



Introducing MySQL Shell

Administration Made Easy with Python

—
Charles Bell

Apress®

www.allitebooks.com

Introducing MySQL Shell

Administration Made Easy
with Python

Charles Bell

Apress®

Introducing MySQL Shell: Administration Made Easy with Python

Charles Bell
Warsaw, VA, USA

ISBN-13 (pbk): 978-1-4842-5082-2
<https://doi.org/10.1007/978-1-4842-5083-9>

ISBN-13 (electronic): 978-1-4842-5083-9

Copyright © 2019 by Charles Bell

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Jonathan Gennick
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484250822. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

*I dedicate this book to my big brother, William E. Bell,
who left this world too soon. I miss you, Bill.*

Table of Contents

About the Author	xiii
About the Technical Reviewer	xv
Acknowledgments	xvii
Introduction	xix
Chapter 1: Introducing the MySQL Shell.....	1
Getting To Know the MySQL Shell	2
Overview.....	2
Features.....	5
Old Features New Again.....	9
JSON Data Type	9
X Plugin, X Protocol, and X DevAPI	15
InnoDB Improvements	16
New Features.....	17
Data Dictionary	18
Account Management.....	19
Removed Options, Variables, and Features	21
Paradigm Shifting Features	23
Document Store.....	23
Group Replication	24
InnoDB Cluster.....	25
Summary.....	26

TABLE OF CONTENTS

- Chapter 2: Installing the MySQL Shell 29**
 - Preparing to Install the MySQL Shell..... 29
 - Prerequisites 29
 - How to Get the MySQL Shell..... 30
 - Installing on Windows with the MySQL Installer 32
 - Installing on macOS 49
 - Installing MySQL Server 51
 - Installing the MySQL Shell..... 57
 - Installing on Linux (Ubuntu) with the APT Repository 62
 - Downloading the APT Repository 63
 - Installing the APT Repository..... 65
 - Installing MySQL Server 66
 - Installing MySQL Shell..... 70
 - Summary..... 73

- Chapter 3: MySQL Shell Tutorial 75**
 - Commands and Options 75
 - Starting the MySQL Shell..... 76
 - Commands 77
 - Options 80
 - Getting Started with the MySQL Shell..... 83
 - Sessions and Modes..... 85
 - Using Connections 88
 - Using a URI 88
 - Using Individual Options..... 90
 - Using Connections in Scripts..... 90
 - Using SSL Connections..... 91
 - Working with the MySQL Shell..... 93
 - Installing the Sample Database..... 93
 - Working with Data..... 96
 - Using Formatting Modes 104
 - Code/Command History..... 108

Saving Passwords	109
Customizing the Shell.....	111
Working with Saved Passwords	115
Changing the Prompt.....	116
Summary.....	118
Chapter 4: Using the Shell with SQL Databases	119
Revisiting Relational Databases.....	119
Working with MySQL Commands and Functions	124
Terminology	124
Creating Users and Granting Access	125
Creating Databases and Tables	126
Storing Data.....	129
Updating Data.....	130
Deleting Data.....	131
Selecting Data (Results)	132
Creating Indexes.....	139
Creating Views.....	140
Simple Joins	141
Additional Advanced Concepts	144
Managing Your Database with Python	146
MySQL X Module	147
CRUD Operations (Relational Data).....	154
Getting Started Writing Python Scripts	170
Summary.....	175
Chapter 5: Example: SQL Database Development.....	177
Getting Started	177
Sample Application Concept.....	178
Database Design.....	181
Code Design	190
Setup and Configuration	192

TABLE OF CONTENTS

- Demonstration..... 194
 - MyGarage Class..... 195
 - Location Class 202
 - Vendor Class..... 209
 - Handtool Class..... 218
 - Organizer Class 220
 - Place Class 221
 - Powertool Class..... 221
 - Storage Class 222
 - Testing the Class Modules..... 222
- Summary..... 230
- Chapter 6: Using the Shell with a Document Store..... 231**
 - Overview 231
 - Origins: Key, Value Mechanisms 232
 - Application Programming Interface..... 234
 - NoSQL Interface..... 234
 - Document Store..... 235
 - JSON..... 235
 - Introducing JSON Documents in MySQL..... 236
 - Quick Start 237
 - Combining SQL and JSON..... 239
 - Formatting JSON Strings in MySQL..... 240
 - Using JSON Strings in SQL Statements..... 241
 - Path Expressions 244
 - JSON Functions 249
 - Summary..... 272

Chapter 7: Example: Document Store Development	275
Getting Started	275
Sample Application Concept.....	276
Schema Design.....	281
Code Design	291
Setup and Configuration	292
Converting Relational Data to a Document Store	293
Importing Data to a Document Store	308
Demonstration.....	310
MyGarage Class.....	311
Collection Base Class	319
Testing the Class Modules.....	328
Summary.....	334
Chapter 8: Using the Shell with Group Replication.....	337
Overview	338
What is High Availability?	338
MySQL High Availability Features	341
What is MySQL Replication?.....	342
What is Group Replication?	344
Setup and Configuration	348
Tutorial	349
Initialize the Data Directories	350
Configure the Master.....	351
Configure the Slaves	353
Start the MySQL Instances	355
Create the Replication User Account	357
Connect the Slaves to the Master.....	358
Start Replication.....	360
Verify Replication Status	363
Shutting Down Replication	365
Summary.....	366

- Chapter 9: Example: Group Replication Setup and Administration 367**
 - Getting Started 367
 - Concepts, Terms, and Lingo 368
 - Group Replication Fault Tolerance 370
 - Setup and Configuration 371
 - Tutorial 371
 - Initialize the Data Directories 373
 - Configure the Primary 374
 - Start the MySQL Instances 379
 - Create the Replication User Account 381
 - Start Group Replication on the Primary 382
 - Connect the Secondaries to the Primary 382
 - Start Group Replication on the Secondaries 383
 - Verify Group Replication Status 383
 - Shutting Down Group Replication 388
 - Demonstration of Failover 388
 - Summary 391

- Chapter 10: Using the Shell with InnoDB Cluster 393**
 - Overview 393
 - InnoDB Storage Engine 397
 - MySQL Shell 403
 - X DevAPI 403
 - AdminAPI 403
 - MySQL Router 404
 - Using InnoDB with Applications 405
 - Setup and Configuration 407
 - Upgrade Checker 407
 - Overview of Installing InnoDB Cluster 411
 - Summary 412

Chapter 11: Example: InnoDB Cluster Setup and Administration	413
Getting Started	413
dba.....	414
cluster	419
Setup and Configuration	421
Create and Deploy Instances in the Sandbox	423
Create the Cluster.....	426
Add the Instances to the Cluster	428
Check the Status of the Cluster	430
Failover Demonstration	432
Using MySQL Router	437
Administration.....	445
Common Tasks	446
Example Tasks	447
Summary.....	450
Chapter 12: Appendix	451
Setup Your Environment.....	451
Installing Flask	453
Installing Flask-Script.....	454
Installing Flask-Bootstrap.....	455
Installing Flask-WTF	456
Installing WTForms	457
Installing Connector/Python	457
Flask Primer	458
Terminology	459
Initialization and the Application Instance.....	461
HTML Files and Templates.....	470
Error Handlers	477
Redirects	479
Additional Features.....	480

TABLE OF CONTENTS

Flask Review: Sample Application 480

 Preparing Your PC..... 481

 Running the Sample Application 483

 How to Use the Application..... 489

 CRUD Operations in the Application..... 490

 Shutting Down the Sample Application 491

Index..... 493

About the Author



Charles Bell conducts research in emerging technologies. He is a member of the Oracle MySQL Development team and is a senior software developer for the MySQL Enterprise Backup team. He lives in a small town in rural Virginia with his loving wife. He received his Doctor of Philosophy in Engineering from Virginia Commonwealth University in 2005.

Charles is an expert in the database field and has extensive knowledge and experience in software development and systems engineering. His research interests include 3D printers, microcontrollers, three-dimensional printing, database systems, software engineering, high availability systems, and cloud and sensor networks. He spends his limited free time as a practicing Maker, focusing on microcontroller projects and refinement of 3D printers.

About the Technical Reviewer



Valerie Parham-Thompson has experience with a variety of open source data storage technologies, including MySQL, MongoDB, and Cassandra, as well as a foundation in web development in software-as-a-service environments. Her work in both development and operations in startups and traditional enterprises has led to solid expertise in web-scale data storage and data delivery.

Valerie has spoken at technical conferences on topics such as database security, performance tuning, and container management and speaks often at local meetups and volunteer events. She holds a bachelor's degree from the Kenan Flagler Business School at UNC-Chapel Hill, has certifications in MySQL and MongoDB, and is a Google Certified Professional Cloud Architect. She currently works in the Open Source Database Cluster at Pythian, headquartered in Ottawa, Ontario.

Follow Valerie's contributions to technical blogs on Twitter at [@dataindataout](https://twitter.com/dataindataout).

Acknowledgments

I would like to thank all the many talented and energetic professionals at Apress. I appreciate the understanding and patience of my editor, Jonathan Gennick, and managing editor, Jill Balzano. They were instrumental in the success of this project. I would also like to thank the army of publishing professionals at Apress for making me look so good in print with a special thank you to the reviewers for their wise counsel and gentle nudges in the right direction. Thank you all very much!

I am also indebted to the technical reviewer for her insight and guidance in making this book the best book (so far, the only book) on MySQL Shell for all levels of MySQL enthusiasts and professionals.

Most importantly, I want to thank my wife, Annette, for her unending patience and understanding while I spent so much time with my laptop.

Introduction

MySQL has been around a long time. I have had the pleasure of witnessing its evolution firsthand as a software developer for Oracle working on MySQL. I have watched MySQL grow from a small database server for web applications to an enterprise-grade high availability database system. The road has not always been smooth as there have been some bumps along the way, but overall Oracle has demonstrated its commitment to the product and continued its evolution.

A perfect example of this dedication is shown in the creation of MySQL Shell, which is a completely new look at how to create a MySQL client. Not only does it replace the existing venerable client, it expands productivity to include programmatic access to the new MySQL Document store (a NoSQL interface) as well as mechanisms for working with MySQL InnoDB Cluster – the next iteration of MySQL high availability.

If you have used the older MySQL client, you will be especially surprised to see how much easier the new MySQL Shell is to use. If that wasn't enough of an incentive, consider you can now write, debug, and execute Python and JavaScript code right from the shell! Yes, we now have scripting capability native to the new client. Yippee!

This book will give you an introduction to MySQL Shell and teach you how to use it for SQL development, working with databases (SQL), writing scripts to interact with the MySQL Document Store to create NoSQL applications, and even how to use Python to work with high availability features such as MySQL Replication, Group Replication and InnoDB Cluster. As you will see, MySQL Shell is the one source for all these uses.

Intended Audience

I wrote this book to share my passion for MySQL and the giant leap forward for MySQL users. I especially wanted to show just how easy and sophisticated MySQL Shell has become. Now, anyone can use MySQL Shell to increase their productivity no matter whether they're working with SQL, NoSQL, or even InnoDB Cluster. The intended audience includes anyone interested in learning about working with MySQL such as database administrators, developers, information technology managers, systems architects, and strategic planners.

How This Book Is Structured

The book was written to guide the reader from a general knowledge of MySQL Shell by introducing its features using example scenarios such as SQL and NoSQL development with the X Developer API (X DevAPI) via Python examples, managing MySQL Replication and Group Replication, working with MySQL InnoDB Cluster via Python scripts, and how to set up and configure MySQL Shell.

The first several chapters cover general topics such as what MySQL Shell is, its features, and how to install it on your system. Later chapters present four scenarios for using MySQL Shell including working with SQL databases with Python, working with MySQL Document Store with Python, configuring MySQL Replication and Group Replication, and setup and managing MySQL InnoDB Cluster with Python. Each of these chapters is presented with an introduction for each topic followed by a companion chapter that presents a detailed example to illustrate the concepts.

- Chapter 1, “Introducing MySQL Shell”: This chapter introduces MySQL Shell including a brief tour of the new features in MySQL realized in the shell.
- Chapter 2, “Installing the MySQL Shell”: This chapter discusses and presents an example of how to download and install MySQL Shell on Windows, macOS, and Linux.
- Chapter 3, “MySQL Shell Tutorial”: This chapter presents a short tutorial on the commands and options used in MySQL Shell, how to use the shell to connect to MySQL servers, and how to work with the shell.
- Chapter 4, “Using the Shell with SQL Databases”: This chapter briefly discusses working with relational databases including a brief look at the more common SQL commands and functions.
- Chapter 5, “Example: SQL Database Development”: This chapter presents a complete Flask Python web application that demonstrates how to use MySQL Shell to develop the Python modules for the SQL application.
- Chapter 6, “Using the Shell with a Document Store”: This chapter briefly introduces JSON documents and the MySQL Document Store

including a brief demonstration of the Document Store. The chapter also demonstrates how to use JSON in SQL databases.

- Chapter 7, “Example: Document Store Development”: This chapter presents a complete Flask Python web NoSQL application that demonstrates how to use MySQL Shell to develop the Python modules for the NoSQL application.
- Chapter 8, “Using the Shell with MySQL Replication”: This chapter presents an overview of the high availability features in MySQL including a brief tutorial on MySQL Replication.
- Chapter 9, “Example: Group Replication Setup and Administration”: This chapter presents a short tutorial on Group Replication including a demonstration of how to use MySQL Shell to configure MySQL Group Replication. The chapter also demonstrates how failover works in MySQL Group Replication.
- Chapter 10, “Using the Shell with InnoDB Cluster”: This chapter introduces MySQL InnoDB Cluster as well as how InnoDB Cluster can be used with applications.
- Chapter 11, “Example: InnoDB Cluster Setup and Administration”: This chapter presents a complete tour of MySQL InnoDB Cluster setup and administration using MySQL Shell and the AdminAPI with Python.
- Appendix: This bonus chapter presents a short primer on Flask and how to set up the example applications in Chapters 5 and 7. You learn how to get started writing Flask Python web applications.

How to Use This Book

This book is designed to guide you through learning more about MySQL Shell, discovering the power of X DevAPI as well as the AdminAPI for working with MySQL InnoDB Cluster, and seeing how to build applications with the X DevAPI and Python.

If you are new to MySQL Shell, you should spend some time going through the first three chapters and installing MySQL Shell on your own system and learning how to use it.

INTRODUCTION

The remaining chapters can be read in pairs with the first introducing one of the four scenarios covered and the second providing a complete example walk through. You can read the pairs of chapters in any order. Even if you are not familiar with some of the scenarios or do not plan to use the knowledge or examples presented as a basis, reading about how MySQL Shell supports the scenario can be helpful in the future as your infrastructure grows.

Finally, those interested in migrating existing applications or perhaps want to write new applications using the X DevAPI may find Chapters 4–8 enlightening as these chapters demonstrate both an SQL application (without any SQL commands) and a NoSQL application in Python.

Downloading the Code

The code for the examples shown in this book is available on the Apress web site, www.apress.com. You can find a link on the book's information page on the Source Code/Downloads tab. This tab is in the “Related Titles” section of the page.

Contacting the Author

Should you have any questions or comments – or even spot a mistake you think I should know about – you can contact me at drcharlesbell@gmail.com.

CHAPTER 1

Introducing the MySQL Shell

Oracle has continued to live up to its commitment to making MySQL better. This has empowered the MySQL engineering division to reach higher and further with each new release. The newest version, MySQL 8, contains more new features and enhancements than any other release. As a result, MySQL continues to be the world's most popular open source database system.

To fully understand the significance of the MySQL 8 release, let us consider that while past releases of MySQL have continued to improve the product, the releases tended to contain a few new features with emphasis on improving the more popular features. Thus, previous releases were largely evolutionary rather than revolutionary.

MySQL 8 breaks with this tradition in several ways. Most notably perhaps is the version number itself. Previous versions were in the 5.X range of numbers, but Oracle has chosen to use the 8.X series signifying the revolutionary jump in technological sophistication and finally breaking away from continuous development of the 5.X codebase that has lasted for over 14 years.

The revolutionary changes to MySQL 8.0 include features dedicated to high availability, greater reliability, and sophistication as well as a completely new user experience and revolutionary way to work with your data. This book examines one of the most important additions that enables the new user experience – the MySQL Shell. In this chapter, we will get a short overview of the newest features in MySQL 8. But first, let's get to know the MySQL Shell better.

Getting To Know the MySQL Shell

One of the pain points for many MySQL users has been the limitations of the default client utility. For several decades, the client of choice (because there wasn't anything else) has been the MySQL client utility named `mysql`, which is included with the server releases.

Perhaps the biggest missing feature in the old MySQL client (`mysql`) was the absence of any form of scripting capability. One could argue that scripting SQL commands is possible with the old client to process a batch of SQL commands. And others may point out that there is limited support in the SQL language supported by MySQL for writing stored routines (procedures and functions). However, those who wanted to create and use a scripting language for managing their databases (and server), there have been external tool options including the MySQL Workbench and MySQL Utilities (now retired), but nothing dedicated to incorporating scripting languages.

Note MySQL Workbench is a GUI tool designed as a workstation-based tool with a host of features including design and modeling, development, database migration, and more. See <http://dev.mysql.com/doc/workbench/en/> for more information about MySQL Workbench.

Aside from these products, there has been no answer to requests to add scripting languages to the MySQL client. That is, until now.

Note I use the term “shell” to refer to features or objects supported by the MySQL Shell. I use “MySQL Shell” to refer to the product itself.

Overview

The MySQL Shell is the next generation of command-line client for MySQL. Not only can you execute traditional SQL commands, you can also interact with the server using one of several programming languages including Python and JavaScript. Furthermore, if you also have the X Plugin installed, you can use MySQL Shell to work with both traditional relational data as well as JavaScript Object Notation (JSON) documents. How cool is that?

If you're thinking, "It is about time!" that Oracle has made a new MySQL client, you're not alone. The MySQL Shell represents a bold new way to interact with MySQL. There are many options and even different ways to configure and use the shell. While we will see more about the shell in the upcoming chapters, let's take a quick look at the shell. Figure 1-1 shows a snapshot of the new MySQL Shell. Notice it provides a very familiar interface albeit a bit more modern and far more powerful. Notice also the new prompt. Not only is it more colorful, it also provides a quick check to see what mode you are in. In this case, it is showing JS, which is JavaScript mode (default mode¹). You can also modify the prompt to your liking.

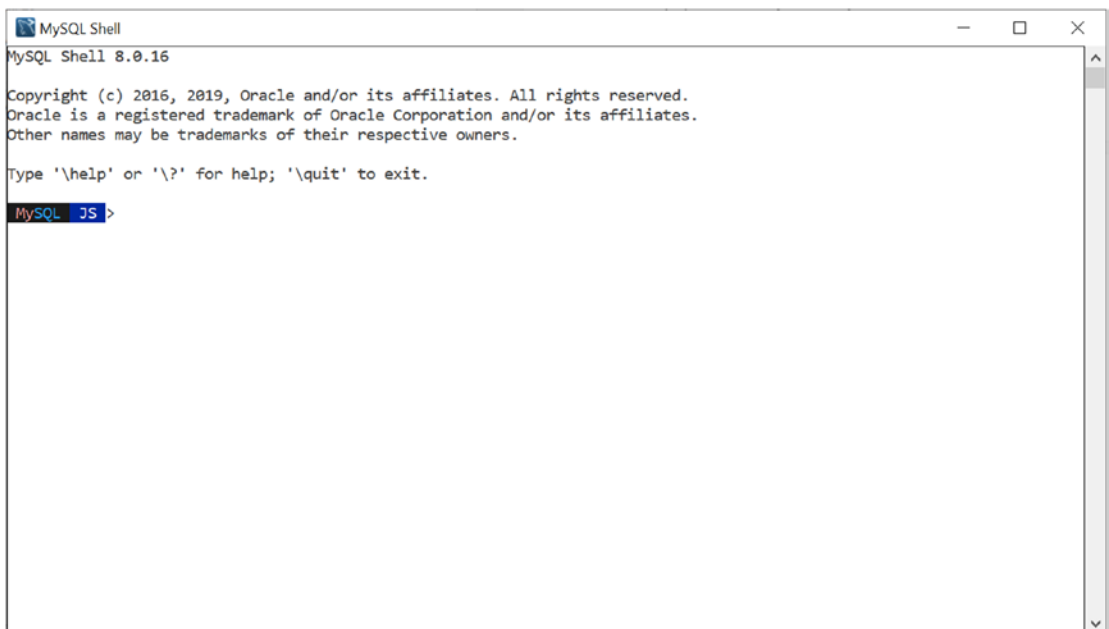


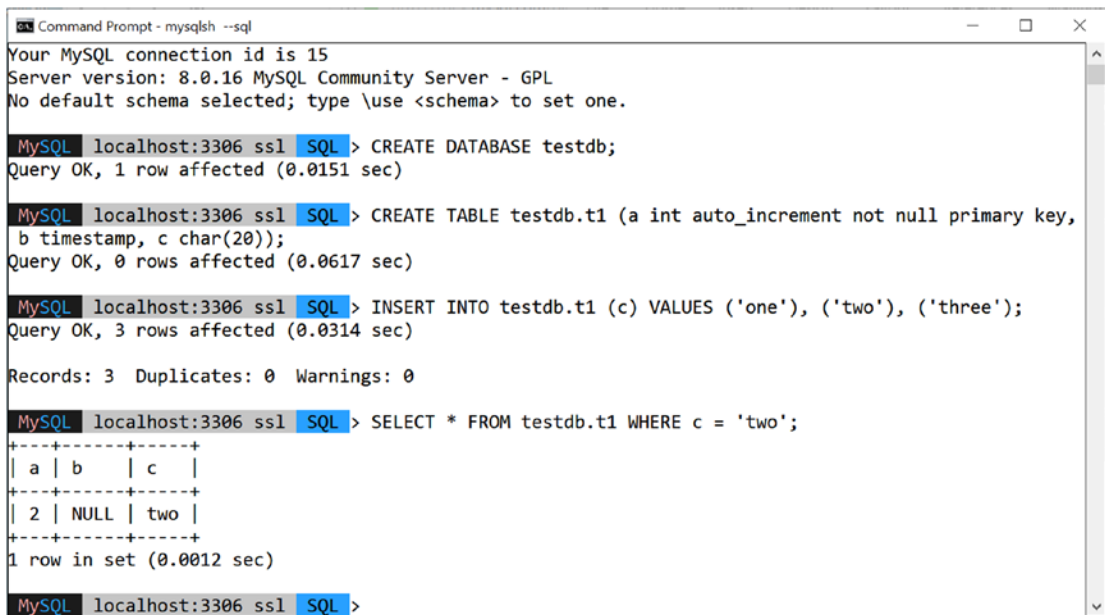
Figure 1-1. *The MySQL Shell*

Tip If you'd like to keep tabs on the MySQL Shell releases, bookmark <https://dev.mysql.com/downloads/shell/>, which includes links to the documentation and downloads for popular platforms.

¹This may be the source of rumors regarding the shell not supporting SQL – the default mode is JavaScript, but as you can see, SQL is also supported. We'll see how to set the mode on start in the next chapter.

If you've read about the MySQL Shell using an entirely new mechanism for accessing data and that you must learn all new commands, you may have been led astray. While the MySQL Shell does indeed support a new application programming interface (API) to access data using a scripting language and in that sense there are new commands (methods) to learn, the MySQL Shell continues to support a SQL interface to your data. In fact, all the SQL commands you've come to know are fully supported. In fact, the MySQL Shell is designed to be your primary tool for working with discrete SQL commands.

Let's see an example of using the MySQL Shell with SQL commands. Figure 1-2 shows a typical set of SQL commands to create a database, insert data, and select some of the data for viewing.



```
Command Prompt - mysqlsh --sql
Your MySQL connection id is 15
Server version: 8.0.16 MySQL Community Server - GPL
No default schema selected; type \use <schema> to set one.

MySQL localhost:3306 ssl SQL > CREATE DATABASE testdb;
Query OK, 1 row affected (0.0151 sec)

MySQL localhost:3306 ssl SQL > CREATE TABLE testdb.t1 (a int auto_increment not null primary key,
b timestamp, c char(20));
Query OK, 0 rows affected (0.0617 sec)

MySQL localhost:3306 ssl SQL > INSERT INTO testdb.t1 (c) VALUES ('one'), ('two'), ('three');
Query OK, 3 rows affected (0.0314 sec)

Records: 3 Duplicates: 0 Warnings: 0

MySQL localhost:3306 ssl SQL > SELECT * FROM testdb.t1 WHERE c = 'two';
+-----+-----+-----+
| a | b | c |
+-----+-----+-----+
| 2 | NULL | two |
+-----+-----+-----+
1 row in set (0.0012 sec)

MySQL localhost:3306 ssl SQL >
```

Figure 1-2. Using the MySQL Shell with SQL Commands

Here, we see several things happening. First, we change the mode of the shell from JavaScript to SQL using the `\sql` command. Then, we connect to the server using the `\connect` command. Notice the command requests the user password and, if this is the first time using that connection, you can have the shell save the password for you (a nice security feature). From there, we see several mundane examples of running SQL commands. Listing 1-1 shows the commands used in this example.

Listing 1-1. Sample Commands (Getting Started with MySQL Shell)

```
\sql
\connect root@localhost:3306
CREATE DATABASE testdb;
CREATE TABLE testdb.t1 (a int auto_increment not null primary key,
b timestamp, c char(20));
INSERT INTO testdb.t1 (c) VALUES ('one'), ('two'), ('three');
SELECT * FROM testdb.t1 WHERE c = 'two';
```

Now that we've been introduced to the MySQL Shell, let's look at its impressive list of features. You are likely to find there are several that you will find can make your experience with MySQL better.

Features

The MySQL Shell has many features including support for traditional SQL command processing, script prototyping, and even support for customizing the shell. The following lists some of the major features of the shell. Most of the features can be controlled via command line options or with special shell commands. The list is presented to give you an idea of the breadth of features in the shell and is presented without examples. We take a deeper look at some of the more critical features in later chapters.

Tip Some of the jargon here may seem unfamiliar. It is not important to understand these at this point, but we will discover each of these in later chapters.

- *Auto Completion:* The shell allows auto completion for keywords in SQL mode and all the major classes and methods in either JavaScript or Python. Simply type a few characters, then press the TAB key to autocomplete keywords. This can be a very handy tool when learning the new APIs and trying to recall the spelling of a seldom used SQL keyword or MySQL function.

- *APIs*: The shell supports JavaScript and Python that interact with the following application programming interfaces:
 - *XDevAPI*: This API permits you to interact with the MySQL server using either relational data or the document store (JSON).
 - *AdminAPI*: This API permits you to interact with the MySQL InnoDB Cluster for setup, configuration, and maintenance of a high-availability cluster.
- *Batch Code Execution*: If you want to run your script without the interactive session, you can use the shell to run the script in batch mode – just like the old client.
- *Command History*: The shell saves the commands you enter allowing you to recall them using the up and down arrow keys.
- *Customize the Prompt*: You can also change the default prompt by updating a configuration file named `~/.mysqlsh/prompt.json` using a special format or by defining an environment variable named `MYSQLSH_PROMPT_THEME`.
- *Global Variables*: The shell provides a few global variables you can access when using the interactive mode. These include the following. We will learn more about working with sessions and the variables in [Chapter 3](#).
 - `session`: Global session object if established
 - `db`: Schema if established via a connection
 - `dba`: The AdminAPI object for working with the InnoDB Cluster
 - `shell`: General purpose functions for using the shell
 - `util`: Utility functions for working with servers
- *JSON Import*: Typing JavaScript Object Notation (JSON) can be a bit tedious. The shell makes working with JSON even easier by permitting users to import JSON documents into the shell. The import feature is enabled in both interactive commands and API functions.

- *Interactive Code Execution:* The default mode for using the shell is interactive mode, which works like the old MySQL client where you enter a command and get a response.
- *Logging:* You can create a log of your session for later analysis or to keep a record of messages. You can set the level of detail with the `--log-level` option ranging from 1 (nothing logged) to 8 (max debug).
- *Multi-Line Support:* The shell permits you to enter commands caching them to be executed as a single command.
- *Output Formats:* The shell supports three format options; table (`--table`), which is the traditional grid format you're used to from the old client, tabbed (`--tabbed`), which presents information using tabs for spacing and is used for batch execution, and JSON (`--json`), which formats the JSON documents in an easier to read manner. These are command-line options you specify when launching the shell.
- *Scripting Languages:* The shell supports both JavaScript and Python, although you can use only one at a time.

Note In this book, we will focus on Python, but the API for using JavaScript is the same. The only difference is in how the classes and methods are spelled. This is because JavaScript uses a different convention for capitalization and multiword identifiers. Savvy JavaScript developers will have no trouble translating the examples in this book.

- *Sessions:* Sessions are essentially connections to servers. The shell allows you to work with sessions including storing and retrieving them when needed.
- *Startup Scripts:* You can define a script to execute when the shell starts. You can write the script in either JavaScript or Python.

- *Upgrade Checker*: The shell also includes a handy upgrade checking tool that lets you check a given server to see if it can be upgraded to MySQL 8. It is a real time saver for those who have existing MySQL servers migrating to MySQL 8.
- *User Credentials “Secret” Store*: Perhaps the most time saving feature of all is the ability to save user passwords to a “secret store” or encrypted credential storage mechanism common to platforms or platform specific stores. The shell supports the MySQL login-path, MacOS keychain, and the Windows API. This feature is turned on by default but may be disabled on a user credential basis (you don’t have to store the password if you don’t want to). If you’re working with a single system or a protected account across several systems, this will save you time by recalling the password for that user from the secret store. We’ll see more about this feature in Chapter 3.

The latest release of MySQL Shell (8.0.16) includes a host of bug fixes as well as some new features that are sure to be appreciated and used frequently. These include the following.

- *User Defined Reports*: You can now setup reports to display live information from the server such as status and performance data. See <https://dev.mysql.com/doc/mysql-shell/8.0/en/mysql-shell-reporting.html> for more information about this new feature if you want to monitor your server metadata and status variables.
- *SQL Mode Execution*: If you are using the shell in Python or JavaScript mode and want to run an SQL command, the `\sql` shell command now allows you to specify a SQL command to run. For example, you can execute `\sql SHOW DATABASES` and not have to switch to the SQL mode (and back).
- *AdminAPI*: Now reports information about the server version in the `status()`, `describe()`, and `rescan()` methods.

The MySQL Shell isn’t the only thing that is new in MySQL 8. In fact, there are a lot of things to like and explore in the latest release of MySQL. Indeed, there are features that have been improved, new features introduced like the MySQL Shell, as well as some extraordinarily unique features that will change how you use MySQL. Some

of these features incorporate the MySQL Shell as a key component. Since we plan to explore how to use these features with the shell, let's take a few moments and learn what is new in MySQL 8.

Old Features New Again

This category includes those features that were introduced in earlier versions of MySQL either as a separate download or as a plugin. Some were considered experimental even though they may have been introduced during a general announcement (GA) release cycle (the feature may not have been GA). While some features may be introduced in this manner in the future, currently all of these are now part of the MySQL 8 GA in a much more refined form. These include the following. What is not listed here are the hundreds of small-to-moderate enhancements and defect repairs included in the release.

- *JSON Data Type*: the most revolutionary change to data includes the incorporation of the JSON data type, which permits the use of MySQL as a true NoSQL database system.
- *X Plugin, X Protocol, and X DevAPI*: The server now supports the new client protocol upon which all the new APIs have been built.
- *InnoDB Improvements*: Aside from being the default storage engine, InnoDB has become a much more robust, enterprise-grade atomicity, consistency, isolation, and durability (ACID) compliant storage engine.

JSON Data Type

As of MySQL version 5.7.8, MySQL supports the JSON data type. The JSON data type can be used to store JSON documents in a relational table. Thus, you can have JSON columns in your table! You can have more than one JSON column (field) in a single table.

The JSON data type is also a key component to using MySQL as a document store. In short, JSON is a markup language used to exchange data. Not only is it human readable, it can be used directly in your applications to store and retrieve data to and from other applications, servers, and even MySQL.

Note The following is a brief overview of the JSON data type and JSON documents. We will see an in-depth look at JSON in Chapter 6.

In fact, JSON looks familiar to programmers because it resembles other markup schemes. JSON is also very simple in that it supports only two types of structures: (1) a collection containing (name, value) pairs and (2) an ordered list (or array). Of course, you can also mix and match the structures in an object. When we create a JSON object, we call it a JSON document.²

The JSON data type, unlike the normal data types in MySQL, allows you to store JSON formatted objects (documents) in a column for a row. While you could do this with TEXT or BLOB fields (and many people do), there is no facility built into MySQL to interact with the data in TEXT and BLOB fields. Thus, the manipulation of the data is largely application dependent. Additionally, the data is normally structured such that every row has the same "format" for the column. Storing data in TEXT and BLOB fields is not new and many have done this for years.

With the JSON data type, we don't have to write any specialized code to store and retrieve data. This is because JSON documents are well understood and many programming environments and scripting languages support it natively. JSON allows you to store data that you have at the time. Unlike a typical database table, we don't have to worry about default values (they're not allowed) or whether we have enough columns or even master/detail relationships to normalize and store all the data in a nice, neat, structured package.

Let's take a sneak peek at the JSON data type. Let's assume you want to store addresses in your database, but you cannot guarantee all the items you are storing will have an address and some may have multiple addresses. Worse, it may be that the address information you have is inconsistent. That is, the addresses vary in form and what data is provided. For example, you may have one, two, or even three lines of "street" address but other addresses may have a single line with a post office box number. Or, some addresses include a five-digit zip code while others have a nine-digit zip code or even some may have a postal code (like in Canadian post).

²Think of JSON as an outgrowth or extension of what XML documents were supposed to be. That is, they offer a flexible way to store data that may differ from one entry to another.

In this situation, you could either add the address fields to your existing table (but this does not solve the case where rows could have more than one address) or, better, create a relational table to store the addresses and shoehorn the data into the fields. For addresses that do not conform, you may be forced to use default values or even store NULL for the missing items. While all this is possible, it forces a layer of complexity in your relational database that may mean additional code for processing the missing data.

Figure 1-3 shows a schema for a typical relational database that contains addresses stored as a separate table. This excerpt, albeit very terse and incomplete, demonstrates the typical approach database designers take when dealing with data like addresses that can vary from one item to another.

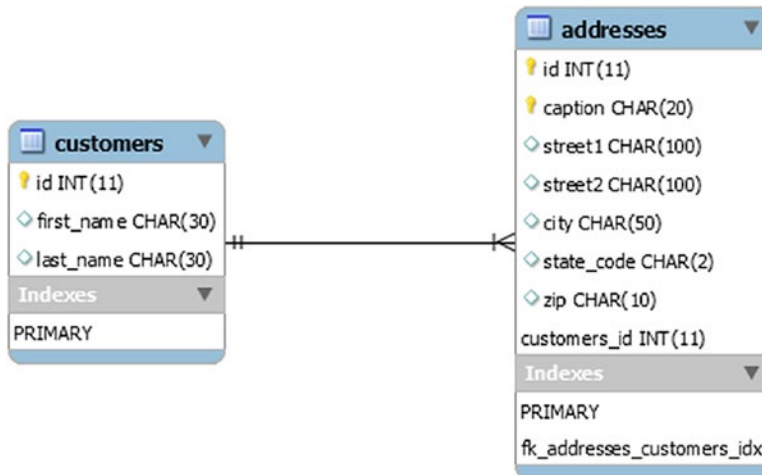


Figure 1-3. Sample Relational Database Excerpt

There's nothing wrong with this approach, but to appreciate the advantages the JSON data type gives us, let's look at a typical set of SQL statements to create the sample and insert some data. Listing 1-2 shows the example SQL statements used to create the tables. I include the shell-specific commands to switch to SQL mode and connect to the server.

Listing 1-2. Sample Relational Database SQL (no JSON)

```
DROP DATABASE IF EXISTS my dB;
CREATE DATABASE mydb;
CREATE TABLE mydb.customers (id int auto_increment NOT NULL PRIMARY KEY,
first_name char(30), last_name char(30));
```

```
CREATE TABLE mydb.addresses (id int NOT NULL, caption char(20) NOT NULL,
street1 char(100), street2 char(100), city char(50), state_code char(2),
zip char(10), PRIMARY KEY(id, caption));
INSERT INTO mydb.customers VALUES (NULL, 'Sam', 'Blastone');
SELECT LAST_INSERT_ID() INTO @last_id;
INSERT INTO mydb.addresses VALUES (@last_id, 'HOME', '9001 Oak Row Road',
Null, 'LaPlata', 'MD', '33532');
INSERT INTO mydb.addresses VALUES (@last_id, 'WORK', '123 Main Street',
Null, 'White Plains', 'MD', '33560');
SELECT first_name, last_name, addresses.* FROM mydb.customers JOIN mydb.
addresses ON customers.id = addresses.id \G
```

So far, the addresses added are somewhat normal. But consider the possibility that we want to add another address but this one is incomplete. For example, we know only the city and state for a warehouse location for this customer where we learn the customer spends some of his time. We want to store this information so that we can know the customer has a presence in that area, but we may not know any more details. If we continue to use this relational example, we will be adding several empty fields (which is Ok). When we run the SELECT query after inserting the incomplete address, we get this information.

```
> SELECT first_name, last_name, addresses.* FROM mydb.customers JOIN mydb.
addresses ON customers.id = addresses.id \G
```

...

***** 2. ROW *****

```
first_name: Sam
last_name: Blastone
      id: 1
caption: WAREHOUSE
street1: NULL
street2: NULL
      city: Carson Creek
state_code: CO
      zip: NULL
```

What we end up with is a single row in the one table (customers) and three rows in the other table (addresses) but with several empty (Null) fields. Now, let's see this same example only this time, we will use a JSON data type to store the addresses.

In this next example, we replace the second table (the address detail table) with a single column in the customers table assigning the JSON data type. Besides the obvious removal of a second table and the relationship that one must traverse to query data, we also gain the ability to store only what we need. Listing 1-3 shows the modified SQL statements to build this version.

Listing 1-3. Sample Relational Database SQL (JSON)

```
DROP DATABASE IF EXISTS mydb_json;
CREATE DATABASE mydb_json;
CREATE TABLE mydb_json.customers (id int auto_increment NOT NULL PRIMARY
KEY, first_name char(30), last_name char(30), addresses JSON);
INSERT INTO mydb_json.customers VALUES (NULL, 'Sam', 'Blastone',
'{"addresses":[
  {"caption":"HOME","street1":"9001 Oak Row
  Road","city":"LaPlata","state_code":"MD","zip":"33532"},
  {"caption":"WORK","street1":"123 Main Street","city":"White
  Plains","state_code":"MD","zip":"33560"},
  {"caption":"WAREHOUSE","city":"Carson Creek","state_code":"CO"}
]}' );
SELECT first_name, last_name, JSON_PRETTY(addresses) FROM mydb_json.
customers \G
```

Here, we see the SQL to define the table is a lot shorter. To use the JSON data type, we simply specify JSON where we would any other data type. However, entering data with JSON values takes a bit more typing, but as you can see, it allows us to use expressions that describe the code in a language that is easily understood. Querying this data will return the JSON strings as a single string, but we can use one of the MySQL JSON functions to help make the output more readable. For this, we use the `JSON_PRETTY()` function as shown in Listing 1-4, which puts newlines and spacing in the string as it is returned from the server.

Listing 1-4. Querying Rows with JSON Data

```
> SELECT first_name, last_name, JSON_PRETTY(addresses) FROM mydb_json.
customers \G
***** 1. ROW *****
      first_name: Sam
      last_name: Blastone
JSON_PRETTY(addresses): {
  "addresses": [
    {
      "zip": "33532",
      "city": "LaPlata",
      "caption": "HOME",
      "street1": "9001 Oak Row Road",
      "state_code": "MD"
    },
    {
      "zip": "33560",
      "city": "White Plains",
      "caption": "WORK",
      "street1": "123 Main Street",
      "state_code": "MD"
    },
    {
      "city": "Carson Creek",
      "caption": "WAREHOUSE",
      "state_code": "CO"
    }
  ]
}
1 row in set (0.0036 sec)
```

Notice we have a single table now and the addresses have been “collapsed” into the JSON column where each row stores the array of addresses. And, only the data known is stored. So, in the case of the warehouse address, we store only the city and state. While not as easy to read in the shell output (we will see some ways to improve readability in

output format later), we can still easily see the data. And, when used in our applications, ingesting the JSON will be much easier than having to check each column for data. We'll learn more about this in Chapter 6.

As we discovered, the JSON data type enables building flexibility into our data storage. As we will discover in Chapter 6, we can take that concept a step further by storing all our data as JSON documents using the document store through the support built into MySQL and the MySQL Shell via the X Plugin, X Protocol, and X DevAPI. In fact, let's now discover what makes the shell powerful by examining the new X Plugin and X Protocol.

X Plugin, X Protocol, and X DevAPI

MySQL has introduced a new protocol and API to work with JSON documents. Along with supporting the JSON data type, we have three technologies prefixed with the simple name “X”: the X Plugin, X Protocol, and X DevAPI. The X Plugin is a plugin that enables the X Protocol. The X Protocol is designed to communicate with the server using the X DevAPI. The X DevAPI is an application-programming interface that, among many things, permits you to develop NoSQL solutions for MySQL and use MySQL as a document store. We will learn more about the document store in a later section.

You may be wondering how the shell and the plugin interact with the server. Figure 1-4 shows how the components are “stacked”.

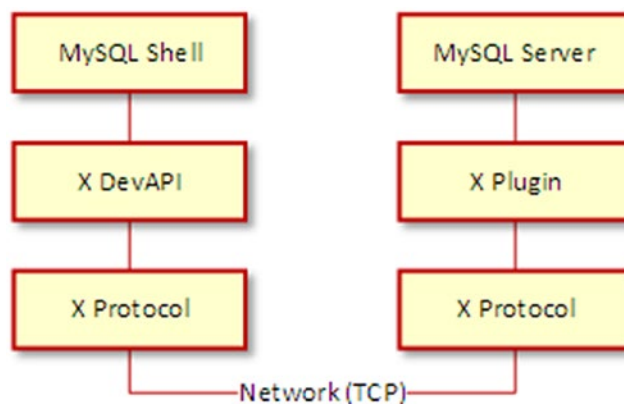


Figure 1-4. X Protocol Stack

Notice we have the shell that permits use of the X DevAPI, which is communicated over the wire to the server via the X Plugin. Thus, the X Plugin is an enabling technology with the real power being the X Protocol and X DevAPI.

Now that we've seen the technologies that enable using MySQL as a document store, let's look at how the InnoDB storage engine has changed in recent releases.

InnoDB Improvements

Since MySQL 5.6, InnoDB has been the flagship storage engine (and the default engine) for MySQL. Oracle has slowly evolved away from the multiple storage engine model focusing on what a modern database server should do – support transactional storage mechanisms. InnoDB is the answer to that requirement and much more.

WHAT IS A STORAGE ENGINE?

A storage engine is a mechanism to store data in various ways. For example, there is a storage engine that allows you to interact with comma-separated values (text) files (CSV), another that is optimized for writing log files (Archive), one that stores data in memory only (Memory), and even one that doesn't store anything at all (Blackhole). You can use them with your tables by using the `ENGINE = table` option. Along with InnoDB, the MySQL server ships with the Archive, Blackhole, CSV, Memory, and MyISAM storage engines. The InnoDB storage engine is the only one that supports transactions. For more information about the other storage engines including the features of each and how they are used, see the "Alternative Storage Engines" section in the online reference manual.

In the early days, InnoDB was a separate company and thus a separate product that was not part of MySQL nor was it owned by MySQL AB (the original owner of MySQL now fully owned by Oracle). Eventually, Oracle came to own both InnoDB and MySQL, so it made sense to combine the two efforts since they have mutually inclusive goals. While there still exists a separate InnoDB engineering team, they are fully integrated with the core server development team.

This tight integration has led to many improvements in InnoDB including a host of performance enhancements and even support for fine-tuning and more. This is readily apparent in how InnoDB continues to evolve with refinements and never more so than the state of InnoDB in MySQL 8.

While most of the improvements are rather subtle in the sense you won't notice them (except through better performance and reliability, which are not to be taken lightly), most show a dedication to making InnoDB the best transactional storage mechanism and through extension MySQL a strong transactional database system.

The most significantly improved areas include performance and stability. Once again, you may not see a lot of differences for smaller databases, but larger databases and enterprise-grade systems will see a noticeable improvement. For example, crash recovery and logging have been improved considerably making recovery faster as well as normal shutdown and startup faster.

Similarly, improvements in deadlock detection, temporary tables, auto-increment, and even Memcached support demonstrate Oracle's desire to leave no stone unturned correcting defects and improving InnoDB. While this list seems focused on minor improvements, some of these are very important to system administrators looking for help tuning and planning their database server installations.

Tip If you would like to know more about any of these improvements or see a list of all the latest changes, see the online MySQL 8 Reference Manual (<http://downloads.mysql.com/docs/refman-8.0-en.pdf>).

The next section describes those features that have been added to and are unique to MySQL 8.

New Features

Aside from those features that have been in development during the 5.7 server releases, there are features that are unique to MySQL 8. They are not currently (or even likely to be incorporated) in the older releases. Part of this is because of how much the server code base was changed to accommodate the new features. Those new features available in MySQL 8.0 include the following:

- *Data Dictionary*: A transactional metadata storage mechanism for all objects in the system
- *Account Management*: Major improvements in user, password, and privilege management

- *Removed Options, Variables, and Features:* Those wanting to upgrade from older versions should review the smaller details that have changed in the new release such as options, variables, and features that have been removed

Aside from those features that have been in development during the 5.7 server releases, there are features that are unique to MySQL 8. In fact, they are not currently (or even likely to be incorporated) in the older releases. Part of this is because of how much the server code base was changed to accommodate the new features. Those new features available in MySQL 8.0 include the new data dictionary and a new account management system.

Data Dictionary

If you have ever worked with MySQL trying to get information about the objects contained in the databases, either to discover what objects are there, searching for objects with a specific name prefix, or trying to discover what indexes exist, chances are you have had to access the tables and views in INFORMATION_SCHEMA or the special mysql databases you've had to navigate. Perhaps worst is some of the definitions of the tables were stored in a special structured file called an .frm³ (form) file. For example, a table named table1 in database1 has an .frm file named /data/database1/table1.frm.

Note The INFORMATION_SCHEMA and mysql databases are still visible and the information in those views and tables can still be used in a similar manner before the data dictionary but lack the additional information in the data dictionary.

This combination required database administrators to learn how to find things by learning where the data resided. Depending on what you were looking for, you may have had to query one of the databases or, in dire situations, decipher the .frm file. More importantly, since the data was in non-transactional tables (and metadata files), the mechanisms were not transactional and, by extension, not crash safe.

³The .frm file has been a source of diabolical difficulties when the files are lost or corrupt.

The new data dictionary changes all of that for us making a single, transactional (same ACID support as InnoDB) repository to store all the metadata for objects in the system. All the file-based metadata has been moved into the data dictionary including the `.frm` files, partition, trigger, and other options files (e.g., `.par`, `.trn`, `.trg`, `.isl`, and `.opt`).

However, you won't see the data dictionary in a list of the databases (e.g., `SHOW DATABASES`). The data dictionary tables are invisible and cannot be accessed directly. You won't find the data dictionary tables easily (although it is possible if you look hard enough).

This was done primarily to make the data dictionary crash safe and something you don't have to manage. Fortunately, you can access the information stored in the data dictionary via the `INFORMATION_SCHEMA` database and even the `SHOW` commands. The `mysql` database still exists, but it mainly contains extra information such as time zones, help, and similar non-vital information. In fact, the `INFORMATION_SCHEMA` and `SHOW` commands use the data dictionary to present information.

So, how do you use the data dictionary if you can't see it? Simply, the `INFORMATION_SCHEMA` views derive information from the data dictionary. So, you can continue to use the same queries you're used to using, but in this case, the data is more reliable and transactional. Cool!

For more information about the data dictionary including details on what is stored and how it interacts with the `INFORMATION_SCHEMA` views, see the section "MySQL Data Dictionary" in the online MySQL reference manual (<https://dev.mysql.com/doc/refman/8.0/en/>).

Adding the data dictionary has finally made possible several features that many have wanted to implement for some time. One of the newest is a major overhaul in account management.

Account Management

Another pain point for MySQL database administrators, especially those that work with enterprise-grade systems, is the need to assign the same privileges to a group of users and manage passwords. MySQL 8 provides numerous improvements in account management and the privilege system in MySQL. The following list the most significant improvements.

- *Roles*: Administrators can assign grant statements to a role, which can be assigned to multiple users.
- *User account limits*: Administrators can set resource limits to help restrict access for critical data.
- *Password management*: Administrators can set conditions for password formation and expiration.
- *User account locking*: Administrators can temporarily lock user accounts from accessing data.

Note MySQL 8 disables the ability to create the user account with the GRANT statement. You must explicitly create the user first with the CREATE USER statement.

Roles

A very common scenario involves having to create a set of users with the same permissions. In the past, you would have to save or archive the GRANT statements and repeat them for each user. That is, you would reuse the GRANT statements for two or more users. Fortunately, with the advent of the data dictionary, supporting roles in MySQL has become a reality in MySQL 8!

Roles can be created, dropped, privileges granted or revoked. We can also grant or revoke roles to/from users. Roles finally make the tedium of managing user accounts on MySQL much easier.

User Account Limits

Another administrative problem for enterprise systems includes the need to further restrict access to user accounts during certain time periods or even restrict the account from issuing a certain number of statements.

In MySQL 8, administrators can set limits on user accounts for number of queries per hour, number of transactions per hour, connections per hour, and even the number of simultaneous connections per hour. This permits administrators to set limits for security goals, productivity limits, and more.

Password Management

One of the most requested changes to the account management features is the ability to set limits and standards for passwords. In today's challenging security climate, we must ensure our passwords are not easily hacked and to do that we need to exert control over how long passwords must be as well as how many characters in lower or upper case or special characters are included.

Fortunately, MySQL 8 has these features and you can set password expiration, reuse of old password restrictions, verification of passwords, and of course password strength. These features and the previous two mentioned have helped propel MySQL 8 a giant leap forward in better security.

User Account Locking

Sometimes it is the case that you must temporarily restrict access to one or more user accounts. This may be due to maintenance schedules, diagnostics, or even the temporary furlough of an employee. Whatever the reason, MySQL in the past required either changing of the password (and not telling the user – but this doesn't prevent the account from being used) or deleting the account and recreating it later. If your user accounts have complex privileges granted to them (or several roles), this is problematic at best.

MySQL 8 includes a feature that allows support for locking and unlocking user accounts using the `ACCOUNT LOCK` clause to lock the account and the `ACCOUNT UNLOCK` clause to unlock the account. You can use these clauses in either the `CREATE USER` statement or the `ALTER USER` statement.

There are many more minor improvements to the account management feature. To read more about the changes, see the section “User Account Management” in the online reference manual (<https://dev.mysql.com/doc/refman/8.0/en/>).

Removed Options, Variables, and Features

If you read the release notes for MySQL 8, you may notice about MySQL 8 is a host of small changes to startup options, variables, and the like. A complete list of all changes for each release of MySQL 8 can be found at <https://dev.mysql.com/doc/relnotes/mysql/8.0/en/>.

Fortunately, most of the changes are related to supporting the newest features and removal of old and obsolete settings. So, most won't have much effect on those wanting to start using MySQL. However, a few may be of minor concern when upgrading from older versions.

Keep in mind that many of those options, variables, and features were marked as deprecated in MySQL 5.7 (and prior). They are now officially removed in MySQL 8. If you have been using MySQL for some time, then you most likely have already prepared for the changes.

However, there are a couple of changes that may affect some wanting to upgrade to MySQL 8. These include the following.

- The `--bootstrap` option was removed. It was used to control how the server started and was typically used to create the MySQL privilege tables without having to start a full MySQL server.
- The `--innodb_file_format_*` options were changed. These were used to configure the file format for the InnoDB storage engine.
- The `--partition` and `--skip partition` options were removed. They were used to control user-defined partitioning support in the MySQL Server.
- The `.frm` and related metadata files were removed as part of the data dictionary feature.
- Some of the SSL options have changed and the introduction of a new authentication plugin (`caching_sha2_password`) to improve secure connections.
- Many error codes were changed in the latest release including the removal of dozens of lesser known (used) error codes. If your applications use the MySQL server error codes, you should check the documentation to ensure the error codes have not changed or been removed.

Changes like these are typical of major releases. In all cases, you should consider the release notes as you plan any upgrades. Places where changes like these are likely to cause problems are in your customizations and configurations. For example, if you have defined tuning procedures, stored procedures, DevOps, or other mechanisms that use or interact with options and variables, you should carefully examine the entry in the MySQL 8 documentation to ensure you can modify your tools accordingly.

Tip See <http://dev.mysql.com/doc/refman/8.0/en/added-deprecated-removed.html> for a complete list of features to be removed in MySQL 8.

Paradigm Shifting Features

Some of the features in MySQL 8 are truly groundbreaking for the MySQL ecosystem. Indeed, they will likely change how people use MySQL and expand the growing list of use cases for MySQL. These include the following paradigm shifting features.

- *Document Store*: A new structured storage mechanism that will change what you can store and indeed how you can interact with MySQL to store data for applications where data can change allowing your application to adapt without having to rebuild the storage layers
- *Group Replication*: A new, powerful self-healing high-availability option
- *InnoDB Cluster*: A new way to manage high availability built on group replication and incorporating the new shell, and the MySQL Router for an easy to setup and easy to maintain high-availability installation

Document Store

We have already learned some things about the document store when we discussed the JSON data type. The MySQL Document Store takes the JSON storage concept to a new level. While the JSON data type permits the introduction of unstructured data in our relational databases, the document store is a true NoSQL data store.

More specifically, the document store allows storing of unstructured data in the form of JSON documents natively in MySQL. That is, MySQL now supports SQL and NoSQL options. The NoSQL option uses the X technologies we discovered earlier including the X Protocol and X DevAPI. These allow you to write applications that interface directly with MySQL without using any SQL or relational structures. How cool is that?

I KNOW SQL, BUT WHAT IS NOSQL?

If you have worked with relational databases systems, you are no doubt very familiar with Structured Query Language (SQL) where we use special statements (commands) to interact with the data. In fact, most database systems have their own version of SQL that includes commands to manipulating the data (DML) as well as defining the objects to store data (DDL) and even administrative commands to manage the server.

That is, you get result sets and must use commands to search for the data then convert results into internal programming structures making the data seem like an auxiliary component rather than an integral part of the solution. NoSQL interfaces break this mold by allowing you to use APIs to work with the data. More specifically, you use programming interfaces rather than command-based interfaces.

Sadly, NoSQL can mean several things depending on your perspective including “non-SQL”, “not only SQL”, or “non-relational”. But they all refer to the fact that the mechanism you’re using is not using a command-based interface and most uses of the term indicate you’re using a programming interface. For MySQL 8, access to JSON documents can be either through SQL or NoSQL using the X Protocol and X DevAPI through the X Plugin.

The origins of the MySQL document store lie in several technologies that are leveraged together to form the document store. Specifically, Oracle has combined a key, value mechanism with a new data type, a new programming library, and a new access mechanism to create what is now the document store. Not only does this allow us to use MySQL with a NoSQL interface, it also allows us to build hybrid solutions that leverage the stability and structure of relational data while adding the flexibility of JSON documents.

We will learn more about the document store in Chapter 6.

Group Replication

If you have used MySQL replication, you are no doubt very familiar with how to leverage it when building high-availability solutions. Indeed, it is likely you have discovered a host of ways to improve availability in your applications with MySQL replication.

Moreover, it has become apparent that the more your high-availability needs, and your solution expands (grows in sophistication), the more you need to employ better ways to manage the loss of nodes, data integrity, and general maintenance of the clusters (groups of servers replicating data – sometimes called replicaset). In fact, most high-availability solutions have outgrown the base master and slaves topology evolving into tiers consisting of clusters of servers, some replicating a portion of the data for faster throughput and even for compartmental storage. All of these have led many to discover they need more from MySQL replication. Oracle has answered these needs and more with Group Replication.

Group Replication enables you to establish a set of servers to be used in a group that enables the mitigation of not only transactions among the servers, but also automatic failover, and fault tolerance. In addition, Group Replication can also be used with the MySQL Router to allow your applications to have a layer of isolation from the cluster. We will see a bit about the router when we examine the InnoDB Cluster.

One important distinction between Group Replication and standard replication is that all the servers in the group can participate in updating the data with conflicts resolved automatically. Yes, you no longer must carefully craft your application to send writes (updates) to a specific server! However, you can configure Group Replication to allow updates by only one server (called the primary) with the other servers acting as secondary servers or as a backup (for failover).

We will learn much more about Group Replication in [Chapter 8](#).

InnoDB Cluster

Another new and emerging feature is called InnoDB Cluster. It is designed to make high availability easier to setup, use, and maintain. InnoDB Cluster works with the X AdminAPI via the MySQL Shell and the AdminAPI, Group Replication, and the MySQL Router to take high availability and read scalability to a new level. That is, it combines new features in InnoDB for cloning data with Group Replication and the MySQL Shell and MySQL Router to provide a new way to setup and manage high availability.

Note The AdminAPI is a special API available via the MySQL Shell for configuring and interacting with InnoDB Cluster. Thus, the AdminAPI has features designed to make working with InnoDB Cluster easier.

In this use case, the cluster is setup with a single primary (think master in standard replication parlance), which is the target for all write (updates). Multiple secondary servers (slaves) maintain replicas of the data, which can be read from and thus enable reading data without burdening the primary thus enabling read out scalability (but all servers participate in consensus and coordination). The incorporation of Group Replication means the cluster is fault tolerant and group membership is managed automatically. The MySQL router caches the metadata of the InnoDB Cluster and performs high-availability routing to the MySQL Server instances making it easier to write applications to interact with the cluster.

You may be wondering what makes this different from a read-out scalability setup with standard replication. At a high level, it may seem that the solutions are solving the same use case. However, with InnoDB Cluster, you can create, deploy, and configure servers in your cluster from the MySQL shell providing a complete high-availability solution that can be managed easily. That is, you can use the InnoDB Cluster AdminAPI via the shell to create and administer an InnoDB Cluster programmatically using either JavaScript or Python.

We will learn more about InnoDB Cluster in [Chapter 10](#).

Summary

MySQL 8 has a lot of new features. In many ways, it represents a major leap forward in several areas including high availability and NoSQL. However, one of the less advertised but immensely important new features is the new MySQL Shell. As you will see in the upcoming chapters, the new shell is the glue that makes all the new features work together in a seamless manner.

For example, without the shell, using the MySQL Document Store would require using a third-party programming environment to interact with the X DevAPI. The shell makes working with the new API much easier since it provides an environment not only familiar (think the old client) but also more user-friendly.

Similarly, without the shell to realize the AdminAPI, working with InnoDB Cluster would be no better than the manual administration needed for MySQL Replication. While you can still manually configure any of the high-availability features in MySQL, now that we have the shell to make it all so much easier, it is not necessary to do so except in certain, specific cases.

However, to truly appreciate how the importance of and significant contributions that the shell makes to MySQL 8, we must see it in action with each of the new features. The rest of this book will present short tutorials on using each of the major features in MySQL 8 as well as examples of how to use the shell with the feature. These include the following:

- Using the shell with SQL databases
- Using the shell with Document Store
- Using the shell with Group Replication
- Using the shell with InnoDB Cluster

But first, we will see how to install the MySQL Shell in the next chapter.

CHAPTER 2

Installing the MySQL Shell

While the MySQL server still includes the old MySQL client (`mysql`), the new MySQL Shell should be considered the default client to interact with your MySQL servers. It has many advantages over the previous client that was bundled with the server; the most powerful being the ability to use Python or JavaScript directly from the shell. So, how do we get the new MySQL Shell?

In this chapter, we will discover how to download and install the MySQL Shell for three of the most popular platforms; Windows, macOS, and Linux. For Windows, we will use a special all-in-one installer that makes installing any MySQL product easy. For macOS, we will see how to download and install the shell using macOS-friendly installers. For Linux, we will see how to use Oracle's Advanced Packaging Tool (APT) repository to make adding MySQL products easier on Linux.

Let's begin by downloading the MySQL Shell and checking its prerequisites.

Preparing to Install the MySQL Shell

The MySQL Shell can be installed on any platform that Oracle supports for MySQL Server. On most platforms, the shell is contained in a separate installation. The one exception is the Windows platform where the shell is included in the MySQL Windows Installer. In this section, we will see a quick overview of downloading the shell for various platforms.

Prerequisites

If you do not have MySQL Server 8.0 installed, you may want to install it on one of your systems before working through this tutorial. While you can use the shell on older versions of MySQL, you will need the latest version of the server to use all the features.

Aside from having MySQL Server 8.0 available on a system (or your desktop or laptop for experimentation or development purposes), the shell requires you also have the following installed.

- Connector/Python 8.0.16 or later (<https://dev.mysql.com/downloads/connector/python/>)
- Connector/J 8.0.16 or later (<https://dev.mysql.com/downloads/connector/j/>)
- Python 3.7.1 or later (<https://www.python.org/downloads/>)
- (Windows only) C++ Redistributable for Visual Studio 2015 (available at the Microsoft Download Center).

Note For the examples in this book, you need only install Python and Connector/Python. Check your system to see if you have either of these installed. Note that on Windows, you can install Connector/Python together with the shell.

Let's now discover how to download the MySQL Shell from Oracle's web site.

How to Get the MySQL Shell

Like most MySQL products, the MySQL Shell is available in both the Community and Enterprise Editions. The Community Edition is open source and thus free to download and use. If you are an Oracle Enterprise customer, you can obtain the installers for MySQL via your preferred customer channel. However, you are also free to download the community edition if you wish. In this section, we will see how to download the Community Edition.

To download the Community Edition of any of the MySQL products, visit Oracle's MySQL download page at <https://dev.mysql.com/downloads/>. Here, you will see a list of all the products.

Click the product you want to download (<https://dev.mysql.com/downloads/shell/>), and the web site will present you with the files available for download for your operating system. That is, the web site will preselect the operating system on which your browser is running. For example, if you were running macOS and clicked on the MySQL Shell, you will see a list of files like those shown in Figure 2-1. You can click the *Select Operating System* drop-down control and select a different operating system if you'd like.

MySQL Shell 8.0.16

Select Operating System:

macOS

! Packages for Mojave (10.14) are compatible with High Sierra (10.13)

macOS 10.14 (x86, 64-bit), DMG Archive <small>(mysql-shell-8.0.16-macos10.14-x86-64bit.dmg)</small>	8.0.16	15.1M	Download
	MD5: 0d51c893e1ea2ec9da4ab2b4c52103f2 Signature		
macOS 10.14 (x86, 64-bit), Compressed TAR Archive <small>(mysql-shell-8.0.16-macos10.14-x86-64bit.tar.gz)</small>	8.0.16	14.9M	Download
	MD5: 4c5168cf840eb11c00c4f045f420d06e Signature		

Figure 2-1. Downloading the MySQL Shell (macOS)

If you are using Windows, you have a different option available. For Windows, Oracle provides a comprehensive guided installer known as MySQL Installer for Windows. If you visit the download web site for any of the MySQL products, you will see an entry for the MySQL Installer at the top of the list of files. You can still download the individual installer if you'd like, but the recommended mechanism is to use the MySQL Installer.

To download the MySQL Installer (<https://dev.mysql.com/downloads/installer/>), click the link shown. This will take you to another page as shown in Figure 2-2, which shows the files available to download. Choose the link that matches your system (32- or 64-bit).

MySQL Shell 8.0.16

Select Operating System:

Microsoft Windows

Recommended Download:



MySQL Installer
for Windows

All MySQL Products. For All Windows Platforms.
In One Package.

Starting with MySQL 5.6 the MySQL Installer package replaces the standalone MSI packages.

Windows (x86, 32 & 64-bit), MySQL Installer MSI [Go to Download Page >](#)

Other Downloads:

Windows (x86, 64-bit), ZIP Archive	8.0.16	29.6M	Download
(mysql-shell-8.0.16-windows-x86-64bit.zip)	MD5: ec192841a5f62ca84fb4980f1ead9fe0		Signature

Figure 2-2. Downloading the MySQL Installer for Windows

Be sure to choose the correct download option for your Windows system and download it now if you want to follow along as we see the installer in action.

There are two versions of the MySQL Installer for Windows; one that includes the products that are commonly installed, and a web version that will download only those products you want to install. If you only want to install a few products, the web version may be a better choice. On the other hand, if you intend to install the server and shell as well as the documentation and connectors, you should download the full version.

Let’s now look at how to install the shell on Windows, macOS, and Linux (Ubuntu). You can read the section that matches your choice of platform.

Installing on Windows with the MySQL Installer

Installing the MySQL Shell with the MySQL Installer for Windows, hence MySQL Installer or simply installer, follows a similar pattern of installing applications on Windows. The MySQL Installer includes all the MySQL products permitting you to

install those components that you want. While the MySQL Installer is the recommended installation option for Windows, you can download the shell and install it separately. But it is best to use the Windows installer.

In this demonstration, we will also install the MySQL Server and the minimal components for using the shell with the examples in this book. You may want to follow along and install MySQL on your system. If you already have MySQL Server installed, you can skip the portions that install the server.

When you launch the installer for the first time (subsequent launches display the add, modify, upgrade dialog), you will be presented with a welcome dialog that presents the license. You must accept the license in order to continue. Figure 2-3 shows the welcome panel for the installer.

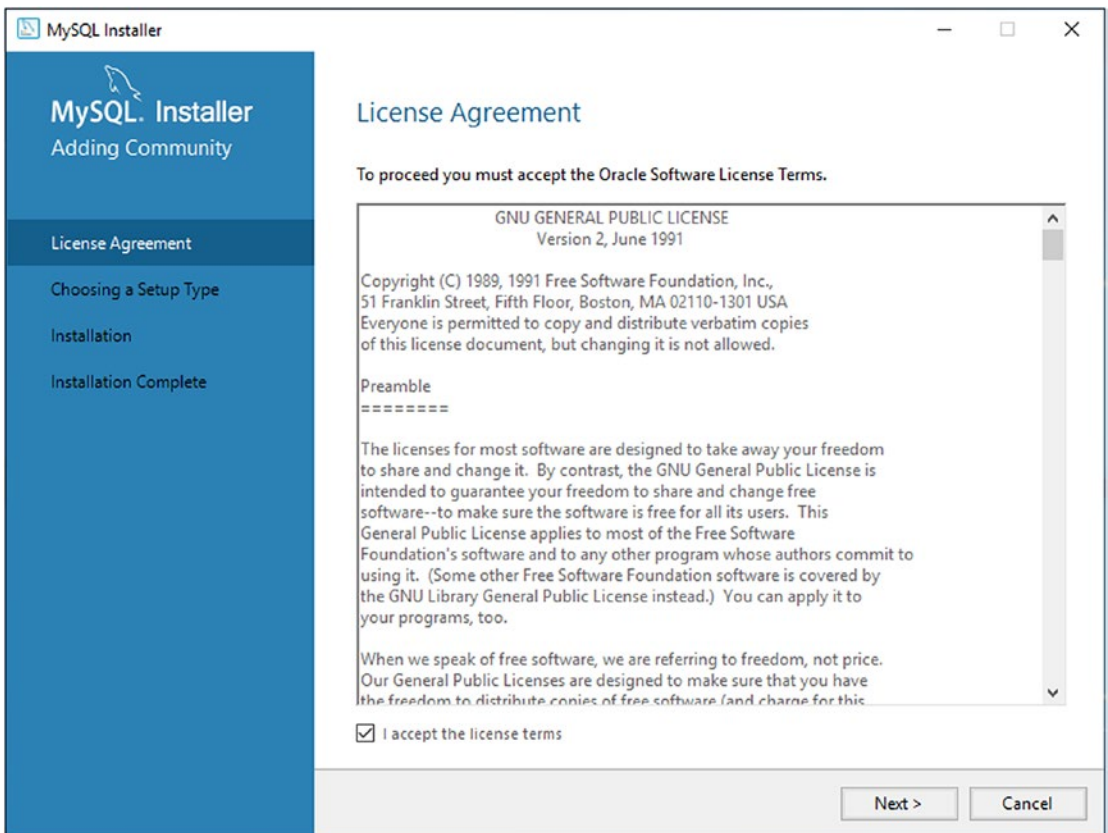


Figure 2-3. *Installer Welcome Panel – License Agreement*

To accept the license terms, tick the checkbox, and click *Next* to proceed. This will move you to the setup type dialog where you can choose one of several options including presets for installing a typical developer setup, installing only the server, installing only the client, or customizing the install. If you want to install multiple products or remove some, you should use the custom setup type. Figure 2-4 shows the Choosing a Setup Type dialog.

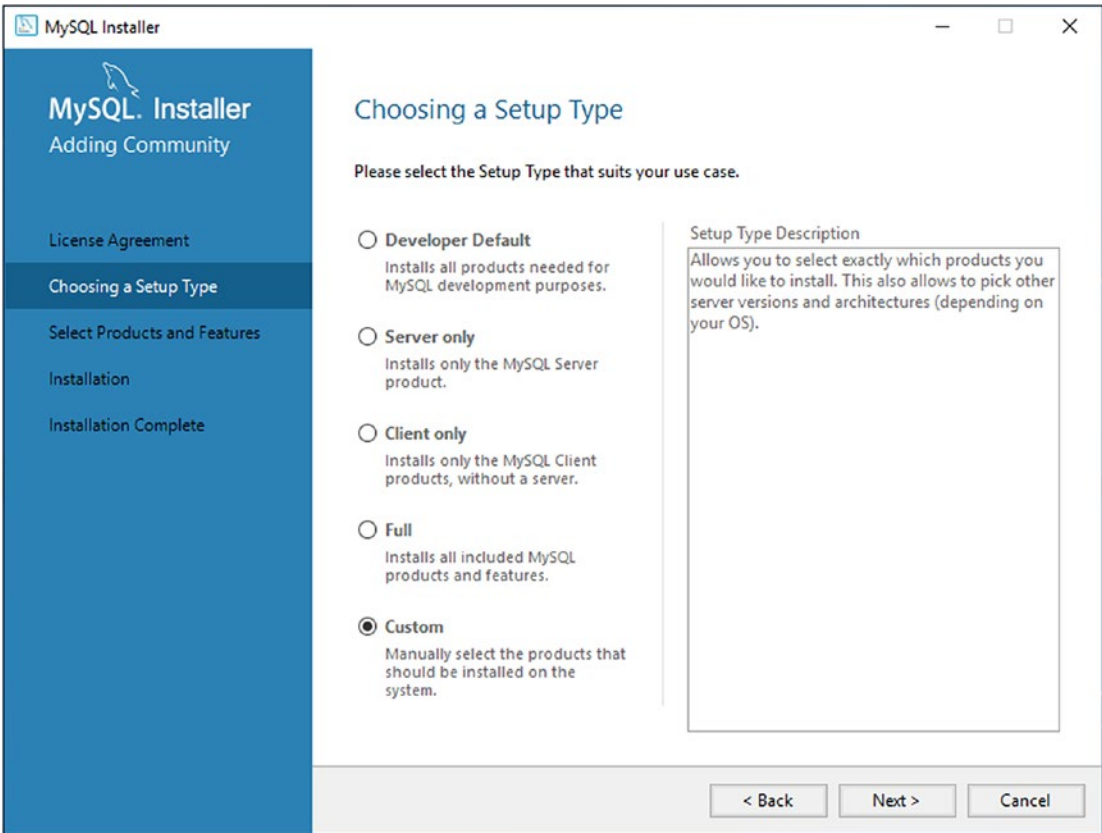


Figure 2-4. *Choosing a Setup Type*

Since we will be installing MySQL Server as well as MySQL Shell and related components, you should tick the *Custom* setup type and click *Next*. This will display the Select Products and Features dialog. Here, we will choose the components we want to install. For this tutorial, we must install the following components.

- MySQL Server
- MySQL Shell

- MySQL Router
- Connector/J
- Connector/Python
- MySQL Documentation
- Samples and Examples
- (Optional) MySQL Workbench

We use the custom option because the other selections will include additional components that we may not need (but it won't hurt to install them).

Figure 2-5 shows the Select Products and Features dialog. The dialog shows two columns where the column on the left contains all the products in the installer and the column on the right are those selected to be installed.

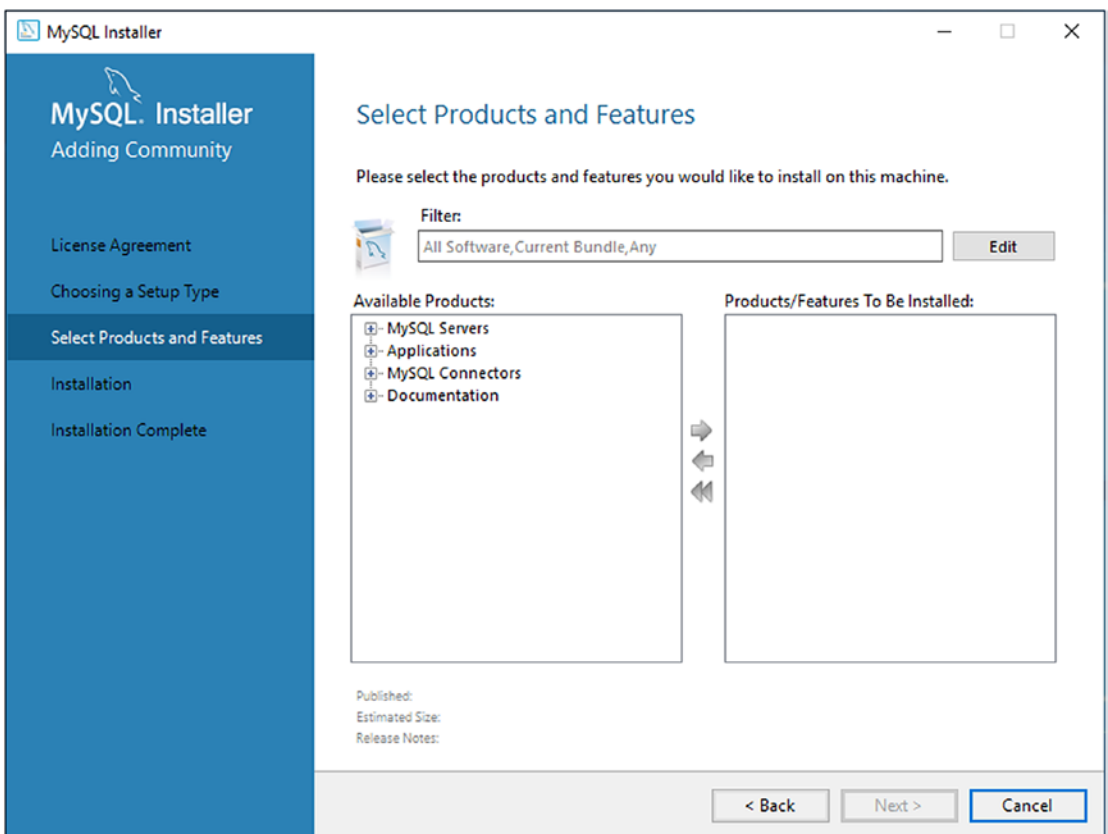


Figure 2-5. Select Products and Features (Default Selections)

To remove components from installation, select the product on the right and then click the left arrow icon. This removes the product. To add products, simply navigate the tree on the left to find the component (e.g., the MySQL Shell), select it, and then click the green arrow to add it to the column on the right. Figure 2-6 shows the correct selections.

Tip To remove all components, click the double left arrow. Similarly, to add all components, click the double right arrow.

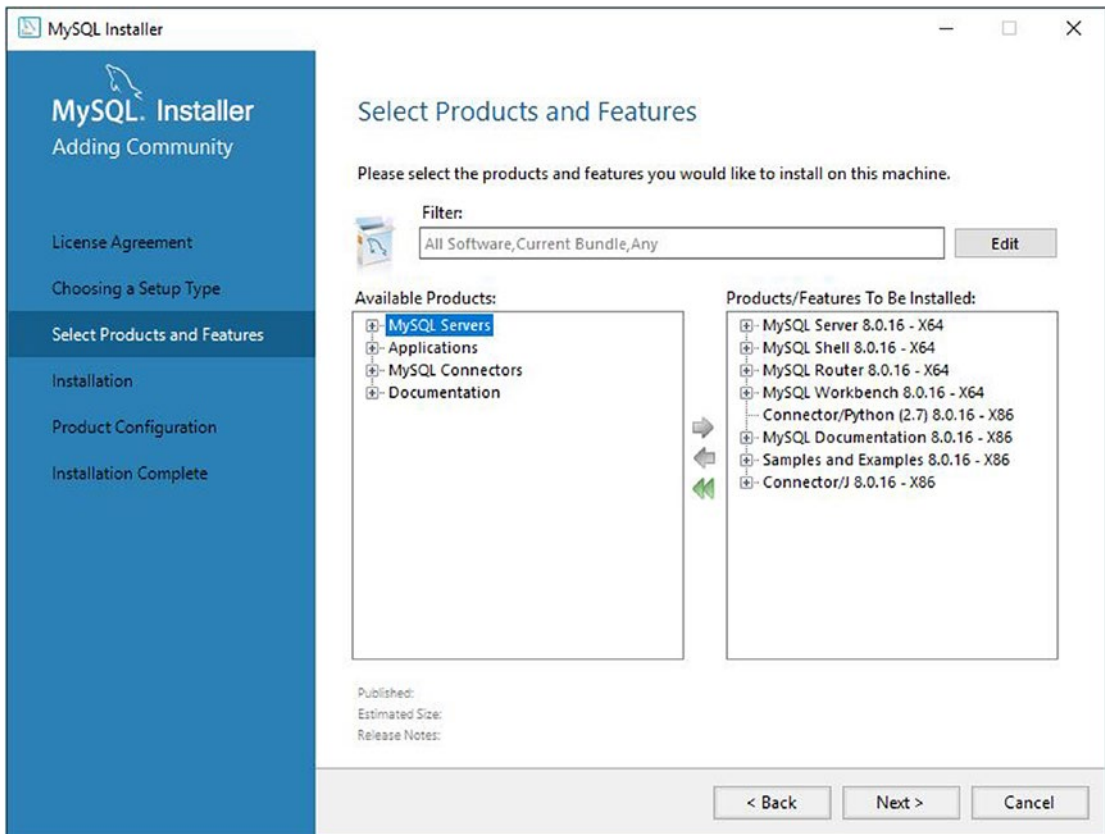


Figure 2-6. *Select Products and Features (Components Selected)*

When all the components listed above have been added to the column on the right, click the *Next* button to proceed. This presents the installation dialog panel, which lists the components to be installed along with whether the component must be downloaded or not. This is a common misconception with the installer. While it covers

all components, it may not include all the components when you download the installer. Fortunately, this means we can download only those components we want to install and nothing more. Figure 2-7 shows the installation dialog.

Take a moment to examine the list to ensure you've got all the components you want to install queued. If you need to make changes, you can click the *Back* button to return the previous dialog and select the missing components.

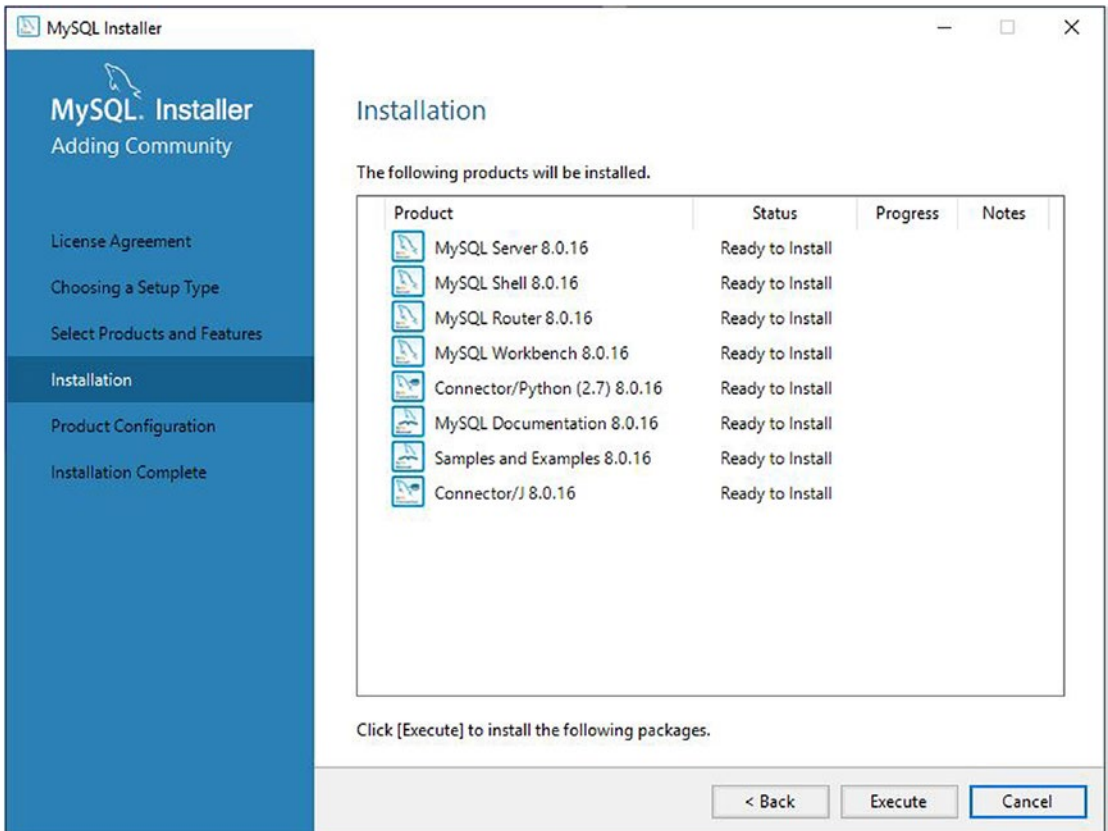


Figure 2-7. *Installation (Staging)*

When you are ready to proceed, click the *Execute* button. This will not display a new dialog, rather, you will see the status of each component change as it is downloaded and installed. Figure 2-8 shows a typical example of the dialog with component installations in progress.

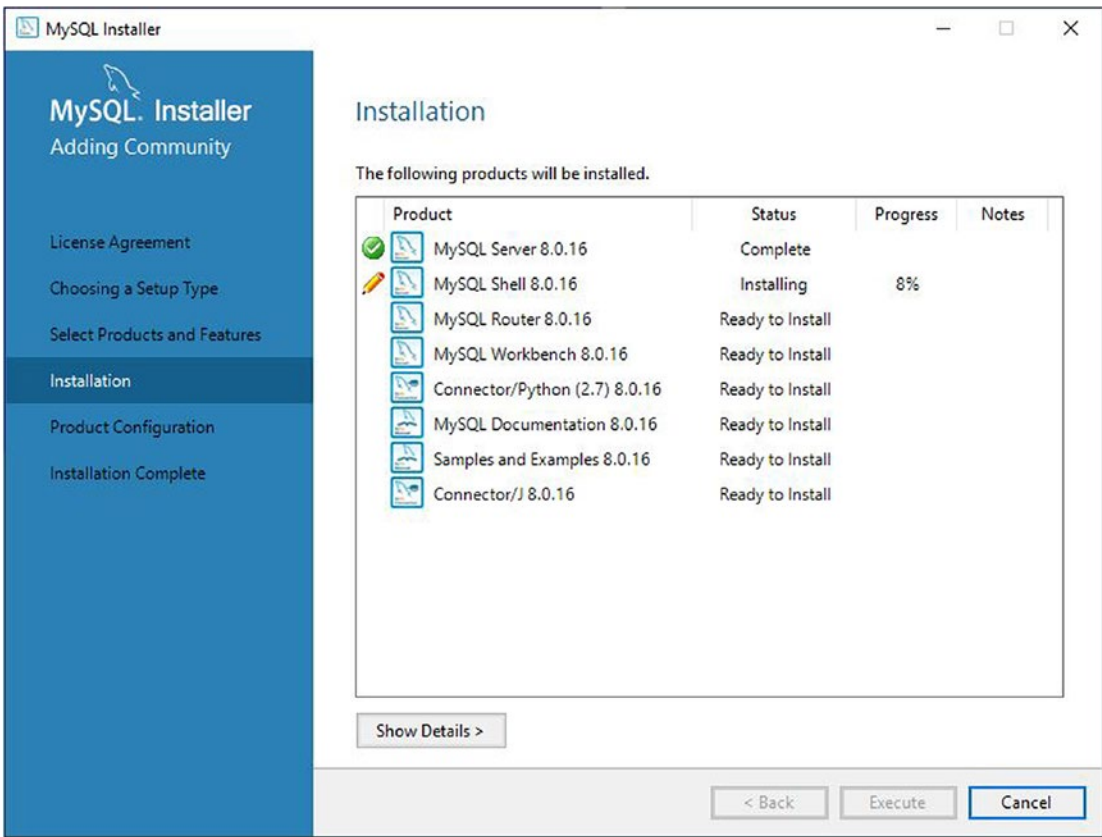


Figure 2-8. Installation (in Progress)

Once all components are installed, the installation dialog panel will show the status of all installations as complete and change the buttons at the bottom to show *Next* as shown in Figure 2-9. When ready, click *Next* once all products are installed.

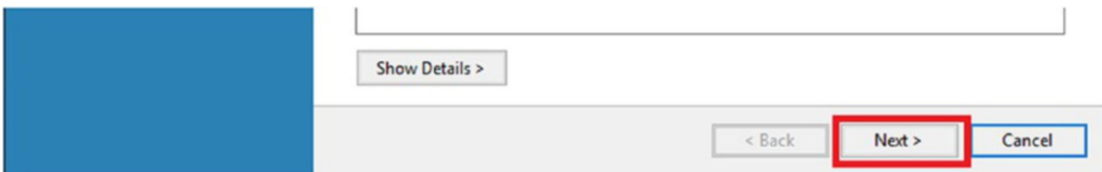


Figure 2-9. Installation (Installation Complete)

Once you click *Next*, the Product Configuration dialog is displayed as shown in Figure 2-10. At this point, the installer will return to this dialog after each component is configured.

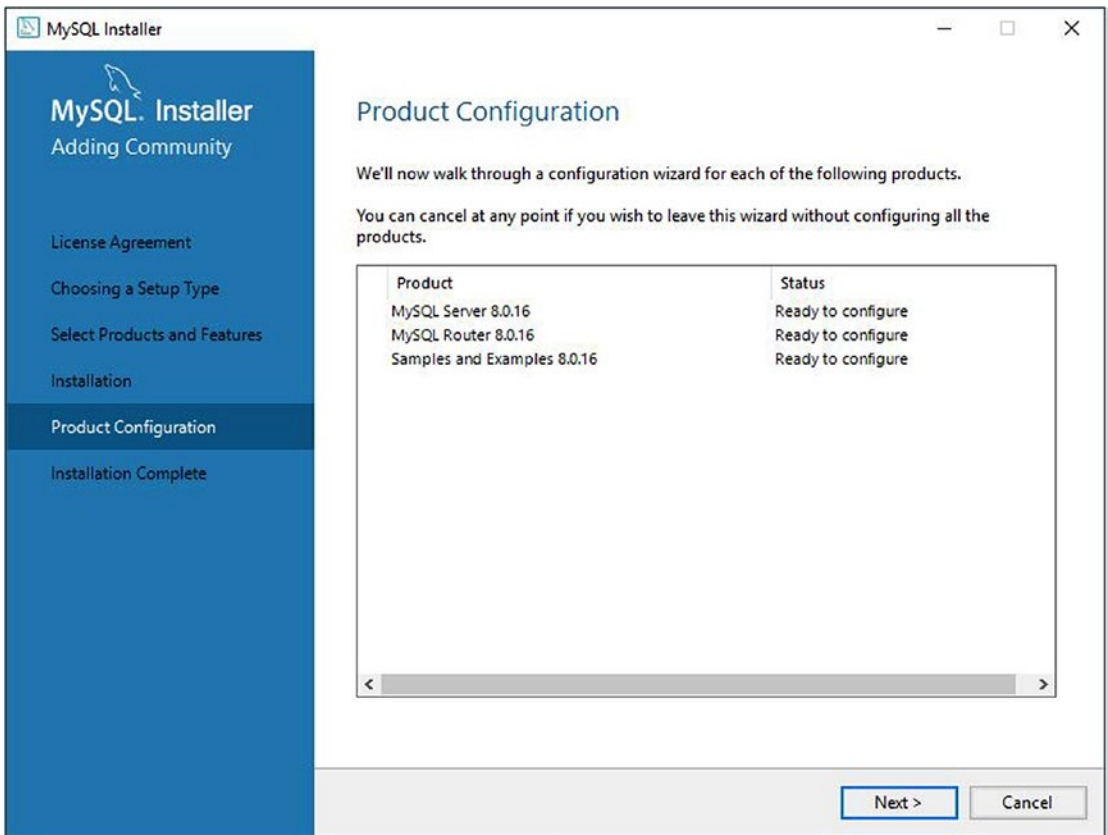


Figure 2-10. *Product Configuration*

Notice we have three components to configure; the server, router, and the samples and examples. The installer will do these in order. Simply click *Next* to get started configuring the server. Figure 2-11 shows the first step in configuring the server – setting up group replication.

The options include installing group replication normally (by choosing the standalone option) or installing group replication in a sandbox. The sandbox option may be helpful if you want to test group replication on your system and do not want to install additional servers. Except for this scenario, you should always choose the standalone option.

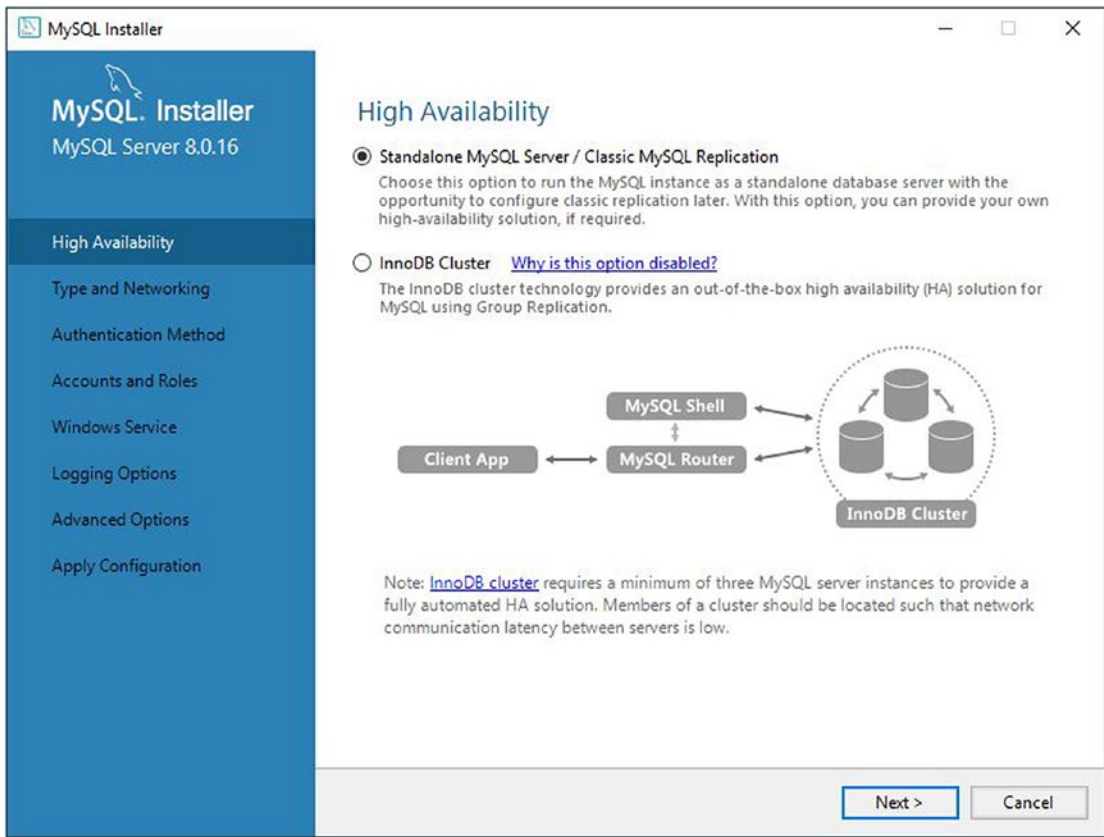


Figure 2-11. Group Replication

Tick the standalone option, then click *Next*. This will display the Type and Networking dialog as shown in Figure 2-12.

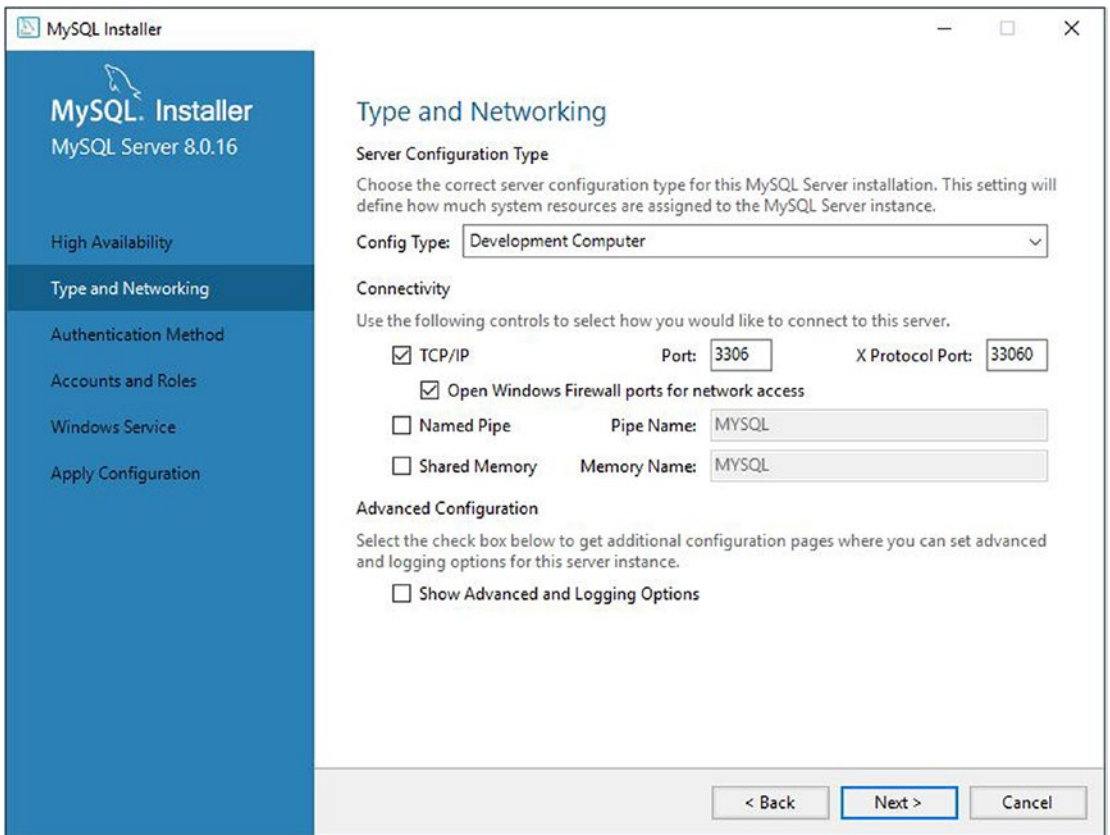


Figure 2-12. *Type and Networking*

On this dialog, you can use one of the preconfigured option or usage sets such as development (e.g., a development machine), server with applications, or dedicated server (only MySQL installed). For most installations on your laptop or desktop, you should choose the *Development Computer* option.

Choose the *Development Computer* option if you want to use the defaults for listening ports (3306 and 33060), named pipes, or shared memory. If the Windows firewall option is not ticked, you will need to tick that to ensure MySQL runs correctly and you can connect to it on Windows. When you have all the settings to your liking, click *Next*. This will display the Authentication Method dialog.

The Authentication Method dialog allows you to choose between the newest option, which includes strong password encryption (highly recommended) or use the legacy authentication method (not recommended for production).

Note If you choose the strong password encryption method and you want to use an older version of the MySQL client (not the shell), you may encounter errors connecting. You must use the newer client with strong password encryption.

Figure 2-13 shows the Authentication Method dialog. Notice there is a significant amount of text that describes each option and the default is to use Strong Password Encryption.

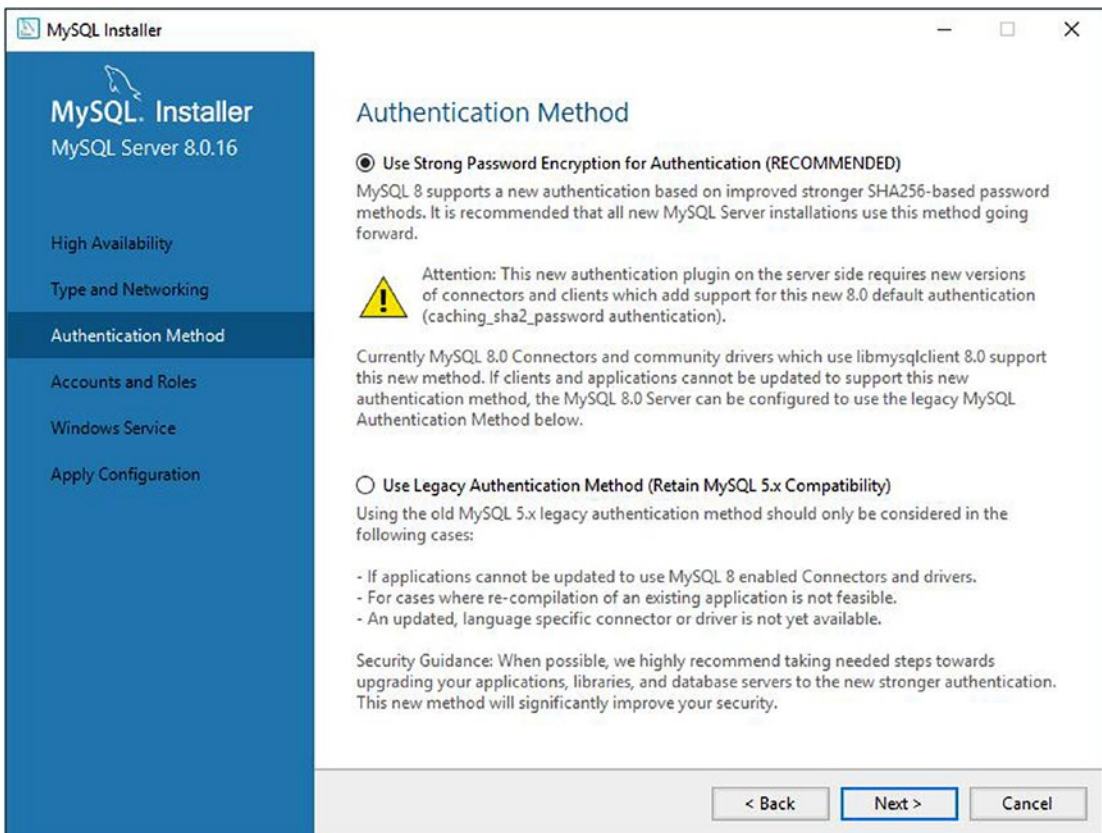


Figure 2-13. *Authentication Method*

You should tick the Strong Password Encryption option and click Next. This will display the Accounts and Roles dialog. This dialog is used to set the root password and optionally create any roles you want to use. This can be helpful if you plan to set up a server for use with applications and many users. However, for this demonstration, we need only select a root password.

Since we are using the strong password and encryption option, the password we enter in the dialog will be evaluated against best practices (strong passwords). However, until we configure the server to use the password validation plugin, we can use whatever password we want. It is always best to use strong passwords whether you have the plugin installed or not.

Tip See <https://dev.mysql.com/doc/refman/8.0/en/password-management.html> to set up password options and validation.

Figure 2-14 shows the Accounts and Roles dialog. Go ahead and type in the root user password of your choice, then type it again in the repeat dialog. Notice the password strength display. In this case, my 10-character password was only medium strength despite using special characters and no dictionary words. Harsh, eh?

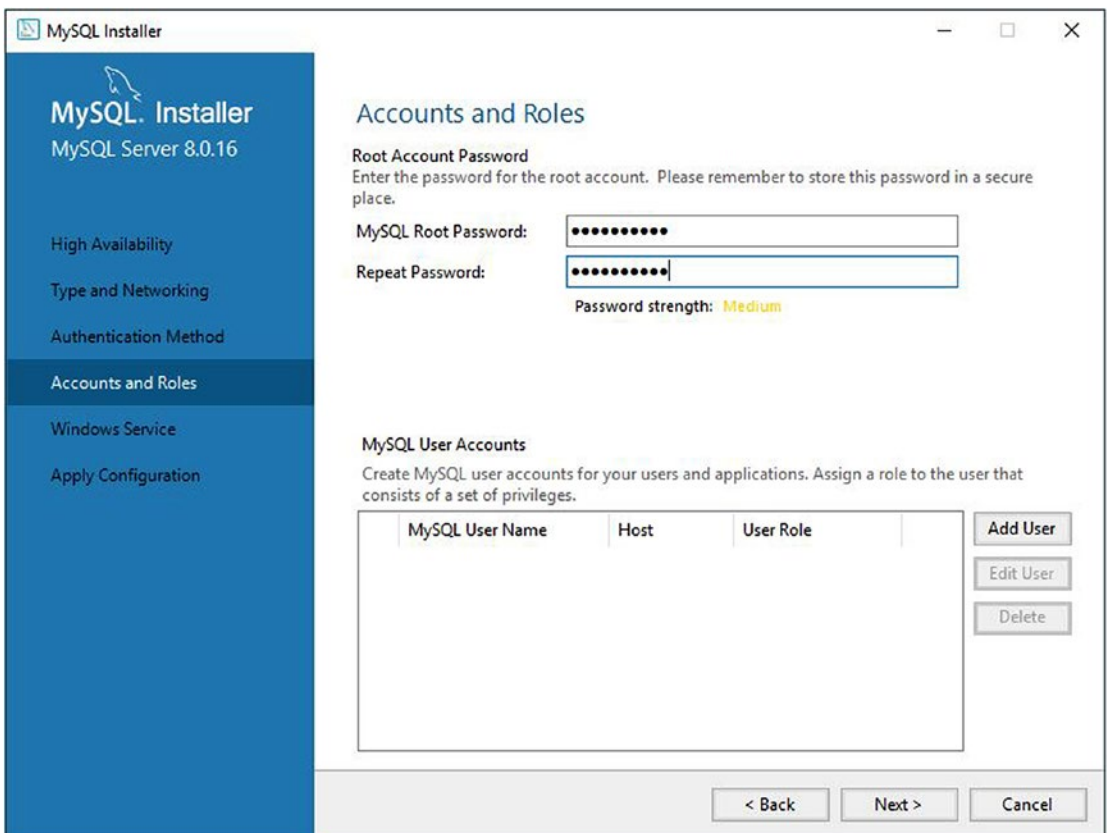


Figure 2-14. *Accounts and Roles*

Once you have the root password entered, click *Next*. This will display the Windows Service dialog where we can choose to start the server on Windows and run it as a service. Figure 2-15 shows the Windows Service dialog.

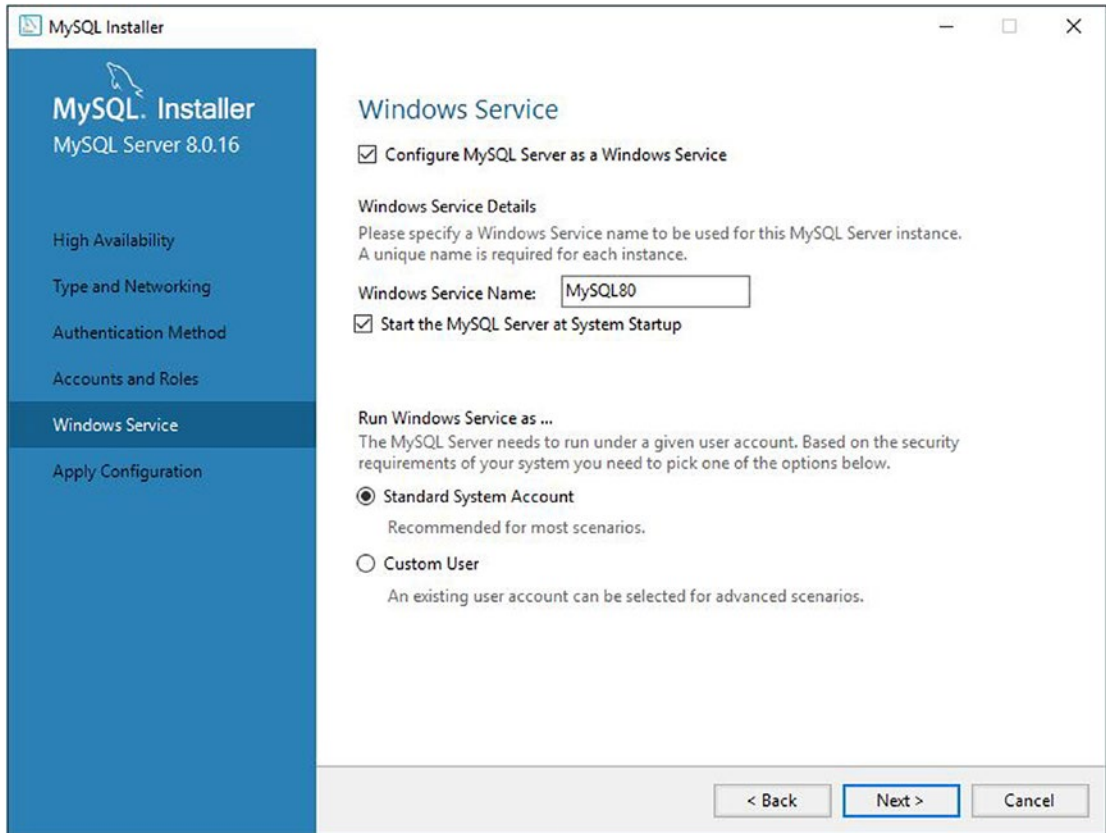


Figure 2-15. *Windows Service*

It is recommended to use the default selections here, which include configuring MySQL Server as a Windows service named MySQL80, starting the server at startup, and using the system user to launch the server. However, if you want to change these and you are familiar with how to start MySQL on Windows manually, you can make those changes.

When ready, click the *Next* button. This will display the Apply Configuration dialog. There is nothing to select on this dialog, so when ready, just click the *Execute* button. This will initiate the configuration process providing feedback in the form of green check marks that appear when each step is complete.

When all the steps are complete, the Finish dialog will display. Figure 2-16 shows the Finish dialog. Click *Finish* to begin the next phase of the configuration (MySQL Router). This will return to the Product Configuration dialog (omitted for brevity – see Figure 2-10).

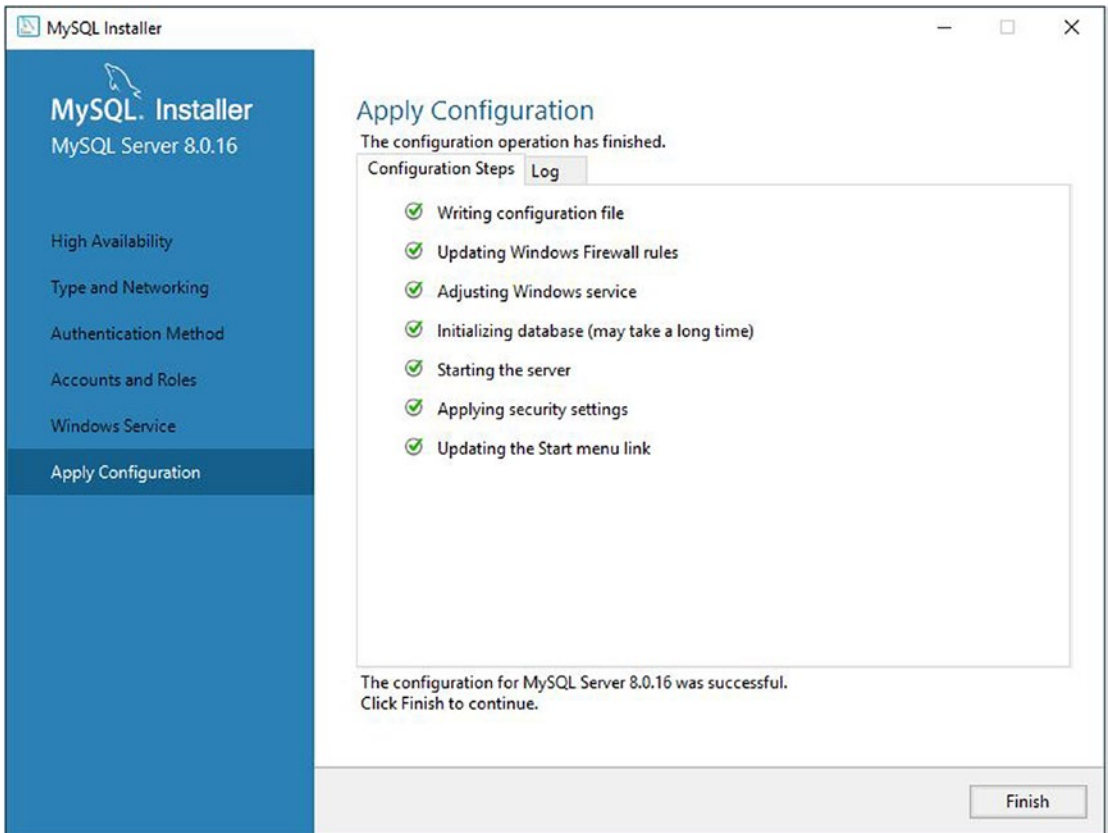


Figure 2-16. *Server Configuration Finished*

At the Product Configuration dialog, click *Next* to proceed to the MySQL Router configuration dialog, which permits you to make changes in how the router is installed and which ports it listens on. While we have not discussed the router, some of this may not be familiar. Fortunately, the default values are acceptable and include ports to use for both the classic client protocol and X Protocol clients.

Note We will discuss the MySQL Router including how to set up and use it in Chapters 8 and 10.

Figure 2-17 shows the MySQL Router configuration dialog. We can use the defaults, so we need only click *Finish* to proceed back to the Product Configuration dialog (omitted for brevity – see Figure 2-10).

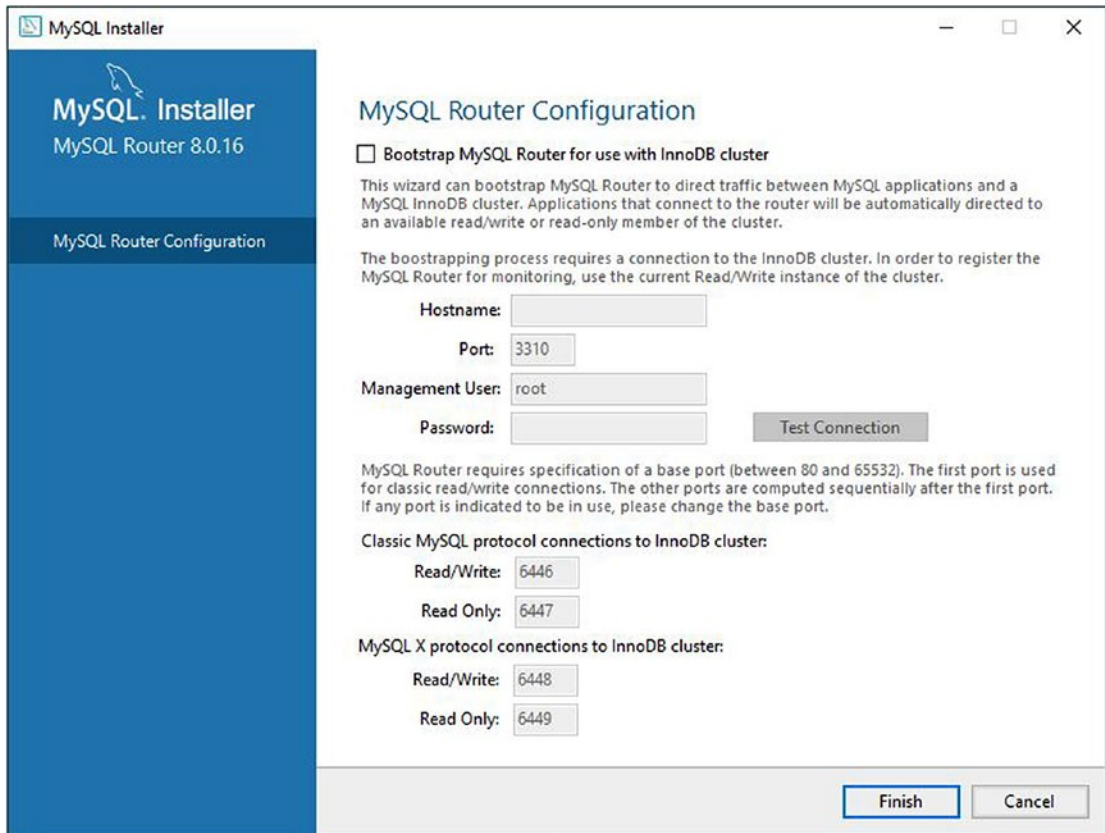


Figure 2-17. *MySQL Router*

At the Product Configuration dialog, click *Next* to proceed. The next component to configure is the Samples and Examples components. While it may appear that there isn't anything to do here (after all, they're only samples and examples for using MySQL), the dialog has a very interesting feature. The dialog allows you to check your server installation by connecting to the server. This not only tests your server but also ensures your installation is working correctly.

When the dialog appears, type in the root password you chose earlier and click the Check button. This will check your connection (and your password) to the server.

You may also notice the dialog shows both the standalone server option as well as a sandbox option. Had you installed the sandbox option, you could also check the MySQL server running in the sandbox. For this demonstration, we have only the standalone option checked.

Figure 2-18 shows the Connect to Server dialog.

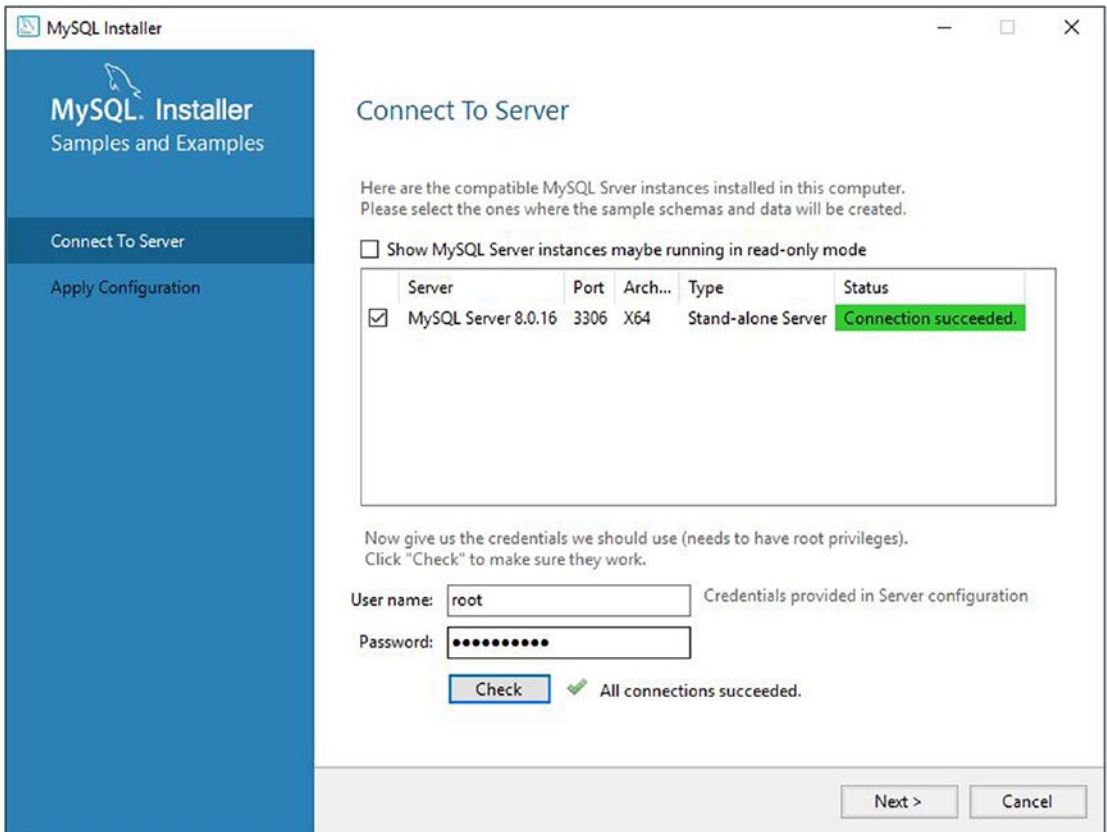


Figure 2-18. *Connect to Server*

Once you've checked your connection, click *Next* to proceed. The next dialog simply applies the configuration like we've seen previously, but in this case, the installer configures all the subcomponents for the samples and examples. Simply click *Execute* to begin. When all configurations are complete, you can click *Finish* to move to the next step. Figure 2-19 shows the completed configuration dialog for the samples and examples.

When you click *Finish*, you will return to the Product Configuration dialog, which will show all components configured (omitted for brevity – see Figure 2-10). Click *Next* to proceed.

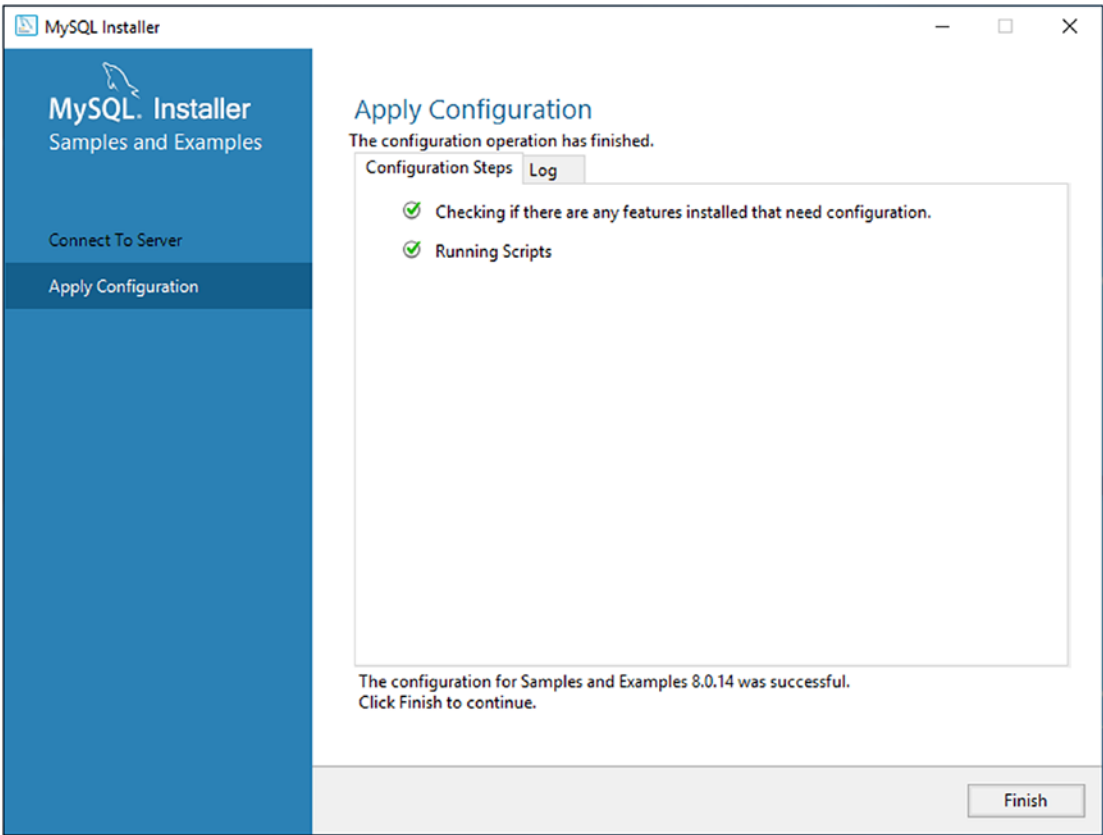


Figure 2-19. *Samples and Examples Finished*

Once all products are configured, you will see the Installation Complete dialog as shown in Figure 2-20. Click Finish to exit the installer.

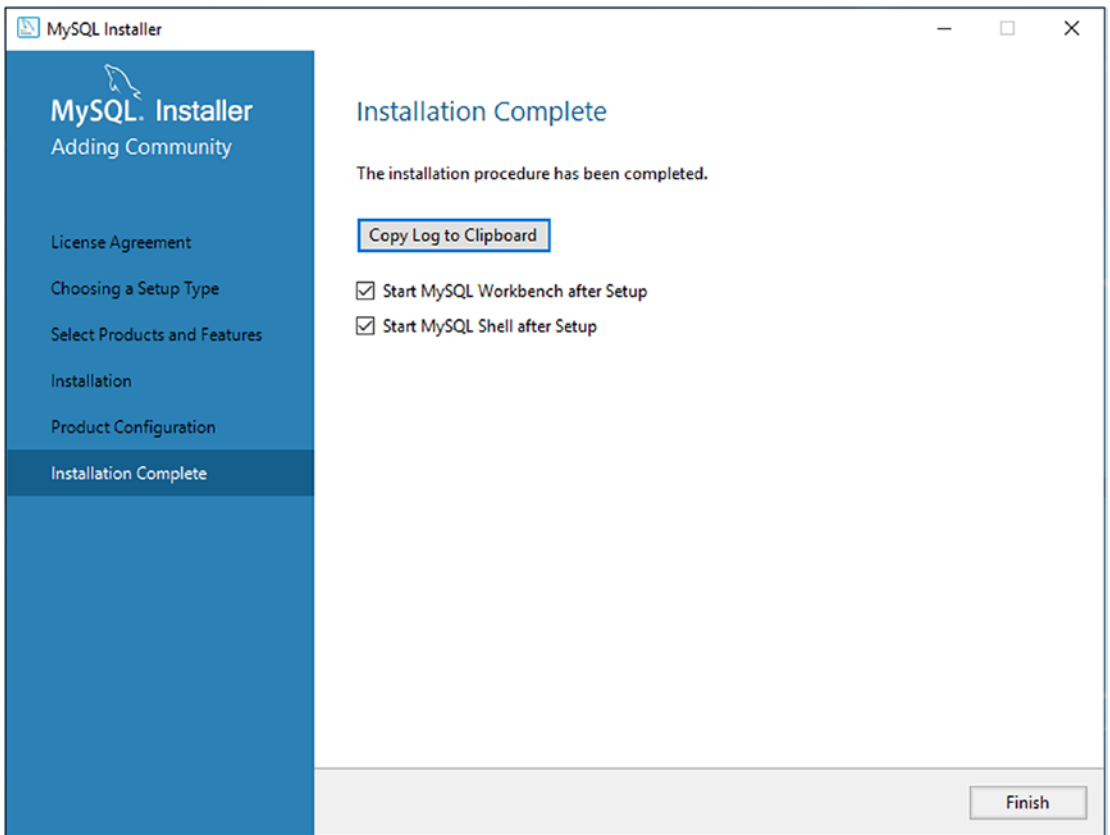


Figure 2-20. *Installation Complete*

If you have been following along installing MySQL on your own machine, congratulations! You now have MySQL Server and MySQL Shell as well as other components you need to complete the examples in this book.

Installing on macOS

Installing MySQL products on macOS is accomplished by downloading each product from the MySQL web site as a separate package. For example, if you want to install MySQL Server and MySQL Shell on macOS, you will need to download the installer for each. In this section, we will see a walkthrough of installing the server and shell on macOS.

We will install MySQL Server first, then install MySQL Shell. Recall we must download the installer for each from <https://dev.mysql.com/downloads/>. For example, after selecting the entry for the community server (<https://dev.mysql.com/downloads/mysql/>) and selecting the macOS entry in the operating system drop-down list, we will see the files available for installing the server on macOS. Figure 2-21 shows the files available for MySQL 8.0.16.

MySQL Community Server 8.0.16

Select Operating System: Looking for previous GA versions?

macOS

! Packages for Mojave (10.14) are compatible with High Sierra (10.13)

macOS 10.14 (x86, 64-bit), DMG Archive <small>(mysql-8.0.16-macos10.14-x86_64.dmg)</small>	8.0.16	256.5M	Download
macOS 10.14 (x86, 64-bit), Compressed TAR Archive <small>(mysql-8.0.16-macos10.14-x86_64.tar.gz)</small>	8.0.16	146.1M	Download
macOS 10.14 (x86, 64-bit), Compressed TAR Archive Test Suite <small>(mysql-test-8.0.16-macos10.14-x86_64.tar.gz)</small>	8.0.16	108.8M	Download
macOS 10.14 (x86, 64-bit), TAR <small>(mysql-8.0.16-macos10.14-x86_64.tar)</small>	8.0.16	270.0M	Download

MDS: a5cd42f14c21eb5800a10e9e9fa3894c | [Signature](#)

MDS: a081e9311a58a9f47768791b36e441bf | [Signature](#)

MDS: ceb30efbe401060d9bf25ddc8e9407c3 | [Signature](#)

MDS: 73818b656158d5381076a6fc5e9a4d2c | [Signature](#)

Figure 2-21. Downloading MySQL Server for macOS

Notice you will see several options including a mountable disk image with a guided installer (.dmg) as well as tape archives (.tar). On macOS, you should use the disk image option. Go ahead and download that now.

Similarly, you can go back to the community download page and click the entry for the MySQL Shell and, after selecting macOS in the operating system drop-down box, you will see the files available for downloading an installer for the shell. Figure 2-22 shows an example of the files available for MySQL Shell 8.0.16.

MySQL Shell 8.0.16

Select Operating System:

! Packages for Mojave (10.14) are compatible with High Sierra (10.13)

macOS 10.14 (x86, 64-bit), DMG Archive (mysql-shell-8.0.16-macos10.14-x86-64bit.dmg)	8.0.16	15.1M	Download
macOS 10.14 (x86, 64-bit), Compressed TAR Archive (mysql-shell-8.0.16-macos10.14-x86-64bit.tar.gz)	8.0.16	14.9M	Download

MDS: 0d51c893e fea2ec9da4ab2b4c52103f2 | [Signature](#)

MDS: 4c5168cf840eb11c00c4f045f420d06e | [Signature](#)

Figure 2-22. Downloading MySQL Shell for macOS

Here, we see there are once again a mountable disk image with a guided installer (.dmg) and a tape archive (.tar). You should download the disk image.

Now, let's see how to install the server.

Installing MySQL Server

To install MySQL Server, open the mountable disk image file (e.g., mysql-8.0.16-macos10.14-x86_64.dmg) and then open the installer (e.g., mysql-8.0.16-macos10.14-x86_64.pkg). This will start the installation.

The first dialog you will see is the welcome dialog, which presents a list of links for the documentation and a brief summary of the steps. For savvy macOS enthusiasts, the install progress will be very familiar. Figure 2-23 shows the welcome dialog.

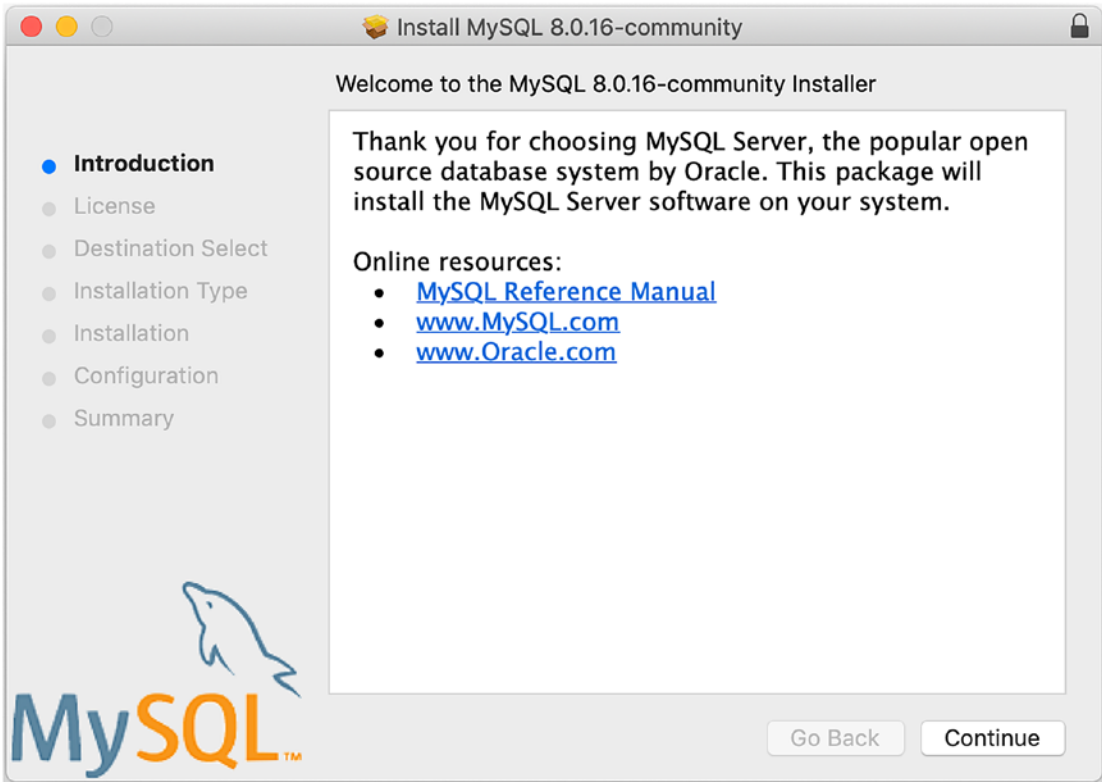


Figure 2-23. *Welcome Dialog*

Once you’ve read the welcome text and optionally explored the links, click *Continue* to proceed. The next dialog is the license dialog where you can choose to read the GNU General Public License (GNU GPL) license (or enterprise license if you chose to download the enterprise edition). You can also print or save the license to a file for later reading. Figure 2-24 shows the license dialog.

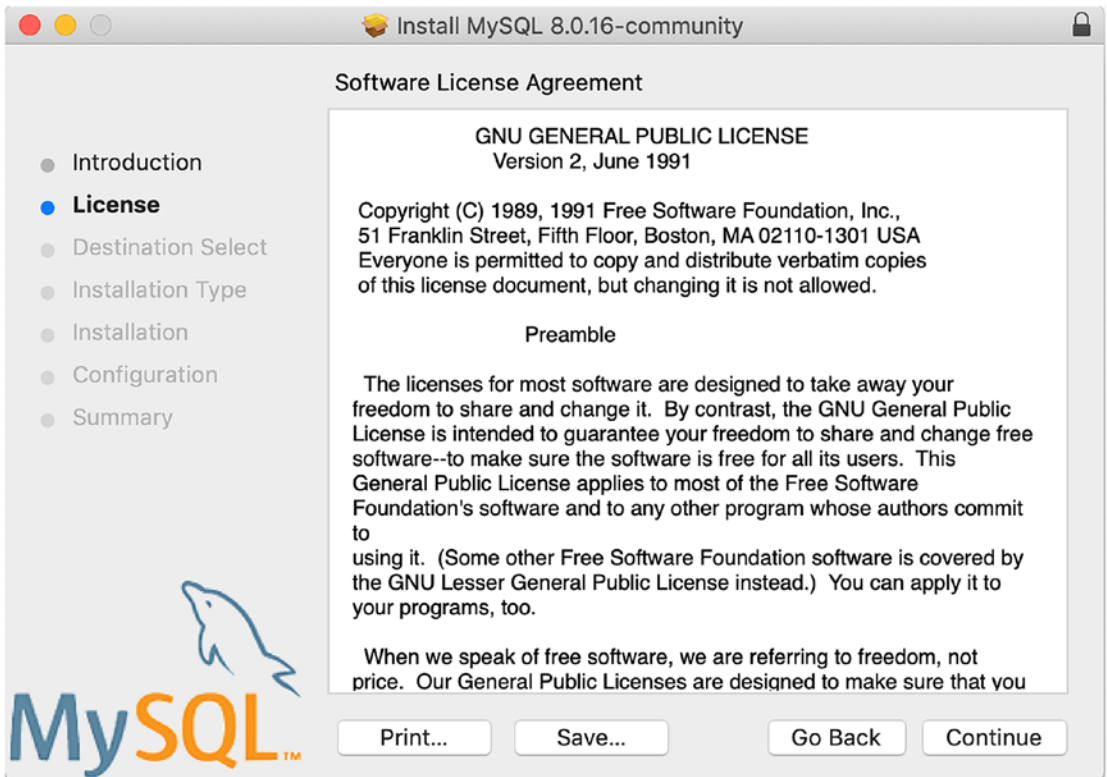


Figure 2-24. License Dialog

Click *Continue* to see the license acceptance dialog. Figure 2-25 shows the license agreement dialog. Once again, you can read the license, but you must accept the license to continue. To accept the license, click *Accept*.

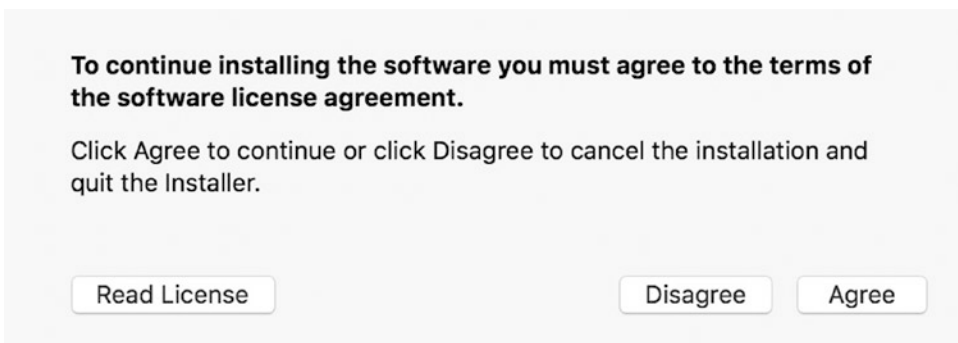


Figure 2-25. Accept License Dialog

Once you accept the license, you will move to the installation type dialog. Note that the destination select dialog is skipped. Figure 2-26 shows the installation type dialog.

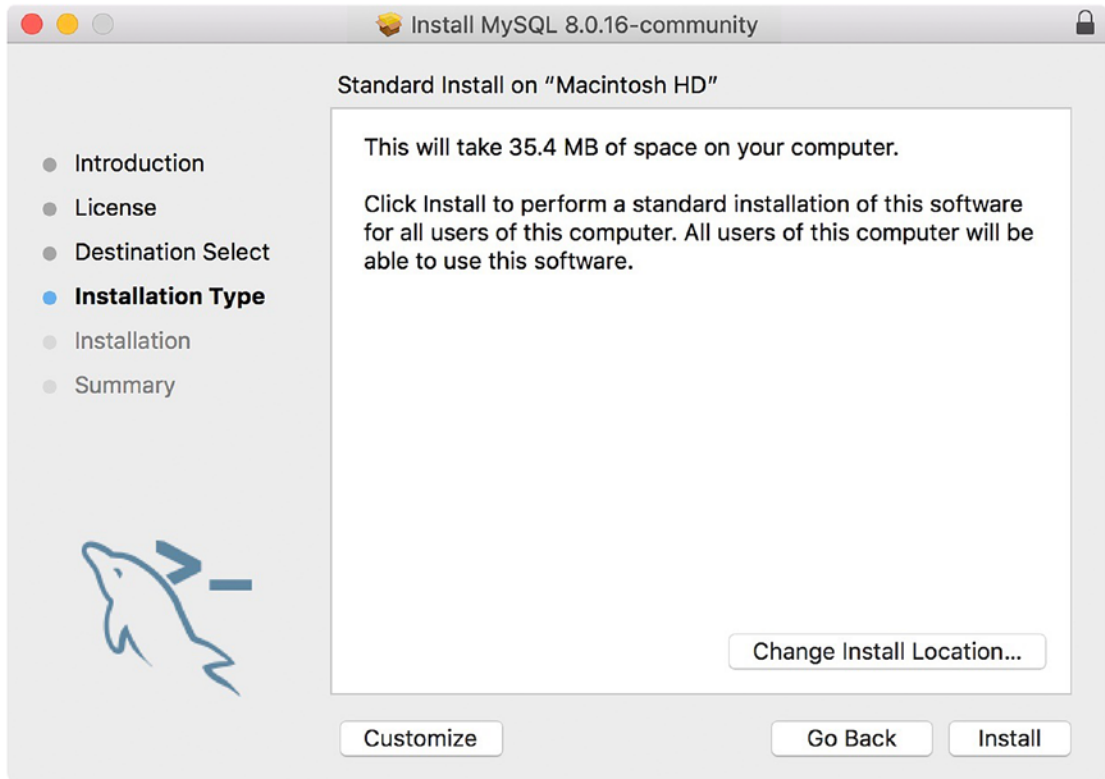


Figure 2-26. *Installation Type Dialog*

On this dialog, you can choose the installation destination. For most systems, you can accept the default. If you need to change the destination, you can click the *Customize* button. To proceed with the installation destination, click *Install*.

The next dialog is a progress dialog that shows the progress of the files being installed. You won't see much here, so let's look at the next dialog, which begins the configuration process. Once that dialog is dismissed, you will then start the configuration stage. Figure 2-27 shows the Configure MySQL Server dialog.



Figure 2-27. *Configure MySQL Server (Password) Dialog*

Here, we must choose to use either strong password encryption (highly recommended) or use the legacy authentication method (not recommended for production). Strong encryption is selected by default, so we simply click *Next* to move to the next configuration item.

Note If you choose the strong password encryption method and you want to use an older version of the MySQL client (not the shell), you may encounter errors connecting. You must use the newer client with strong password encryption.

The next configuration item is where we choose the password for the root user account as shown in Figure 2-28. Choose a password that you will remember and is sufficiently complex that no one will guess easily. You are encouraged to use a password with at least eight characters that are a mix of letters, numbers, and other characters.

We can also choose to start the MySQL server after installation is complete. It is recommended to start the server after installation, so you can check that it works correctly.

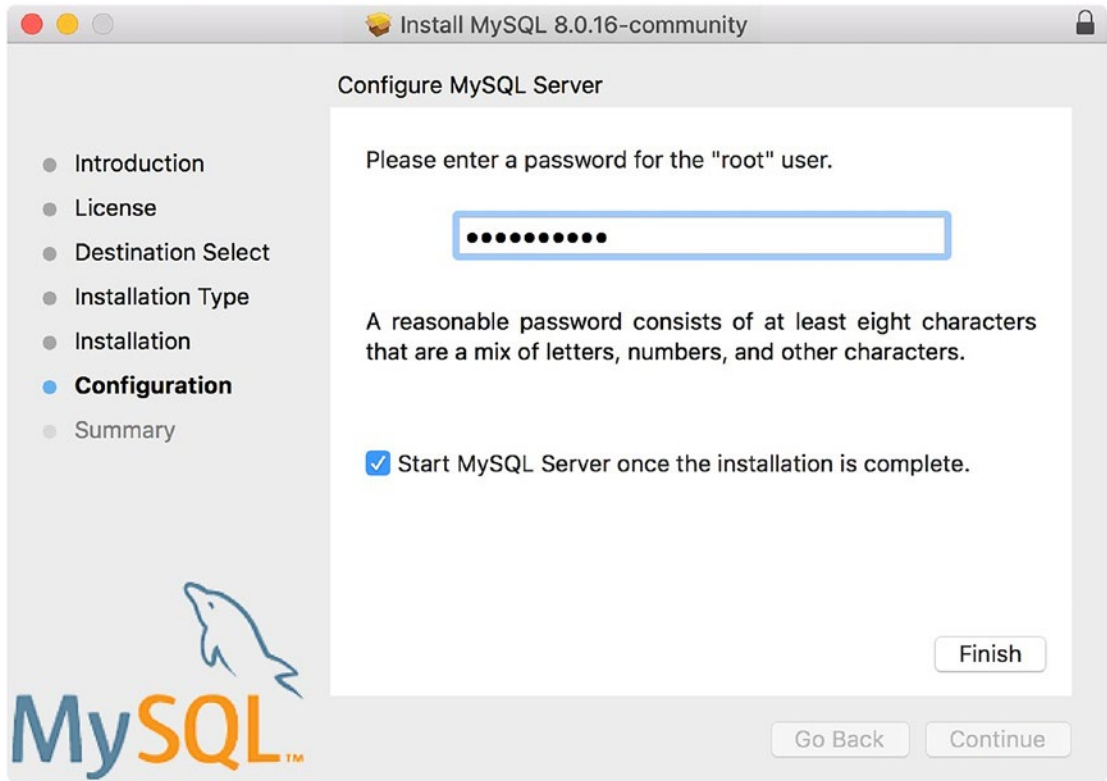


Figure 2-28. *Configure MySQL Server (Root User Password/Start Server) Dialog*

Once you've entered the root user password and decided if you want to start the server, click Finish to complete the configuration and move to the summary dialog. Figure 2-29 shows the Summary dialog.

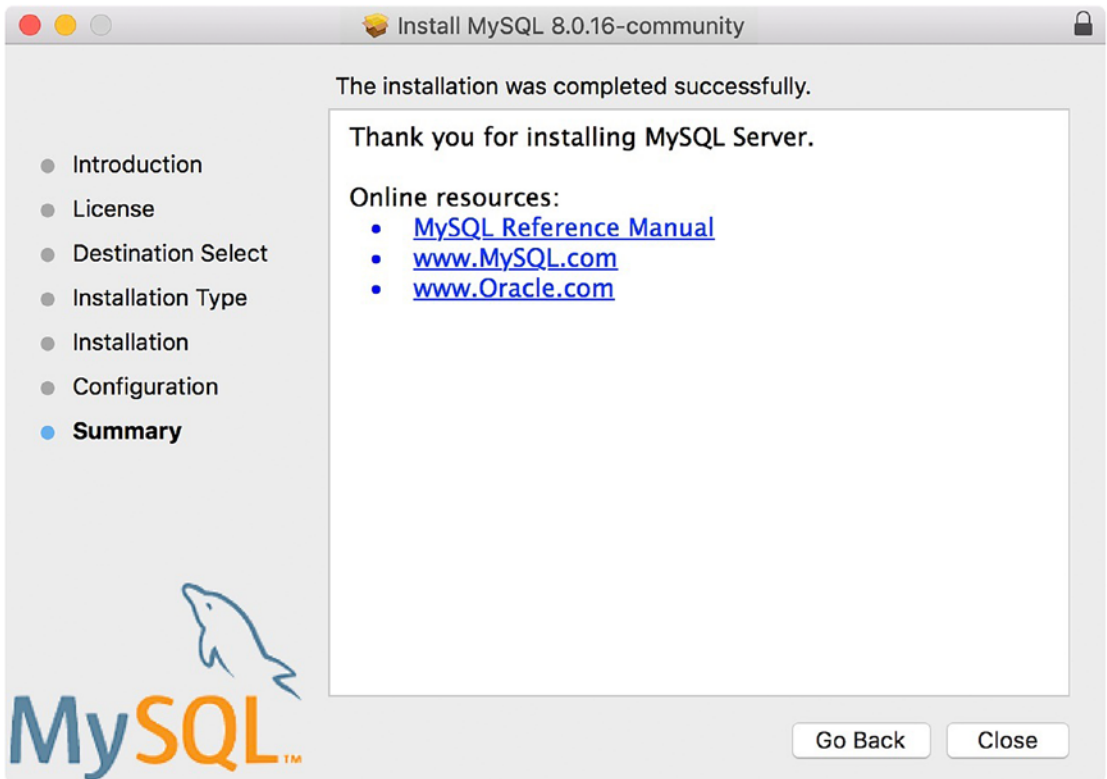


Figure 2-29. Summary Dialog

To complete the installation, click *Close*. You may want to close the mountable disk image (it will be closed on shutdown if you don't close it). You can now connect to the server if you want, but you will have to use the old client. Rather than doing that, let's install the shell.

Installing the MySQL Shell

Installing the MySQL Shell is very similar to installing the server. The exception is there isn't a configuration step.

To install MySQL Shell, open the mountable disk image file (e.g., `mysql-shell-8.0.16-macos10.14-x86-64bit.dmg`) and then open the installer (e.g., `mysql-shell-8.0.16-macos10.14-x86-64bit.pkg`). This will start the installation.

The first dialog you will see is the welcome dialog, which presents a list of links for the documentation and a brief summary of the steps. For savvy macOS enthusiasts, the install progress will be very familiar. Figure 2-30 shows the welcome dialog.

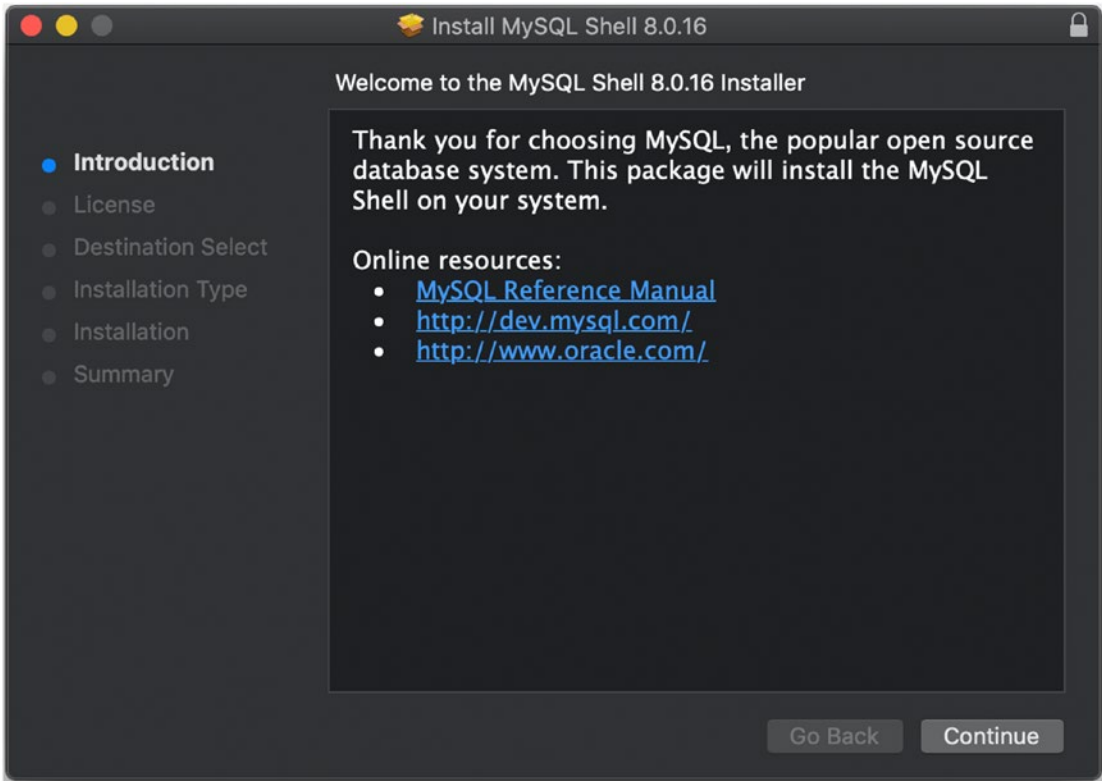


Figure 2-30. Welcome Dialog

Once you’ve read the welcome text and optionally explored the links, click *Continue* to proceed. The next dialog is the license dialog where you can choose to read the GPL license (or enterprise license if you chose to download the enterprise edition). You can also print or save the license to a file for later reading. Figure 2-31 shows the license dialog.

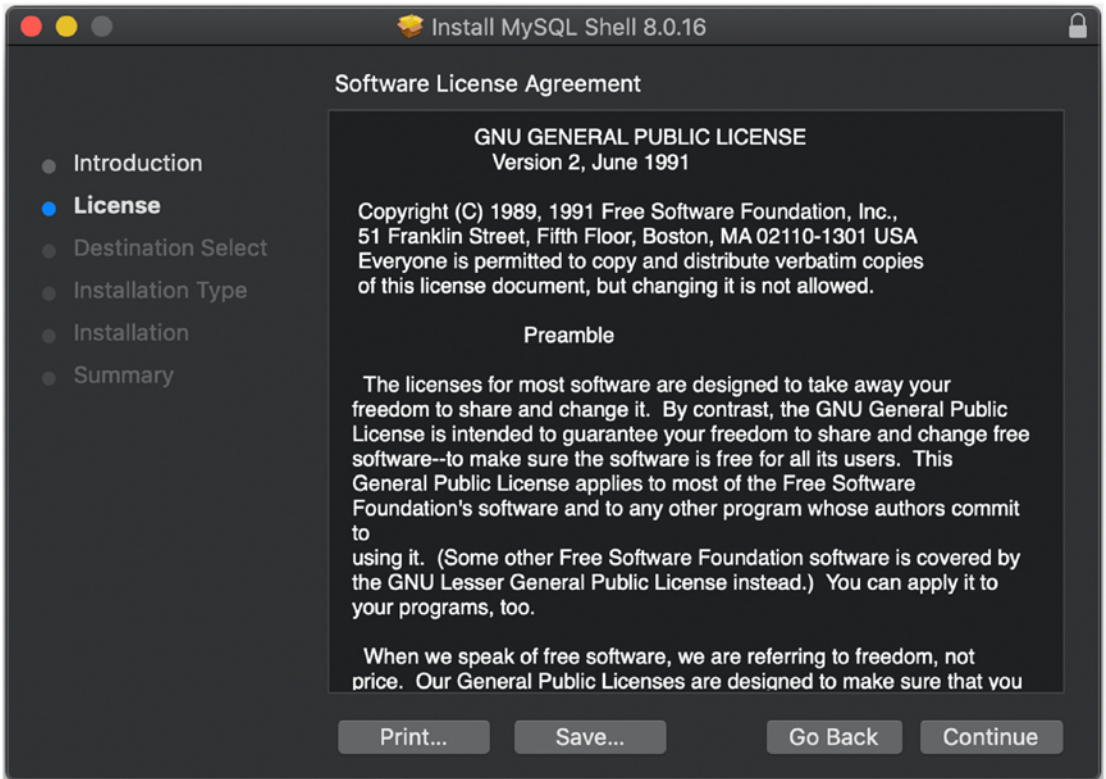


Figure 2-31. License Dialog

Click *Continue* to see the license acceptance dialog. Figure 2-32 shows the license agreement dialog. Once again, you can read the license, but you must accept the license to continue. To accept the license, click *Accept*.

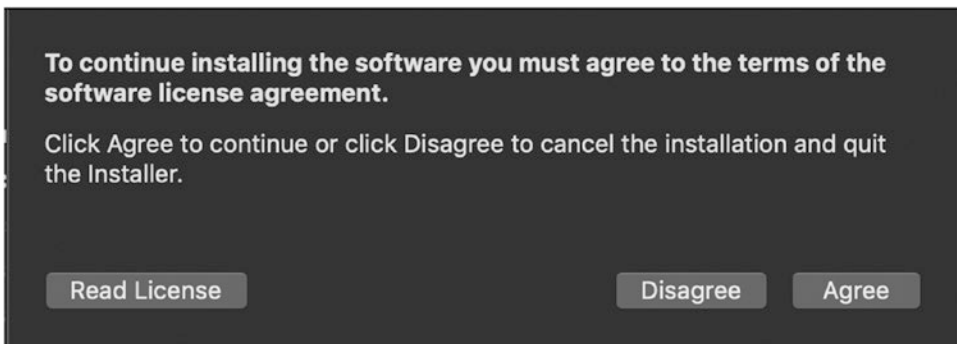


Figure 2-32. Accept License Dialog

Once you accept the license, you will move to the installation type dialog. Note that the destination select dialog is skipped. Figure 2-33 shows the installation type dialog.

On this dialog, you can choose the installation destination. For most systems, you can accept the default. If you need to change the destination, you can click the *Customize* button. To proceed with the installation destination, click *Install*.



Figure 2-33. *Installation Type Dialog*

The next dialog is a progress dialog that shows the progress of the files being installed. You won't see much here, so let's look at the next dialog, which presents the summary dialog. Figure 2-34 shows the Summary dialog.

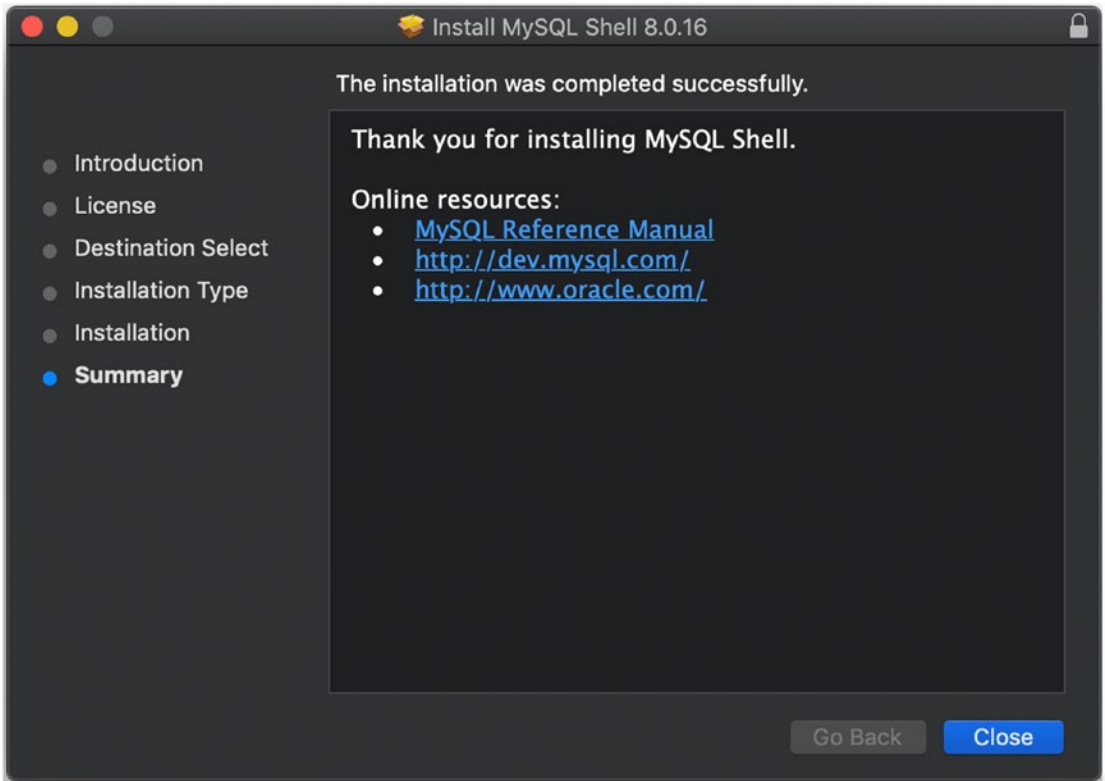


Figure 2-34. Summary Dialog

To complete the installation, click *Close*. You may want to close the mountable disk image (it will be closed on shutdown if you don't close it). You can now connect to the server using the MySQL Shell as shown in Figure 2-35. Here, I opened a terminal and entered the command *mysqlsh* to start the shell.

```

MacBook-Pro-2:~ cbell$ mysqlsh
MySQL Shell 8.0.16

Copyright (c) 2016, 2019, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

Type '\help' or '\?' for help; '\quit' to exit.

MySQL JS > \sql
Switching to SQL mode... Commands end with ;

MySQL SQL > \connect root@localhost:3306
Creating a session to 'root@localhost:3306'
Please provide the password for 'root@localhost:3306':
[Save password for 'root@localhost:3306'? [Y]es/[N]o/[M]e[v]er (default No):
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 11
Server version: 8.0.16 MySQL Community Server - GPL
No default schema selected; type \use <schema> to set one.

MySQL localhost:3306 ssl SQL >

```

Figure 2-35. Using the MySQL Shell (macOS)

If you have been following along installing MySQL on your own machine, congratulations! You now have MySQL Server and MySQL Shell as well as other components you need to complete the examples in this book.

Installing on Linux (Ubuntu) with the APT Repository

Installing the MySQL Server and Shell on other platforms is best done by using the platform-specific repository. That is, you can go to the MySQL download site and download the platform-specific installers and install MySQL products following the common methods for your Linux distribution.

In fact, the MySQL Server distribution packages are not a single download like other platforms. This is because the server packages are built in a modular manner. That is, you can install the server in parts including the clients, common libraries, server core, and more.

However, Oracle has built an easier way to install for the more popular Linux distributions, specifically the Ubuntu and Debian distributions. This is made possible through the APT Repository, which establishes the packages and references for your

platform. For example, once you've installed the repository on Ubuntu, you can use apt to install whichever server product you want, which will automatically download the appropriate installer.

But the APT Repository is more than that. Once installed, you can keep up with the latest versions of MySQL Server as they are released. Overall, it is much easier than downloading the installers manually every time you want to install a new version.

In this section, we see a demonstration of installing MySQL Server and MySQL Shell using the MySQL APT Repository on Ubuntu. Let's begin by downloading and installing the repository.

Downloading the APT Repository

To download the APT Repository, navigate to the MySQL Community downloads page (<https://dev.mysql.com/downloads/>) and click the APT Repository menu item at the top of the page as shown in Figure 2-36.

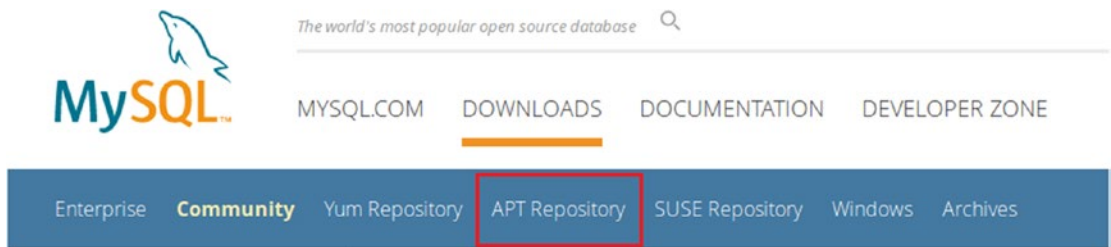


Figure 2-36. Select APT Repository

You will then be presented with the files available for the APT repository. For the MySQL 8.0.16 release, there is only one option as shown in Figure 2-37. To download the file, click *Download*.

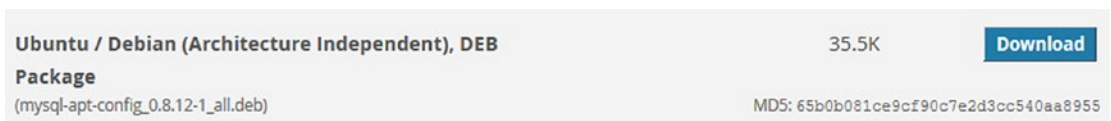


Figure 2-37. Downloading the APT Repository

You may be asked to log in or sign up for a free MySQL account. This is optional, and you can bypass the step by clicking the No thanks, just start my download link below the buttons as shown in Figure 2-38.

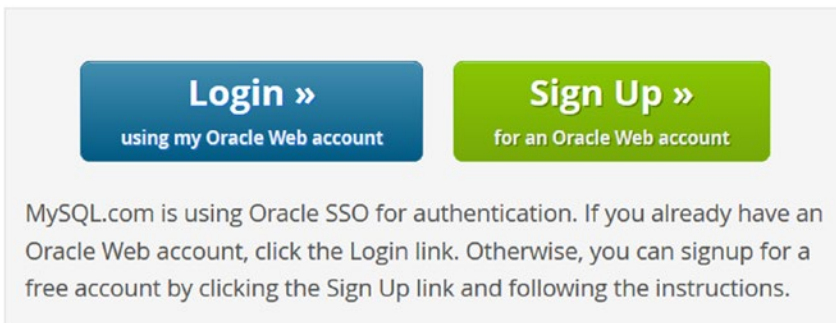
Begin Your Download

mysql-apt-config_0.8.13-1_all.deb

Login Now or Sign Up for a free account.

An Oracle Web Account provides you with the following advantages:

- Fast access to MySQL software downloads
- Download technical White Papers and Presentations
- Post messages in the MySQL Discussion Forums
- Report and track bugs in the MySQL bug system



The screenshot shows two buttons: a blue 'Login »' button with the text 'using my Oracle Web account' below it, and a green 'Sign Up »' button with the text 'for an Oracle Web account' below it. Below the buttons is a paragraph of text: 'MySQL.com is using Oracle SSO for authentication. If you already have an Oracle Web account, click the Login link. Otherwise, you can sign up for a free account by clicking the Sign Up link and following the instructions.'

No thanks, just start my download.

Figure 2-38. *Skipping the Login for Downloading the APT Repository*

Depending on which browser you used, you will next be asked to open or save the file. You should save the file in your Downloads folder or someplace where you can find it as shown in Figure 2-39.

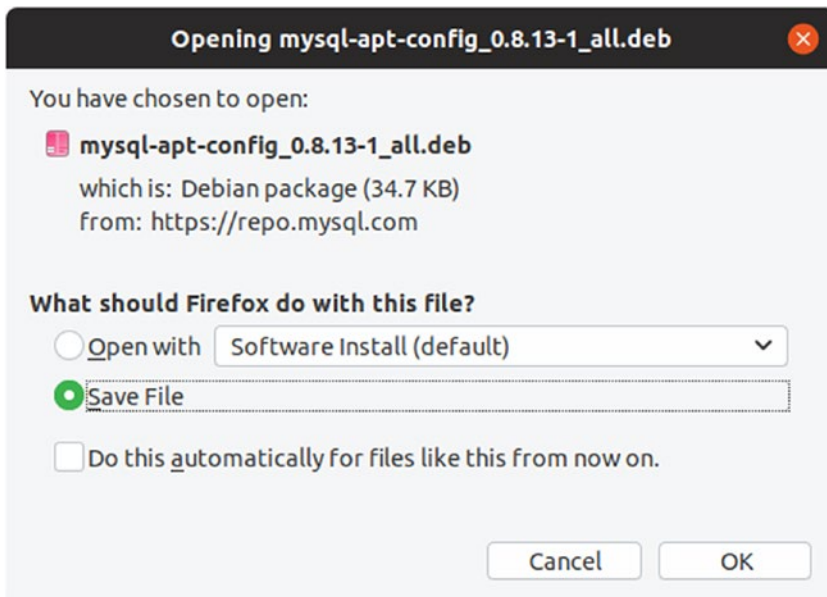


Figure 2-39. *Save File*

Now that we have the APT Repository downloaded, we can install it.

Installing the APT Repository

To install the APT Repository, open a terminal and change to the directory where you saved it (e.g., your Downloads folder). The steps needed to install the APT Repository include the following:

```
$ sudo dpkg -i mysql-apt-config_0.8.13-1_all.deb
$ sudo apt-get update
```

The first command installs the package sources needed for linking the Oracle repositories. The second command is used to update the package sources and enable the new sources. Listing 2-1 shows the transcript of installing the APT Repository. Your results should be similar.

Listing 2-1. Installing the APT Repository

```
$ sudo dpkg -i mysql-apt-config_0.8.13-1_all.deb
[sudo] password for cbell:
Selecting previously unselected package mysql-apt-config.
```

```
(Reading database ... 212217 files and directories currently installed.)
Preparing to unpack mysql-apt-config_0.8.13-1_all.deb ...
Unpacking mysql-apt-config (0.8.13-1) ...
Setting up mysql-apt-config (0.8.13-1) ...
OK
$ sudo apt-get update
Get:1 http://repo.mysql.com/apt/ubuntu xenial InRelease [19.1 kB]
Hit:2 http://us.archive.ubuntu.com/ubuntu xenial InRelease
Get:3 http://security.ubuntu.com/ubuntu xenial-security InRelease [107 kB]
Get:4 http://repo.mysql.com/apt/ubuntu xenial/mysql-8.0 Sources [994 B]
Get:5 http://us.archive.ubuntu.com/ubuntu xenial-updates InRelease [109 kB]
Get:6 http://repo.mysql.com/apt/ubuntu xenial/mysql-apt-config amd64
Packages [567 B]
Get:7 http://repo.mysql.com/apt/ubuntu xenial/mysql-apt-config i386
Packages [567 B]
Get:8 http://repo.mysql.com/apt/ubuntu xenial/mysql-8.0 amd64 Packages [7,150 B]
Get:9 http://repo.mysql.com/apt/ubuntu xenial/mysql-8.0 i386 Packages [7,143 B]
Get:10 http://repo.mysql.com/apt/ubuntu xenial/mysql-tools amd64 Packages
[3,353 B]
Get:11 http://repo.mysql.com/apt/ubuntu xenial/mysql-tools i386 Packages
[2,632 B]
Get:12 http://us.archive.ubuntu.com/ubuntu xenial-backports InRelease [107 kB]
Fetched 364 kB in 5s (64.6 kB/s)
Reading package lists... Done
```

At this point, your system is ready for installing MySQL products simply by specifying the product name in the APT command. Next, let's install MySQL Server.

Installing MySQL Server

Installing the server with the APT Repository can be done by installing the meta-package named `mysql server`. The following shows the command you can use to install the server and its most common components including the clients.

```
$ sudo apt-get install mysql-server
```

The output of the command is typical of most Linux installations and the contents are not particularly interesting for most, so we will skip examining the output. If you're curious, you can see all the packages and dependencies for the packages downloaded and installed.

During the installation, you will be prompted to select the packages you want to enable. For most, you can use the defaults. Select the *Ok* entry from the list to proceed with the defaults. The *Ok* at the bottom of the dialog moves to the next screen to configure the package chosen from the list. Figure 2-40 shows the dialog presented that allows you to configure certain packages. If you want to configure one of the packages, select the package using the *UP* and *DOWN* arrow keys, then press the *TAB* key to move to the *Ok* selection at the bottom of the screen to proceed.

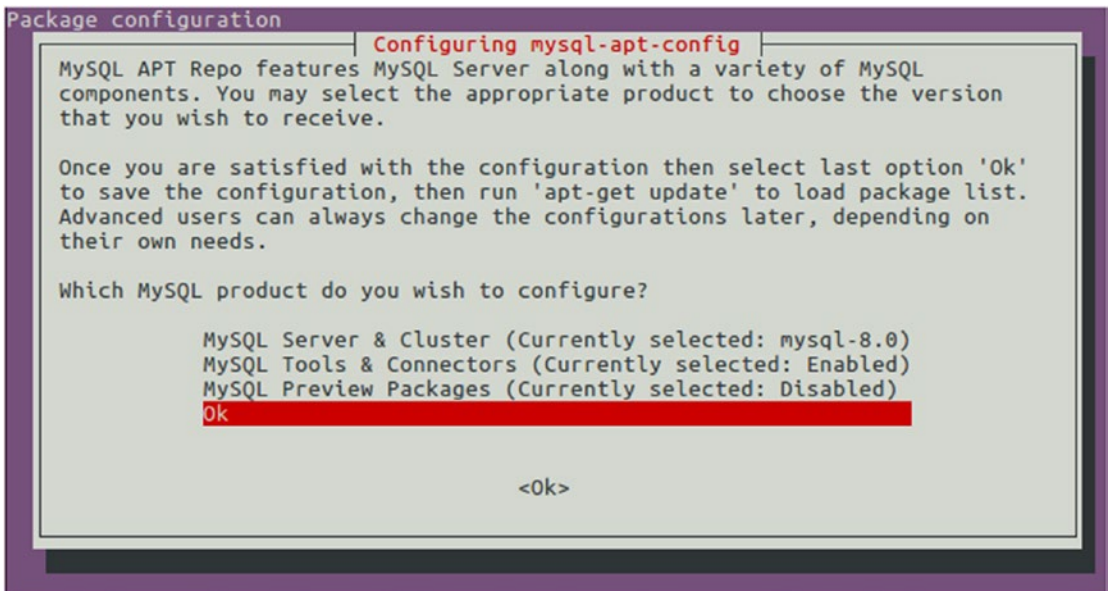


Figure 2-40. *Configuring Packages in the APT Repository*

The next step you will see is selecting the root password. You should choose a password that you will remember and is sufficiently complex that no one will guess easily. You are encouraged to use a password with at least eight characters that are a mix of letters, numbers, and other characters. Figure 2-41 shows the dialog for setting the root user password.

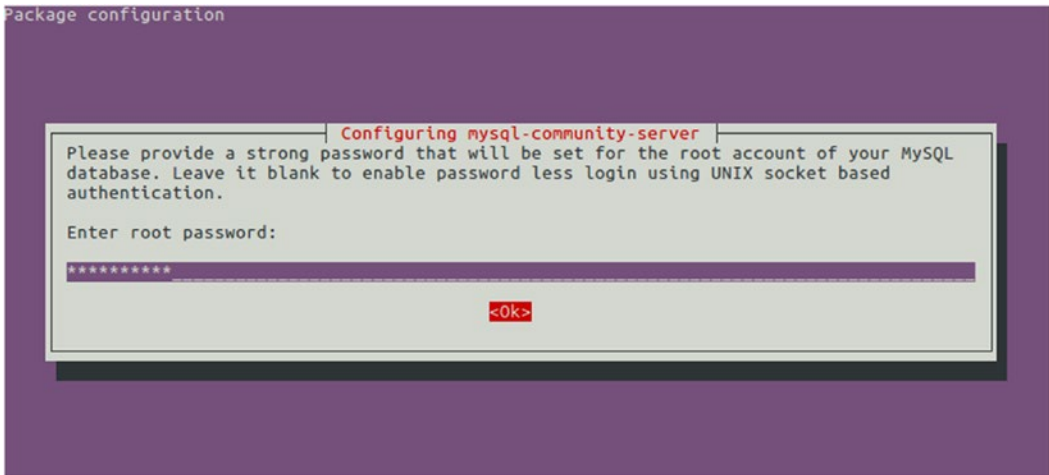


Figure 2-41. *Setting the Root User Password*

The next dialog is the last one you will see before returning to the terminal prompt. You must choose to use either strong password encryption (highly recommended) or use the legacy authentication method (not recommended for production). Strong encryption is selected by default, so we simply use the *TAB* key and select *Ok* to continue. Figure 2-42 shows the password encryption configuration dialog.

Note If you choose the strong password encryption method and you want to use an older version of the MySQL client (not the shell), you may encounter errors connecting. You must use the newer client with strong password encryption.

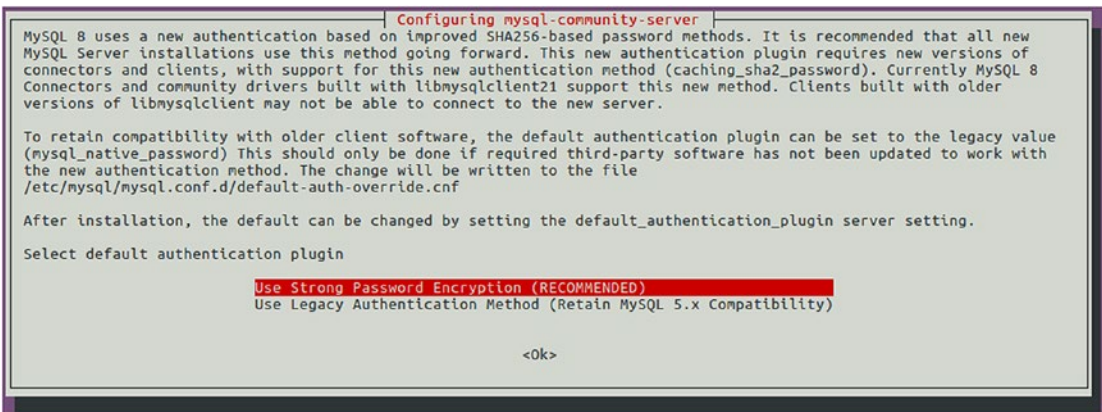


Figure 2-42. *Password Encryption Dialog*

If all went well, you should be unceremoniously returned to the terminal. You should not see any errors and may see the results of installing any additional components such as the following excerpt.

```
...
update-alternatives: using /var/lib/mecab/dic/ipadic-utf8 to provide /var/
lib/mecab/dic/debian (mecab-dictionary) in auto mode
Setting up mysql-server (8.0.16-1ubuntu16.04) ...
Processing triggers for libc-bin (2.23-0ubuntu10) ...
```

Next, we start the server with the following command. Note that this could take a while to run on the first start since the server must set up the data directory.

```
$ sudo service mysql start
```

Now, you can connect to the server. Since we haven't installed the shell yet, we can use the old client using the following command. The options specify the user (root) and the option to prompt for the user password.

```
$ mysql -uroot -p
```

This will result in launching the old client, which looks very similar to the examples we've seen thus far for the MySQL Shell. If you look closely, you will see the subtle differences in the welcome statements, but the biggest clue is the new prompt. Listing 2-2 shows an example of using the old client to connect to the server.

Listing 2-2. Connecting to MySQL (Old Client)

```
$ mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.16 MySQL Community Server - GPL

Copyright (c) 2000, 2019, Oracle and/or its affiliates. All rights reserved.
```

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

```
mysql> \q  
Bye!
```

Next, let's install MySQL Shell and test it by connecting to the server.

Installing MySQL Shell

To install the MySQL Shell, we must download the MySQL Shell distribution package. You can do this by visiting <https://dev.mysql.com/downloads/shell/>. You must then change the operating system to match your system (e.g., Ubuntu) and the version (e.g., 18.04) as shown in Figure 2-43. Or, you can leave the version set to "All" to see all the packages available. Click on the *Download* button for the package that matches your system.

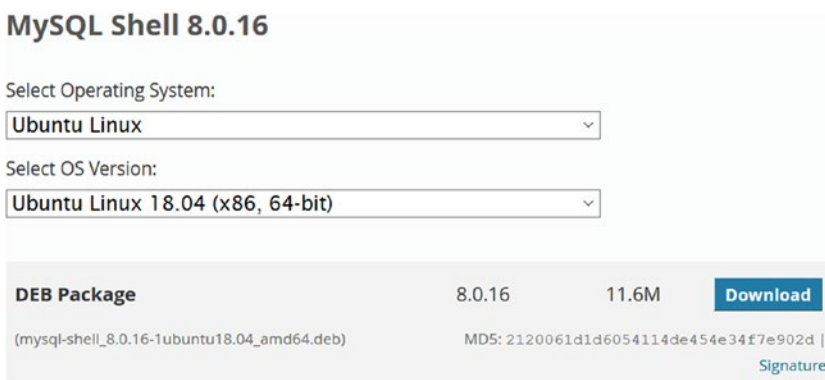


Figure 2-43. Downloading the MySQL Shell (Ubuntu)

For example, I was using Ubuntu 18.04 running the 64-bit version. Thus, I downloaded the file named `mysql-shell_8.0.16-1ubuntu18.04_amd64.deb`. Recall, depending on what browser you are using, you may need to click through the MySQL download acceptance dialog (see Figure 2-38) and save the file (see Figure 2-39). Once downloaded, you can install the shell with the following command.

```
$ sudo dpkg -i ./mysql-shell_8.0.16-1ubuntu18.04_amd64.deb
```

The output of running the command is very short as shown here.

```
(Reading database ... 151363 files and directories currently installed.)
Preparing to unpack .../mysql-shell_8.0.16-1ubuntu18.04_amd64.deb ...
Unpacking mysql-shell:amd64 (8.0.16-1ubuntu18.04) ...
Setting up mysql-shell:amd64 (8.0.16-1ubuntu18.04) ...
```

You can also install the shell using the APT repository by using the following command. Whichever you use is fine, but using the APT repository is the preferred method.

```
$ sudo apt-get install mysql-shell
```

At this point, we have the MySQL Server and Shell installed and configured. Now, let's test both by issuing the following command in a terminal window.

```
$ mysqlsh
```

Once you enter the command, you will see the shell launch as shown in Figure 2-44.

```

cbell@oracle-pc:~$ mysqlsh
MySQL Shell 8.0.16

Copyright (c) 2016, 2019, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

Type '\help' or '?' for help; '\quit' to exit.

MySQL JS > \sql
Switching to SQL mode... Commands end with ;

MySQL SQL > \connect root@localhost:3306
Creating a session to 'root@localhost:3306'
Please provide the password for 'root@localhost:3306': *****
Save password for 'root@localhost:3306'? [Y]es/[N]o/[e]v[er] (default No):
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 9
Server version: 8.0.16 MySQL Community Server - GPL
No default schema selected; type \use <schema> to set one.

MySQL localhost:3306 ssl SQL > SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.0014 sec)

MySQL localhost:3306 ssl SQL >

```

Figure 2-44. Using the MySQL Shell (Ubuntu)

Notice in this example, I issued several commands. Those that start with a slash (\) are shell commands. The first is to change the mode to SQL (\sql) for working with SQL commands, then I connected to the server with the \connect command passing the user id, host, and port as root@localhost:3306. Finally, I issued an SQL command to show all the databases. We will learn more about using the shell in the next chapter.

If you have been following along installing MySQL on your own machine, congratulations! You now have MySQL Server and MySQL Shell as well as other components you need to complete the examples in this book.

Summary

The MySQL Shell is a huge leap forward in technology for MySQL clients. Not only is it designed to work with SQL in MySQL in a smarter way, it is also designed to enable prototyping of JavaScript and Python. You can work with any language you want and switch between them easily without having to restart the application or drop the connection. How cool is that?

In this chapter, we learned how to install MySQL Shell. We also learned how to install MySQL Server. Demonstrations were presented installing MySQL on Windows using the MySQL Installer, macOS, and Linux (Ubuntu) using the APT Repository.

In the next chapter, we will see a short tutorial on the shell and its major features.

CHAPTER 3

MySQL Shell Tutorial

Now that we know what the MySQL Shell is and where it fits in the suite of MySQL products, it is time to learn what the shell can do for us. Specifically, what commands does it support, how does it connect to the server, and what features does it support?

In this chapter, we explore the MySQL Shell in more detail. We will learn more about its major features and options as well as see how to use the new shell to execute scripts interactively. As you will see, the MySQL Shell is an important element of the future of MySQL. Let's begin with the commands and options supported by the shell.

Commands and Options

If you're thinking the new shell is nothing more than an improved version of the original client, you could not be further from the truth. The shell is much more than a simple replacement for the original client. To get you started learning how it differs and how it is much more sophisticated than the original shell, let's begin by examining the commands and options supported.

Commands are those special entries that you can provide at the prompt that interacts with the MySQL Shell application directly. These are often referred to as shell commands and begin with a slash (\). Options refer to the many parameters (options) that you can specify when you launch the MySQL Shell. Thus, the shell supports customization at launch such as connecting to a server, setting the mode, and much more.

We will learn more about the commands and options for the shell in the following sections, but first let's briefly discuss how to start the shell.

Starting the MySQL Shell

Depending on your platform and how you installed the shell, you may be able to start the shell from the system (e.g., Start) menu. However, on all platforms, the installation places the shell executable in a place where it can be executed from the command line. The shell executable on Windows is named `mysqlsh.exe`. On other platforms, it is simply `mysqlsh`.

For example, on Windows, you can start the shell in a special terminal using the Start menu as shown in Figure 3-1.

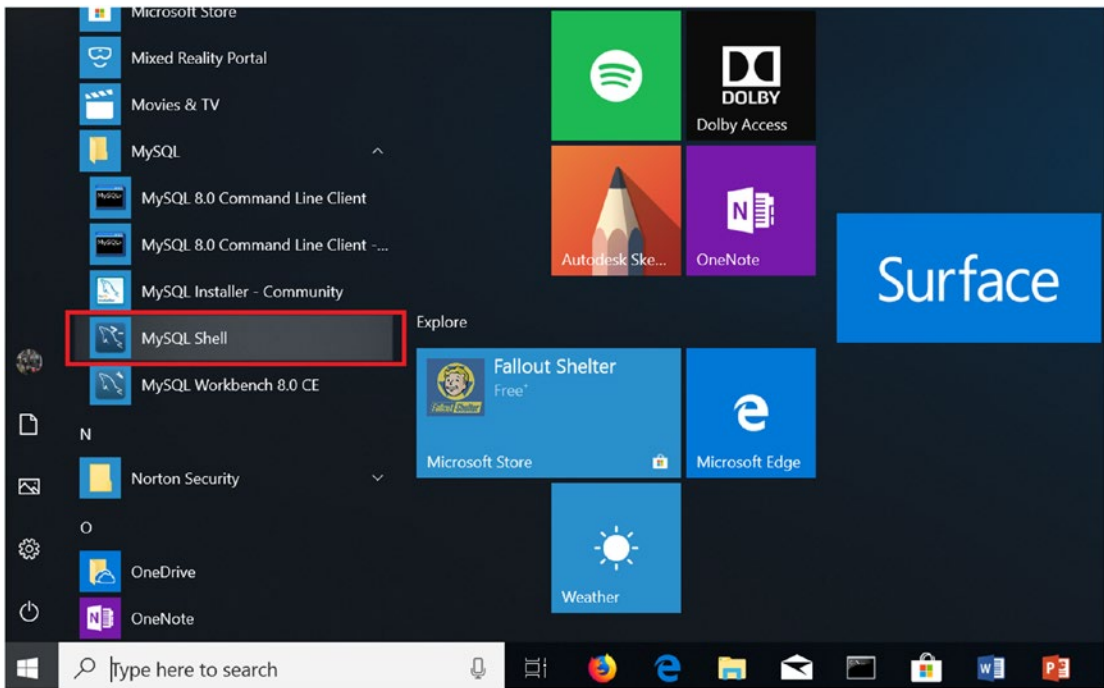


Figure 3-1. *Launching the Shell from the Start Menu (Windows)*

Most people will launch the shell from the command line. We can do this by opening a terminal (or console) and running the shell executable. For example, Figure 3-2 shows how to launch the shell on Windows from a command window. You can do the same on other platforms.

```

Command Prompt
C:\Users\olias>mysqlsh
MySQL Shell 8.0.16

Copyright (c) 2016, 2019, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

Type '\help' or '\?' for help; '\quit' to exit.

MySQL JS > \q
Bye!

C:\Users\olias>

```

Figure 3-2. *Launching the Shell from the Command Line (Windows)*

Commands

Like the original MySQL client, there are some special commands that control the application itself rather than interact with data (via SQL or the X DevAPI). To execute a shell command, issue the command with a slash (\). For example, `\help` prints the help for all the shell commands. Table 3-1 lists some of the more frequently used shell commands.

Table 3-1. *Shell Commands*

Command	Shortcut	Description
<code>\</code>		Start multiline input (SQL mode only)
<code>\connect</code>	<code>\c</code>	Connect to a server
<code>\help</code>	<code>\?, \h</code>	Print the help text
<code>\history</code>		View and edit command line history
<code>\js</code>		Switch to JavaScript mode
<code>\nowarnings</code>	<code>\w</code>	Don't show warnings
<code>\option</code>		Query and change MySQL Shell configuration options
<code>\py</code>		Switch to Python mode
<code>\quit</code>	<code>\q, \exit</code>	Quit

(continued)

Table 3-1. (continued)

Command	Shortcut	Description
<code>\reconnect</code>		Reconnect to the same MySQL server
<code>\rehash</code>		Manually update the autocomplete name cache
<code>\source</code>	<code>\.</code>	Executes the script file specified
<code>\sql</code>		Switch to SQL mode
<code>\status</code>	<code>\s</code>	Print information about the connection
<code>\use</code>	<code>\u</code>	Set the schema for the session
<code>\warnings</code>	<code>\W</code>	Show warnings after each statement

Some of the commands, such as the `\connect` command, take one or more parameters. The best way to learn how to use the shell commands is the `\help` command. You can use this command without parameters to get help about it. For example, to learn more about the `\connect` command, enter `\help connect` as shown in Listing 3-1. Here, we start the shell from the command line without any options, which starts in the default JavaScript mode, issue a few help commands, then quit the shell with the `\q` command. Portions of the output have been omitted for brevity and the commands bolded for easier reading.

Listing 3-1. Getting Help in MySQL Shell

```
C:\>mysqlsh
```

```
MySQL Shell 8.0.16
```

```
...
```

```
MySQL JS > \help sql
```

```
Found several entries matching sql
```

```
The following topics were found at the X DevAPI category:
```

- `mysqlx.Session.sql`
- `mysqlx.SqlExecute.sql`

```
For help on a specific topic use: \? <topic>
```

e.g.: \? mysqlx.Session.sql

MySQL JS > \help connect

NAME

connect - Establishes the shell global session.

SYNTAX

```
shell.connect(connectionData[, password])
```

WHERE

connectionData: the connection data to be used to establish the session.

password: The password to be used when establishing the session.

DESCRIPTION

This function will establish the global session with the received connection data.

The connection data may be specified in the following formats:

- A URI string
- A dictionary with the connection options

...

MySQL JS > \q

Bye!

Tip Use the `\help <command>` to discover how to use a new command.

You can use the `\sql`, `\js`, and `\py` shell commands to switch the mode on the fly. Not only does this mean you can change modes without restarting the shell, it also makes working with SQL and NoSQL data much easier by allowing you to switch the mode whenever you want. For example, you can execute a few SQL commands, then connect to the X DevAPI to run JavaScript, return to SQL, then switch to running Python scripts. Furthermore, you can use these shell commands even if you used the startup option to set the mode.

Notice the way you exit the shell is with the `\q` (or `\quit`) command. If you type `quit` like you're used to in the old client, the shell will respond differently depending on the mode you're in. Listing 3-2 presents an example of what happens in each mode. Let's start with the default mode (JavaScript) then switch to Python and finally SQL mode.

Listing 3-2. Results of Using `quit` in Different Modes

```
MySQL JS > quit
ReferenceError: quit is not defined

MySQL JS > \py
Switching to Python mode...

MySQL Py > quit
Traceback (most recent call last):
  File "<string>", line 1, in <module>
NameError: name 'quit' is not defined

MySQL Py > \sql
Switching to SQL mode... Commands end with ;

MySQL SQL > quit;
ERROR: Not connected.

MySQL SQL >
MySQL SQL > \quit
Bye!
```

You may see similar oddities if you are used to the old MySQL client and accidentally use an old client command, but it only takes a bit of regular use to remind you of the correct commands to use. Now, let's look at the command line options for the shell.

Options

The shell can be launched using several startup options that control the mode, connection, behavior, and more. This section introduces some of the more common options that you may want to use. We will see the more about the connection options in a later section. Table 3-2 shows the common shell options. These are just a few of the many options available.

Table 3-2. *Common MySQL Shell Options*

Option	Description
--auth-method=method	Authentication method to use
--cluster	Ensures the target is part of an InnoDB Cluster
--compress	Enable compression between client and server
--database=name	An alias for --schema
--dba=enableXProtocol	Enable the X Protocol in the server connected to. Must be used with --mysql
--dbpassword=name	Password to use when connecting to server
--dbuser=name, -u	User to use for the connection
--execute=<cmd>, -e	Execute command and quit
--file=file, -f	Process file for execution
--host=name, -h	Hostname to use for connection
--import	Import one or more JSON documents
--interactive[=full], -i	To use in batch mode, it forces emulation of interactive mode processing. Each line on the batch is processed as if it were in interactive mode
--js	Start in JavaScript mode
--json=[raw pretty]	Produce output in JSON format in raw format (no formatting) or pretty (human readable format)
--log-level=value	The log level. Value must be an integer between 1 and 8 or any of [none, internal, error, warning, info, debug, debug2, debug3]
--mc --mysql	Create a classic (old protocol) session
--mx --mysqlx	Create an X protocol session (simply called “Session”)
--name-cache	Enable loading of table names for default schema
--no-name-cache	Disable loading of table names for default schema
--nw, --no-wizard	Disables wizard mode (noninteractive) for executing scripts

(continued)

Table 3-2. (continued)

Option	Description
-p	Request password prompt to set the password
--password=name	An alias for dbpassword
--port=#, -P	Port number to use for connection
--py	Start in Python mode
--schema=name, -D	Schema to use
--socket=sock, -S	Socket name to use for connection in UNIX or a named pipe name in Windows (only classic sessions)
--sql	Start in SQL mode
--sqlc	Start in SQL mode using a classic session
--sqlx	Start in SQL mode using Creating an X protocol session
--ssl-ca=name	CA file in PEM format (check OpenSSL docs)
--ssl-capath=dir	CA directory
--ssl-cert=name	X509 cert in PEM format
--ssl-cipher=name	SSL Cipher to use
--ssl-crl=name	Certificate revocation list
--table	Show results in table format
--tabbed	Show results in tabbed format
--uri	Provide connection information in the form of user@host:port
--vertical	Show results in vertical format (like \G)

Notice there are aliases for some of the options that have the same purpose as the original client. This makes switching to the shell a bit easier if you have scripts for launching the client to perform operations. Notice also there is a set of options for using a secure socket layer (SSL) connection. There are also options for controlling how the output is viewed – as a traditional table (think SQL results), vertical orientation, or even as JSON. Take a few moments and scan through this list to familiarize yourself

with what is available. However, there are other options that are not commonly used. For a complete list of the available options, see <https://dev.mysql.com/doc/mysql-shell/8.0/en/mysqlsh.html>. Don't worry about memorizing these now or how to use them – we will see many of these in action in later chapters.

Getting Started with the MySQL Shell

As we've learned, the MySQL Shell is a new and exciting addition to the MySQL portfolio. Not only is it a new client, it is also an excellent scripting environment for developing new tools and applications for working with data. Cool!

Note We won't examine every aspect of MySQL Shell; rather, we focus on the commonly used features. We will also see many of the features demonstrated in later chapters. See the MySQL Shell online user's manual for more information about additional features such as application logs, startup scripts, and working with environment variables (<https://dev.mysql.com/doc/mysql-shell/8.0/en/>).

Let's see the shell in action once more. Listing 3-3 shows an example of using the command line options to connect to our MySQL server via a Uniform Resource Identifier (URI) setting the default schema, mode to SQL, and the output format to vertical.

Note In future examples, I will use a listing to show the shell in action rather than a figure.

Listing 3-3. Starting the MySQL Shell with Options

```
C:\Users\cbell>mysqlsh --sql --uri=root@localhost -p -D world --vertical
Creating a session to 'root@localhost/world'
Please provide the password for 'root@localhost': *****
Save password for 'root@localhost'? [Y]es/[N]o/[N]e[v]er (default No):
Fetching schema names for autocompletion... Press ^C to stop.
```

CHAPTER 3 MYSQL SHELL TUTORIAL

Fetching table and column names from `world` for auto-completion... Press ^C to stop.

```
Your MySQL connection id is 11 (X protocol)
Server version: 8.0.16 MySQL Community Server - GPL
Default schema set to `world`.
MySQL Shell 8.0.16
```

Copyright (c) 2016, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type '\help' or '\?' for help; '\quit' to exit.

```
MySQL localhost:33060+ ssl world SQL > SHOW TABLES;
***** 1. ROW *****
Tables_in_world: city
***** 2. ROW *****
Tables_in_world: country
***** 3. ROW *****
Tables_in_world: countrylanguage
3 rows in set (0.0014 sec)

MySQL localhost:33060+ ssl world SQL > \q
Bye!
```

As you can see, you can start the shell with or without options. You can choose to connect on start or wait and connect to the server after you've started the shell. In fact, you can also change connections to another server without restarting the application whether you used command line options to connect or not. That's a nice feature when working with multiple MySQL servers and, as we will see, is also key to effectively setting up high-availability solutions.

As we saw in the table and previous examples, you must use the \connect shell command to connect to a server. These connections are called sessions and the shell has several features for working with sessions. Connections can use either the original client/server protocol or the new X Protocol for communicating with the server via the

X Plugin. What this means is the shell allows you to work with both relational (SQL) and JSON documents (NoSQL), or both. We will discover more about connections in a later section.

Along with sessions, it is important to understand the various modes that the shell supports. Recall, these include an SQL, JavaScript, and Python mode. Like connections, you can specify the mode on the command line and change the mode at any time while using the shell. This allows you to switch from Python to SQL and back as you need to – all without leaving the application.

The following sections present the session and mode features of the shell at a high level. Learning these features is key to understanding how to make connections in the shell. We will return to learning about making connections in a later section. For more information about the MySQL Shell, see the section entitled, “MySQL Shell User Guide” in the online MySQL reference manual.

Sessions and Modes

Like the original client and indeed most MySQL client applications, you will need to connect to a MySQL server so that you can run commands. The MySQL Shell supports several ways to connect to a MySQL server and a variety of options for interacting with the server (called a session). Within a session, you can change the way the shell accepts commands (called modes) to include SQL, JavaScript, or Python commands.

Given all the different and new concepts of working with servers, those new to using the shell may find the difference subtle and even at times confusing. Indeed, the online reference manual and various blogs and other reports sometimes use mode and session interchangeably, but as you will see, they are different (however subtle). Let’s begin by looking at the session objects available.

Session Objects

The first thing to understand about sessions is that a session is a connection to a single server. The second thing to understand is that each session can be started using one of two session objects that exposes a specific object for use in working with the MySQL server using a specific communication protocol. That is, sessions are connections to servers (with all parameters defined) and a session object is what the shell uses to interact with a server in one of several ways. More specifically, a MySQL Shell session

object simply defines how you interact with the server including what modes are supported and even how the shell communicates with the server. The shell supports two session objects as follows:

- *Session*: An X Protocol session is used for application development and supports the JavaScript, Python, and SQL modes. Typically used to develop scripts or execute scripts. To start the shell with this option, use the `--mx (--mysqlx)` option.
- *Classic Session*: Uses the older server communication protocol with very limited support for the DevAPI. Use this mode with older servers that do not have the X Plugin or do not support the X Protocol. Typically used for SQL mode with older servers. To start the shell with this option, use the `--mc (--mysqlc)` option.

You can specify the session object (protocol) to use when you use the `\connect` shell command by specifying `-mc` for classic session or `-mx` for X Protocol session. The following shows each of these in turn. Note that `<URI>` specifies a uniform resource identifier.

- `\connect -mx <URI>`: Use the X Protocol (session)
- `\connect -mc <URI>`: Use the classic protocol (classic session)

The URI referenced in this chapter and elsewhere refers to a specific format or layout of the connection information used when connecting to a MySQL server. The following shows the format for constructing a URI.

```
[scheme://][user[:[password]]@]target[:port][/schema][?attribute1=value1&attribute2=value2...
```

Notice we can specify the connection (session) type, which is optional, the user, password (not recommended), the target or host, port, and even the schema as well as any options used by the session type. For example, a simple URI for connecting to the local MySQL server is shown in the following. Here, we use the user name `root` and port `3306`. The shell will prompt for the password if not included in the URI.

```
/connect root@localhost:3306
```

Of course, you can still use the individual command line options for user, host, and port. However, the norm is to use URIs with the `/connect` command or command line option.

Tip See <https://dev.mysql.com/doc/refman/8.0/en/connecting-using-uri-or-key-value-pairs.html> for more information about using URIs.

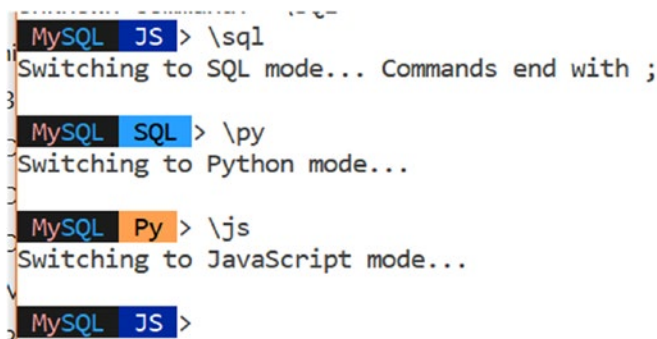
Recall sessions are loosely synonymous with a connection. However, a session is a bit more than just a connection since all the settings used to establish the connection including the session object are included as well as the protocol to use to communicate with the server. Thus, we sometimes encounter the term “protocol” for describing a session.

Modes Supported

The shell supports three modes (also called language support or simply the active language); SQL, JavaScript, and Python. Recall we can initiate any one of these modes by using a shell command. You can switch modes (languages) as often as you want without disconnection each time. The following lists the three modes and how to switch to each.

- `\sql`: Switch to the SQL language
- `\js`: Switch to the JavaScript language (default mode)
- `\py`: Switch to the Python language

You can switch modes any time you want, or you can start the shell in a particular mode. The default mode is JavaScript. Thus, if you do not specify a mode on the command line, you will be presented with the JavaScript prompt. This is how you will know which mode you are in. Figure 3-3 shows the various modes. They are also color-coded with JavaScript yellow, SQL orange, and Python blue. Nice!



```
MySQL JS > \sql
Switching to SQL mode... Commands end with ;

MySQL SQL > \py
Switching to Python mode...

MySQL Py > \js
Switching to JavaScript mode...

MySQL JS >
```

Figure 3-3. MySQL Shell Mode Prompts

Now that we understand sessions and modes, we can look at how to make connections to MySQL servers.

Using Connections

Making connections in the shell is one area that may take some getting used to doing differently than the original MySQL client, which required the use of several options on the command line. You can use a specially formatted URI string or connect to a server using individual options by name (like the old client). SSL connections are also supported. Connections can be made via startup options, shell commands, and in scripts. However, all connections are expected to use a password. Thus, unless you state otherwise, the shell will prompt for a password if one is not given.

Note If you want to use a connection without a password (not recommended), you must use the `--password` option or, if using a URI, include an extra colon to take the place of the password.

Rather than discuss all the available ways to connect and all the options to do so, the following presents one example of each method of making a connection in the following sections.

Using a URI

Recall, a URI is a string that uses a special format to include the values for the various parameters. The password, port, and schema are optional, but the user and host are required. Schema in this case is the default schema (database) that you want to use when connecting. The default port for the old client/server protocol is 3306 and the default port for the X Protocol is 33060. To connect to a server using a URI on the command line when starting the shell, specify it with the `--uri` option as follows.

```
$ mysqlsh --uri root:secret@localhost:3306
```

Tip If you omit `--uri` but still include a URI, the shell will process the string presented as a URI.

The shell assumes all connections require a password and will prompt for a password if one is not provided. Listing 3-4 shows the same preceding connection made without the password. Notice how the shell prompts for the password.

Listing 3-4. Connecting with a URI

```
C:\Users\cbell>mysqlsh --uri root@localhost:33060/world_x --sql
Creating a session to 'root@localhost:33060/world_x'
Please provide the password for 'root@localhost:33060': *****
Save password for 'root@localhost:33060'? [Y]es/[N]o/[N]ever (default No):
Fetching schema names for autocompletion... Press ^C to stop.
Fetching table and column names from `world_x` for auto-completion...
Press ^C to stop.
Your MySQL connection id is 16 (X protocol)
Server version: 8.0.16 MySQL Community Server - GPL
Default schema set to `world_x`.
MySQL Shell 8.0.16

Copyright (c) 2016, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type '\help' or '\?' for help; '\quit' to exit.

MySQL localhost:33060+ ssl world_x SQL > \q
Bye!
```

Notice we also specified the default schema (*world_x*) with the schema option in the URI. The *world_x* database is a sample database you can download from <https://dev.mysql.com/doc/index-other.html>. We will install this database during the tutorial on MySQL Shell in a later section.

Using Individual Options

You can also specify connections on the shell command line using individual options. The available connection options available are those shown in Table 3-1. Listing 3-5 shows how to connect to a MySQL server using individual options. Notice I changed the mode (language) to Python with the `--py` option.

Listing 3-5. Connecting Using Individual Options

```
C:\Users\cbell>mysqlsh --user root --host localhost --port 33060 --schema
world_x --py --mx
Creating an X protocol session to 'root@localhost:33060/world_x'
Please provide the password for 'root@localhost:33060': *****
Save password for 'root@localhost:33060'? [Y]es/[N]o/[N]e[v]er (default No):
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 17 (X protocol)
Server version: 8.0.16 MySQL Community Server - GPL
Default schema `world_x` accessible through db.
MySQL Shell 8.0.16

Copyright (c) 2016, 2019, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type '\help' or '\?' for help; '\quit' to exit.

MySQL localhost:33060+ ssl world_x Py > \q
Bye!
```

Using Connections in Scripts

If you plan to use the shell to create scripts or simply as a prototyping tool, you will also want to use sessions in your scripts. While some of the examples may seem strange and are presented without detailed explanations, it is always a good idea to see how things in action before learning the details of how to do those steps (and

why). Thus, you should read this section to become familiar with what is possible. In this section, we will explore how to use session in scripts. We will learn how to use the examples in later chapters.

What makes using sessions in scripts powerful is that we can save them and reuse them later. In this case, we will create a variable to contain the session once it is fetched. A session created in this manner is called a global session because once it is created, it is available to any of the modes. However, depending on the session object we're using (recall this is either Classic or X Protocol), we will use a different method of the `mysqlx` object to create an X or Classic session. We use the `get_session()` method for an X Protocol session object, and the `get_classic_session()` method for a classic session object.

Note We will focus on the Python scripts but many of the examples also apply to JavaScript albeit with a slightly different capitalization of the classes and methods.

The following demonstrates getting an X Protocol session object in Python. Notice I specify the password in a URI and the password as a separate parameter. I omit the shell prompt for brevity.

```
> my_session = mysqlx.get_session('root@localhost:33060', 'secret');
> print(my_session)
<Session:root@localhost:33060>
```

The following demonstrates getting a Classic session object in Python.

```
Py > my_classic = mysql.get_classic_session('root@localhost:3306', 'secret');
Py > print(my_classic)
<ClassicSession:root@localhost:3306>
```

Using SSL Connections

You can also create SSL connections for secure connections to your servers. To use SSL, you must configure your server to use SSL. To use SSL on the same machine where MySQL is running, you can use the `--ssl-mode=REQUIRED` option.

Use the SHOW VARIABLES command to view the state of the SSL variables to determine if your server has SSL enabled. Listing 3-6 shows the results of running the query with ssl surrounded by %, which are wildcards. This will result in showing all variables with ssl in the name. As we can see, SSL is indeed enabled (see have_ssl, which shows YES). If your server is not set up to use SSL and you want to use SSL connections, see the online MySQL Reference Manual (<https://dev.mysql.com/doc/refman/8.0/en/encrypted-connections.html>) to learn how to set up SSL on your server.

Listing 3-6. Checking for SSL Support

```
SQL > SHOW VARIABLES LIKE '%ssl%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| have_openssl  | YES   |
| have_ssl      | YES   |
| mysqlx_ssl_ca |       |
| mysqlx_ssl_capath |       |
| mysqlx_ssl_cert |       |
| mysqlx_ssl_cipher |       |
| mysqlx_ssl_crl |       |
| mysqlx_ssl_crlpath |       |
| mysqlx_ssl_key |       |
| ssl_ca        | ca.pem |
| ssl_capath    |       |
| ssl_cert      | server-cert.pem |
| ssl_cipher    |       |
| ssl_crl       |       |
| ssl_crlpath   |       |
| ssl_fips_mode | OFF   |
| ssl_key       | server-key.pem |
+-----+-----+
17 rows in set (0.0207 sec)
```

You can also specify the SSL options as shown in Table 3-1. You can specify them on the command line using the command line options or as an extension to the `\connect` shell command. The following shows how to connect to a server using SSL and command line options.

```
C:\Users\cbell>mysqlsh -uroot -h127.0.0.1 --port=33060 --ssl-mode=REQUIRED
```

Note Older versions of the MySQL server may not have the X Plugin enabled. See the online MySQL Reference manual for how to enable the X Plugin on version 5.7 and earlier releases of 8.0.

Now, let's see the MySQL Shell in action by way of a demonstration of its basic features.

Working with the MySQL Shell

The following sections demonstrate how to use the MySQL shell in the most basic of operations – selecting and inserting data. The examples use the *world_x* database and are designed to present an overview rather than a deep dive. If you do not know anything about the MySQL Document Store or JSON data, do not despair; the tutorial is meant to demonstrate working with the MySQL Shell and since the shell is intended for use with JSON documents, we will do so.

Thus, the objective in this tutorial is to insert new data in the *world_x* database and then execute a search to retrieve rows that meet criteria that contains the new data. I will use a relational table to illustrate the concepts since that is easier for those of us familiar with “normal” database operations.

Before we begin our journey, let's take a moment to install the sample database we will need, the *world_x* sample MySQL database from Oracle.

Installing the Sample Database

Oracle provides several sample databases for you to use in testing and developing your applications. Sample databases can be downloaded from <http://dev.mysql.com/doc/index-other.html>. The sample database we want to use is named *world_x* to indicate it contains JSON documents and is intended for testing with the X DevAPI, the shell, etc.

Go ahead and navigate to that page and download the database. The sample database contains several relational tables (country, city, and countrylanguage) as well as a collection (countryinfo).

Once you’ve downloaded the file, uncompress it and note the location of the files. You will need that when we import it. Next, start the MySQL Shell and make a connection to your server. Use the `\sql` shell command to switch to SQL mode, then the `\source` shell command to read the `world_x.sql` file and process all its statements. Listing 3-7 demonstrates how to use these options and install the sample database. Responses from running the `\source` command are omitted for brevity.

Listing 3-7. Installing the world_x Sample Database

```

JS > \sql
Switching to SQL mode... Commands end with ;
SQL > \source world_x.sql
Query OK, 0 rows affected (0.0034 sec)
Query OK, 0 rows affected (0.0004 sec)
Query OK, 0 rows affected (0.0003 sec)
...
Query OK, 0 rows affected (0.0003 sec)
Query OK, 0 rows affected (0.0002 sec)
Query OK, 0 rows affected (0.0003 sec)
Query OK, 0 rows affected (0.0003 sec)
Query OK, 0 rows affected (0.0002 sec)
Query OK, 0 rows affected (0.0003 sec)

MySQL localhost:3306 ssl SQL > show databases;
+-----+
| Database          |
+-----+
| information_schema |
| mysql             |
| performance_schema |
| sakila            |
| sys               |
| world_x           |
+-----+

```

```

6 rows in set (0.0045 sec)
SQL > USE world_x;
Query OK, 0 rows affected (0.00 sec)
SQL > SHOW TABLES;
+-----+
| Tables_in_world_x |
+-----+
| city      |
| country  |
| countryinfo      |
| countrylanguage |
+-----+
4 rows in set (0.00 sec)

```

Notice I have another sample database installed named *sakila*, which you can also find on the Oracle web site with the *world_x* database. I also show the tables located in the sample database.

Tip If the path to the file has spaces in it, you should include the path within double quotes.

You can also install the sample database using the `--recreate-schema` option on the command line as follows. Note that this will delete and recreate the database if it already exists. This is another example of running the SQL commands as a batch.

```

C:\Users\cbell\Downloads\world_x-db>mysqlsh -uroot -hlocalhost --sql
--recreate-schema --schema=world_x < world_x.sql
Please provide the password for 'root@localhost': *****
Please pick an option out of [Y]es/[N]o/Ne[v]er (default No):
Recreating schema world_x...

```

Of course, you could install the sample database with the old client by using the similar source command, but where's the fun in that?

Now, let's see how we can work with data.

Working with Data

In this section, we will see some simple examples of selecting and inserting data in the database. I use the city table in the *world_x* database demonstrating the JSON data type in queries. As you will see, this opens a new way to work with data. Once again, we will see more about the JSON data type in the next chapter. For this section, you should focus on the interactions within the shell. That is, how to use the shell to run the queries. If you're a pro at SQL databases, all of this except for the JSON data type will be very familiar. Let's start with querying data.

The task we want to accomplish is to see what rows are in the city table. In this case, we will retrieve (select) those rows that include cities in the United States. We will sort the rows by name and for brevity only show the first 20 rows. Listing 3-8 shows how to execute the query. Even if you don't know SQL, the query reads easy.

Listing 3-8. Selecting Rows

```
SQL > SELECT Name, District, Info FROM city WHERE CountryCode = 'USA' ORDER
BY Name DESC LIMIT 20;
```

Name	District	Info
Yonkers	New York	{"Population": 196086}
Worcester	Massachusetts	{"Population": 172648}
Winston-Salem	North Carolina	{"Population": 185776}
Wichita Falls	Texas	{"Population": 104197}
Wichita	Kansas	{"Population": 344284}
Westminster	Colorado	{"Population": 100940}
West Valley City	Utah	{"Population": 108896}
West Covina	California	{"Population": 105080}
Waterbury	Connecticut	{"Population": 107271}
Washington	District of Columbia	{"Population": 572059}
Warren	Michigan	{"Population": 138247}
Waco	Texas	{"Population": 113726}
Visalia	California	{"Population": 91762}
Virginia Beach	Virginia	{"Population": 425257}
Vancouver	Washington	{"Population": 143560}

```

| Vallejo          | California          | {"Population": 116760} |
| Tulsa           | Oklahoma            | {"Population": 393049} |
| Tucson          | Arizona             | {"Population": 486699} |
| Torrance        | California          | {"Population": 137946} |
| Topeka          | Kansas              | {"Population": 122377} |
+-----+-----+-----+
20 rows in set (0.0024 sec)

```

Notice the Info column. This is a JSON data type column and the data is displayed as a JSON document (e.g., {"Population": 122377}). This shows that the JSON document contains a single key, value pair representing the population for each of the cities. We can see how the table was constructed using a `SHOW CREATE TABLE` query as shown in Listing 3-9. Here, we see the JSON data type for the Info column.

Listing 3-9. SHOW CREATE TABLE Example

```

SQL > SHOW CREATE TABLE city\G
***** 1. TOW *****
      Table: city
Create Table: CREATE TABLE `city` (
  `ID` int(11) NOT NULL AUTO_INCREMENT,
  `Name` char(35) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL
  DEFAULT '',
  `CountryCode` char(3) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL
  DEFAULT '',
  `District` char(20) CHARACTER SET utf8 COLLATE utf8_general_ci NOT NULL
  DEFAULT '',
  `Info` json DEFAULT NULL,
  PRIMARY KEY (`ID`)
) ENGINE=InnoDB AUTO_INCREMENT=4080 DEFAULT CHARSET=utf8
1 row in set (0.0038 sec)

```

You may be wondering if we can use the JSON document in the Info column in our queries. The answer is, yes, you can! We can do so using a special function (one of the dozens designed to work with the JSON data type) to extract the key, value pairs. In this case, we will use the `JSON_EXTRACT()` function passing in the column name and a specially formatted string called a path expression. In this case, we want to select those

rows representing cities with a population of 500,000 or more. Listing 3-10 shows an example of the modified query. In this case, we will limit the output to only the first 10 rows but keep the ordering options and limiting the columns to only the name, district, and information columns. Also, we will use the shell in a batch mode to launch the shell, run the query, and then exit.

Listing 3-10. Selecting Rows Using JSON Data Type (Batch Mode)

```
C:\Users\cbell\Downloads\world_x-db>mysqlsh --uri=root@localhost:3306 --sql
--table -e "SELECT Name, District, Info FROM world_x.city WHERE CountryCode
= 'USA' AND JSON_EXTRACT(Info, '$.Population') > 500000 ORDER BY Name DESC
LIMIT 10;"
```

Please provide the password for 'root@localhost:3306': *****
 Save password for 'root@localhost:3306'? [Y]es/[N]o/Ne[v]er (default No):

```
+-----+-----+-----+
| Name          | District                | Info                                |
+-----+-----+-----+
| Washington    | District of Columbia   | {"Population": 572059} |
| Seattle       | Washington              | {"Population": 563374} |
| San Jose      | California              | {"Population": 894943} |
| San Francisco | California              | {"Population": 776733} |
| San Diego     | California              | {"Population": 1223400}|
| San Antonio   | Texas                   | {"Population": 1144646}|
| Portland      | Oregon                  | {"Population": 529121} |
| Phoenix       | Arizona                 | {"Population": 1321045}|
| Philadelphia  | Pennsylvania            | {"Population": 1517550}|
| Oklahoma City | Oklahoma                | {"Population": 506132} |
+-----+-----+-----+
```

Let’s look at the command a little closer. Notice we start the shell with the --uri to provide the login information, --sql to turn on SQL mode, --table to display the output in table mode (the default is tab), and the -e option followed by the query to execute and return the results. You can use this mechanism to insert the shell in any batch job.

The next task we want to do is to insert data. To make things interesting, we will modify some of the rows to include interesting sites you can visit while in town. In a pure relational database, this would require changing the table or adding a new table to store the new information. But the JSON data type allows us to add information in a free-form

manner via more key, value pairs. Thus, we will modify the information column to include additional information about the interesting sites. In other words, we can add our own comments about places we've visited and would recommend to others.

For this example, we will add two sites; the Smithsonian National Air and Space Museum in Washington, D.C. (<https://airandspace.si.edu/>) and National Harbor in Baltimore, MD (<https://www.nationalharbor.com/>). These are two excellent sites worth visiting when you are in the area.

We know there exists a row in the table for Washington, D.C. but what about Baltimore? We can run a quick batch query like the preceding one, but this time we will see the result in tab form.

```
C:\Users\cbell\Downloads\world_x-db>mysqlsh --uri=root@localhost:3306
--sql -e "SELECT Name, Info FROM world_x.city WHERE CountryCode = 'USA'
AND DISTRICT = 'Maryland'"
Please provide the password for 'root@localhost:3306': *****
Save password for 'root@localhost:3306'? [Y]es/[N]o/[e]x[ist]ing (default No):
Name      Info
Baltimore {"Population": 651154}
```

Now that we know there are rows for both cities, all we need do is construct the JSON we want to insert. In this case, we will use another JSON function to add the new data. But first, let's see what that data looks like. In this case, we want to include the name of the site, the URL for the web site, and the type of site (e.g., museum, attraction). To make it possible to add more than one site for each city, we will add the new data as a JSON array. This may seem rather confusing, but let's look at the example for Baltimore. The following shows the complete JSON document with the existing and new data. I've formatted it in a typical manner that you will see JSON document using indentation.

```
{
  "Population": 651154,
  "Places_of_interest": [
    {
      "name": "National Harbor",
      "URL": "https://www.nationalharbor.com/",
      "type": "attraction"
    }
  ]
}
```

At this point, you may be wondering how we can ensure the JSON document is formatted correctly. For most, this comes with experience working with JSON. However, MySQL provides a JSON function named `JSON_VALID()` that you can use to validate a JSON document. Just pass the string as a parameter. You should get a value of 1 for a valid document or 0 for a document with errors as demonstrated in the following. Here, we see the document is valid.

```
SQL > SELECT JSON_VALID('{ "Population": 651154, "Places_of_interest":
[{"name": " National Harbor ", "URL": " https://www.nationalharbor.com/", "ty
pe": "attraction"}]}' )\G
***** 1. ROW *****
JSON_VALID('{ "Population": 651154, "Places_of_interest": [{"name":
" National Harbor ", "URL": " https://www.nationalharbor.
com/", "type": "attraction"}]}' ): 1
1 row in set (0.0005 sec)
```

Now we're ready to update the data in the table. To do so, we will use the `UPDATE` SQL statement replacing the information column with new JSON document from above. However, we must reformat the document in a more traditional string. You may be able to leave the spaces and line breaks in the string, but it is not common to do that. Rather, we want to form a single string enclosed in single quotes. But first, we need the key for the row in the table. The city table has a key field named `ID`. Let's get the IDs for both cities.

```
SQL > SELECT ID, Name FROM world_x.city WHERE Name IN ('Washington',
'Baltimore');
```

ID	Name
3809	Baltimore
3813	Washington

```
2 rows in set (0.0053 sec)
```

Now, let's do the update. The following is typical of an SQL `UPDATE` command to replace a column for a specific row in the table. There's nothing unusual here, but notice we use the `\G` option of the SQL execution to show the result in an easier reading form.

```
SQL > UPDATE world_x.city set Info = '{"Population": 651154,"Places_of_
interest":[{"name":"National Harbor","URL":"https://www.nationalharbor.
com/","type":"attraction"}]}' WHERE ID = 3809;
Query OK, 1 row affected (0.0499 sec)
```

```
Rows matched: 1 Changed: 1 Warnings: 0
```

```
SQL > SELECT Name, District, Info FROM world_x.city WHERE ID = 3809\G
***** 1. ROW *****
      Name: Baltimore
      District: Maryland
      Info: {"Population": 651154, "Places_of_interest": [{"URL":
"https://www.nationalharbor.com/", "name": "National Harbor", "type":
"attraction"}]}
```

```
1 row in set (0.0008 sec)
```

Here, we see we have successfully updated (replaced) the JSON document. But the output isn't easy to read. To make it easier to read, let's use the `JSON_PRETTY()` function, which reformats the output in a more pleasing layout. The following shows the same query with the function added. Isn't that easier to read?

```
SQL > SELECT Name, District, JSON_PRETTY(Info) FROM world_x.city
WHERE ID = 3809\G
```

```
***** 1. ROW *****
      Name: Baltimore
      District: Maryland
      JSON_PRETTY(Info): {
        "Population": 651154,
        "Places_of_interest": [
          {
            "URL": "https://www.nationalharbor.com/",
            "name": "National Harbor",
            "type": "attraction"
          }
        ]
      }
1 row in set (0.0005 sec)
```

Now, let's see the other update done using a true update of the column. Here, we will use another special JSON function named `JSON_MERGE_PRESERVE()`, which merges two JSON documents and preserves the arrays (when merging two arrays). For this example, it effectively merges the existing JSON document with the new information we're adding.

Note There are several ways we could approach this update, but this is one that is common among savvy database administrators.

To effect this, let's first get the current JSON document from the row. We will use a local variable (starts with `@`) to store the result using the `SELECT...INTO` version of the `SELECT` statement as shown in the following. Here, we save the value to `@var1` and display it.

```
SQL > SELECT Info FROM world_x.city WHERE ID = 3813 INTO @var1;
Query OK, 1 row affected (0.0007 sec)
```

```
SQL > SELECT @var1;
+-----+
| @var1          |
+-----+
| {"Population": 572059} |
+-----+
1 row in set (0.0004 sec)
```

Now we can construct the new information we want to add. The following shows the JSON document with only the new data (population is already in the row). But don't worry, we're going to merge the documents.

```
{
  "Places_of_interest":[
    {
      "name":"Smithsonian National Air and Space Museum",
      "URL":"https://airandspace.si.edu/",
      "type":"museum"
    }
  ]
}
```

Next, we can use that JSON document along with the variable to update the row as shown in the following.

```
SQL > UPDATE world_x.city SET Info = JSON_MERGE_PRESERVE(@var1, '{"Places_of_interest":[{"name":"Smithsonian National Air and Space Museum",
"URL":"https://airandspace.si.edu/","type":"museum"}]}') WHERE ID = 3813;
Query OK, 1 row affected (0.0784 sec)
```

```
Rows matched: 1 Changed: 1 Warnings: 0
```

Notice, we now have the new information added and have retained the population. Cool.

```
SQL > SELECT Name, District, JSON_PRETTY(Info) FROM world_x.city WHERE ID = 3813\G
```

```
***** 1. ROW *****
```

```
      Name: Washington
```

```
      District: District of Columbia
```

```
JSON_PRETTY(Info): {
```

```
  "Population": 572059,
```

```
  "Places_of_interest": [
```

```
    {
```

```
      "URL": "https://airandspace.si.edu/",
```

```
      "name": "Smithsonian National Air and Space Museum",
```

```
      "type": "museum"
```

```
    }
```

```
  ]
```

```
}
```

```
1 row in set (0.0005 sec)
```

Now, what if we wanted to only retrieve the values for the `Places_of_interest` array? In this case, we can use another JSON function called `JSON_EXTRACT()` to extract keys and values from the array. The following demonstrates the technique. Notice the portion highlighted in bold. Here, we extract the key using the path expression like we saw earlier. And, we're doing the query on the entire table, so we'll get all rows that have the `Places_of_interest` key in the JSON document.


```
SQL > SELECT Name, District, JSON_EXTRACT(info, '$.Places_of_interest[*].
name') as Sights FROM world_x.city WHERE JSON_EXTRACT(info, '$.Places_of_
interest') IS NOT NULL \G
```

```
***** 1. ROW *****
```

```
    Name: Baltimore
```

```
District: Maryland
```

```
    Sights: ["National Harbor"]
```

```
***** 2. ROW *****
```

```
    Name: Washington
```

```
District: District of Columbia
```

```
    Sights: ["Smithsonian National Air and Space Museum"]
```

```
2 rows in set (0.0119 sec)
```

Ok, now that's a lot easier to read, isn't it? It's also a bit of a messy SQL command. And if all of that seemed a bit painful, you're right, it was. Working with JSON data in SQL works with the help of the JSON functions, but it is an extra step and can be a bit confusing in syntax. See the online MySQL reference manual for full explanations of each of the JSON functions.¹ We will see more about the JSON functions in the next chapter.

Using Formatting Modes

If you've used the old MySQL client much to query data with wide rows, chances are you've used the \G option like we did above to display the results in a vertical format, which makes reading the data easier. With the shell, we can display data in several ways. In this section, we see brief examples of running the shell in batch mode displaying data in a variety of formats. As you will see, choosing the format can help make the data easier to read (or ingest if reading the output in a script).

Recall, we can set the format using the command line options `--table`, `--tabbed`, `--vertical`, or `--json` with two choices for the JSON format. The output chosen will affect how the output looks in the execution modes (SQL, Python, or JavaScript). For this tutorial, we will see only the SQL mode.

¹<https://dev.mysql.com/doc/refman/8.0/en/json.html>

Let's begin with the table format. The following shows the results of executing the last query in batch mode with the table format. Notice we see output like we would when running interactively in SQL mode.

```
C:\Users\cbell>mysqlsh --uri=root@localhost:3306 --sql -e "SELECT Name,
District, JSON_EXTRACT(info, '$.Places_of_interest[*].name') as Sights FROM
world_x.city WHERE JSON_EXTRACT(info, '$.Places_of_interest') IS NOT NULL"
--table
Please provide the password for 'root@localhost:3306': *****
Save password for 'root@localhost:3306'? [Y]es/[N]o/[v]er (default No):
+-----+-----+-----+
| Name      | District                | Sights                |
+-----+-----+-----+
| Baltimore | Maryland                | ["National Harbor"]  |
| Washington | District of Columbia | ["Smithsonian National Air and
                        | Space Museum"]        |
+-----+-----+-----+
```

Next, let's see the tabbed format. The following shows the results of executing the last query in batch mode with the tabbed format. Notice in this case the output is a tabbed-separated view. It is not as easy to see in the listing, but when run in a script, it makes ingesting the output easy to separate with tabs.

```
C:\Users\cbell>mysqlsh --uri=root@localhost:3306 --sql -e "SELECT Name,
District, JSON_EXTRACT(info, '$.Places_of_interest[*].name') as Sights FROM
world_x.city WHERE JSON_EXTRACT(info, '$.Places_of_interest') IS NOT NULL"
--tabbed
Please provide the password for 'root@localhost:3306': *****
Save password for 'root@localhost:3306'? [Y]es/[N]o/[v]er (default No):
Name      District                Sights
Baltimore  Maryland                ["National Harbor"]
Washington District of Columbia    ["Smithsonian National Air and
Space Museum"]
```

Next, let's see the vertical format. The following shows the results of executing the last query in batch mode with the vertical format. Notice in this case the output is like using the \G option in the SQL mode.

```
C:\Users\cbell>mysqlsh --uri=root@localhost:3306 --sql -e "SELECT Name,
District, JSON_EXTRACT(info, '$.Places_of_interest[*].name') as Sights FROM
world_x.city WHERE JSON_EXTRACT(info, '$.Places_of_interest') IS NOT NULL"
--vertical
```

```
Please provide the password for 'root@localhost:3306': *****
Save password for 'root@localhost:3306'? [Y]es/[N]o/Ne[v]er (default No):
```

```
***** 1. ROW *****
```

```
    Name: Baltimore
District: Maryland
    Sights: ["National Harbor"]
```

```
***** 2. ROW *****
```

```
    Name: Washington
District: District of Columbia
    Sights: ["Smithsonian National Air and Space Museum"]
```

Next, let's see the raw JSON format. The following shows the results of executing the last query in batch mode with the raw JSON format. Notice in this case the output is very different from what we'd normally see in SQL mode. In fact, we see the output as a series of JSON documents albeit without formatting.

```
C:\Users\cbell>mysqlsh --uri=root@localhost:3306 --sql -e "SELECT Name,
District, JSON_EXTRACT(info, '$.Places_of_interest[*].name') as Sights FROM
world_x.city WHERE JSON_EXTRACT(info, '$.Places_of_interest') IS NOT NULL"
--json=raw
```

```
{"password":"Please provide the password for 'root@localhost:3306': "}
```

```
*****
```

```
{"prompt":"Save password for 'root@localhost:3306'? [Y]es/[N]o/Ne[v]er
(default No): "}
```

```
{"executionTime":"0.0063 sec","info":"","rows":[{"Name":"Baltimore","District":"Maryland","Sights":["National Harbor"]}, {"Name":"Washington","District":"District of Columbia","Sights":["Smithsonian National Air and Space Museum"]}], "warningCount":0, "warningsCount":0, "warnings":[], "hasData":true, "affectedRowCount":0, "affectedItemsCount":0, "autoIncrementValue":0}
```

Finally, let's look at the pretty JSON format. Up to this point, the output has been rather concise. However, the pretty JSON format is a bit more verbose. Listing 3-11 shows the results of executing the last query in batch mode with the pretty JSON format. In this case, it adds whitespace and new lines to make it easier to read.

Listing 3-11. The JSON pretty Format

```
C:\Users\cbell>mysqlsh --uri=root@localhost:3306 --sql -e "SELECT Name,
District, JSON_EXTRACT(info, '$.Places_of_interest[*].name') as Sights FROM
world_x.city WHERE JSON_EXTRACT(info, '$.Places_of_interest') IS NOT NULL"
--json=pretty
{
  "password": "Please provide the password for 'root@localhost:3306': "
}
*****
{
  "prompt": "Save password for 'root@localhost:3306'? [Y]es/[N]o/[e]x[er]t
(default No): "
}
{
  "executionTime": "0.0033 sec",
  "info": "",
  "rows": [
    {
      "Name": "Baltimore",
      "District": "Maryland",
      "Sights": "[\"National Harbor\"]"
    },
    {
      "Name": "Washington",
      "District": "District of Columbia",
      "Sights": "[\"Smithsonian National Air and Space Museum\"]"
    }
  ],
}
```

```

    "warningCount": 0,
    "warningsCount": 0,
    "warnings": [],
    "hasData": true,
    "affectedRowCount": 0,
    "affectedItemsCount": 0,
    "autoIncrementValue": 0
}

```

Notice the output is more verbose and even the messages from the shell are in JSON format, but it does make for reading JSON data much nicer.

Code/Command History

Like its predecessor, the shell allows you to recall the last commands entered through a command history list. Unlike the original client, the shell also allows you to search through the history. This is especially helpful when writing scripts as it allows you to search for operations you've used previously.

To search the command history, press *CTRL+R* at any point. This initiates a reverse search, which searches back through the commands. The prompt changes to indicate you are searching and you can type in the first few characters of the command you're searching for. When the search will display the first command found as shown in the following. If that isn't the command you're looking for, you can press *CTRL+R* again to get the next match or *CTRL+C* to cancel. If you press *ENTER*, the command found will be executed.

```
(reverse-i-search)`SHOW': SHOW TABLES FROM world_x;
```

The forward search is a little different and works when you are already in a search mode. That is, pressing *CTRL+S* will search forward through the history but if you're not in the search mode, you cannot search forward. However, it is very handy to use once you get the hang of it. Like the reverse search, the forward search changes the prompt as shown in the following.

```
(i-search)`SHOW': SHOW DATABASES;
```

You can also configure the number of items to store in the history (default is 100) as well as see the history in a file named `~/mysqlsh/history` (`%AppData%\MySQL\mysqlsh\history` on Windows). However, you must either use the `\history` command to save the history or have the shell configured to automatically save the history. We will see how to do this in a later section. You can see the history list at any time by using the `\history` command as shown in the following.

```
SQL > \history
1  \history
2  \connect root@localhost:3306
3  SHOW DATABASES;
4  SHOW TABLES FROM world_x;
5  SHOW VARIABLES LIKE '%ssh%';
6  SHOW VARIABLES LIKE '%ssl%';
```

The `\history` command also allows you to delete one or more entries in the history, clear the history (useful if you're switching modes), and save the history. The command options are demonstrated in the following.

- `\history del 2-4`: Deletes entries 2, 3, and 4 from history.
- `\history clear`: Clear history for this session.
- `\history save`: Save history to file.

Note Only those commands entered interactively are saved in the history. Batch execution does not save commands to the history file.

Saving Passwords

Now let's discuss one of the newest and most productive features of the shell – the secret store. This is made possible with a feature called the pluggable password store, which supports several storage mechanisms including the secret store, keychain, and more.

It allows you to securely store commonly used passwords, which makes working with MySQL Shell easier and more secure. You can save a password for a server connection using a secret store, such as a keychain. You enter the password for a connection interactively and it is stored with the server URL as credentials for the connection.

Tip See <https://dev.mysql.com/doc/mysql-shell/8.0/en/mysql-shell-pluggable-password-store.html> for more information about working with the pluggable password store.

In fact, by this point you've seen the message appear in the output of the shell numerous times and in all the examples I've replied "no" by simply pressing enter to this prompt, which tells the shell to not store the password.

```
Please provide the password for 'root@localhost:3306': *****
Save password for 'root@localhost:3306'? [Y]es/[N]o/[v]er (default No):
```

Now, let's remedy that and save that password once and for all on our system! Listing 3-12 shows a transcript of running the shell to execute a simple query. The first time, I tell the shell to remember the password. The second time, I no longer must remember the password and the shell doesn't prompt me for it.

Listing 3-12. Saving Passwords with the Secret Store

```
C:\Users\cbell>mysqlsh --uri=root@localhost:3306 --sql -e "SELECT Name,
District, JSON_EXTRACT(info, '$.Places_of_interest[*].name') as Sights FROM
world_x.city WHERE JSON_EXTRACT(info, '$.Places_of_interest') IS NOT NULL"
--vertical
Please provide the password for 'root@localhost:3306': *****
Save password for 'root@localhost:3306'? [Y]es/[N]o/[v]er (default No): Y
***** 1. IOW *****
    Name: Baltimore
    District: Maryland
    Sights: ["National Harbor"]
***** 2. IOW *****
    Name: Washington
    District: District of Columbia
    Sights: ["Smithsonian National Air and Space Museum"]

C:\Users\cbell>mysqlsh --uri=root@localhost:3306 --sql -e "SELECT Name,
District, JSON_EXTRACT(info, '$.Places_of_interest[*].name') as Sights FROM
world_x.city WHERE JSON_EXTRACT(info, '$.Places_of_interest') IS NOT NULL"
--vertical
```

```

***** 1. IOW *****
Name: Baltimore
District: Maryland
Sights: ["National Harbor"]
***** 2. IOW *****
Name: Washington
District: District of Columbia
Sights: ["Smithsonian National Air and Space Museum"]

```

If you want to reset the password should you forget it or change it, you can use a special object (global variable) that is a class that you can use to customize the pluggable password feature. Since customizing the shell is a larger topic, we will discuss how to customize the shell in the next section where we will also see how to reset passwords stored.

Customizing the Shell

The last tutorial we will cover is how to customize the shell. You can change several parameters governing how the shell operates. This includes changing the prompt as well as autocompletion, wait times, output format, and more. Let's begin with how it works.

There are three ways to set configuration options in the shell. You can either use the `\option` command to list, set, and even unset options and their values, or you can use the `shell.option` object in either Python or JavaScript mode, or you can modify the configuration file on disk. Let's see each of these in action.

Using the `\option` Command

The `\option` command works in any mode. The `\option` command is the more common method of setting options. Listing 3-13 shows the help text for the `\option` command along with a list of the options for configuring command history.

Listing 3-13. Using the `\option` Command

```

Py > \help \option
NAME
    \option - Allows working with the available shell options.

SYNTAX
    \option [args]

```


DESCRIPTION

The given [args] define the operation to be done by this command, the following values are accepted

- -h, --help [<filter>]: print help for the shell options matching filter.
- -l, --list [--show-origin]: list all the shell options.
- <shell_option>: print value of the shell option.
- <shell_option> [=] <value> sets the value for the shell option.
- --persist causes an option to be stored on the configuration file
- --unset resets an option value to the default value.

```
Py > \option --help history
```

```
history.autoSave          Shell's history autosave.
history.maxSize           Shell's history maximum size
history.sql.ignorePattern Shell's history ignore list.
```

```
Py > \option history.autoSave
```

```
false
```

```
Py > \option history.maxSize
```

```
1000
```

```
Py > \option history.sql.ignorePattern
```

```
*IDENTIFIED*:*PASSWORD*
```

Here, we see an example of listing the options available for the command history feature using the `\option \help` command. While this only shows the names of the options, we can use the `\option` command to see the value for each specific option. If you want to see all the options available and their values, use the `\option --list` command.

Tip If you set options without the `--persist` argument, the change is not saved when the shell is closed. You must use the argument to save the changes so that it persists (is saved) for later executions.

Recall from a previous section we can have the shell save history automatically, so we don't have to do it manually. As we saw in Listing 3-13, we can set the `history.autoSave` option to `true` as shown in the following. Notice here, we simply use an assignment argument (the equals sign) to set the value and then save (persist) the changes.

```
Py > \option --persist history.autoSave = true
```

Using the `shell.option` Object

We can also set options in one of the scripting modes by using the `shell.option` object and the `set()` method (to set the value for the current execution of the shell) or the `set_persist()` method to save the value permanently. To set an option, we use the name of the option category and option as a dotted string within quotes (single or double). For example, the following sets the automatic history save like we saw in the last section.

```
Py > shell.options.set_persist("history.autoSave", True)
```

The shell object has several methods you can use with options as shown in the following.

- `shell.options.set(<option_name>, <value>)`: sets the `<option_name>` to value for this session, the change is not saved.
- `shell.options.set_persist(<option_name>, <value>)`: sets the `<option_name>` to value for this session and saves the change to the configuration file.
- `shell.options.unset(<option_name>)`: resets the `<option_name>` to the default value for this session, the change is not saved to the configuration file.
- `shell.options.unset_persist(<option_name>)`: resets the `<option_name>` to the default value for this session and saves the change to the configuration file.

Listing 3-14 shows an example of using the shell object along with the autocompletion feature (initiated by pressing `TAB` twice) to unset the value changing it to the default and persisting it. Finally, we see how to return all options to their default values.

Listing 3-14. Using the shell.option Object

```

Py > shell.options.<TAB><TAB>
autocomplete.nameCache          devapi.dbObjectHandles          pager
batchContinueOnError            history.autoSave                passwordsFromStdin
credentialStore.excludeFilters  history.maxSize                 sandboxDir
credentialStore.helper          history.sql.ignorePattern       showWarnings
credentialStore.savePasswords   interactive                      useWizards
dba.gtidWaitTimeout            logLevel
defaultMode                    outputFormat
Py > shell.options.set("history.autoSave", True)
Py > print(shell.options["history.autoSave"])
True
Py > shell.options.unset_persist("history.autoSave")

```

Using the Configuration File

The options that are changed and persisted are stored in a configuration file in JSON format. Values are read at startup, and when you use the persist feature, settings are saved to the configuration file. As a result, you can also change options by adding them to the configuration file or if the option already exists, you can change them in the file and restart the shell to have them take effect. The following shows an example of what the configuration file looks like. All options and values are stored as key, value pairs in the same JSON document.

```

C:\Users\cbell\AppData\Roaming\MySQL\mysqlsh>more options.json
{
  "history.autoSave": "true"
}

```

The location of the configuration file is the user configuration path and the file is named `options.json`. On Windows, the file is found at `%APPDATA%\MySQL\mysqlsh\options.json` or on Linux, at `~/mysqlsh/options.json`.

The configuration file is created the first time you change an option. While you can edit this file to make changes to options, you must do so with great care. This is because the file is considered an internal file and is not intended to be changed by the user. If you make a mistake and set the wrong option (such as misspelling the name), the

shell may not start and throw an error. Thus, you should take care to set the option first interactively, then edit the file to change it. While this is still not considered “safe,” it is possible to change options via editing the file.

Caution Directly editing the `options.json` file is not recommended.

Working with Saved Passwords

One of the things that you can configure in the shell is the pluggable password authentication or, in more practical terms, credentials saved in the secret store. In this case, we may want to revoke certain passwords because we’ve changed them or perhaps we want to see which credentials are stored.

We can interact with this feature using one or more of the functions in the shell object. For example, the following shows how to list the credentials stored. This will return a list (or, if the variable is missing, print the list returned) of all the credentials stored. In this case, only one credential is stored. Notice no passwords are printed.

```
Py > shell.list_credentials()
[
  "root@localhost:3306"
]
```

The following functions are those that allow you to work with the Pluggable Password store. You can list the available Secret Store Helpers, as well as list, store, and retrieve credentials. To use any of these, you must execute them in one of the scripting modes (Python or JavaScript). However, it is worth noting that due to the difference in naming conventions, method names differ slightly between the Python and JavaScript modes. For example, JavaScript names follow a different pattern similar to camelCase where Python uses underscores in names. We will use Python examples in this book.

- `cred_list = list_credentials()`: return list of all credentials stored (no passwords!)
- `delete_credential(<URI>)`: delete a credential for a given URI
- `delete_all_credentials()`: delete all credentials currently stored

- `cred_helpers = list_credential_helpers()`: return a list of the credential helpers
- `store_credential(<URI>, [<password>])`: store a credential for a given URI optionally specifying the password (if not provided, password is prompted)

If you want to replace one of the credentials – say to change the password associated with it – you can use the `shell.store_credential()` method supplying the same URI as shown in the following. In this case, I left off the password parameter so the shell prompts for the password.

```
Py > shell.store_credential("root@localhost:3306")
Please provide the password for 'root@localhost:3306': *****
```

If we want to flush all credentials, we can use the `shell.delete_all_credentials()` method.

Changing the Prompt

Finally, the shell allows you to change the prompt for the interactive session. You may want to do this if you want to display a reminder or similar cue such as working with different databases or servers.

Changing the prompt requires editing a file on the system. This file is called a prompt theme file and can be specified using the `MYSQLSH_PROMPT_THEME` environment variable or saving a theme template file to a file named `prompt.json` in the `~/mysqlsh` folder on Linux or the `%AppData%\MySQL\mysqlsh` directory on Windows.

You can find the sample prompt theme files in the `share\mysqlsh\prompt` directory where the shell is installed. For example, in Windows, the files are stored in `c:\Program Files\MySQL\MySQL Shell 8.0`. The following shows a list of the prompt theme files.

```
10/04/2018 02:54 AM          1,245 prompt_16.json
10/04/2018 02:54 AM          2,137 prompt_256.json
10/04/2018 02:54 AM          1,622 prompt_256inv.json
10/04/2018 02:54 AM          2,179 prompt_256pl+aw.json
10/04/2018 02:54 AM          1,921 prompt_256pl.json
10/04/2018 02:54 AM           183 prompt_classic.json
10/04/2018 02:54 AM          2,172 prompt_dbl_256.json
```

```

10/04/2018 02:54 AM          2,250 prompt_dbl_256pl+aw.json
10/04/2018 02:54 AM          1,992 prompt_dbl_256pl.json
10/04/2018 02:54 AM          1,205 prompt_nocolor.json

```

Here, we see several files that are preformatted for a variety of common prompt customizations. There are ones for different color themes as well as what is displayed in the prompt. For example, the following shows the classic theme:

```

{
  "symbols" : {
    "separator" : "-",
    "separator2" : "-",
    "ellipsis" : "-"
  },
  "segments": [
    {
      "text": "mysql"
    },
    {
      "text": "%mode%"
    }
  ]
}

```

If you want to make your own changes, you can find the format documented in the `README.prompt` file located in the application installation under the `share\mysqlsh\prompt` directory. You can use a theme to specify a special font, terminal colors, and more.

However, take care when creating a prompt theme file because if an error is found in the prompt theme file, an error message is printed and a default prompt theme is used. I recommend spending some time reading the `README.prompt` file and study the examples before building your own prompt theme file. Also, remember that these files may specify settings that are platform dependent and may not apply universally.

Summary

The MySQL Shell represents a major leap forward in productivity for MySQL users. The shell is not only a better MySQL client, it is also a code editor and testing environment. In this chapter, we have taken a short tour of the shell and its major features including built-in commands, how to format output, and even how to customize the shell. Once again, we will apply what we learned thus far as we explore using the shell in a variety of tasks in the following chapters.

While we haven't learned all there is to know about the MySQL Shell in this chapter, we have learned quite a bit about how it works and how to become productive using the MySQL Shell. If you want to learn all the nuances that make up the MySQL Shell, please see the online users guide (<https://dev.mysql.com/doc/mysql-shell/8.0/en/>).

In the next chapter, we will take a guided tour of working with SQL databases. We will see a brief overview of using the SQL interface, but we will focus on working with relational databases using the X DevAPI. If you are well acquainted with SQL databases and using SQL commands, you may want to skim the SQL portions of the chapter and then work through the examples in Chapter 5 that demonstrate how to use MySQL Shell with SQL databases including how to work with the new X DevAPI for SQL databases.

CHAPTER 4

Using the Shell with SQL Databases

Most people who work with MySQL leverage the relational database capabilities using the Structured Query Language (SQL) interface to interact with their data. As we have seen, MySQL Shell is a very capable client that you can use to work with your data using Structured Query Language (SQL) statements. However, MySQL Shell is also a powerful scripting language editor and execution engine.

In this chapter, we will take a brief look at what SQL databases are and how to work with them in the shell, including a brief overview of using SQL. However, we won't spend a lot of time there since many are familiar with SQL. Whether you are new to MySQL and SQL in general or not, I suggest you read these sections so that you understand the access methods we will use later in the chapter.

While we will see some short examples, Chapter 5 includes a more detailed example that demonstrates how to use MySQL Shell with SQL databases including how to work with the new X DevAPI for SQL databases.

Let's begin with a brief overview of MySQL's SQL interface.

Revisiting Relational Databases

As you know, MySQL runs as a background process (a service in Windows). You can also run it as a foreground process if you launch it from the command line.

Like most database systems, MySQL supports SQL. You can use SQL to create databases and objects (using data definition language [DDL]), write or change data (using data manipulation language [DML]), and execute various commands for managing the server.

DDL statements are those we use to create the storage mechanisms (the objects such as tables) in the database – including the database itself. DML statements on the other hand are those designed to store and retrieve data (rows). There are additional, utilitarian SQL commands supported by MySQL such as those that display system status, variables, and similar metadata. Listing 4-1 shows an example of each form (DDL and DML) as well as a few utility SQL commands.¹

Listing 4-1. Example DDL and DML Statements

```
C:\Users\cbell>mysqlsh --sql --uri root@localhost:3306
...
SQL > CREATE DATABASE test_db;
Query OK, 1 row affected (0.0586 sec)

SQL > USE test_db;
Query OK, 0 rows affected (0.0003 sec)

SQL > CREATE TABLE test_tbl (id int auto_increment, name char(20), primary
key(id));
Query OK, 0 rows affected (0.0356 sec)

SQL > INSERT INTO test_tbl VALUES (NULL, 'one');
Query OK, 1 row affected (0.1117 sec)

SQL > INSERT INTO test_tbl VALUES (NULL, 'two');
Query OK, 1 row affected (0.0078 sec)

SQL > INSERT INTO test_tbl VALUES (NULL, 'three');
Query OK, 1 row affected (0.0109 sec)

SQL > SELECT * FROM test_tbl;
+----+-----+
| id | name |
+----+-----+
|  1 | one  |
|  3 | three|
```

¹Not all SQL commands in the MySQL command list are true, standard SQL commands. Many of the utility commands are nonstandard SQL commands. Thus, if you work with other database systems, the commands may seem similar but differ slightly.

```
| 4 | one |
| 5 | two |
| 6 | three |
```

```
+-----+-----+
```

5 rows in set (0.0011 sec)

```
SQL > DELETE FROM test_tbl WHERE id = 2;
```

```
Query OK, 0 rows affected (0.0005 sec)
```

```
SQL > SELECT * FROM test_tbl;
```

```
+-----+-----+
```

```
| id | name |
```

```
+-----+-----+
```

```
| 1 | one |
```

```
| 3 | three |
```

```
| 4 | one |
```

```
| 5 | two |
```

```
| 6 | three |
```

```
+-----+-----+
```

5 rows in set (0.0005 sec)

```
SQL > SHOW TABLES;
```

```
+-----+
```

```
| Tables_in_test_db |
```

```
+-----+
```

```
| test_tbl |
```

```
+-----+
```

1 row in set (0.0015 sec)

```
SQL > SHOW DATABASES;
```

```
+-----+
```

```
| Database |
```

```
+-----+
```

```
| information_schema |
```

```
| mysql |
```

```
| performance_schema |
```

```
| sakila |
```

```
| sys          |
| test_db     |
| world       |
| world_x     |
+-----+
8 rows in set (0.0009 sec))
```

```
SQL > SELECT @@version;
```

```
+-----+
| @@version |
+-----+
| 8.0.16    |
+-----+
1 row in set (0.0004 sec)
```

```
SQL > DROP DATABASE test_db;
Query OK, 1 row affected (0.2659 sec)
```

Note You must terminate each SQL command with a semicolon (;) or \G.

This example demonstrates DDL statements in the form of the `CREATE DATABASE` and `CREATE TABLE` statements, DML in the form of the `INSERT`, `DELETE`, and `SELECT` statements, and a couple of utility statements including a simple administrative command to retrieve a global server variable (`@@version`).

Notice the creation of a database and a table to store the data, the addition of several rows in the table, deleting a row, and finally the retrieval of the data in the table. Notice how I used capital letters for SQL command keywords. This is a common practice and helps make the SQL commands easier to read and easier to find user-supplied options or data, which is in lower case.

A great many commands are available in MySQL. Fortunately, you need master only a few of the more common ones. The following are the commands you will use most often. The portions enclosed in `<>` indicate user-supplied components of the command, and `[...]` indicates that additional options are needed.

- `CREATE DATABASE <database_name>`: Creates a database
- `USE <database>`: Sets the default database (not an SQL command)

- `CREATE TABLE <table_name> [...]`: Creates a table or structure to store data
- `INSERT INTO <table_name> [...]`: Adds data to a table
- `UPDATE [...]`: Changes one or more values for a specific row
- `DELETE FROM <table_name> [...]`: Removes data from a table
- `SELECT [...]`: Retrieves data (rows) from the table
- `SHOW [...]`: Shows a list of the objects, system variables, and more

Although this list is only a short introduction and nothing like a complete syntax guide, there is an excellent online reference manual that explains every command (and much more) in detail. You should refer to the online reference manual whenever you have a question about anything in MySQL. You can find explanation and details for every SQL command supported by MySQL at <https://dev.mysql.com/doc/refman/8.0/en/sql-syntax.html>.

One of the more interesting commands shown allows you to see a list of objects. For example, you can see the databases with `SHOW DATABASES`, a list of tables (once you set the default database with the `USE` command) with `SHOW TABLES`, and even the permissions for users with `SHOW GRANTS`. I find myself using these commands quite frequently.

If you are thinking that there is a lot more to MySQL than a few simple commands, you are correct. Despite its ease of use and fast startup time, MySQL is a full-fledged relational database management system (RDBMS). There is much more to it than you've seen here. For more information about MySQL, including all the advanced features, see the online reference manual.

WHAT IS A RELATIONAL DATABASE MANAGEMENT SYSTEM?

An RDBMS is a data storage and retrieval service based on the Relational Model of Data as proposed by E. F. Codd in 1970. These systems are the standard storage mechanism for structured data. A great deal of research is devoted to refining the essential model proposed by Codd, as discussed by C. J. Date in *The Database Relational Model: A Retrospective Review and Analysis*. This evolution of theory and practice is best documented in Date's *The Third Manifesto*.

The relational model is an intuitive concept of a storage repository (database) that can be easily queried by using a mechanism called a query language to retrieve, update, and insert data. Many vendors have implemented the relational model because it has a sound systematic theory, a firm mathematical foundation, and a simple structure. The most commonly used query mechanism is SQL, which resembles natural language. Although SQL is not included in the relational model, it provides an integral part of the practical application of the relational model in RDBMSs.

Now that you know what MySQL is and have seen a terse example of SQL commands used for working with data, let's discover some of the more common concepts and operations needed to successfully deploy and use MySQL to store and retrieve your data.

Working with MySQL Commands and Functions

Learning and mastering a database system requires training, experience, and a good deal of perseverance. Chief among the knowledge needed to become proficient is how to use the common SQL commands and concepts. This section completes the primer on MySQL and SQL by introducing the most common commands and concepts.

Rather than regurgitate the reference manual, this section introduces the commands and concepts at a high level. If you decide to use any of the commands or concepts and need more information, please refer to the online reference manual for additional details, complete command syntax, and additional examples. But first, let's clarify some of the terms we use when working with MySQL using traditional relation databases (SQL).

Terminology

In MySQL, like other relational database systems, we store data in a fixed manner where we use *databases* to store data for a given task, job, application, domain, etc. and we use *tables* to store like data (data that has the same format). Inside a *table*, the data is represented as *rows* each having the same format (or schema).

If you have never worked with a database before, you can loosely associate a relational database table like a spreadsheet² where the columns are defined, and each row contains values for each of the columns. Thus, inserting data or retrieving data requires forming or viewing data as rows from the table.

Creating Users and Granting Access

To begin working with data, you need to know about two administrative operations before working with MySQL: creating user accounts and granting access to databases. MySQL can perform these with the `CREATE USER` and `GRANT` statements. To create a user, you issue a `CREATE USER` command followed by one or more `GRANT` commands. For example, the following shows the creation of a user named *jane* and grants the user access to the database named *store_inventory*:

```
CREATE USER 'jane'@'%' IDENTIFIED BY 'secret';
GRANT SELECT, INSERT, UPDATE ON store_inventory.* TO 'jane'@'%';
```

The first command creates the user named *jane*, but the name also has an `@` followed by another string. This second string is the host name of the machine with which the user is associated. That is, each user in MySQL has both a user name and a host name, in the form `user@host`, to uniquely identify them. That means the user and host `jane@10.0.1.16` and the user and host `jane@10.0.1.17` are not the same. However, the `%` symbol can be used as a wildcard to associate the user with any host. The `IDENTIFIED BY` clause sets the password for the user.

Caution It is always a good idea to create the user accounts for your application without full access to the MySQL system and reserve full access for administrators. Furthermore, you should avoid using the wildcard for the host so that you can restrict users to known machines (IP addresses), subnets, etc. This is so you can minimize any accidental changes and to prevent exploitation.

²While wildly inaccurate on several levels, the similarity for novice database users is valid.

Be careful about using the wildcard % for the host. Although it makes it easier to create a single user and let the user access the database server from any host, it also makes it much easier for someone bent on malice to access your server from anywhere (once they discover the password).

The second command allows access to databases. There are many privileges that you can give a user. The example shows the most likely set that you would want to give a user of a database: read (SELECT), add data (INSERT), and change data (UPDATE). See the online reference manual for more about security and account access privileges.³

The command also specifies a database and objects to which to grant the privilege. Thus, it is possible to give a user read (SELECT) privileges to some tables and write (INSERT, UPDATE) privileges to other tables. This example gives the user access to all objects (tables, views, etc.) in the *store_inventory* database.

Tip Newer versions of MySQL no longer permit you to create a user with a GRANT statement. You must explicitly create the user first.

Creating Databases and Tables

The most basic commands you will need to learn and master are the CREATE DATABASE and CREATE TABLE commands. Recall that database servers such as MySQL allow you to create any number of databases that you can add tables and store data in a logical manner.

Creating a Database

To create a database, use CREATE DATABASE followed by a name for the database. Once you issue the command, the shell does not “switch” context to that database (like some other clients). Rather, if you want to set the default database, you must use the USE <database> command. This is needed whenever you decide to omit using the database in the name qualifiers in the SQL commands.

³<https://dev.mysql.com/doc/refman/8.0/en/access-control.html>

For example, you can use the `SELECT` command to select rows from any table in any database by specifying the database and table in the form of `<database>.<table>`. Notice we separate the names with a period. Further, `SELECT * FROM db1.table1` will execute regardless of the default database set. You should get in the habit of always specifying the database in your commands. The following shows two commands to create and change the focus of the database:

```
CREATE DATABASE factory_sensors;
USE factory_sensors;
```

Creating a Table

To create a table, use the `CREATE TABLE` command. This command has many options allowing you to specify not only the columns and their data types but also additional options such as indexes, foreign keys, and so on. The following shows how to create a simple table for storing sensor data for an assembly line.

```
CREATE TABLE `factory_sensors`.`trailer_assembly` (
  `id` int auto_increment,
  `sensor_name` char(30) NOT NULL,
  `sensor_value` float DEFAULT NULL,
  `sensor_event` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
  CURRENT_TIMESTAMP,
  `sensor_units` char(15) DEFAULT NULL,
  PRIMARY KEY `sensor_id` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Notice here that I specified the table name (*trailer_assembly*) and four columns (*sensor_name*, *sensor_value*, *sensor_event*, and *sensor_units*). I used several data types. For *sensor_name*, I used a character field with a maximum of 30 characters, a floating-point data type for *sensor_value*, a timestamp value for *sensor_event*, and another character field for *sensor_units* of 15 characters.

I also added an index with an auto increment column to ensure we can store sensor values with the same name. That is, we may be sampling the same sensors several times over a period. An index can also be created using the `CREATE INDEX` command.

Note This fictional table was taken from a working concept where sensor data is placed in a table for a given period (say, a 24-hour period) and then moved to another system for analysis. Thus, the table is not designed to store sensor data for long periods of time.

Notice the `TIMESTAMP` column. A column with this data type is of particular use in sensor network or Internet of Things (IOT) solutions or any time you want to record the date and time of an event or action. For example, it is often helpful to know when a sensor value is read. By adding a `TIMESTAMP` column to the table, you do not need to calculate, read, or otherwise format a date and time at data collection.

Notice also that I specified that the `sensor_name` column be defined as a key, which creates an index. In this case, it is also the primary key. The `PRIMARY KEY` phrase tells the server to ensure there exists one and only one row in the table that matches the value of the column. You can specify several columns to be used in the primary key by repeating the keyword. Note that all primary key columns must not permit nulls (`NOT NULL`).

Note This example is a high-level concept of a typical sensor network in a factory setting and is representative for tutorial purposes.

If you cannot determine a set of columns that uniquely identify a row (and you want such a behavior – some favor tables without this restriction, but a good database administrator (DBA) would not), you can use an artificial data type option for integer fields called `AUTO INCREMENT`. When used on a column (must be the first column), the server automatically increases this value for each row inserted. In this way, it creates a default primary key. For more information about auto increment columns, see the online reference manual.

However, best practices suggest using a primary key on a character field is suboptimal in some situations such as tables with large values for each column or many unique values. This can make searching and indexing slower. In this case, you could use an auto increment field to artificially add a primary key that is smaller in size (but somewhat more cryptic).

There are far more data types available than those shown in the previous example. You should review the online reference manual for a complete list of data types. See the section “Data Types.” If you want to know the layout or “schema” of a table, use the `SHOW CREATE TABLE` command as demonstrated in the following.

```
SQL > SHOW CREATE TABLE factory_sensors.trailer_assembly \G
***** 1. ROW *****
      Table: trailer_assembly
Create Table: CREATE TABLE `trailer_assembly` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `sensor_name` char(30) NOT NULL,
  `sensor_value` float DEFAULT NULL,
  `sensor_event` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  `sensor_units` char(15) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=latin1
1 row in set (0.0009 sec)
```

Like databases, you can also get a list of all the tables in the database with the `SHOW TABLES` command.

Storing Data

Now that you have a database and tables created, you will want to load or insert data into the tables. You can do so using the `INSERT INTO` statement. Here, we specify the table and the data for the row. The following shows a simple example:

```
INSERT INTO factory_sensors.trailer_assembly (sensor_name, sensor_value,
sensor_units) VALUES ('paint_vat_temp', 32.815, 'Celsius');
```

In this example, I am inserting data manually for one of the sensors in the trailer assembly line. What about the other columns, you wonder? In this case, the other columns include a timestamp column, which will be filled in by the database server. All other columns (just the one) will be set to `NULL`, which means no value is available, the value is missing, the value is not zero, or the value is empty. For auto increment and

timestamp columns, NULL triggers their behavior such as setting the value to the next unique integer or capturing the current date and time. The following shows an example of inserting this one row in the table.

```
SQL > SELECT * FROM factory_sensors.trailer_assembly \G
***** 1. ROW *****
sensor_name: paint_vat_temp
sensor_value: 32.815
sensor_event: 2019-02-01 14:59:35
sensor_units: Celsius
1 row in set (0.0005 sec)
```

Notice I specified the columns before the data for the row. This is necessary whenever you want to insert data for fewer columns than what the table contains. More specifically, leaving the column list off means you must supply data (or NULL) for all columns in the table. Also, the order of the columns listed can be different from the order they are defined in the table. Leaving the column list off will result in the ordering of the column data based on how they appear in the table.

You can also insert several rows using the same command by using a comma-separated list of the row values, as shown here:

```
INSERT INTO factory_sensors.trailer_assembly (sensor_name, sensor_value,
sensor_units) VALUES ('tongue_height_variance', 1.52, 'mm'), ('ambient_
temperature', 24.5, 'Celsius'), ('gross_weight', 1241.01, 'pounds');
```

Here I've inserted several rows with the same command. Note that this is just a shorthand mechanism and, except for automatic commits, no different than issuing separate commands.

Updating Data

There are times when you want to change or update data. You may have a case where you need to change the value of one or more columns, replace the values for several rows, or correct formatting or even the scale of numerical data. To update data, we use the UPDATE command.

You can update a column, update a set of columns, perform calculations on one or more columns, and more. The example used in this section – a factory sensor network – isn't likely to require changing data (IOT is all about storing data as it was recorded storing it only for as long as it is relevant), but sometimes in the case of mistakes in sensor-reading code or similar data entry problems, it may be necessary.

What may be more likely is you or your users will want to rename an object in your database. For example, suppose we determine the plant on the deck is not actually a fern but was an exotic flowering plant. In this case, we want to change all rows that have a plant name of *gross_weight* to *trailer_weight*. The following command performs the change. Notice the key operator here is the SET operator. This tells the database to assign a new value to the column(s) specified. You can list more than one set operation in the command.

```
UPDATE factory_sensors.trailer_assembly SET sensor_name = 'trailer_weight'
WHERE sensor_name = 'gross_weight';
```

Notice also I used a WHERE clause here to restrict the UPDATE to a set of rows. This is the same WHERE clause as you saw in the SELECT statement, and it does the same thing; it allows you to specify conditions that restrict the rows affected. If you do not use the WHERE clause, the updates will apply to all rows.

Deleting Data

Sometimes you end up with data in a table that needs to be removed. Maybe you used test data and want to get rid of the fake rows, or perhaps you want to compact or purge your tables or want to eliminate rows that no longer apply. To remove rows, use the DELETE FROM command.

Let's look at an example. Suppose you have a plant-monitoring solution under development, and you've discovered that one of your sensors or sensor nodes is reading values that are too low, because of a coding, wiring, or calibration error. In this case, we want to remove all rows with a sensor value less than 0.001 (presumably spurious data). The following command does this:

```
DELETE FROM factory_sensors.trailer_assembly WHERE sensor_value < 0.001;
```

You should take care when forming WHERE clauses. I like to use the WHERE clause with a SELECT to make sure I am acting on the rows I want. Using the SELECT to test the potential affected rows in this manner makes it much safer than simply issuing

the command blindly. For example, I would issue the following first to check that I am about to delete the rows I want and only those rows. Notice it is the same WHERE clause.

```
SELECT * FROM factory_sensors.trailer_assembly WHERE sensor_value < 0.001;
```

Caution Issuing an UPDATE or DELETE command without a WHERE clause will affect all rows in the table!

Selecting Data (Results)

The most used basic command you need to know is the command to return the data from the table (also called a result set or rows). To do this, you use the SELECT statement. This SQL statement is the workhorse for a database system. All queries for data will be executed with this command. As such, we will spend a bit more time looking at the various clauses (parts) that can be used, starting with the column list.

The SELECT statement allows you to specify which columns you want to choose from the data. The list appears as the first part of the statement. The second part is the FROM clause, which specifies the table(s) you want to retrieve rows from. The FROM clause can also permit you to combine data from two or more tables. This is called a join and uses the JOIN operator to link the tables. You will see a simple example of a join in a later section.

The order that you specify the columns determines the order shown in the result set. If you want all the columns, use an asterisk (*) instead. Listing 4-2 demonstrates three statements that generate the same result sets. That is, the same rows will be displayed in the output of each. In fact, I am using a table with only four rows for simplicity.

Listing 4-2. Example SELECT Statements

```
SQL > SELECT sensor_name FROM factory_sensors.trailer_assembly;
```

```
+-----+
| sensor_name          |
+-----+
| ambient_temperature  |
| paint_vat_temp       |
```

```
| tongue_height_variance |
| trailer_weight         |
+-----+
4 rows in set (0.0006 sec)
```

```
SQL > SELECT sensor_name, sensor_value, sensor_event, sensor_units FROM
factory_sensors.trailer_assembly \G
```

```
***** 1. ROW *****
```

```
sensor_name: ambient_temperature
sensor_value: 24.5
sensor_event: 2019-02-01 15:04:08
sensor_units: Celsius
```

```
***** 2. ROW *****
```

```
sensor_name: paint_vat_temp
sensor_value: 32.815
sensor_event: 2019-02-01 14:59:35
sensor_units: Celsius
```

```
***** 3. ROW *****
```

```
sensor_name: tongue_height_variance
sensor_value: 1.52
sensor_event: 2019-02-01 15:04:08
sensor_units: mm
```

```
***** 4. ROW *****
```

```
sensor_name: trailer_weight
sensor_value: 1241.01
sensor_event: 2019-02-01 15:06:17
sensor_units: pounds
```

```
4 rows in set (0.0004 sec)
```

```
SQL > SELECT * FROM factory_sensors.trailer_assembly \G
```

```
***** 1. ROW *****
```

```
sensor_name: ambient_temperature
sensor_value: 24.5
sensor_event: 2019-02-01 15:04:08
sensor_units: Celsius
```

CHAPTER 4 USING THE SHELL WITH SQL DATABASES

***** 2. ROW *****

sensor_name: paint_vat_temp
sensor_value: 32.815
sensor_event: 2019-02-01 14:59:35
sensor_units: Celsius

***** 3. ROW *****

sensor_name: tongue_height_variance
sensor_value: 1.52
sensor_event: 2019-02-01 15:04:08
sensor_units: mm

***** 4. ROW *****

sensor_name: trailer_weight
sensor_value: 1241.01
sensor_event: 2019-02-01 15:06:17
sensor_units: pounds

4 rows in set (0.0005 sec)

SQL > SELECT sensor_value, sensor_name, sensor_units FROM factory_sensors.
trailer_assembly;

sensor_value	sensor_name	sensor_units
24.5	ambient_temperature	Celsius
32.815	paint_vat_temp	Celsius
1.52	tongue_height_variance	mm
1242.00	trailer_weight	pounds

4 rows in set (0.0005 sec)

Notice that the first statement lists the sensor names in the table. The next two statements result in the same rows as well as the same columns in the same order, but the third statement, while it generates the same rows minus the sensor event, displays the columns in a different order.

You can also use functions in the column list to perform calculations and similar operations. One special example is using the `COUNT()` function to determine the number of rows in the result set, as shown here. Notice we pass in the wildcard (`*`) to count all rows. See the online reference manual for more examples of functions supplied by MySQL.⁴

```
SELECT COUNT(*) FROM factory_sensors.trailer_assembly;
```

The next clause in the `SELECT` statement is the `WHERE` clause. Like we saw with updating and deleting rows, this is where you specify the conditions you want to use to restrict the number of rows in the result set. That is, only those rows that match the conditions. The conditions are based on the columns and can be quite complex. That is, you can specify conditions based on calculations, results from a join, and more. But most conditions will be simple equalities or inequalities on one or more columns in order to answer a question. For example, suppose you wanted to see the plants where the sensor value read is less than 10.00. In this case, we issue the following query and receive the results. Notice I specified only two columns: the sensor name and the value read from sensor.

```
SQL > SELECT sensor_name, sensor_value FROM factory_sensors.trailer_
assembly WHERE sensor_value < 10.00;
```

```
+-----+-----+
| sensor_name          | sensor_value |
+-----+-----+
| tongue_height_variance |          1.52 |
+-----+-----+
1 row in set (0.0008 sec)
```

There are additional clauses you can use including the `GROUP BY` clause, which is used for grouping rows for aggregation or counting, and the `ORDER BY` clause, which is used to order the result set. Let's take a quick look at each starting with aggregation.

⁴<https://dev.mysql.com/doc/refman/8.0/en/functions.html>

Suppose you wanted to average the sensor values read in the table for each sensor. In this case, we have a table that contains sensor readings over time for a variety of sensors. While the example contains only four rows (and thus may not be statistically informative), the example demonstrates the concept of aggregation quite plainly, as shown in Listing 4-3. Notice what we receive is simply the average of the four sensor values read.

Listing 4-3. GROUP BY Example

```
SQL > SELECT sensor_name, sensor_value FROM factory_sensors.trailer_
assembly WHERE sensor_name = 'gross_weight';
```

```
+-----+-----+
| sensor_name | sensor_value |
+-----+-----+
| gross_weight |          1250 |
| gross_weight |          1235 |
| gross_weight |          1266 |
| gross_weight |          1242 |
+-----+-----+
```

4 rows in set (0.0040 sec)

```
SQL > SELECT sensor_name, AVG(sensor_value) as avg_value FROM factory_sensors.
trailer_assembly WHERE sensor_name = 'gross_weight' GROUP BY sensor_name;
```

```
+-----+-----+
| sensor_name | avg_value |
+-----+-----+
| gross_weight |    1248.25 |
+-----+-----+
```

1 row in set (0.0006 sec)

```
SQL > SELECT sensor_name, sensor_value FROM factory_sensors.trailer_
assembly WHERE sensor_name = 'gross_weight' ORDER BY sensor_value ASC;
```

```
+-----+-----+
| sensor_name | sensor_value |
+-----+-----+
| gross_weight |          1235 |
| gross_weight |          1242 |
+-----+-----+
```

```
| gross_weight |          1250 |
| gross_weight |          1266 |
+-----+-----+
4 rows in set (0.0007 sec)
```

```
SQL > SELECT sensor_name, sensor_value FROM factory_sensors.trailer_
assembly WHERE sensor_name = 'gross_weight' ORDER BY sensor_value DESC;
+-----+-----+
| sensor_name | sensor_value |
+-----+-----+
| gross_weight |          1266 |
| gross_weight |          1250 |
| gross_weight |          1242 |
| gross_weight |          1235 |
+-----+-----+
4 rows in set (0.0009 sec)
```

Notice in the second example, I specified the average function, `AVG()`, in the column list and passed in the name of the column I wanted to average. There are many such functions available in MySQL to perform some powerful calculations. Clearly, this is another example of how much power exists in the database server that would require many more resources on a client computer (not to mention for large data sets, it means transporting the data to the client before the operation).

Notice also that I renamed the column with the average with the `AS` keyword. You can use this to rename any column specified, which changes the name in the result set, as you can see in the listing.

The last two examples show how we can see the results of our result set ordered by sensor value. We order the rows by sensor value in ascending and descending order using the `ORDER BY` clause. If you combine this with the `LIMIT` clause, you can see the largest (max) and smallest (min) values as shown in the following. But it is more preferred to use the `MIN()` and `MAX()` functions – see <https://dev.mysql.com/doc/refman/8.0/en/function-summary-ref.html> for a complete list of functions available in MySQL.

```
SQL > SELECT sensor_value AS min FROM factory_sensors.trailer_assembly
WHERE sensor_name = 'gross_weight' ORDER BY sensor_value ASC LIMIT 1;
+-----+
| min |
```

```
+-----+
| 1235 |
+-----+
1 row in set (0.0008 sec)
SQL > SELECT sensor_value as max FROM factory_sensors.trailer_assembly
WHERE sensor_name = 'gross_weight' ORDER BY sensor_value DESC LIMIT 1;
+-----+
| max |
+-----+
| 1266 |
+-----+
1 row in set (0.0005 sec)
```

Another use of the GROUP BY clause is counting. In this case, we replaced AVG() with COUNT() and received the number of rows matching the WHERE clause. More specifically, we want to know how many sensor values were stored for each sensor.

```
SQL > SELECT sensor_name, COUNT(sensor_value) as num_values FROM factory_
sensors.trailer_assembly GROUP BY sensor_name;
+-----+-----+
| sensor_name          | num_values |
+-----+-----+
| paint_vat_temp      |          1 |
| tongue_height_variance |          1 |
| ambient_temperature |          1 |
| trailer_weight      |          1 |
| gross_weight        |          4 |
+-----+-----+
5 rows in set (0.0008 sec)
```

As I mentioned, there is a lot more to the SELECT statement than shown here, but what we have seen here will get you very far, especially when working with data typical of most small to medium-sized solutions.

Creating Indexes

Tables are created without the use of any ordering. That is, tables are unordered. While it is true MySQL will return the data in the same order each time, there is no implied (or reliable) ordering unless you create an index.⁵ The ordering I am referring to here is not like you think when sorting (that's possible with the `ORDER BY` clause in the `SELECT` statement).

Rather, indexes are mappings that the server uses to read the data when queries are executed. For example, if you had no index on a table and wanted to select all rows with a value greater than a certain value for a column, the server will have to read all rows to find all the matches. However, if we added an index on that column, the server would have to read only those rows that match the criteria.

To create an index, you can either specify the index in the `CREATE TABLE` statement or issue a `CREATE INDEX` command. We can use this command to add an index on the `sensor_name` column. Listing 4-4 shows the effects on the table structure (schema) before and after the index is added. Recall, we added the index on the primary key when we created the table earlier.

Listing 4-4. Adding Indexes

```
SQL > SHOW CREATE TABLE factory_sensors.trailer_assembly \G
***** 1. IOW *****
      Table: trailer_assembly
Create Table: CREATE TABLE `trailer_assembly` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `sensor_name` char(30) NOT NULL,
  `sensor_value` float DEFAULT NULL,
  `sensor_event` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
  `sensor_units` char(15) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=latin1
1 row in set (0.0005 sec)
```

⁵So, you should never expect the results to be in the same order without using an index! It is possible to demonstrate how the same data entered on one system can differ when entered on another, like system. There are many factors involved including character set, operating system, etc. that can cause the order to differ. If order is a concern, use an index.

```
SQL > CREATE INDEX sensor_name ON factory_sensors.trailer_assembly
(sensor_name);
```

```
Query OK, 0 rows affected (0.2367 sec)
```

```
Records: 0 Duplicates: 0 Warnings: 0
```

```
SQL > SHOW CREATE TABLE factory_sensors.trailer_assembly \G
```

```
***** 1. ROW *****
```

```
Table: trailer_assembly
```

```
Create Table: CREATE TABLE `trailer_assembly` (
```

```
  `id` int(11) NOT NULL AUTO_INCREMENT,
```

```
  `sensor_name` char(30) NOT NULL,
```

```
  `sensor_value` float DEFAULT NULL,
```

```
  `sensor_event` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
```

```
  `sensor_units` char(15) DEFAULT NULL,
```

```
  PRIMARY KEY (`id`),
```

```
  KEY `sensor_name` (`sensor_name`)
```

```
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=latin1
```

```
1 row in set (0.0009 sec)
```

Indexes created like this do not affect the uniqueness of the rows in the table, in other words, making sure there exists one and only one row that can be accessed by a specific value of a specific column (or columns). What I am referring to is the concept of a primary key (or primary index), which is a special option used in the creation of the table as described earlier.

You can remove indexes with the `DROP INDEX` command as shown here.

```
DROP INDEX sensor_name ON factory_sensors.trailer_assembly;
```

Creating Views

Views are logical mappings of results of one or more tables. They can be referenced as if they were tables in queries, making them a powerful tool for creating subsets of data to work with. You create a view with `CREATE VIEW` and give it a name like a table. The following shows a simple example where we create a test view to read values from a table. In this case, we limit the size of the view (number of rows), but you could use a

wide variety of conditions for your views, including combining data from different tables. Thus, views can be used in queries just like tables. They're a handy way of working with a subset of the data (when constructed correctly).

```
SQL > CREATE VIEW list_weights AS SELECT * FROM factory_sensors.trailer_
assembly WHERE sensor_units = 'pounds' LIMIT 3;
```

```
Query OK, 0 rows affected (0.0525 sec)
```

```
SQL > SELECT * FROM factory_sensors.list_weights;
```

```
+----+-----+-----+-----+-----+
| id | sensor_name | sensor_value | sensor_event | sensor_units |
+----+-----+-----+-----+-----+
| 4 | trailer_weight | 1241.01 | 2019-02-01 15:40:35 | pounds |
| 5 | gross_weight | 1250 | 2019-02-01 15:40:35 | pounds |
| 6 | gross_weight | 1235 | 2019-02-01 15:40:35 | pounds |
+----+-----+-----+-----+-----+
```

```
3 rows in set (0.0047 sec)
```

Views are not normally encountered in small or medium-sized database solutions, but I include them to make you aware of them in case you decide to do additional analysis and want to organize the data into smaller groups for easier reading.

Simple Joins

One of the most powerful concepts of database systems is the ability to make relationships (hence the name relational) among the data. That is, data in one table can reference data in another (or several tables). The most simplistic form of this is called a master-detail relationship where a row in one table references or is related to one or more rows in another.

A common (and classic) example of a master-detail relationship is from an order-tracking system where we have one table containing the data for an order and another table containing the line items for the order. Thus, we store the order information such as customer number and shipping information once and combine or “join” the tables when we retrieve the order proper.

Let’s look at an example from the sample database named `world_x`. You can find this database on the MySQL web site (<http://dev.mysql.com/doc/index-other.html>). Feel free to download it and any other sample database. They all demonstrate various designs of database systems. You will also find it handy to practice querying the data as it contains more than a few, simple rows.

Note If you want to run the following examples, you need to install the `world_x` sample database as described in Chapter 3.

Listing 4-5 shows an example of a simple join. There is a lot going on here, so take a moment to examine the parts of the `SELECT` statement, especially how I specified the `JOIN` clause. You can ignore the `LIMIT` option because that simply limits the number of rows in the result set.

Listing 4-5. Simple JOIN Example

```
SQL > SELECT Name, Code, Language FROM world_x.Country JOIN world_x.
CountryLanguage ON Country.Code = CountryLanguage.CountryCode LIMIT 10;
```

```
+-----+-----+-----+
| Name      | Code | Language |
+-----+-----+-----+
| Aruba     | ABW  | Dutch    |
| Aruba     | ABW  | English  |
| Aruba     | ABW  | Papiament|
| Aruba     | ABW  | Spanish  |
| Afghanistan | AFG  | Balochi  |
| Afghanistan | AFG  | Dari     |
| Afghanistan | AFG  | Pashto   |
| Afghanistan | AFG  | Turkmenian |
| Afghanistan | AFG  | Uzbek    |
| Angola    | AGO  | Ambo     |
+-----+-----+-----+
10 rows in set (0.0165 sec)
```

Caution If the file system for your system supports case-sensitive names, be sure to use naming consistently. For example, `world_x` and `World_X` are two different names on some platforms. See <https://dev.mysql.com/doc/refman/8.0/en/identifier-case-sensitivity.html> for more information about case-sensitive identifiers.

Here I used a JOIN clause that takes two tables specified such that the first table is joined to the second table using a specific column and its values (the ON specifies the match). What the database server does is read each row from the tables and returns only those rows where the value in the columns specified a match. Any rows in one table that are not in the other are not returned.

Notice also that I included only a few columns. In this case, I specified the country code and continent from the Country table and the language column from the CountryLanguage table. If the column names were not unique (the same column appears in each table), I would have to specify them by table name such as Country.Name. In fact, it is considered good practice to always qualify the columns in this manner.

There is one interesting anomaly in this example that I feel important to point out. In fact, some would consider it a design flaw. Notice in the JOIN clause I specified the table and column for each table. This is normal and correct but notice the column name does not match in both tables. While this really doesn't matter and creates only a bit of extra typing, some DBAs would consider this erroneous and would have a desire to make the common column name the same in both tables.

Another use for a join is to retrieve common, archival, or lookup data. For example, suppose you had a table that stored details about things that do not change (or rarely change) such as cities associated with ZIP codes or names associated with identification numbers (e.g., social security number (SSN)). You could store this information in a separate table and join the data on a common column (and values) whenever you needed. In this case, that common column can be used as a foreign key, which is another advanced concept.

Foreign keys are used to maintain data integrity (i.e., if you have data in one table that relates to another table, but the relationship needs to be consistent). For example, if you wanted to make sure when you delete the master row that all the detail rows are also

deleted, you could declare a foreign key in the master table to a column (or columns) to the detail table. See the online reference manual for more information about foreign keys.⁶

This discussion on joins touches only the very basics. Indeed, joins are arguably one of the most difficult and often confused areas in database systems. If you find you want to use joins to combine several tables or extend data so that data is provided from several tables (outer joins), you should spend some time with an in-depth study of database concepts such as Clare Churcher's book *Beginning Database Design* (Apress, 2012).

Additional Advanced Concepts

There are more concepts and commands available in MySQL, but two that may be of interest are PROCEDURE and FUNCTION, sometimes called routines. I introduce these concepts here so that if you want to explore them, you understand how they are used at a high level.

Suppose you need to run several commands to change data. That is, you need to do some complex changes based on calculations. For these types of operations, MySQL provides the concept of a stored procedure. The stored procedure allows you to execute a compound statement (a series of SQL commands) whenever the procedure is called. Stored procedures are sometimes considered an advanced technique used mainly for periodic maintenance, but they can be handy in even the more simplistic situations.

For example, suppose you want to develop your solution, but since you are developing it, you need to periodically start over and want to clear out all the data first. If you had only one table, a stored procedure would not help much, but suppose you have several tables spread over several databases (not unusual for larger solutions). In this case, a stored procedure may be helpful.

When entering commands with compound statements in the shell, you need to change the delimiter (the semicolon) temporarily so that the semicolon at the end of the line does not terminate the command entry. For example, use `DELIMITER //` before writing the command with a compound statement, use `//` to end the command, and change the delimiter back with `DELIMITER ;`.

⁶<https://dev.mysql.com/doc/refman/8.0/en/create-table-foreign-keys.html>

Suppose you want to execute a compound statement and return a result – you want to use it as a function. You can use functions to fill in data by performing calculations, data transformation, or simple translations. Functions therefore can be used to provide values to populate column values, provide aggregation, provide date operations, and more.

You have already seen a couple of functions (COUNT, AVG). These are considered built-in functions, and there is an entire section devoted to them in the online reference manual. However, you can also create your own functions. For example, you may want to create a function to perform some data normalization on your data. More specifically, suppose you have a sensor that produces a value in a specific range, but depending on that value and another value from a different sensor or lookup table, you want to add, subtract, average, and so on the value to correct it. You could write a function to do this and call it in a trigger to populate the value for a calculation column.

Since stored procedures can be quite complicated, if you decide to use them, read the “*CREATE PROCEDURE and CREATE FUNCTION Syntax*” section of the online reference manual before trying to develop your own.⁷ There is more to creating stored procedures than described in this section.

WHAT ABOUT CHANGING OBJECTS?

You may be wondering what you do when you need to modify a table, procedure, trigger, and so on. Rest easy, you do not have to start over from scratch! MySQL provides an ALTER command for each object. That is, there is an ALTER TABLE, ALTER PROCEDURE, and so on. See the online reference manual section entitled “*Data Definition Statements*” for more information about each ALTER command.⁸

Now that we’ve had a high-level view of working with SQL commands and MySQL to store and retrieve relational data, let’s see how we can use the X DevAPI to write Python code that works with the same relational data. We’re going to write some Python finally!

⁷<https://dev.mysql.com/doc/refman/8.0/en/create-procedure.html>

⁸<https://dev.mysql.com/doc/refman/8.0/en/sql-syntax-data-definition.html>

Managing Your Database with Python

Now that we've had a good introduction to SQL and MySQL commands, we can turn our attention to an exciting new method of working with SQL databases in MySQL – writing Python scripts to merge code and data using the X DevAPI.

Rather than embark on an arduous, perhaps tedious tour of all that the X DevAPI has to offer, we will explore the API beginning with a view of working with relational data. In this section, we will discover just enough about the X DevAPI to be able to write Python code to work with relational database objects (databases and tables). We leave the more complex look at the X DevAPI's support of JSON documents for Chapter 6.

However, we still need to know what the X DevAPI is and its major features. As mentioned, we will explore a few of these in this chapter and the next from the viewpoint of working with relational data. There is far more to the API than that but starting from a known (or at least familiar) ground will help those new to incorporating database support in applications. After all, the X DevAPI is all about making it easy to access your data from code!

The first thing you should know is the shell has built-in support for several library objects that we will need. For example, the `mysqlx` module listed earlier is one of the built-in modules in the shell. We refer to the built-in modules as global variables. The shell also includes the following libraries as built-in modules (sometimes called libraries or objects) as global variables.

- *session*: Represents the global session if one has been established.
- *db*: Represents a schema if one has been defined, for example, by a URI type string.
- *dba*: Represents the AdminAPI, a component of InnoDB cluster that enables you to administer clusters of server instances. See Chapter 10 for more information about InnoDB Cluster.
- *mysqlx*: Provides operations on session objects resulting from connection to a MySQL server.
- *shell*: Provides general purpose functions, for example, to configure MySQL Shell.
- *util*: Provides utility functions, for example, to check server instances before an upgrade.

The key concepts to understand when learning to use the X DevAPI for relational data include the following. These are represented (realized) as objects in your Python code. That is, we will be using one or more methods to create the object, then use one or more of its methods to execute our code to work with the data.

- *CRUD*: Create, Read, Update, and Delete – the basic operations on data
- *Database/Schema*: A container of one or more database-level objects such as tables, views, triggers, etc.
- *Result*: A set of zero or more rows from a read operation (SELECT) or other operations resulting in values returned from the server
- *Session*: A connection to a MySQL server including attributes governing the connection
- *Table*: A container for data formatted with a specific layout for storing data in predefined columns with data types

Most of these concepts should be familiar or at least familiar enough that learning to work with them won't require a lot of effort. For example, as SQL database users, we understand the basic concepts of databases (schemas), tables, and result sets. The objects that represent these concepts are nothing more than models of their behavior, which we can use to invoke as methods.

However, the three newest concepts are likely the `mysqlx` module, sessions, and CRUD operations. Let's look at each of those.

MySQL X Module

The `mysqlx` module is the entry point for writing your applications with the X DevAPI. You can think of this module as a library that contains several objects that we can use in our Python scripts. The most notable objects we will need are those classes and methods available for connecting to and working with relational data in MySQL, but there is more available for use with JSON documents.

The key concept to understand is that the objects are generated from methods called on other objects. More specifically, when we call method `x()` on object `a`, it returns and instance of object `b`. For example, we call the method and assign the returning object to

another variable like this `b = a.x()`. Once you understand this, you can check the return type of a method and then reference the object type returned to find out what methods it provides.

When working with the `mysqlx` module, it all begins with the connection that returns a `Session` object⁹ – the same returned from the `get_session()` method. From there, we can call the methods on the session, and they return different objects. Let’s see what classes are available in the `mysqlx` module. Table 4-1 shows the classes available in the module.

Table 4-1. *Classes in the mysqlx Module*

Class	Description
BaseResult	Base class for the different types of results returned by the server
Collection	A Collection is a container that may be used to store Documents in a MySQL database
CollectionAdd	Handler for document addition on a Collection
CollectionFind	Handler for document selection on a Collection
CollectionModify	Operation to update documents on a Collection
CollectionRemove	Operation to delete documents on a Collection
DocResult	Allows traversing the DbDoc objects returned by a Collection.find operation
LockContention	Constants to represent lock contention types
Result	Allows retrieving information about nonquery operations performed on the database
RowResult	Allows traversing the Row objects returned by a Table.select operation
Schema	Represents a Schema as retrieved from a session created using the X Protocol
Session	Enables interaction with a MySQL Server using the X Protocol
SqlExecute	Handler for execution SQL statements, supports parameter binding

(continued)

⁹An object in this case is an executable instance of a class.

Table 4-1. (continued)

Class	Description
SqlResult	Allows browsing through the result information after performing an operation on the database done through Session.sql
Table	Represents a Table on a Schema, retrieved with a session created using mysqlx module
TableDelete	Operation to delete data from a Table
TableInsert	Handler for Insert operations on Tables
TableSelect	Handler for record selection on a Table
TableUpdate	Handler for record update operations on a Table
Type	Constants to represent data types on Column objects

As you can see, there are several classes that are provided by the `mysqlx` module. Don't worry if this seems a bit overwhelming. We won't need all of these for working with relational data; most are designed for use with JSON documents (also called the document store). However, we will need to use the `Session` class.

Note The tables in this book are referencing the Python X DevAPI as implemented in/for MySQL Shell as documented in the online Doxygen documents (<https://dev.mysql.com/doc/dev/mysqlsh-api-python/8.0/>). The X DevAPI for other languages may differ slightly in organization as well as naming schemes for the classes and methods.

We have already discovered sessions in Chapter 3. Recall, we interacted with sessions in the shell using the `\connect` shell command, which allows you to make a connection in the shell interactive session. Using sessions in Python is a little different. Let's look at the `Session` class next.

Session Class

The `Session` class is the major class we will use when writing Python applications that interact with data. We use this module to pass connection information to the server in the form of a connection string or a language-specific construct (a dictionary in Python)

to pass the connection parameters either as a URI or a connection dictionary as the parameter (not both). The most commonly used method to get a session object is shown in the following.

```
get_session(<URI or connection dictionary>)
```

The following shows examples of getting a session object instance using a dictionary of connection options and getting a session object instance using a connection string (URI).

```
import mysqlx
mysqlx_session1 = mysqlx.get_session({'host': 'localhost', 'port': 33060,
'user': 'root', 'password': 'secret'})
mysqlx_session2 = mysqlx.get_session('root:secret@localhost:33060')
```

The resulting variable will point to an object instance should the connection succeed. If it fails, you could get an error or an uninitialized connection as the result. We'll see how to deal with this in the next chapter.

Once we have a session, we can begin working with our data by getting a Schema object.

Schema Class

The X DevAPI uses the term “schema” to refer to a set of collections, which are a collection of documents. However, when working with relational data, we use “database” to refer to a collection of tables and similar objects. One may be tempted to conclude “schema” is synonymous with “database,” and for older versions of MySQL, that is true. However, when working with the document store and the X DevAPI, you should use “schema” and when you refer to relational data, you should use “database.”

SCHEMA OR DATABASE: DOES IT MATTER?

Since MySQL 5.0.2, the two terms have been synonyms via the CREATE DATABASE and CREATE SCHEMA SQL commands. However, other database systems make a distinction. That is, some state a schema is a collection of tables and a database is a collection of schemas. Others state a schema is what defines the structure of data. If you use other database systems, be sure to check the definitions so that you use the terms correctly.

When starting work with data, the first thing you will need to do is either select (get) an existing schema, delete an existing schema, or create a new one. You may also want to list the schemas on the server. The `Session` class provides several methods for performing these operations, all of which return a `Schema` object. Table 4-2 lists the methods, parameters, and return values for the methods concerning schemas.

Table 4-2. *Session Class – Schema Methods*

Method	Returns	Description
<code>create_schema</code> (<code>str name</code>)	Schema	Creates a schema on the database and returns the corresponding object
<code>get_schema</code> (<code>str name</code>)	Schema	Retrieves a <code>Schema</code> object from the current session through its name
<code>get_default_schema</code> ()	Schema	Retrieves the <code>Schema</code> configured as default for the session
<code>get_current_schema</code> ()	Schema	Retrieves the active schema on the session
<code>set_current_schema</code> (<code>str name</code>)	Schema	Sets the current schema for this session and returns the schema object for it
<code>get_schemas</code> ()	List	Retrieves the <code>Schemas</code> available on the session
<code>drop_schema</code> (<code>str name</code>)	None	Drops the schema with the specified name

Let us now look at the transactional methods for performing ACID compliant transactions.

Transaction Methods

Transactions provide a mechanism that permits a set of operations to execute as a single atomic operation. For example, if a database were built for a banking institution, the macro operations of transferring money from one account to another would preferably be executed completely (money removed from one account and placed in another) without interruption.

Transactions permit these operations to be encased in an atomic operation that will back out any changes should an error occur before all operations are complete, thus avoiding data being removed from one table and never making it to the next table. A sample set of operations in the form of SQL statements encased in transactional commands is:

```
START TRANSACTION;
UPDATE SavingsAccount SET Balance = Balance - 100
WHERE AccountNum = 123;
UPDATE CheckingAccount SET Balance = Balance + 100
WHERE AccountNum = 345;
COMMIT;
```

MySQL's InnoDB storage engine (the default storage engine) supports ACID transactions that ensure data integrity with the ability to only commit (save) the resulting changes if all operations succeed or rollback (undo) the changes if any one of the operations fails.

The Session classes implement methods for transaction processing that mirror the SQL commands shown earlier. Table 4-3 lists the transaction methods.

Table 4-3. Transaction Methods

Method	Returns	Description
start_transaction()	None	Starts a transaction context on the server
commit()	None	Commits all the operations executed after a call to startTransaction()
rollback()	None	Discards all the operations executed after a call to startTransaction()
set_savepoint(str name="")	str	Creates or replaces a transaction savepoint with the given name
release_savepoint(str name)	None	Removes a savepoint defined on a transaction
rollback_to(str name)	None	Rolls back the transaction to the named savepoint without terminating the transaction

Notice the last three methods allow you to create a named transaction savepoint, which is an advanced form of transaction processing. See the server online reference manual for more information about savepoints and transactions.¹⁰

Now, let's look at the methods that concern the connection to the server.

Connection Methods

There are two methods for the underlining connection. One to check to see if the connection is open and another to close the connection. Table 4-4 shows the remaining utility methods available in the `Session` class.

Table 4-4. Connection Methods

Method	Returns	Description
<code>close()</code>	None	Closes the session
<code>is_open()</code>	Bool	Returns true if session is known to be open

Miscellaneous Methods

There are also several utility methods in the `Session` class. Table 4-5 lists the additional functions. See the X DevAPI online reference for more information about these methods.

Table 4-5. Miscellaneous Methods

Method	Returns	Description
<code>quote_name(str id)</code>	str	Escapes the identifier
<code>get_uri()</code>	str	Returns the URI for the session
<code>set_fetch_warnings(bool enable)</code>	None	Enables or disables warning generation
<code>sql(str sql)</code>	SqlStatement	Creates a <code>SqlStatement</code> object to allow running the received SQL statement on the target MySQL Server

¹⁰<https://dev.mysql.com/doc/refman/8.0/en/savepoint.html>

Notice the `sql()` method. We can use this method to issue SQL statements, but in general, we don't need to when working with data because there is a `Table` object. We will examine that class in more detail once we look at what CRUD operations are available for working with relational data.

CRUD Operations (Relational Data)

The X DevAPI implements a create, read, update, and delete (CRUD) model for working with the objects that are contained in a schema. A schema can contain any number of collections, documents, tables, views, and other relational data objects (such as triggers). In this section, we see an overview of the schema and tables classes. The CRUD model is implemented for all objects in the schema that can contain data for both document store and relational data.

As we will see in Chapter 6, the document store data CRUD operations use the verbs `add`, `find`, `modify`, and `remove`, whereas relational data uses terms that match the equivalent SQL command (`insert`, `select`, `update`, and `delete`). Table 4-6 provides a quick look at how the methods are named as well as a brief description of each. Note that we use the `Collection` class for document store data and the `Table` class for relational data.

Table 4-6. *CRUD Operations for Document Store and Relational Data*

CRUD operation	Description	Document store	Relational data
Create	Add a new item/object	<code>collection.add()</code>	<code>table.insert()</code>
Read	Retrieve/search for data	<code>collection.find()</code>	<code>table.select()</code>
Update	Modify data	<code>collection.modify()</code>	<code>table.update()</code>
Delete	Remove item/object	<code>collection.remove()</code>	<code>table.delete()</code>

We will see the methods specific to each class that we will need for working with relational data (`Schema` and `Table`) in the following sections. Let's begin with a look at the details of the `Schema` class.

Schema Class

The schema is a container for the objects that store your data. Recall this can be a collection for document store data or a table or view for relational data. Much like the old days working with relational data, you must select (or use) a schema for storing data in either a collection, table, or view.

While you can mix the use of document store data (collections) and relational data (tables, views), to keep things easy to remember, we will examine the Schema class methods as they pertain to working with relational data.

Table 4-7 shows the methods for working with both collections and tables. Once again, we will only be using the methods to work with tables in this and the next chapter but it doesn't hurt to get a glimpse of the document store methods. Notice the create and get methods return an instance of an object. For example, the `get_table()` method returns a Table object.

Table 4-7. *Schema Class Methods*

Method	Returns	Description
<code>get_tables()</code>	list	Returns a list of Tables for this Schema
<code>get_collections()</code>	list	Returns a list of Collections for this Schema
<code>get_table(str name)</code>	Table	Returns the Table of the given name for this schema
<code>get_collection(str name)</code>	Collection	Returns the Collection of the given name for this schema
<code>get_collection_as_table(str name)</code>	Table	Returns a Table object representing a Collection on the database
<code>create_collection(str name)</code>	Collection	Creates in the current schema a new collection with the specified name and retrieves an object representing the new collection created
<code>drop_collection(str name)</code>	None	Drops the specified collection

Now, let's look at the methods for the Table class.

Table Class

The table concept is the major organizational mechanism for relational data. In the X DevAPI, a table is the same relational data construct with which we are all familiar. The X DevAPI has a `Table` (you can use them with views too) class complete with CRUD operations (select, insert, update, and delete) as well as additional methods for counting the rows or whether the base object is a view. Table 4-8 shows the methods for the `Table` class.

Table 4-8. *Table Class*

Method	Returns	Description
<code>insert()</code>	<code>TableInsert</code>	Creates <code>TableInsert</code> object to insert new records into the table
<code>insert(list columns)</code>	<code>TableInsert</code>	Insert a row using a list of columns
<code>insert(str col1, str col2,...)</code>	<code>TableInsert</code>	Insert a row using a parameter list of columns
<code>select()</code>	<code>TableSelect</code>	Creates a <code>TableSelect</code> object to retrieve rows from the table
<code>select(list columns)</code>	<code>TableSelect</code>	Creates a <code>TableSelect</code> object to retrieve rows from the table
<code>update()</code>	<code>TableUpdate</code>	Creates a record update handler
<code>delete()</code>	<code>TableDelete</code>	Creates a record deletion handler
<code>is_view()</code>	<code>bool</code>	Indicates whether this <code>Table</code> object represents a View on the database
<code>count()</code>	<code>int</code>	Returns number of rows in the table
<code>get_name()</code>	<code>str</code>	Returns the name of the object
<code>get_session()</code>	<code>object</code>	Returns the <code>Session</code> object of this database object
<code>get_schema()</code>	<code>object</code>	Returns the <code>Schema</code> object of this database object
<code>exists_in_database()</code>	<code>bool</code>	Verifies if this object exists in the database

Notice there aren't methods for creating the table. We must use the `CREATE TABLE` SQL command to do this or the `sql()` method to execute the SQL statement. In fact, there are no methods to create any relational data objects. You must use SQL to issue the appropriate create statement to create the objects. For example, to create a table for our *factory_sensors* data in the previous example, we can use the following `CREATE TABLE` statement. While we saw this earlier, the following shows a Python code snippet where we declare a variable to hold the query and demonstrate executing the query using the `sql()` method.

```
...
CREATE_TBL = """
CREATE TABLE `factory_sensors`.`trailer_assembly` (
  `id` int auto_increment,
  `sensor_name` char(30) NOT NULL,
  `sensor_value` float DEFAULT NULL,
  `sensor_event` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
  CURRENT_TIMESTAMP,
  `sensor_units` char(15) DEFAULT NULL,
  PRIMARY KEY `sensor_id` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
"""

my_session = mysqlx.get_session(user_info)
my_db = my_session.create_schema('factory_sensors')
sql_res = my_session.sql(CREATE_TBL).execute()
my_tbl = my_db.get_table('trailer_assembly')
...
```

Tip There are no create methods to create tables or views. You must pass the SQL command to the `sql()` method to create these (and other relational data) objects.

Now that we've got the table created, we can insert data using the `Table` object. Recall, we have a set of objects in play here. We started with the `Session` object `create_schema()` method, which returned a `Database` object that we saved to a variable named `my_db`. After we created the table, we called the `my_db.get_table()` method to get the `Table` object.

Before we see an example of working with the data, let's look at the other classes and methods for working with relational data. Table 4-9 lists the methods for each of the classes related to the CRUD operations for relational data.

Table 4-9. *Classes for CRUD Operations for Relational Data*

Class	Method	Returns	Description
TableSelect			A statement for record retrieval operations on a Table
	<code>select (list searchExprStr)</code>	TableSelect	Defines the columns to be retrieved from the table
	<code>where (str expression)</code>	TableSelect	Sets the search condition to filter the records to be retrieved from the Table
	<code>group_by (list searchExprStr)</code>	TableSelect	Sets a grouping criteria for the retrieved rows
	<code>having (str condition)</code>	TableSelect	Sets a condition for records to be considered in aggregate function operations
	<code>order_by (list sortExprStr)</code>	TableSelect	Sets the order in which the records will be retrieved
	<code>limit (int numberOfRows)</code>	TableSelect	Sets the maximum number of rows to be returned on the select operation
	<code>offset (int numberOfRows)</code>	TableSelect	Sets number of rows to skip on the result set when a limit has been defined
	<code>bind (str name, Value value)</code>	TableSelect	Binds a value to a specific placeholder used on this operation
	<code>execute ()</code>	RowResult	Executes the select operation with all the configured options

(continued)

Table 4-9. *(continued)*

Class	Method	Returns	Description
TableInsert	A statement for insert operations on Table		
	<code>insert ()</code>	TableInsert	Initializes the record insertion handler
	<code>insert (list columns)</code>	TableInsert	Initializes the record insertion handler with the received column list
	<code>insert (str col1, str col2,...)</code>	TableInsert	Initializes the record insertion handler with the received column list
	<code>values (Value, Value value,...)</code>	TableInsert	Adds a new row to the insert operation with the given values
	<code>execute ()</code>	Result	Executes the insert operation
TableUpdate	A statement for record update operations on a Table		
	<code>update ()</code>	TableUpdate	Initializes the update operation
	<code>set (str attribute, Value value)</code>	TableUpdate	Adds an update operation
	<code>where (str expression)</code>	TableUpdate	Sets the search condition to filter the records to be updated
	<code>order_by (list sortExprStr)</code>	TableUpdate	Sets the order in which the records will be updated
	<code>limit (int numberOfRows)</code>	TableUpdate	Sets the maximum number of rows to be updated by the operation
	<code>bind (str name, Value value)</code>	TableUpdate	Binds a value to a specific placeholder used on this operation
	<code>execute ()</code>	Result	Executes the delete operation with all the configured options

(continued)

Table 4-9. (continued)

Class	Method	Returns	Description
TableDelete	A statement that drops a table		
	<code>delete ()</code>	TableDelete	Initializes this record deletion handler
	<code>where (str expression)</code>	TableDelete	Sets the search condition to filter the records to be deleted from the Table
	<code>order_by (list sortExprStr)</code>	TableDelete	Sets the order in which the records will be deleted
	<code>limit (int numberOfRows)</code>	TableDelete	Sets the maximum number of rows to be deleted by the operation
	<code>bind (str name, Value value)</code>	TableDelete	Binds a value to a specific placeholder used on this operation
	<code>execute ()</code>	Result	Executes the delete operation with all the configured options

Wow, there's a lot of methods! Notice there are some similarities among the statement classes. For example, most have methods for binding parameters, search conditions, and more. To understand this better, let's look at the syntax diagrams for the CRUD operations from the X DevAPI Users' Guide (<https://dev.mysql.com/doc/x-devapi-userguide/en/>).

The way we will use these classes and methods is a concept called method chaining where we can combine our class and method calls into a "chain" where we call the method for a returned object by using dot notation to extend the syntax. In other words, if `method_a()` returns an instance of an object that has a method named `count()`, we can chain it together like this: `method_a().count()` thereby avoiding the need to store an intermediate object.

WHAT IS METHOD CHAINING?

Method chaining (also known as named parameter idiom) is a design constraint in object-oriented programming where each method (that supports chaining) returns an instance of an object. Thus, one can access (call) any method on the returned object simply by adding the call to the end of the first method.

For example, if a class X has a method `a()` that returns object Y with a method `b()`, we can chain calls together as follows.

```
x = something.get_x()
res = x.a().b()
```

In this case, the `x.a()` method executes first, then when it returns with a Y object instance, it calls the `b()` method on the Y object instance.

For more information about the concepts of method chaining, see https://en.wikipedia.org/wiki/Method_chaining.

The following sections demonstrate simple examples of the CRUD operations for relational data. Recall, we will use the Table object (instance) we retrieved from the session to execute the CRUD operations. Let's look at an example of each.

Creating Data

The create operation uses a Table object method named `insert()`, which takes as parameters a list of columns. We can then use the `values()` method for the TableInsert object using method chaining (see later) passing as a parameter a list of values. This is because the `insert()` method returns an instance of the TableInsert class. For example, we added a row in the earlier section with the following INSERT query.

```
INSERT INTO factory_sensors.trailer_assembly (sensor_name, sensor_value,
sensor_units) VALUES ('paint_vat_temp', 32.815, 'Celsius');
```

To execute this in Python, we use the following statements. Notice we have used a variable to store the list of columns. We also used method chaining to call the `values()` and `execute()` methods to complete the row insert in a single statement.

```
...
COLUMNS = ['sensor_name', 'sensor_value', 'sensor_units']
my_tbl.insert(COLUMNS).values('paint_vat_temp', 32.815, 'Celsius').
execute()
...
```

Figure 4-1 shows the syntax diagram for the read operation. Here, the chain is rather small since we have only two intermediate objects (the “owner” of values() and execute()). If this seems a little strange, don’t feel bad. Coming from the world of SQL statements transitioning to advanced coding techniques can be a challenge, but with practice and more examples, chaining the methods like this will seem quite natural.

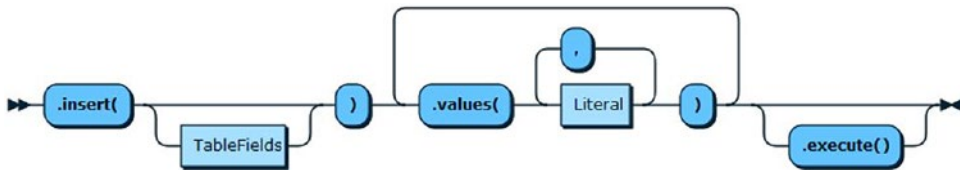


Figure 4-1. Syntax Diagram - Table.insert()

Reading Data

Reading data from the Table object is another case where we use method chaining to link the object methods together. If you consider the complexity of a typical SELECT statement, it should come as no surprise that the read operation can also be quite complex. However, we’ll keep it simple for this demonstration and look at more complex examples in the next chapter.

The following is the SELECT statement to get all the rows in the table.

```
SELECT sensor_value, sensor_name, sensor_units FROM factory_sensors.
trailer_assembly;
```

The same method chaining is true for the select() method, which returns a Table object where we chained the where() clause. In this simple example, we don’t have a WHERE clause, so we leave the parameters off. We still add the execute() method to run the query.

```

...
COLUMNS = ['sensor_name', 'sensor_value', 'sensor_units']
my_res = my_tbl.select(COLUMNS).execute()
...

```

Figure 4-2 shows the syntax diagram for the `select()` method of the `Table` class. Here, we see how the various classes and methods can be chained together with the `execute()` method at the end of the chain. The various methods depicted in the chain are optional, but most read operations will include the `where()` method (clause).

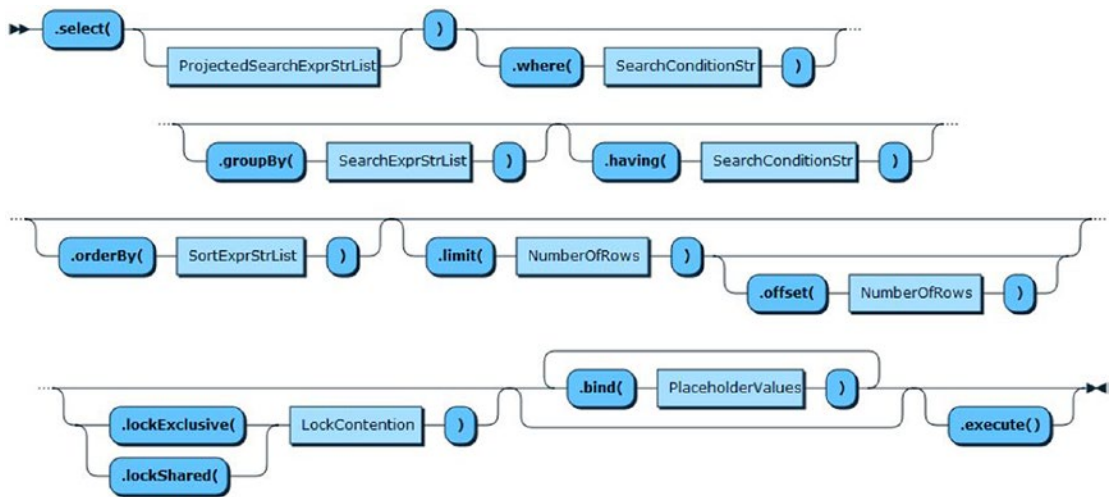


Figure 4-2. Syntax Diagram - `Table.select()`

Updating Data

The update operation is like the read operation where we use one or more of the methods to specify the subclauses we would find in a typical UPDATE SQL query. The following is a simple example where we update rows with a sensor value equal to 1.52 changing the sensor units. Notice we used a trick of MySQL's function library to convert our floating point value to a specific decimal to remove errors resulting from rounding (`sensor_units = 1.52` doesn't work).

```

UPDATE factory_sensors.trailer_assembly SET sensor_units = 'inches' WHERE
sensor_value LIKE 1.52;

```

To execute this in Python, we use the Table object’s update() method and chain it with the TableUpdate object’s set() and where() methods and passing in our parameters.

```
my_tbl.update().set().where('sensor_value LIKE 1.52').execute()
```

Figure 4-3 shows the syntax diagram for the update operation. This is like the read operation since we have several common intermediate steps such as the WHERE clause, order by, etc.

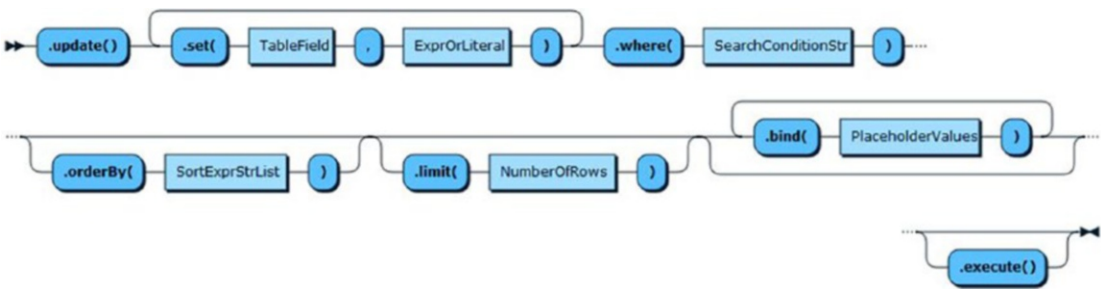


Figure 4-3. Syntax Diagram - Table.update()

Deleting Data

The delete operation is like the read and update operations where we use one or more of the methods to specify the subclauses we would find in a typical DELETE SQL query. The following is a simple example where we delete rows with a sensor value >30.

```
DELETE FROM factory_sensors.trailer_assembly WHERE sensor_value > 30;
```

To execute this in Python, we use the Table object’s delete() method and chain it with the TableDelete object’s where() method and pass in our parameters.

```
my_tbl.delete().where('sensor_value > 30').execute()
```

Figure 4-3 shows the syntax diagram for the update operation. This is like the update operation since we have several common intermediate steps such as the WHERE clause, order by, etc.

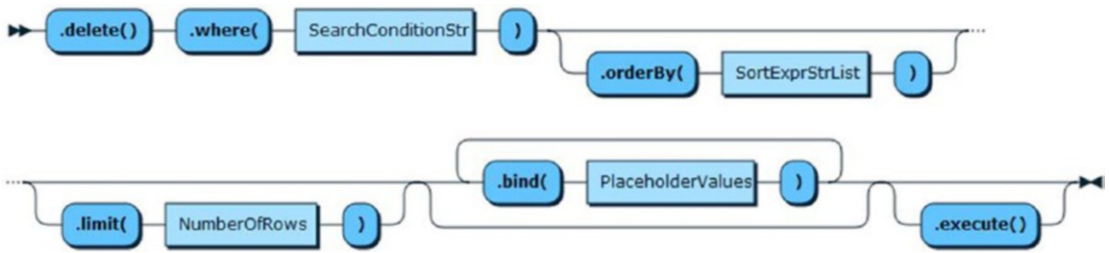


Figure 4-4. Syntax Diagram - `Table.delete()`

There is just one more thing we should explore before we see a complete Python example of working with relational data - working with result sets (sometimes called data sets).

Working with Results

Until now, we have seen a few simple examples of working with results and while it may appear all results are the same class, there are several result classes. The object instance for the Result class returned depends on the operation. For example, there is a separate class for each type of result. Results are sometimes called data sets or result sets.

Table 4-10 shows all the classes and their methods that you will encounter when working with data sets and results. All result classes are derived from the BaseResult object, which provides a set of properties and base methods. I've repeated these in the table for completeness.

Note Class methods are indicated with `()` and properties are indicated without `()`.

Table 4-10. *Classes and Methods for Working with Data Sets and Results*

Class	Method	Returns	Description
Result	Allows retrieving information about nonquery operations performed on the database		
	<code>get_affected_item_count()</code>	int	The number of affected items for the last operation
	<code>get_auto_increment_value()</code>	int	The last insert id autogenerated (from an insert operation)
	<code>get_generated_ids()</code>	list	Returns the list of document ids generated on the server
	<code>affected_item_count</code>	int	Same as <code>get_affected_itemCount()</code>
	<code>auto_increment_value</code>	int	Same as <code>get_auto_increment_value()</code>
	<code>generated_ids</code>	list	Same as <code>get_generated_ids()</code>
	<code>affected_items_count</code>	int	Same as <code>get_affected_items_count()</code>
	<code>warning_count</code>	int	Same as <code>get_warning_count()</code>
	<code>warnings_count</code>	int	Same as <code>get_warnings_count()</code>
	<code>warnings</code>	list	Same as <code>get_warnings()</code>
	<code>execution_time</code>	str	Same as <code>get_execution_time()</code>

(continued)

Table 4-10. *(continued)*

Class	Method	Returns	Description
RowResult	Allows traversing the Row objects returned by a Table.select operation		
	fetch_one()	Row	Retrieves the next Row on the RowResult
	fetch_all()	list	Returns a list of DbDoc objects, which contains an element for every unread document
	get_column_count()	int	Retrieves the number of columns on the current result
	get_column_names()	list	Gets the columns on the current result
	get_columns()	list	Gets the column metadata for the columns on the active result
	column_count	int	Same as get_column_count()
	column_names	list	Same as get_column_names()
	columns	list	Same as get_columns()
	affected_items_count	int	Same as get_affected_items_count()
	warning_count	int	Same as get_warning_count()
	warnings_count	int	Same as get_warnings_count()
	warnings	list	Same as get_warnings()
	execution_time	str	Same as get_execution_time()

(continued)

Table 4-10. *(continued)*

Class	Method	Returns	Description
SqlResult	Represents a result from a SQL statement		
	get_auto_increment_value()	int	Returns the identifier for the last record inserted
	get_affected_row_count()	int	Returns the number of rows affected by the executed query
	has_data()	bool	Returns true if the last statement execution has a result set
	next_data_set()	bool	Prepares the SqlResult to start reading data from the next Result (if many results were returned)
	next_result()	bool	Prepares the SqlResult to start reading data from the next Result (if many results were returned)
	auto_increment_value	int	Same as get_auto_increment_value()
	affected_row_count	int	Same as get_affected_row_count()
	column_count	int	Same as get_column_count()
	column_names	list	Same as get_column_names()
	columns	list	Same as get_columns()
	affected_items_count	int	Same as get_affected_items_count()
	warning_count	int	Same as get_warning_count()
	warnings_count	int	Same as get_warnings_count()
	warnings	list	Same as get_warnings()
	execution_time	str	Same as get_execution_time()

(continued)

Table 4-10. (continued)

Class	Method	Returns	Description
DocResult	Allows traversing the DbDoc objects returned by a Collection.find operation		
	fetch_one()	Document	Retrieves the next DbDoc on the DocResult
	fetch_all()	list	Returns a list of DbDoc objects, which contains an element for every unread document
	affected_items_count	int	Same as get_affected_items_count()
	warning_count	int	Same as get_warning_count()
	warnings_count	int	Same as get_warnings_count()
	warnings	list	Same as get_warnings()
	execution_time	str	Same as get_execution_time()

The three classes that have iterators implement two methods: `fetch_one()` and `fetch_all()`. They work like you would imagine and return either a data set or a set of objects for a set of documents. The `fetch_one()` method returns the next data item in the data set or NULL if there are no more data items and `fetch_all()` returns all the data items. More specifically, `fetch_one()` retrieves one data item at a time from the server whereas `fetch_all()` retrieves all the data from the server in one pass. Which you use will depend on the size of the data set and how you want to process the data.

So, what does this look like in Python? The following shows a simple example of performing a read operation to get all the rows in the table (there is no WHERE clause). Here, we first retrieve a list of the table columns so we can print them. We will use two loops: one to loop through the list of column names and another to loop through the rows returned from the read operation.

```
column_names = my_res.get_column_names()
column_count = my_res.get_column_count()
```

```

for i in range(0,column_count):
    if i < column_count - 1:
        print "{0}, ".format(column_names[i]),
    else:
        print "{0}".format(column_names[i]),
print

```

Here, we see a few Python statements for getting the results. In this case, we are working with the `TableSelect` class, but since most of the result classes have the same methods, your code will be similar for other results. You may notice some rudimentary formatting code to make the output comma-separated. This is only for demonstration. Your own applications would likely consume the data a row at a time and do something with it. However, the concepts of getting the columns and fetching rows are the same. Once you get those concepts down, we need only add the concept of how to get started.

Getting Started Writing Python Scripts

Now it's time to get our hands on some actual code that we can use to reinforce the concepts discussed thus far. Let's do so by looking at a simple example. At first, this code may seem a little intimidating, but it is a very simple example that contains the boilerplate code you will need to make a connection to the server by opening a session, then creating the new schema and creating the table. From there, we see examples of the CRUD operations starting with a `select()` call on the table object and a demonstration of working with the CRUD operations.

Listing 4-6 shows a Python script to create the database and table used previously including adding data and performing a simple select query. Only, this time we're doing it in Python! If you want to follow along, open the shell and connect to your server as shown.

Listing 4-6. Simple Relational Data Example

```

#
# Introducing the MySQL 8 Shell
#
# This example shows a simple X DevAPI script to work with relational data
#
# Dr. Charles A. Bell, 2019

```

```
from mysqlsh import mysqlx # needed in case you run the code outside of the
shell
```

```
# SQL CREATE TABLE statement
```

```
CREATE_TBL = """
```

```
CREATE TABLE `factory_sensors`.`trailer_assembly` (
```

```
  `id` int auto_increment,
```

```
  `sensor_name` char(30) NOT NULL,
```

```
  `sensor_value` float DEFAULT NULL,
```

```
  `sensor_event` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
```

```
  `sensor_units` char(15) DEFAULT NULL,
```

```
  PRIMARY KEY `sensor_id` (`id`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

```
"""
```

```
# column list, user data structure
```

```
COLUMNS = ['sensor_name', 'sensor_value', 'sensor_units']
```

```
user_info = {
```

```
  'host': 'localhost',
```

```
  'port': 33060,
```

```
  'user': 'root',
```

```
  'password': 'secret',
```

```
}
```

```
print("Listing 4-6 Example - Python X DevAPI Demo with Relational Data.")
```

```
# Get a session (connection)
```

```
my_session = mysqlx.get_session(user_info)
```

```
# Precautionary drop schema
```

```
my_session.drop_schema('factory_sensors')
```

```
# Create the database (schema)
```

```
my_db = my_session.create_schema('factory_sensors')
```

```
# Execute the SQL statement to create the table
```

```
sql_res = my_session.sql(CREATE_TBL).execute()
```

```
# Get the table object
```

```
my_tbl = my_db.get_table('trailer_assembly')
```

```
# Insert some rows (data)
```

```

my_tbl.insert(COLUMNS).values('paint_vat_temp', 32.815, 'Celsius').execute()
my_tbl.insert(COLUMNS).values('tongue_height_variance', 1.52, 'mm').execute()
my_tbl.insert(COLUMNS).values('ambient_temperature', 24.5, 'Celsius').execute()
my_tbl.insert(COLUMNS).values('gross_weight', 1241.01, 'pounds').execute()
# Execute a simple select (SELECT * FROM)
print("\nShowing results after inserting all rows.")
my_res = my_tbl.select(COLUMNS).execute()
# Display the results . Demonstrates how to work with results
# Print the column names followed by the rows
column_names = my_res.get_column_names()
column_count = my_res.get_column_count()
for i in range(0,column_count):
    if i < column_count - 1:
        print "{0}, ".format(column_names[i]),
    else:
        print "{0}".format(column_names[i]),
print

for row in my_res.fetch_all():
    for i in range(0,column_count):
        if i < column_count - 1:
            print "{0}, ".format(row[i]),
        else:
            print "{0}".format(row[i]),
    print

# Update a row
my_tbl.update().set('sensor_units', 'inches').where('sensor_value LIKE
1.52').execute()
print("\nShowing results after updating row with sensor_value LIKE 1.52.")
# Execute a simple select (SELECT * FROM)
my_res = my_tbl.select(COLUMNS).execute()
# Display the results
for row in my_res.fetch_all():
    print row
# Delete some rows

```

```

my_tbl.delete().where('sensor_value > 30').execute()
# Execute a simple select (SELECT * FROM)
print("\nShowing results after deleting rows with sensor_value > 30.")
my_res = my_tbl.select(COLUMNS).execute()
# Display the results
for row in my_res.fetch_all():
    print row
# Delete the database (schema)
my_session.drop_schema('factory_sensors')

```

Take a moment and read through the code to ensure you find the CRUD operations. Once again, these are very simple examples with only small examples of expressions for the WHERE clause. The comment lines and extra print statements form a guide to help make the code easier to read. We will see a more detailed example in the next chapter complete with more explanation of how to use the various methods for restricting the output (the WHERE clause).

Now, let's see the code executing. In this case, we will use the batch execution feature of the shell to read the file we created earlier and execute it. Listing 4-7 shows the command and results of running the script.

Listing 4-7. Executing the Sample Code

```
C:\Users\cbell\MySQL Shell\source\Ch04>mysqlsh --py -f listing4-6.py
```

Listing 4-6 Example - Python X DevAPI Demo with Relational Data.

```
Showing results after inserting all rows.
```

```

sensor_name, sensor_value, sensor_units
paint_vat_temp, 32.815, Celsius
tongue_height_variance, 1.52, mm
ambient_temperature, 24.5, Celsius
gross_weight, 1241.01, pounds

```

```
Showing results after updating row with sensor_value LIKE 1.52.
```

```

[
  "paint_vat_temp",
  32.815,
  "Celsius"
]

```

```
[
    "tongue_height_variance",
    1.52,
    "inches"
]
[
    "ambient_temperature",
    24.5,
    "Celsius"
]
[
    "gross_weight",
    1241.01,
    "pounds"
]
```

Showing results after deleting rows with `sensor_value > 30`.

```
[
    "tongue_height_variance",
    1.52,
    "inches"
]
[
    "ambient_temperature",
    24.5,
    "Celsius"
]
```

The output shows the first read operation that prints the rows using the method of working the results that we saw earlier by printing the column names and rows as comma-separated output with one row per line. The other output shows how the results are returned to Python – they’re a list of lists! That’s why we see the output appear as a list of string values for the rows.

Take a moment to look through the code again and ensure you can see the effects of the CRUD operations on the data. That is, the output of the results should differ slightly after each of the read, updated, and delete operations.

WHAT ABOUT CONNECTOR/PYTHON?

If you're following along and have used the Python database connector named Connector/Python, you may be wondering what is so different here that can't be done with the connector. At this point, your intuition is correct. So far, nothing I've presented cannot be done with the connector and good Python programming. In fact, it is this overlap that shows the X DevAPI has fulfilled one of its goals.

Now, you may be interested in knowing that the connector fully supports the X DevAPI and that the shell uses the connector under the hood. What we are learning then is how to work with our data from a different view – the view of data is code. Once you read Chapter 6, it will all click (if it hasn't already).

Summary

The traditional data storage and retrieval mechanism in MySQL is the SQL interface. This is what most are familiar with and indeed most learn SQL as part of their training to become a developer or database administrator. Thus, for many, learning a new tool like MySQL Shell is done best from the familiar ground of SQL. In fact, that's what this chapter is all about.

In this chapter, we saw a brief tour of using the shell first with a traditional SQL interactive session where we issued SQL statements and worked with relational data. It was familiar, and it demonstrated the basic concepts of relational data. Even if you haven't worked with SQL before, the small demonstration is enough to get you going.

However, the trend is to mix our data into our code, that is, to make our data part of the code. To do that, we need a strong API that lets us work with our data as if it were objects in the code. The X DevAPI is the answer. And, we saw a brief introduction to the X DevAPI for use with relational data. Not only did we learn how to get started using the X DevAPI, but we also saw some working Python code that you can use to get started writing your own Python scripts.

But that was just the beginning. What we really need is a larger example that we can use as a tutorial for writing more advanced Python scripts. The next chapter presents the concepts introduced in this chapter in greater detail.

CHAPTER 5

Example: SQL Database Development

In the last chapter, we explored the shell using two modes: the traditional SQL command execution and a brief tour of using the X DevAPI to write Python code to interact without SQL databases.

In this chapter, we will see a demonstration of how to use the shell to develop Python code modules for working with a traditional relational database. In fact, we will give credence to the claim that the MySQL Shell is a development tool.

We are going to do that by first examining the database for a sample application and then build the database code to access the data in the database. We will do this in a step-wise manner to give you the best view of how to use the shell to develop your own code. Finally, we will see a very brief demonstration of how to use the shell to test the database code.

We won't go into detail about the sample application because the focus is on how to use the shell rather than the sample application itself. However, the appendix for this book contains a tour of the code used to implement the sample application.

Let's begin by examining the sample database and briefly discussing the sample application.

Getting Started

It is one thing to be able to successfully explain topics in prose and to back it up with examples that solidify the reader's understanding, but it is quite a different thing to explain the benefits of a new way of working with data or code. In those cases, one must demonstrate the concepts in an interactive manner so that the concepts are proven through the example rather than simply showing how it is possible. In this section, we

will learn about a sample application that strives to do just that – to prove how you can use the shell to develop your own code.

However, in order to show the full capabilities of the shell in this manner, the sample must be sufficiently complex enough to have the depth (and breadth) to fulfill its role. Thus, for this chapter, we will focus on solving a monumental problem: how to organize your garage!

Ok, that may be too far to reach. Let's scale that back to simply organizing the tools in your garage or workshop. If you have any tools at all or are like me and have a vast array of tools for all manner of building, repairing, and servicing, it can be a real struggle to know where every individual tool is located, especially so when you acquire so many tools that you need multiple storage locations to hold them.

Sample Application Concept

The sample application concept is one of organization. As such, we will be storing descriptions of things we want to organize, including the things in which they are organized. Specifically, we want to know what tools we have and where they are stored. If that storage location is a toolbox or cabinet, we also want to know what drawer or shelf it is on. It may also be that we have tools stored in a box or bin and that, in turn, is stored in some place. Thus, we are modeling not only the tools but also the tool storage.

The garage application was born from a need to get better organized. Indeed, having a cluttered garage or just your tools stored in a cavalier manner may fit the needs of some people, others like me need a bit more structure. Plus, if you ever wanted to know if you had a certain tool, it would be nice to not only know that you have one but also where it is!

Thus, this application is designed primarily as a lookup tool, hence the primary focus on list views that show all the rows in the table.

That is why the storage location is the default view. If you walk into your garage (or workshop), the first thing you should see is the storage equipment – the toolboxes, racks, shelves, etc. When you look for a tool, you generally look in one or more of places (storage equipment) using the memory of the last place or the common place you store it. But if you have many tools, it may not be possible to remember where each tool resides, especially if you haven't used it in some time.

However, the sample application also provides a view to show all your handtools and powertools. The list views for each of these categories show all the items sorted

for you. You need to only skim through the list to find the desired tool, then look at the columns to determine where the tool is stored. So, by clicking a few times in your garage application, you know where to go to get the tool you want. We'll call the sample application MyGarage. Cool, eh?

Tip Rather than explain every nuance of the sample application, we will focus on the portions that are best used to prove the utility of using the shell to develop code – the database access code modules.

Let's take a quick look at a part of the user interface for the sample application. Figure 5-1 shows a detailed view of the handtool record. Here, we see we can specify the vendor, a description, the tool size, type, and location. In this way, we can capture the basic information about a tool, including which company made it and where we've stored it.

The screenshot shows a web form titled "Handtool - Detail". It contains the following fields and controls:

- Vendor:** A dropdown menu with "Vaughan" selected.
- Description:** A text input field containing "Smooth Face Steel Head Wood Tack Hammer".
- ToolSize:** A text input field containing "5-oz".
- Handtool Type:** A dropdown menu with "Hammer" selected.
- Location:** A dropdown menu with "Kobalt 3000 Steel Rolling Tool Cabinet (Black) - Drawer, Bottom" selected.
- Buttons:** Three buttons labeled "Update", "Delete", and "Close" are positioned at the bottom of the form.

Figure 5-1. *Handtool Detail View*

While this view looks rather straightforward, the design of the database underneath is a bit more complicated. For example, one can look at the form and predict we will have some way to store vendors since there can be many tools for one vendor. You can also predict there is a similar situation for storage locations. However, consider for the moment a piece of storage equipment can have one or more drawers or shelves or both. Thus, we may want to model these as well as the tools.

Before we embark on the database design, let's understand better the objects in the sample application. The following lists the objects identified in the application and how they are used. This will go a long way toward understanding how the data is stored (and its design).

- *Handtool*: A tool that is unpowered
- *Powertool*: A tool that runs on air (pneumatic) or electricity, either corded or cordless
- *Storage equipment*: A rack, box, chest, and so on that has one or more places where tools (things) can be stored
- *Storage place*: A feature of storage equipment such as a shelf or drawer
- *Organizer*: A container that can hold one or more tools but requires storing in a storage place
- *Vendor*: A manufacturer of a tool

Let's look at the storage portion of the sample application. This may seem a bit complicated, but once you see it in action it should be clearer. Let's say we have a new tool storage chest that has several drawers and shelves. If we were to make a table and store only the chest, how would we know in which drawer or on which shelf a tool resides?

For example, we could list the tool as being in `tool_chest_1`, but if it has ten drawers and four shelves, that doesn't help us much. Who wants an application that tells you a general location? You'd have to pull open drawers or randomly check shelves until you find your tool. However, if we abstract the drawers and shelves, we can specify the exact location for a given tool in the tool chest by referencing the storage place (drawer, shelf), which references the storage equipment.

Let's look at an example. Figure 5-2 shows a Kobalt tool chest available from a home improvement store (Lowe's). Notice the chest has seven drawers and two shelves.



Figure 5-2. *Kobalt Tool Chest*

If we model or create entries in a table for each drawer, we can then assign a relationship between the tool, drawer, and the tool chest. Not only does this demonstrate how we can categorize (organize) our data better, it also demonstrates one of the key aspects of most applications that use real data – there are several one-to-many relationships in the data.

Now that we understand the goals of the sample application and how we need to model the storage feature, let's see how the database is designed.

Database Design

Let's begin our tour of the database design from the entity-relationship diagram (ERD). Figure 5-3 shows the ERD for the database. If you're not familiar with these diagrams, they typically show the tables, views, or any other object you want along with the relationships between the entities (the dashed lines). Also included in this example are the indexes for each table. There is one view shown as a solid rectangle.

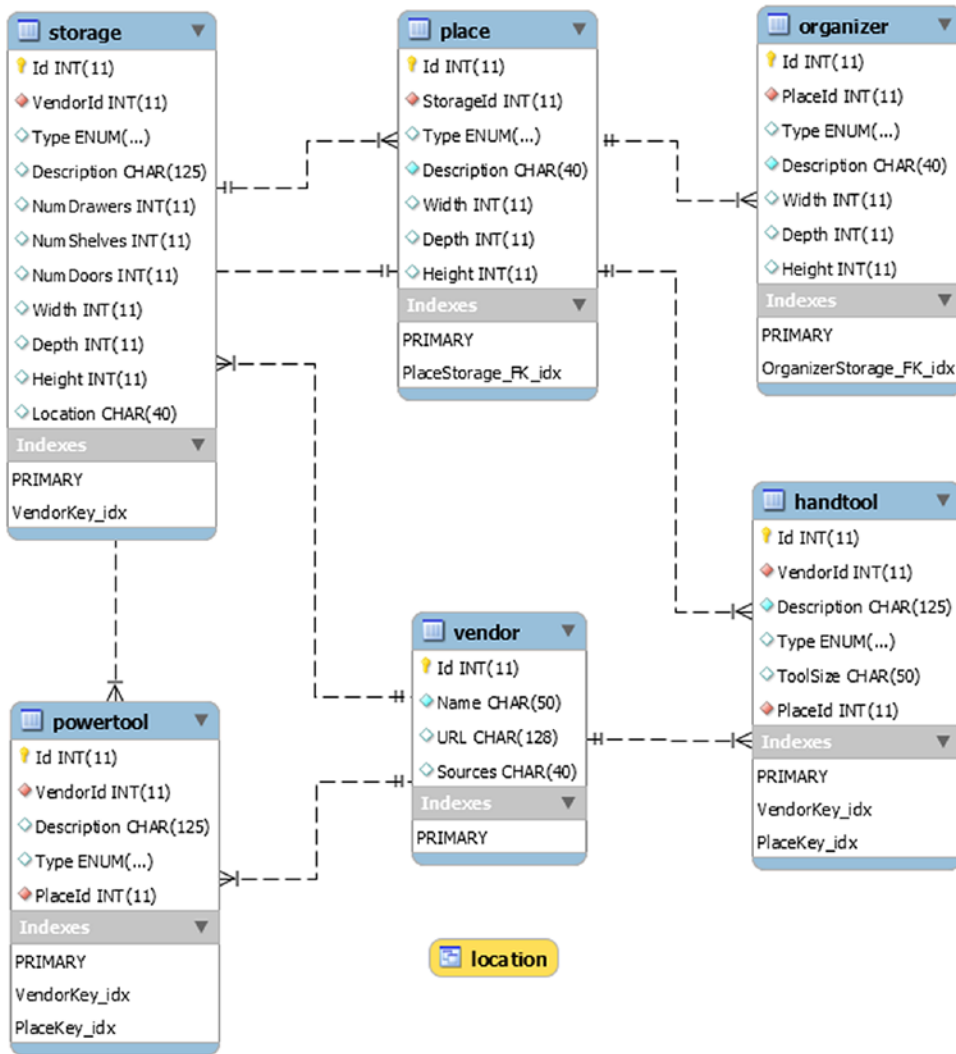


Figure 5-3. MyGarage Database ERD (Version 1)

Take some time to study the diagram so that you're familiar with the tables we will be using. We will see the tables in more detail later in this section. We will name the database garage_v1 because we will see how to migrate this database from a relational model to a NoSQL model in Chapter 7, which will become garage_v2.

One thing you may notice is each table has a surrogate key defined as an auto-increment field. This is a nice, easy way to ensure the rows in your table are unique and an artificial mechanism for allowing storage of more than one of the same item.

For example, in a typical tool collection, one generally has more than one of a certain tool such as a hammer, pliers, adjustable wrench, etc. Using auto-increment keys allows us to give each tool its unique Id.

Now, let's look at each of the entities in the ERD so that we understand what they store. We will start with the tables that have the fewest relationships and build from there so that you can understand how they are constructed.

When reading through this design, savvy readers may see ways to improve the design. However, recall the goal of this sample application was twofold: to be sufficiently complex to demonstrate nontrivial examples and to be something readers can run themselves. Thus, some design compromises were taken to avoid overcomplexity.¹

Vendor Table

The vendor table contains basic information about the vendors or manufacturers of the tools in the database. We record the name, a URL for the vendor's web site, and a short description of where we can purchase products from this vendor. Listing 5-1 shows the SQL CREATE TABLE command to create the vendor table.

Listing 5-1. Vendor Table

```
CREATE TABLE `garage_v1`.`vendor` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `Name` char(50) NOT NULL,
  `URL` char(128) DEFAULT NULL,
  `Sources` char(40) DEFAULT NULL,
  PRIMARY KEY (`Id`)
) ENGINE=InnoDB AUTO_INCREMENT=100 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci
```

Notice we use the auto-increment field for the primary key, which is an integer field. Look closely at the table options. Here, we demonstrate how to set the initial value for the auto-increment field (column). In this case, we chose a starting value of 100.

¹For example, I opt for object (table) names in the singular, while some prefer plural. It is customary to use singular names, but some may debate the issue to the point of placing fault. That's just noise for most of us.

We can choose other starting values for the other tables making each range of Ids somewhat unique. For example, if we set the starting value for another table at 1000, we can know at a glance that a row with an Id of 103 is a vendor while a value of 1022 is from the other table.

Granted, most savvy database administrators would quote chapter and version in some relational database textbook² about how horrible this practice is, but in practice, it can be handy if you use some general form of encoding like this for debugging purposes. Since the rows are in different tables, the fear or “sin” of encoding is not realized. That is, there is no possibility of collision. So, you can relax as this isn’t strictly an antithesis to relational database design; rather, it is a debugging or coding tool.

Organizer Table

The organizer table is used to store information about an organizer, be that a box, bin, molded case, and so on. This helps solve the problem of some tools having their own special cases and tools that must be grouped (and used) together such as socket sets, some types of wrenches, etc. We also employ the auto-increment trick by starting the value at 2000 for the Id column. Listing 5-2 shows the SQL CREATE TABLE command to create the organizer table.

Listing 5-2. Organizer Table

```
CREATE TABLE `garage_v1`.`organizer` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `PlaceId` int(11) NOT NULL,
  `Type` enum('Bin','Box','Case') DEFAULT 'Case',
  `Description` char(40) NOT NULL,
  `Width` int(11) DEFAULT '0',
  `Depth` int(11) DEFAULT '0',
  `Height` int(11) DEFAULT '0',
  PRIMARY KEY (`Id`),
  KEY `OrganizerStorage_FK_idx` (`PlaceId`),
```

²I was one of those at one point in my career. As I gain more and more experience, I’ve come to realize some trade-off can indeed be beneficial if used sparingly and safely.


```

CONSTRAINT `OrganizerStorage_FK` FOREIGN KEY (`PlaceId`) REFERENCES
`place` (`Id`) ON DELETE RESTRICT ON UPDATE RESTRICT
) ENGINE=InnoDB AUTO_INCREMENT=2000 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci

```

Notice this table contains a foreign key. A foreign key can be considered a link or relationship between a row in one table to a row in another. They are primarily used to enforce the relationship. For example, notice the restrictions in the preceding SQL code. Here, we see the foreign key is restricted on delete and update operations such that the row cannot be deleted in the `place` table if a row in this table references its `Id` column. It also specifies that the `Id` column in the `place` table cannot be altered in an update. This is the reason it is called “foreign” because it places restrictions on another table. This is another relational database construct that database designers use to help build robustness (and protection from accidental changes) into the database.

Storage Place Table

The storage place table, named `place` for brevity, is used to store information about places where we can store things such as a drawer or shelf. In fact, this table is limited to those two types through an enumerated column named `Type`. We also store a description, the `Id` of the storage equipment where this storage place resides, its dimensions. Listing 5-3 shows the SQL `CREATE TABLE` for the `place` table.

Listing 5-3. Place Table

```

CREATE TABLE `garage_v1`.`place` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `StorageId` int(11) NOT NULL,
  `Type` enum('Drawer','Shelf') DEFAULT 'Drawer',
  `Description` char(40) NOT NULL,
  `Width` int(11) DEFAULT '0',
  `Depth` int(11) DEFAULT '0',
  `Height` int(11) DEFAULT '0',
  PRIMARY KEY (`Id`),
  KEY `PlaceStorage_FK_idx` (`StorageId`),

```

```

CONSTRAINT `PlaceStorage_FK` FOREIGN KEY (`StorageId`) REFERENCES
`storage` (`Id`) ON DELETE RESTRICT ON UPDATE RESTRICT
) ENGINE=InnoDB AUTO_INCREMENT=1038 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci

```

This table also has a foreign key like the organizer table. Once again, this is so that rows cannot be deleted (or the Id column changed) in the storage table that a row in this table references.

Storage Equipment Table

The storage equipment table, named `storage` for brevity, is used to store information about a tool or general storage construct such as a tool chest, cabinet, workbench, or shelving. In fact, like the storage place table, we use an enumerated column named `Type` to specify the storage equipment type.

Along with the storage equipment type, we also store a description, the number of drawers, shelves, and doors (if applicable) as well as its overall dimensions and a general text field to store the location (physical description of) where it is in the garage or workshop. Listing 5-4 shows the SQL CREATE TABLE command for the storage table.

Listing 5-4. Storage Table

```

CREATE TABLE `garage_v1`.`storage` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `VendorId` int(11) NOT NULL,
  `Type` enum('Cabinet','Shelving','Toolchest','Workbench') DEFAULT
  'Toolchest',
  `Description` char(125) DEFAULT NULL,
  `NumDrawers` int(11) DEFAULT '0',
  `NumShelves` int(11) DEFAULT '0',
  `NumDoors` int(11) DEFAULT '0',
  `Width` int(11) DEFAULT NULL,
  `Depth` int(11) DEFAULT NULL,
  `Height` int(11) DEFAULT NULL,
  `Location` char(40) DEFAULT NULL,
  PRIMARY KEY (`Id`),
  KEY `VendorKey_idx` (`VendorId`),

```

```

CONSTRAINT `StorageVendor_FK` FOREIGN KEY (`VendorId`) REFERENCES
`vendor` (`Id`) ON DELETE RESTRICT ON UPDATE RESTRICT
) ENGINE=InnoDB AUTO_INCREMENT=503 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci

```

Once again, we see a foreign key here between this table and the vendor table so that no vendor can be deleted, or its Id column changed so long as there are rows in this table that reference it.

Handtool Table

The handtool table is used to store the information about each of the non-powered tools. We collect the vendor, description, and size. We also use an enumerated field named Type, which stores the type of tool so that we can group tools by category. This should make for issuing queries such as “show me all of my screwdrivers” much easier – especially when some categories of tools get put in different places. The types permitted can be seen in the SQL statement.

We also store links (the Id values for) vendor and storage place. Thus, we are forming the relationships between these tables. Listing 5-5 shows the SQL CREATE TABLE command for the handtool table.

Listing 5-5. Handtool Table

```

CREATE TABLE `garage_v1`.`handtool` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `VendorId` int(11) NOT NULL,
  `Description` char(125) NOT NULL,
  `Type` enum('Adjustable Wrench','Awl','Clamp','Crowbar','Drill Bit','File',
  'Hammer','Knife','Level','Nutdriver','Pliers','Prybar','Router Bit','Ru
  ler','Saw','Screwdriver','Socket','Socket Wrench','Wrench') DEFAULT NULL,
  `ToolSize` char(50) DEFAULT NULL,
  `PlaceId` int(11) NOT NULL,
  PRIMARY KEY (`Id`),
  KEY `VendorKey_idx` (`VendorId`),
  KEY `PlaceKey_idx` (`PlaceId`),
  CONSTRAINT `HandtoolPlace_FK` FOREIGN KEY (`PlaceId`) REFERENCES `place`
  (`Id`) ON DELETE RESTRICT ON UPDATE RESTRICT,

```

```

CONSTRAINT `HandtoolVendor_FK` FOREIGN KEY (`VendorId`) REFERENCES
`vendor` (`Id`) ON DELETE RESTRICT ON UPDATE RESTRICT
) ENGINE=InnoDB AUTO_INCREMENT=2253 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci

```

In this table, we have two foreign keys; one for the Id of the storage place (place) table and another for the Id in the vendor table.

Powertool Table

The powertool table is like the handtool table except here we store those tools that are powered by air, electricity (mains), or battery. We store the description and an enumerated field named Type for the type of power used by the tool. This might be handy if we wanted a list of all of the pneumatic tools (air).

We also store links (the Id values for) vendor and storage place. Thus, we are forming the relationships between these tables. Listing 5-6 shows the SQL CREATE TABLE command for the powertool table.

Listing 5-6. Powertool Table

```

CREATE TABLE `garage_v1`.`powertool` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `VendorId` int(11) NOT NULL,
  `Description` char(125) DEFAULT NULL,
  `Type` enum('Air','Corded','Cordless') DEFAULT NULL,
  `PlaceId` int(11) NOT NULL,
  PRIMARY KEY (`Id`),
  KEY `VendorKey_idx` (`VendorId`),
  KEY `PlaceKey_idx` (`PlaceId`),
  CONSTRAINT `PowerToolPlace_FK` FOREIGN KEY (`PlaceId`) REFERENCES `place`
(`Id`) ON DELETE RESTRICT ON UPDATE RESTRICT,
  CONSTRAINT `PowertoolVendor_FK` FOREIGN KEY (`VendorId`) REFERENCES
`vendor` (`Id`) ON DELETE RESTRICT ON UPDATE RESTRICT
) ENGINE=InnoDB AUTO_INCREMENT=3022 DEFAULT CHARSET=utf8mb4
COLLATE=utf8mb4_0900_ai_ci

```

We have the same foreign keys in this table as we had in the `handtool` table; one for the `Id` of the storage place (`place`) table and another for the `Id` in the `vendor` table.

Location View

Finally, we are going to employ one view. This view, named `location`, lets us quickly get a lookup table (view) of the combinations of storage places and storage equipment. We can use this to create a nice pull-down list in our sample application. Figure 5-4 shows an example of the result of the pull-down list. We would use the list to create references to the storage place in the `handtool` or `powertool` table as described before. Notice, we see a combination of the tables to make it much easier to see and select the proper location. This is another example of how real-world applications can employ tricks in the database to make the user interface easier to use.

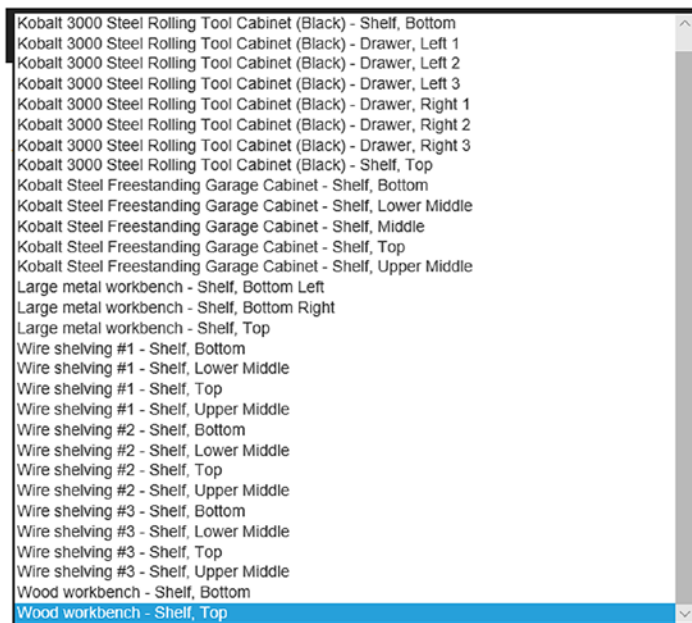


Figure 5-4. Using the Location View in a Drop-Down List

The SQL for this view is rather complex and involves a join (very common among relational databases) that combines the information from both tables. Listing 5-7 shows the SQL `CREATE VIEW` for the `location` view. Notice the view renames some of the

columns (using the AS keyword) to make it easier to differentiate between the fields from each table. This is especially important since we used a generic “Id” for the primary key for each table.³

Listing 5-7. Location View

```
CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`localhost` SQL SECURITY DEFINER
VIEW `garage_v1`.`location` AS
  SELECT `garage_v1`.`storage`.`Id` AS `StorageId`,
         `garage_v1`.`storage`.`Description` AS `StorageEquipment`,
         `garage_v1`.`place`.`Id` AS `PlaceId`,
         `garage_v1`.`place`.`Type` AS `Type`,
         `garage_v1`.`place`.`Description` AS `Location`
FROM (`garage_v1`.`storage` JOIN `garage_v1`.`place` ON
      ((`garage_v1`.`storage`.`Id` = `garage_v1`.`place`.`StorageId`)))
```

Now, let’s take a moment to discuss the database code design.

Code Design

While you may see some things in the database design that you’d do differently⁴ and there are several “right” ways to do things, code design takes that to a much higher level. That is, given two programmers and a set of code to review, they’ll likely spend more time scrutinizing the finer points of this way or that way to code something than it took to write it in the first place. That isn’t to say that the scrutiny isn’t beneficial – it most certainly is – rather, it means there are always ways to do the same thing in code using different mechanisms, structures, and philosophies.

This holds true for the code designed for the sample application. Choices were made to make the code modular, easier to read, and most of all, demonstrate (one way) to build relational database applications. So, what you are about to encounter may not be how you would have written the code, but it should still be usable in its current form for demonstration purposes. More specifically, the code design choices made for the sample application include the following:

³Another “sin” for the relational database purists. Hey, it happens.

⁴And you’re welcome to do so!

- Use Flask framework for web-based interface
- Use a class to represent each table in the database
- Place a single class in its own code module
- Place all database code modules in its own folder (named database)
- Use a class to encapsulate the connection to the database server
- Use class modules to test each of the table/view classes
- Use a code module run from the shell to test the class modules

We will see most of these constraints in the demonstration. As mentioned previously, a description of the user interface is included in the Appendix.

The code we are focusing on in this section includes the code we need to interact with the database. Thus, we will need code that implements the create, read, update, and delete (CRUD) operations. We also need code to help us make a connection to the database server.

Table 5-1 shows the code modules, class names, and description of each of the database code files planned. We will see how each of these is developed using the shell in the next section.

Table 5-1. Database Code Modules

Code module	Class name	Description
garage_v1	MyGarage	Implements connection to server and general server interface
handtool.py	Handtool	Models the handtool table
location.py	Location	Models the location view
organizer.py	Organizer	Models the organizer table
place.py	Place	Models the place table
powertool.py	Powertool	Models the powertool table
storage.py	Storage	Models the storage table
vendor.py	Vendor	Models the vendor table

When we write the code for the sample application to use these code modules, we will use the `MyGarage` class to make a connection to the database server and, when requested, use the class associated with each table to call the CRUD operations on each. The only exception is the `Location` class implements only the read operation because it is a view and views are used as lookup (read) tables.

Now that we understand the goals for the sample application and its design, let's get started with writing the database code for the sample application.

Setup and Configuration

The setup for the following demonstration does not require installing anything or even using the sample application; rather, we need only load the sample database because we will only be working with the database code modules. While images are used to depict certain aspects of the sample application, you don't strictly need it for this chapter. Once again, see the appendix for how to set up and use the complete sample application.

To install the sample database, we must download the sample source code from the book web site (<https://www.apress.com/us/book/9781484250822>). Choose the folder for this chapter and download the files. The sample source code contains a file named `database/garage_v1.sql`, which contains the SQL statements for creating the sample database and populating it with sample data.

This file not only issues the `CREATE DATABASE` and `CREATE TABLE` commands, it also contains a small set of data for each table using `INSERT SQL` commands. That is, it is an inventory for a set of tools in a typical garage or workshop. So, you don't have to spend precious time trying to come up with descriptions, sizes, etc. for a set of data to use - it's been done for you!

Since this file is an SQL file, we will need to use the `--sql` mode for the shell. Fortunately, we can use the options to read this file, import (source) it, and exit as shown in the following. Remember, you must either specify the path to the file or execute the shell from the directory where the file resides.

```
mysqlsh --uri root@localhost:3306 --sql -f garage_v1.sql
```

Change to the database folder and issue the following command to tell the shell to open the file and execute the statements. It won't take but a minute to run and, since we're running in batch mode, will exit the shell when complete. Listing 5-8 shows the

results of running these commands. If you're curious about the commands in the file, feel free to open it and look at how the SQL statements were written. You should notice that this is a dump of the database using the `mysqlpump` server client application.

Tip See <https://dev.mysql.com/doc/refman/8.0/en/mysqlpump.html> for more information about `mysqlpump`.

Listing 5-8. Populating the Example Database (Windows 10)

```
C:\Users\cbell\Documents\mygarage_v1>cd database
C:\Users\cbell\Documents\mygarage_v1\database>mysqlsh --uri root@localhost:3306 --sql -f garage_v1.sql
Records: 31 Duplicates: 0 Warnings: 0
Records: 6 Duplicates: 0 Warnings: 0
Records: 250 Duplicates: 0 Warnings: 0
Records: 3 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 22 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 22 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 2 Duplicates: 0 Warnings: 0
Records: 3 Duplicates: 0 Warnings: 0
```

Now that we have the database created and populated, we're ready to start exploring the database code modules!

Demonstration

While you may have realized when downloading the same source code from the book web site that the code modules are included in the download and are complete. Thus, if you are a savvy Python programmer, you may be tempted to skim through or skip the rest of the chapter. However, you should continue reading because we will be seeing a demonstration of how to use the shell to help develop these modules. That is, we will use the shell to start the development of our Python code for the sample application.

Note The complete, working sample application for this chapter is available for download on the book web site. See the Appendix on how to set up your machine to run the application.

If you have never programmed with Python before, don't worry as it is a very easy language to learn. In fact, you need only follow the examples in this section and by the end you will have a solid grasp on what the code is doing (and why). However, if you want to learn Python or should you require more in-depth knowledge of Python, there are several excellent books on the topic. I list a few of my favorites in the following. A great resource is the documentation on the Python site: python.org/doc/.

- *Pro Python*, Second Edition (Apress 2014), J. Burton Browning, Marty Alchin
- *Learning Python*, 5th Edition (O'Reilly Media 2013), Mark Lutz
- *Automate the Boring Stuff with Python: Practical Programming for Total Beginners* (No Starch Press 2015), Al Sweigart

In the following sections, we will see demonstrations of how to create the simplest class first (Location), then move on to some of the other classes. As you will see, they follow the same design pattern/layout so once you've seen one or two, the others are easy to predict. Thus, we will see detailed walkthroughs using a couple of the classes and the rest will be demonstrated and presented with fewer details for brevity.

If you want to follow along, be sure to have the sample database loaded and MySQL Shell ready to go. You may also want to use a code or text editor to write the code modules. More importantly, you should create a folder named database and start the shell from the parent folder.

For example, you should create a folder named `mygarage_v1`, and in that folder, create the database folder. We would then execute the shell from `mygarage_v1`. Why? Because we will use the Python import directive and name the path to the code module using the folder name (e.g., `from database import Location`). We will also be creating unit tests and thus will need a folder named `unittests` where we will store the test files.

Let's begin with the `MyGarage` class.

MyGarage Class

This class is intended to make it easier to work with the MySQL server by providing a mechanism to login (connect) to the server and encapsulate some of the common operations such as getting the session, current database, checking to see if the connection to MySQL is active, disconnecting, etc. We also will include methods to convert an SQL result or select result to a Python list (array) for easier processing. Table 5-2 shows the complete list of methods we will create for this class including the parameters required (some methods do not require them).

Table 5-2. *MyGarage Class Methods*

Method	Parameters	Description
<code>__init__()</code>	<code>mysqlx_sh</code>	Constructor – provide <code>mysqlx</code> if running from MySQL Shell
<code>connect()</code>	<code>username, passwd, host, port</code>	Connect to a MySQL server at <code>host, port</code>
<code>get_session()</code>		Return the session for use in other classes
<code>get_db()</code>		Return the database for use in other classes
<code>is_connected()</code>		Check to see if connected to the server
<code>disconnect()</code>		Disconnect from the server
<code>make_rows()</code>	<code>sql_select</code>	Return a Python array for the rows returned from a read operation from a select result
<code>make_rows_sql()</code>	<code>sql_res, num_cols</code>	Return a Python array for the rows returned from a read operation from a sql result

Writing the Source Code

For this code module, we won't use the shell to develop the code since this is more of a convenience class and you've already seen examples of most of its methods or at least the methods in the `mysqlx` module that are used in the code. Rather, we will see the complete code and then see how to test the class using the shell. This class maintains the current session and hides much of the mechanism for connecting to and disconnecting from the server.

Some may be inclined to move the connection mechanism into the classes (and you can) but using a separate class to manage that means you won't be duplicating any code, which is always preferred.

Listing 5-9 shows the complete code for the `MyGarage` class. Open a new file in your text or code editor and save this code in the database folder in a file named `garage_v1.py`. Take a few minutes to read through the code. It should be easy to read and understand even if you are learning Python.

Note Comments and nonessential lines have been removed in the source code listings in this chapter for brevity.

Listing 5-9. `garage_v1` Code

```
from __future__ import print_function

# Attempt to import the mysqlx module. If unsuccessful, we are
# running from the shell and must pass mysqlx in to the class
# constructor.
try:
    import mysqlx
except Exception:
    print("Running from MySQL Shell. Provide mysqlx in constructor.")

class MyGarage(object):
    def __init__(self, mysqlx_sh=None):
        self.session = None
```

```

if mysqlx_sh:
    self.mysqlx = mysqlx_sh
    self.using_shell = True
else:
    self.mysqlx = mysqlx
    self.using_shell = False

def connect(self, username, passwd, host, port):
    config = {
        'user': username,
        'password': passwd,
        'host': host,
        'port': port,
    }
    try:
        self.session = self.mysqlx.get_session(**config)
    except Exception as err:
        print("CONNECTION ERROR:", err)
        self.session = None
        raise

def get_session(self):
    return self.session

def get_db(self):
    return self.session.get_schema('garage_v1')

def is_connected(self):
    return self.session and (self.session.is_open())

def disconnect(self):
    try:
        self.session.close()
    except Exception as err:
        print("WARNING: {0}".format(err))

```

```

def make_rows(self, sql_select):
    cols = []
    if self.using_shell:
        cols = sql_select.get_column_names()
    else:
        for col in sql_select.columns:
            cols.append(col.get_column_name())
    rows = []
    for row in sql_select.fetch_all():
        row_item = []
        for col in cols:
            if self.using_shell:
                row_item.append("{0}".format(row.get_field(col)))
            else:
                row_item.append("{0}".format(row[col]))
        rows.append(row_item)
    return rows

@staticmethod
def make_rows_sql(sql_res, num_cols):
    rows = []
    all_rows = sql_res.fetch_all()
    for row in all_rows:
        row_item = []
        for col in range(0, num_cols):
            row_item.append("{0}".format(row[col]))
        rows.append(row_item)
    return rows

def get_last_insert_id(self):
    return self.get_session().sql(
        "SELECT LAST_INSERT_ID()").execute().fetch_one()

```

Notice the import line. This is placed in a try...except block because when using the code module from the shell, the shell does not expose the `mysqlx` module directly (it is one of the built-in modules). Rather, we can provide an instance of the built-in `mysqlx` module in the constructor.

Indeed, `__init__()` takes one parameter, `mysql_sh`, which we can use to run the code from the shell. This is a nifty way to make your code usable from either the shell or interactive (in an application).

Notice also we use a variable `self.using_shell` to store whether we are using the shell. This is needed in the `make_rows*` methods because the `mysqlx` module in the shell differs slightly from the `mysqlx` module provided in the connectors. See the following sidebar for why this is so.

DIFFERENCES IN SHELL AND CONNECTORS

One of the things you will notice as you move into more advanced applications is there are differences in the `mysqlx` module used in MySQL Shell from those used in the MySQL connectors (Connector/Python, Connector/J, etc.). The reason for these differences is primarily due to a desire to keep the operation or mechanics of the methods in the module the same across languages. Since the languages supported by the connectors are many, an attempt to standardize the behavior has resulted in some slight differences in how the shell implements the same methods. Fortunately, the differences are minor and easily rectified.

Now that we have the source code written, let's test the class using the MySQL Shell.

Testing the Class

Before we embark on testing the class, we must set the Python path variable (`PYTHONPATH`) to include the folder from which we want to run our tests. This is because we are using modules that are not installed at the system level, rather, are in a folder relative to the code we're testing. In Windows, you can use the following command to add the path for the execution to the Python path.

```
C:\Users\cbell\Documents\my_garage_v1> set PYTHONPATH=%PYTHONPATH%;c:\
users\cbell\Documents\mygarage_v1
```

Or, on Linux and macOS, you can use this command to set the Python path.

```
export PYTHONPATH=$(pwd);$PYTHONPATH
```

Now we can run the shell. For this, we will start in Python mode using the `--py` option. Let's exercise some of the methods in the class. We can do try all of them out except the `make_rows()` methods. We'll see those later. Listing 5-10 shows how to import the class in the shell, initialize (create) a class instance named `mygarage`, then connect with `connect()`, and execute some of the methods. We close with a call to `disconnect()` to shut down the connection to the server.

Listing 5-10. Testing MyGarage using MySQL Shell

```
C:\Users\cbell\Documents\my_garage_v1> mysqlsh --py
MySQL Py > from database.garage_v1 import MyGarage
Running from MySQL Shell. Provide mysqlx in constructor.
MySQL Py > myg = MyGarage(mysqlx)
MySQL Py > myg.connect('root', 'SECRET', 'localhost', 33060)
MySQL Py > db = myg.get_db()
MySQL Py > db
<Schema:garage_v1>
MySQL Py > s = myg.get_session()
MySQL Py > s
<Session:root@localhost:33060>
MySQL Py > myg.is_connected()
true
MySQL Py > myg.disconnect()
MySQL Py > myg.is_connected()
false
```

Notice here we imported the module, then created an instance of the class passing in the built-in `mysqlx` module. Then, we connected to the server (be sure to use the password for your system), retrieved the database and printed it (by placing the variable on a line and pressing *ENTER*), did the same for the session, and then finally tested the `is_connected()` and `disconnect()` methods.

During the execution, we issued `print()` statements to print some of the results from method calls. A nice feature of the shell is if you print a class instance variable, it displays the class name for the variable. That's another trick you can use to help you learn the classes and help you choose the correct methods to use. This will save you time as you continue to learn the X DevAPI.

Caution The class uses the `mysqlx` module, which requires an X Protocol connection. Be sure to use the X Protocol port (33060 by default).

If you want to save these commands in a file, you can. In fact, this is one form of a manual unit test to test class (units) of the code.⁵ To make it a bit easier to read from executing in batch mode, we will add some `print()` statements. To run this test, create a folder named `unittests` and place the file there named `garage_v1_test.py`. Listing 5-11 shows the complete listing for the file. We also added code to prompt for the user Id and password, which is much nicer than having it hardcoded in the file!

Listing 5-11. `garage_v1_test.py`

```
from getpass import getpass
from database.garage_v1 import MyGarage

print("MyGarage Class Unit test")
mygarage = MyGarage(mysqlx)
user = raw_input("User: ")
passwd = getpass("Password: ")
print("Connecting...")
mygarage.connect(user, passwd, 'localhost', 33060)
print("Getting the database...")
database = mygarage.get_db()
print(database)
print("Getting the session...")
session = mygarage.get_session()
print(session)
print("Connected?")
print(mygarage.is_connected())
print("Disconnecting...")
mygarage.disconnect()
print("Connected?")
print(mygarage.is_connected())
```

⁵These are not the unit tests available in Python using the unit testing framework (https://docs.python.org/2/library/unit_test.html). Rather, they are unit tests in that they test portions of the application (https://en.wikipedia.org/wiki/Unit_testing).

Later, if you want to execute it, you can do so with the following command. Remember to run this from the folder you created earlier (`mygarage_v1`). This is a nifty way to ensure you can test parts of your code without having the entire application sorted. Listing 5-12 shows the execution of the test code.

Listing 5-12. Running the `garage_v1_test` Unit Test

```
> mysqlsh --py -f unittests\garage_v1_test.py
Running from MySQL Shell. Provide mysqlx in constructor.
MyGarage Class Unit test
User: root
Password:
Connecting...
Getting the database...
<Schema:garage_v1>
Getting the session...
<Session:root@localhost:33060>
Connected?
True
Disconnecting...
Connected?
False
```

Now, let's look at the simplest of the classes that model one of the database tables or, in this case, a view.

Location Class

This class is a model of the location view. Recall, the location view performs the join to get a list of all the storage places and storage equipment into a single list that can be used as a lookup table. Thus, this class needs to implement only the read CRUD operation.

In the following section, we will demonstrate how to write the source code for the class using the MySQL Shell.

Writing the Source Code

One of the ways you can use the shell to write your code is to use an interactive session and write the code one line at a time. This allows you to experiment with how to organize the code and, more importantly, learn which methods to use.

For this class, we need only the read operation to populate the drop-down list in the detail forms for the hand tool, power tool, and organizer tables. Since we are using the `database\garage_v1.py` code module for the database connection, we will need to initialize that class first. Once we login and have an instance of the `MyGarage` class, we can use that to get the table and read the rows in the table. Listing 5-13 shows the code that can accomplish these steps.

Listing 5-13. Primitive Code

```
from database.garage_v1 import MyGarage
LOCATION_READ_COLS = ['PlaceId', 'StorageEquipment', 'Type', 'Location']
LOCATION_READ_BRIEF_COLS = ['StorageEquipment', 'Type', 'Location']
mygarage = MyGarage(mysqlx)
mygarage.connect('root', 'SECRET', 'localhost', 33060)
schema = mygarage.get_db()
table = schema.get_table('location')
sql_res = table.select(LOCATION_READ_COLS).order_by(*LOCATION_READ_
BRIEF_COLS).limit(5).execute()
rows = mygarage.make_rows(sql_res)
print(rows)
```

Notice here we use constants to set up the column names. This makes the code in the `select()` method a bit nicer, especially if you use other clauses that require a list of column names. In the case, we also used the `limit()` method, which limits the output to the first five rows, which makes the execution of the code brief. Listing 5-14 shows the execution of this code with the shell.

Listing 5-14. Executing the Primitive Code

```
MySQL Py > from database.garage_v1 import MyGarage
MySQL Py > LOCATION_READ_COLS = ['PlaceId', 'StorageEquipment', 'Type',
'Location']
```

```

MySQL Py > LOCATION_READ_BRIEF_COLS = ['StorageEquipment', 'Type',
'Location']
MySQL Py > mygarage = MyGarage(mysqlx)
MySQL Py > mygarage.connect('root', 'SECRET', 'localhost', 33060)
MySQL Py > schema = mygarage.get_db()
MySQL Py > table = schema.get_table('location')
MySQL Py > sql_res = table.select(LOCATION_READ_COLS).order_by(*LOCATION_
READ_BRIEF_COLS).limit(5).execute()
MySQL Py > rows = mygarage.make_rows(sql_res)
MySQL Py > print(rows)
[['1007', 'Kobalt 3000 Steel Rolling Tool Cabinet (Black)', 'Drawer',
'Bottom'], ['1001', 'Kobalt 3000 Steel Rolling Tool Cabinet (Black)',
'Drawer', 'Left 1'], ['1002', 'Kobalt 3000 Steel Rolling Tool Cabinet
(Black)', 'Drawer', 'Left 2'], ['1003', 'Kobalt 3000 Steel Rolling Tool
Cabinet (Black)', 'Drawer', 'Left 3'], ['1004', 'Kobalt 3000 Steel Rolling
Tool Cabinet (Black)', 'Drawer', 'Right 1']]

```

While the output of the rows isn't printed in a nice-to-read manner (it's not really required), you could add code to do that if you wanted to see the details, but printing the raw Python list is enough to see that five rows were returned.

Now, let's form a class from the preceding example code. We simply apply the coding constructs for creating a class having a single method named `read()`. We also write a constructor to the same using the instance of the `mysqlx` object to get the table. Listing 5-15 shows the modified code.

While the listings in this chapter show how you can type the code needed to create classes, there is a bit of a procedure you must follow to do this. Specifically, you must enter lines with spaces between the class declaration and its methods. This is because the shell will evaluate the class code when you press *ENTER* on a blank line. The same holds true for any multiline code block including dictionaries, lists, etc.

Caution If you encounter errors about unexpected indent even though the code is correct, try using a line with spaces on it for the separation between methods. Note that you can execute the file in batch mode without the need for the lines with spaces.

Listing 5-15. Location Class Primitive

```

from database.garage_v1 import MyGarage
LOCATION_READ_COLS = ['PlaceId', 'StorageEquipment', 'Type', 'Location']
LOCATION_READ_BRIEF_COLS = ['StorageEquipment', 'Type', 'Location']
class Location(object):
    def __init__(self, myg):
        self.table = myg.get_db().get_table('location')

    def read(self):
        sql_res = self.table.select(LOCATION_READ_COLS).order_by(
            *LOCATION_READ_BRIEF_COLS).limit(5).execute()
        return(mygarage.make_rows(sql_res))

mygarage = MyGarage(mysqlx)
mygarage.connect('root', 'SECRET', 'localhost', 33060)
location = Location(mygarage)
rows = location.read()
print(rows)

```

Notice we made a class with a constructor that sets a class variable to contain the class. This is required for running methods from the X DevAPI to implement the CRUD operations. Remember, we have the connection already from the MyGarage class instance (`myg`) and that is being passed into the Location class on the constructor.

That's just the class code. We also need to add code to execute or test the class. We add that after the class. When we place this code in a file (named `listing5-15.py`) and execute it, the shell will create the class as written and execute the lines following the class. For example, we execute the listing using the following command, which tells the shell to open the file and run the contents of the file one line at a time in Python mode.

```
$ mysqlsh --py -f listing5-15.py
```

Now, when we execute that code in the shell, we get the same output as before as shown in Listing 5-16.

Listing 5-16. Executing the Location Class Primitive

```

MySQL Py > from database.garage_v1 import MyGarage
Running from MySQL Shell. Provide mysqlx in constructor.

MySQL Py > LOCATION_READ_COLS = ['PlaceId', 'StorageEquipment', 'Type',
'Location']
MySQL Py > LOCATION_READ_BRIEF_COLS = ['StorageEquipment', 'Type',
'Location']
MySQL Py > class Location(object):
->     def __init__(self, myg):
->         self.table = myg.get_db().get_table('location')
->
->     def read(self):
->         sql_res = self.table.select(LOCATION_READ_COLS).order_
->             by(*LOCATION_READ_BRIEF_COLS).limit(5).execute()
->         return(mygarage.make_rows(sql_res))
->

MySQL Py > mygarage = MyGarage(mysqlx)
MySQL Py > mygarage.connect('root', 'SECRET', 'localhost', 33060)
MySQL Py > location = Location(mygarage)
MySQL Py > rows = location.read()
MySQL Py > print(rows)
[['1007', 'Kobalt 3000 Steel Rolling Tool Cabinet (Black)', 'Drawer',
'Bottom'], ['1001', 'Kobalt 3000 Steel Rolling Tool Cabinet (Black)',
'Drawer', 'Left 1'], ['1002', 'Kobalt 3000 Steel Rolling Tool Cabinet
(Black)', 'Drawer', 'Left 2'], ['1003', 'Kobalt 3000 Steel Rolling Tool
Cabinet (Black)', 'Drawer', 'Left 3'], ['1004', 'Kobalt 3000 Steel Rolling
Tool Cabinet (Black)', 'Drawer', 'Right 1']]

```

As you can see, we not only were able to code the class, we also tested the class at the end. This is a common and easy way to create your class modules. That is, rather than coding them from scratch in a Python code file and later executing them (which many do), the shell makes it possible to write the code on the fly. This is very similar to how the Python interpreter works. The difference is the shell makes it possible to use the X DevAPI directly.

Once you perfect your class, you can create the proper code module to store the class. In the sample application, this code is placed in the database folder with the name of the class. For example, the Location class is stored in a file named database/location.py. The completed code for the Location class is shown in Listing 5-17.

Listing 5-17. Completed Location Class Module (database/location.py)

```
class Location(object):
    """Location class

    This class encapsulates the location view permitting read operations
    on the data.
    """
    def __init__(self, mygarage):
        """Constructor"""
        self.mygarage = mygarage
        self.schema = mygarage.get_db()
        self.tbl = self.schema.get_table('location')

    def read(self):
        """Read data from the table"""
        sql_res = self.tbl.select(LOCATION_READ_COLS).order_by(
            *LOCATION_READ_BRIEF_COLS).execute()
        return self.mygarage.make_rows(sql_res)
```

Notice the completed code differs slightly in that we've added comments and we stored the MyGarage instance and retrieved the schema (database) and stored both in class variables. That is, the retrieval of the table was done in two steps rather than chaining the get_schema() and get_table() methods. This sort of simplification can sometimes make the code easier to read.

Now that we have the code module written, let's write a unit test to test the class.

Testing the Class

We have already seen a primitive of how to test the class in Listing 5-16. So, all we need to do is execute those same lines adding only the import statement for the Location class. Listing 5-18 shows the complete test code for the class. Notice we added [:5] to the print statement for the rows. This limits print to the first five items in the list (rows).

Listing 5-18. Test Code for the Location Class

```

from database.garage_v1 import MyGarage
from database.location import Location
mygarage = MyGarage(mysqlx)
mygarage.connect('root', 'SECRET', 'localhost', 33060)
location = Location(mygarage)
rows = location.read()
print(rows[:5])

```

We could place this code in a file and execute it, but let's use the shell instead. Listing 5-19 shows the execution of the code in the shell.

Listing 5-19. Executing the Location Class Test Code

```

MySQL Py > from database.garage_v1 import MyGarage
Running from MySQL Shell. Provide mysqlx in constructor.

MySQL Py > from database.location import Location
MySQL Py > mygarage = MyGarage(mysqlx)
MySQL Py > mygarage.connect('root', 'SECRET', 'localhost', 33060)
MySQL Py > location = Location(mygarage)
MySQL Py > rows = location.read()
MySQL Py > print(rows[:5])
[['Kobalt 3000 Steel Rolling Tool Cabinet (Black)', 'Drawer', 'Bottom'],
 ['Kobalt 3000 Steel Rolling Tool Cabinet (Black)', 'Drawer', 'Left 1'],
 ['Kobalt 3000 Steel Rolling Tool Cabinet (Black)', 'Drawer', 'Left 2'],
 ['Kobalt 3000 Steel Rolling Tool Cabinet (Black)', 'Drawer', 'Left 3'],
 ['Kobalt 3000 Steel Rolling Tool Cabinet (Black)', 'Drawer', 'Right 1']]

```

If you're thinking that we might want to make the testing code a bit more formal and easier to use, you're on the right track. We will explore that in a later section. But first, let's see how to create the class for the vendor table.

Vendor Class

The `Vendor` class is responsible for encapsulating the create, read, update, and delete (CRUD) operations on the `vendor` table. As such, the class will implement these methods by name. In fact, the other classes will implement the same methods. That way, we have uniformity in the sample application database code.

In this section, we will examine the `Vendor` class code in detail including a look at how we can build the class using the shell as well as how we can test the class in the shell. Now that we've seen a smaller example (just the read operation), the code for this class will be at least familiar in look but much more detailed as you shall see.

We will see a detailed demonstration of how to write the code for the class incrementally (one method at a time) starting with the first operation – create. We will also see the test code added to each example but for brevity will only show the code executing for the method we're focused on (each of the CRUD operations).

Create

The create operation is where we create a new row in the table. Thus, we will need to provide all the data for the row. In this case, that includes the name, URL, and sources fields. Recall, this allows us to give the vendor a name we recognize (e.g., Kobalt, Craftsman), a URL to the vendor's web site, and a sources field that is a text field describing the stores where we can purchase products for that vendor.

Like the `Location` class, we will need to add a few directives to start including the `import` for the `MyGarage` class and a list that includes the column names. The list is purely a bookkeeping measure that allows us to change the columns or use alternate column definitions for different SQL operations. We use this technique in more detail in other classes for the database code.

We also include the class definition like we did previously naming the class appropriately and adding a constructor method that accepts the `MyGarage` instance, stores it in a class variable for later use, gets the schema saving that to a class variable, and finally gets the table class instance.

The `create()` method simply accepts the values from the caller in the form of a dictionary where the column names are the keys, then issues the `insert()` method passing in the list of column names and values for the columns by chaining the `values()` method. Listing 5-20 shows the initial version of the `Vendor` class. Take a moment to read through the class definition.

Notice at the bottom of the listing is additional code to create an instance of the `Vendor` class, create a dictionary of test values, then call the `create()` method. Finally, we use the `MyGarage` method `get_last_insert_id()` to retrieve the last auto-increment value and print it.

Listing 5-20. Vendor Class `create()` Method

```
from database.garage_v1 import MyGarage
VENDOR_COLS_CREATE = ['Name', 'URL', 'Sources']
class Vendor(object):
    def __init__(self, myg):
        self.mygarage = mygarage
        self.schema = mygarage.get_db()
        self.tbl = self.schema.get_table('vendor')
    def create(self, vendor_data):
        vendor_name = vendor_data.get("Name", None)
        link = vendor_data.get("URL", None)
        sources = vendor_data.get("Sources", None)
        self.tbl.insert(VENDOR_COLS_CREATE).values(
            vendor_name, link, sources).execute()

mygarage = MyGarage(mysqlx)
mygarage.connect('root', 'SECRET', 'localhost', 33060)
vendor = Vendor(mygarage)
vendor_data = {
    "Name": "ACME Bolt Company",
    "URL": "www.acme.org",
    "Sources": "looney toons"
}

vendor.create(vendor_data)
last_id = mygarage.get_last_insert_id()[0]
```

When we execute this code in the shell, we will see each of the lines of code validated. Don't forget to use a blank line with spaces on it to separate the class methods and a blank line with no spaces on it to terminate the class and dictionary definitions. See the preceding note for why we need to do this.

Let's concentrate on the test code. Here, we simply created a new row and retrieved the value for its Id column as shown in the following (the rest of the demonstration omitted for brevity).

```
...
MySQL Py > print("Last insert id = {0}".format(last_id))
Last insert id = 177
```

However, that isn't much detail is it? We can't really tell if the row was inserted – only that we got the last insert Id. So, let's implement the read operation and use that to validate the create.

Read

The read operation needs two things: it needs to be able to read all the rows in the table and return them like we did with the Location class, but it also needs to be used to read a single row and return that data. This is because we will either read all the rows for the list views or, when viewing a single row, retrieve the values for that row.

To do this, we will use a parameter named `vendor_id`, which is set to `None` by default. This allows us to test this parameter and if it is `None`, retrieve all rows or retrieve a single row if has a value.

There is one other aspect to the read operation. Recall the create operation used a list to contain the column names. In that case, we did not need the Id field because the create operation (insert) will result in that value being populated by MySQL. However, for reading a row or all the rows, we need to get the Id column. Thus, we build another list to add the Id column for the `select()` method call.

Let's also add some error handling code. In this case, we will use a `try...except` block to catch any errors during the `insert()` and `select()`. We also add a technique to return a Boolean to tell the caller if the operation worked and, if it did not, an error message to be used to display to the user. We do this by returning a tuple such as `(True, None)` for success or `(False, <error>)` for an error. This will help us later if there is a problem. Listing 5-21 shows the class with the `read()` method added and the test code updated.

Listing 5-21. Adding the `read()` Method

```
from database.garage_v1 import MyGarage
VENDOR_COLS_CREATE = ['Name', 'URL', 'Sources']
VENDOR_COLS = []
```

```

VENDOR_COLS.extend(VENDOR_COLS_CREATE)
VENDOR_COLS.insert(0, 'Id') # Add the Id to the list
class Vendor(object):
    def __init__(self, mygarage):
        self.mygarage = mygarage
        self.schema = mygarage.get_db()
        self.tbl = self.schema.get_table('vendor')

    def create(self, vendor_data):
        vendor_name = vendor_data.get("Name", None)
        link = vendor_data.get("URL", None)
        sources = vendor_data.get("Sources", None)
        assert vendor_name, "You must supply a name for the vendor."
        try:
            self.tbl.insert(VENDOR_COLS_CREATE).values(
                vendor_name, link, sources).execute()
        except Exception as err:
            print("ERROR: Cannot add vendor: {0}".format(err))
            return (False, err)
        return (True, None)

    def read(self, vendor_id=None):
        if not vendor_id:
            # return all vendors
            sql_res = self.tbl.select(VENDOR_COLS).order_by("Name").
                execute()
        else:
            # return specific vendor
            sql_res = self.tbl.select(VENDOR_COLS).where(
                "Id = '{0}'".format(vendor_id)).execute()
        return self.mygarage.make_rows(sql_res)

mygarage = MyGarage(mysqlx)
mygarage.connect('root', 'SECRET', 'localhost', 33060)
vendor = Vendor(mygarage)

```

```

vendor_data = {
    "Name": "ACME Bolt Company",
    "URL": "www.acme.org",
    "Sources": "looney toons"
}

vendor.create(vendor_data)
last_id = mygarage.get_last_insert_id()[0]
print("Last insert id = {}".format(last_id))
rows = vendor.read(last_id)
print("{}".format(", ".join(rows[0])))
rows = vendor.read()
print(rows[:5])

```

Notice the last line of code prints only the first five rows returned for brevity. Like before, we will omit the entry of the class and focus on the lines that test the class. The following shows the lines executed to test the `create()` and `read()` methods. We test reading a single row by using the last Id returned after the `create()` and then perform a `read()` to get all the rows.

```

...
MySQL Py > print("Last insert id = {}".format(last_id))
Last insert id = 178
MySQL Py > rows = vendor.read(last_id)
MySQL Py > print("{}".format(", ".join(rows[0])))
178, ACME Bolt Company, www.acme.org, looney toons
MySQL Py > rows = vendor.read()
MySQL Py > print(rows[:5])
[['178', 'ACME Bolt Company', 'www.acme.org', 'looney toons'], ['175',
'ACME Bolt Company', 'www.acme.org', 'looney toons'], ['172', 'ACME Bolt
Company', 'www.acme.org', 'looney toons'], ['171', 'ACME Bolt Company',
'www.acme.org', 'looney toons'], ['170', 'ACME Bolt Company', 'www.acme.
org', 'looney toons']]

```

Once again, the printing of the rows isn't pretty, but for development purposes, it does show the `create()` and `read()` are working. Cool! Now, let's add the update operation.

Update

The update operation is like the create operation in that we will need all the data for the row. But unlike the create operation, we need the `Id` column so that we're updating the correct row. Savvy developers would add a step to validate the columns before issuing the update so that only those columns that changed are updated, but we'll take a simpler approach and supply all the columns and let the database sort it out.

However, since the update operation must have the `Id` column, we will add an assertion to ensure the caller provides that information for the `where()` method. Otherwise, an update would be too dangerous!

We also use a `try...except` block around the `update()` to catch any errors. Listing 5-22 shows the class with the `update()` method added and the constructor, `create()`, and `read()` methods omitted for brevity. Notice how we set the values for the columns using a `for` loop.

Listing 5-22. Adding the `update()` Method

```
from database.garage_v1 import MyGarage
VENDOR_COLS_CREATE = ['Name', 'URL', 'Sources']
VENDOR_COLS = []
VENDOR_COLS.extend(VENDOR_COLS_CREATE)
VENDOR_COLS.insert(0, 'Id') # Add the Id to the list
class Vendor(object):
    ...
    def update(self, vendor_data):
        vendor_id = vendor_data.get("VendorId", None)
        vendor_name = vendor_data.get("Name", None)
        link = vendor_data.get("URL", None)
        sources = vendor_data.get("Sources", None)
        assert vendor_id, "You must supply an Id to update the vendor."
        field_value_list = [('Name', vendor_name),
                            ('URL', link), ('Sources', sources)]
        try:
            tbl_update = self.tbl.update()
            for field_value in field_value_list:
                tbl_update.set(field_value[0], field_value[1])
```

```

tbl_update.where("Id = '{0}'".format(vendor_id)).execute()
except Exception as err:
    print("ERROR: Cannot update vendor: {0}".format(err))
    return (False, err)
return (True, None)

mygarage = MyGarage(mysqlx)
mygarage.connect('root', 'SECRET', 'localhost', 33060)
vendor = Vendor(mygarage)
vendor_data = {
    "Name": "ACME Bolt Company",
    "URL": "www.acme.org",
    "Sources": "looney toons"
}
vendor.create(vendor_data)
last_id = mygarage.get_last_insert_id()[0]
print("Last insert id = {0}".format(last_id))
rows = vendor.read(last_id)
print("{0}".format(", ".join(rows[0])))
rows = vendor.read()
print(rows[:5])
vendor_data = {
    "VendorId": last_id,
    "Name": "ACME Nut Company",
    "URL": "www.acme.org",
    "Sources": "looney toons"
}
vendor.update(vendor_data)
rows = vendor.read(last_id)
print("{0}".format(", ".join(rows[0])))

```

The test code for the update operation simply uses the same dictionary as the create operation, only we change some of the values to test the update. The following shows the output of the test execution starting with the read after the create operation. Notice update() does indeed change the values for that row we created earlier.

```

...
MySQL Py > rows = vendor.read(last_id)
MySQL Py > print("{0}".format(", ".join(rows[0])))
179, ACME Bolt Company, www.acme.org, looney toons
MySQL Py > rows = vendor.read()
...
MySQL Py > vendor_data = {
    ->     "VendorId": last_id,
    ->     "Name": "ACME Nut Company",
    ->     "URL": "www.acme.org",
    ->     "Sources": "looney toons"
    -> }
MySQL Py > vendor.update(vendor_data)
MySQL Py > rows = vendor.read(last_id)
MySQL Py > print("{0}".format(", ".join(rows[0])))
179, ACME Nut Company, www.acme.org, looney toons

```

Now, let's add the last operation – delete.

Delete

The delete operation simply deletes a row in the table. All we need to do that is the Id column. Thus, the `delete()` method is written to use the `vendor_id` as a parameter testing to ensure there is one provided, then issues the delete operation on the table.

Like the other methods, we use the `try...except` block and return a tuple to report whether the operation succeeded or not. Listing 5-23 shows the class with the `delete()` method added and the constructor, `create()`, `read()`, and `update()` methods omitted for brevity.

Listing 5-23. Adding the `delete()` Method

```

from database.garage_v1 import MyGarage
VENDOR_COLS_CREATE = ['Name', 'URL', 'Sources']
VENDOR_COLS = []
VENDOR_COLS.extend(VENDOR_COLS_CREATE)
VENDOR_COLS.insert(0, 'Id') # Add the Id to the list
class Vendor(object):

```



```

...
def delete(self, vendor_id=None):
    """Delete a row from the table"""
    assert vendor_id, "You must supply an Id to delete the vendor."
    try:
        self.tbl.delete().where("Id = '{0}'".format(vendor_id)).
            execute()
    except Exception as err:
        print("ERROR: Cannot delete vendor: {0}".format(err))
        return (False, err)
    return (True, None)

mygarage = MyGarage(mysqlx)
mygarage.connect('root', 'secret', 'localhost', 33060)
vendor = Vendor(mygarage)
vendor_data = {
    "Name": "ACME Bolt Company",
    "URL": "www.acme.org",
    "Sources": "looney toons"
}
vendor.create(vendor_data)
last_id = mygarage.get_last_insert_id()[0]
print("Last insert id = {0}".format(last_id))
rows = vendor.read(last_id)
print("{0}".format(", ".join(rows[0])))
rows = vendor.read()
print(rows[:5])
vendor_data = {
    "VendorId": last_id,
    "Name": "ACME Nut Company",
    "URL": "www.acme.org",
    "Sources": "looney toons"
}
vendor.update(vendor_data)
rows = vendor.read(last_id)
print("{0}".format(", ".join(rows[0])))

```

```

vendor.delete(last_id)
rows = vendor.read(last_id)
if not rows:
    print("Record not found.")

```

Ok, so that is the complete code for the class. The following shows the execution after the `update()` call. Here, we attempt to delete the row we inserted and later updated, then try to read it from the table. If there are no rows returned, the row was not found and hence we show the delete operation succeeding.

```

...
MySQL Py > rows = vendor.read(last_id)
MySQL Py > print("{0}".format(", ".join(rows[0])))
180, ACME Nut Company, www.acme.org, looney toons
MySQL Py > vendor.delete(last_id)
MySQL Py > rows = vendor.read(last_id)
MySQL Py > if not rows:
    -> print("Record not found.")
Record not found.

```

Now that we've had a detailed walkthrough on how to use the shell to create the classes for the database tables (and view), let's see an overview of each of the remaining classes. Each of the classes is written in the same manner as the `Vendor` class with details of specific implementation for the class noted. As you will see, uniformity is our friend.

Handtool Class

The `Handtool` class encapsulates the CRUD operations for the `handtool` table. It uses the same class structure and methods as the other classes. It differs in complexity from the `Vendor` class in three main ways.

First, the `handtool` table has several fields that cannot be blank (Null), so there are a few additional assertions for insert and update operations as follows.

```

assert tool_size, "You must specify a toolsize for the handtool."
assert handtool_type, "You must specify a type for the handtool."
assert description, "You must supply a description for the handtool."
assert place_id, "You must supply an Id for the handtool."

```

Second, the table has an enumerated field, which we represent with another list in the code as follows. This allows us to map the names to the enumerated values. This may appear strange as it is a list of tuples with the values repeated. That is intentional. A map can be defined this way so that the key (the first value in the tuple) is used to “map” to the second value (or simply value). Since we are good database developers, we don’t encode numeric values for the enumerated field values, we must repeat the value.

```
HANDTOOL_TYPES = [
    ('Adjustable Wrench', 'Adjustable Wrench'), ('Awl', 'Awl'),
    ('Clamp', 'Clamp'), ('Crowbar', 'Crowbar'), ('Drill Bit', 'Drill Bit'),
    ('File', 'File'), ('Hammer', 'Hammer'), ('Knife', 'Knife'),
    ('Level', 'Level'),
    ('Nutdriver', 'Nutdriver'), ('Pliers', 'Pliers'), ('Prybar', 'Prybar'),
    ('Router Bit', 'Router Bit'), ('Ruler', 'Ruler'), ('Saw', 'Saw'),
    ('Screwdriver', 'Screwdriver'), ('Socket', 'Socket'),
    ('Socket Wrench', 'Socket Wrench'), ('Wrench', 'Wrench'),
]
```

Third, since the handtool table has several fields, a list of the rows in the table for a lookup or browse operation does not require all the fields. It is also the case that we want to see the values that the foreign keys point to. Thus, we use a SQL SELECT query like the one shown in the following in place of the read all rows feature for the read() method. We store this in a constant to make it easier to read and change if needed.

```
HANDTOOL_READ_LIST = (
    "SELECT handtool.Id, handtool.type, handtool.description, "
    "handtool.toolsize, storage.description as StorageEquipment, "
    "place.type as locationtype, place.description as location FROM "
    "garage_v1.handtool "
    "JOIN garage_v1.place ON "
    "handtool.placeid = place.id JOIN garage_v1.storage ON place."
    "storageid = storage.id "
    "ORDER BY handtool.type, handtool.description"
)
```

This query is used in the read operation as follows. Notice we use the `sql()` method of the session object instead of the `select()` method for issuing the query. Thus, we also capture the session object instance in the constructor.

```

if not handtool_id:
    # return all handtools - uses a JOIN so we have to use the sql()
    # method instead of select, but we arrive at the same results
    sql_res = self.session.sql(HANDTOOL_READ_LIST).execute()
    return self.mygarage.make_rows_sql(sql_res, len(HANDTOOL_READ_COLS))
else:
    # return specific handtool
    sql_res = self.tbl.select(HANDTOOL_COLS).where(
        "Id = '{0}'".format(handtool_id)).execute()
return self.mygarage.make_rows(sql_res)

```

You can find this code in the `database/handtool.py` code module. Take a few moments to study these changes and see for yourself how they fit together.

Organizer Class

The Organizer class encapsulates the CRUD operations for the organizer table. It uses the same class structure and methods as the other classes. It differs in complexity from the Vendor class like the Handtool class; it requires a map for the enumerated column and an SQL SELECT statement for the read operation as shown in the following. Otherwise, the code is the same pattern as the Vendor class.

```

ORGANIZER_TYPES = [('Bin', 'Bin'), ('Box', 'Box'), ('Case', 'Case')]
...
ORGANIZER_READ_LIST = (
    "SELECT organizer.Id, organizer.Type, organizer.Description, "
    "storage.description as StorageEquipment, place.type as LocationType, "
    "place.description as Location FROM garage_v1.organizer JOIN "
    "garage_v1.place ON organizer.placeid = place.ID JOIN "
    "garage_v1.storage ON place.storageid = storage.id "
    "ORDER BY Type, organizer.description"
)

```

Place Class

The Place class encapsulates the CRUD operations for the place table. It uses the same class structure and methods as the other classes. It differs in complexity from the Vendor class like the Handtool class; it requires a map for the enumerated column and an SQL SELECT statement for the read operation as shown in the following. Otherwise, the code is the same pattern as the Vendor class.

```
PLACE_TYPES = [('Drawer', 'Drawer'), ('Shelf', 'Shelf')]
...
PLACE_READ_LIST = (
    "SELECT place.Id, storage.description as StorageEquipment, place.Type
    as LocationType, "
    "place.Description as Location FROM garage_v1.place JOIN "
    "garage_v1.storage ON place.StorageId = storage.ID ORDER BY "
    "StorageEquipment, LocationType, Location"
)
```

Powertool Class

The Powertool class encapsulates the CRUD operations for the powertool table. It uses the same class structure and methods as the other classes. It differs in complexity from the Vendor class like the Handtool class; it requires a map for the enumerated column and an SQL SELECT statement for the read operation as shown in the following. Otherwise, the code is the same pattern as the Vendor class.

```
POWERTOOL_TYPES = [('Corded', 'Corded'), ('Cordless', 'Cordless'),
('Air', 'Air')]
...
POWERTOOL_READ_LIST = (
    "SELECT powertool.Id, powertool.type, powertool.description, "
    "storage.description as StorageEquipment, place.type as locationtype, "
    "place.description as location FROM garage_v1.powertool JOIN garage_
    v1.place "
    "ON powertool.placeid = place.id JOIN garage_v1.storage ON "
    "place.storageid = storage.id ORDER BY powertool.type, powertool.
    description"
)
```

Storage Class

The Storage class encapsulates the CRUD operations for the storage table. It uses the same class structure and methods as the other classes. It differs in complexity from the Vendor class like the Handtool class; it requires a map for the enumerated column and an SQL SELECT statement for the read operation as shown in the following. It also differs in the read operation for returning all rows returns a smaller list of columns. This is used to show all of the storage equipment in the user interface. Otherwise, the code is the same pattern as the Vendor class.

```
STORAGE_TYPES = [
    ('Cabinet', 'Cabinet'), ('Shelving', 'Shelving'),
    ('Toolchest', 'Toolchest'), ('Workbench', 'Workbench')
]
STORAGE_COLS_BRIEF = [
    'storage.Id', 'Type', 'Description', 'Location'
]
...
STORAGE_READ_LIST = (
    "SELECT storage.Id, vendor.name, Type, description, Location FROM "
    "garage_v1.storage JOIN garage_v1.vendor ON storage.VendorId = vendor.Id "
    "ORDER BY Type, Location"
)
```

Testing the Class Modules

One of the tools in a developer's toolbox is a strong set of tests. Since we have already seen how to create the database classes in the shell, let's now see how to develop a testing framework for testing the database classes.

Recall, testing of the classes used a very similar mechanism and, in fact, followed the same sequence of steps. Whenever developers see this, they think "automate" and "class." That is, it is easy to create a base class that contains all the steps and subclasses that implement the class (test) specific to the class it is testing. This is a very common way to approach repeatable tests.

In this case, we create a base class named `CRUDTest` in the `unittests/crud_test.py` code module that implements methods for starting (or setup) of the test, a generic

method to show the rows returned, and one each for the test cases we want to run. Table 5-3 shows the methods implemented in the class.

Table 5-3. *CRUDTest Class Methods*

Method	Parameters	Description
<code>__init__()</code>		Constructor
<code>begin()</code>	mysqlx instance, class name, user name, password	Connect to a MySQL server and setup the MyGarage class. Called by the <code>setup()</code> method
<code>show_rows()</code>	rows (list), number of rows to display	Print the rows in the list up to the number specified
<code>set_up()</code>		Setup the test and initialize the test cases. Override for each class
<code>create()</code>		Run the create test case. Override for each class
<code>read_all()</code>		Run the read test case to return all rows. Override for each class
<code>read_one()</code>		Run the read test case to return a specific row. Override for each class
<code>update()</code>		Run the update test case. Override for each class
<code>delete()</code>		Run the delete test case. Override for each class
<code>tear_down()</code>		Close the test and disconnect from the server

This may seem like a lot of work, but let's look at the class first, then see an example of how we can derive from it to create a test for one of the database classes. Listing 5-24 shows the code for the `CRUDTest` class.

Listing 5-24. Code for the `CRUDTest` Class

```
from __future__ import print_function
from getpass import getpass
from database.garage_v1 import MyGarage
```

```

class CRUDTest(object):
    """Base class for Unit testing table/view classes."""

    def __init__(self):
        """Constructor"""
        self.mygarage = None

    def __begin(self, mysql_x, class_name, user=None, passwd=None):
        """Start the tests"""
        print("\n*** {0} Class Unit test ***\n".format(class_name))
        self.mygarage = MyGarage(mysql_x)
        if not user:
            user = raw_input("User: ")
        if not passwd:
            passwd = getpass("Password: ")
        print("Connecting...")
        self.mygarage.connect(user, passwd, 'localhost', 33060)
        return self.mygarage

    @staticmethod
    def show_rows(rows, num_rows):
        """Display N rows from row result"""
        print("\n\tFirst {0} rows:".format(num_rows))
        print("\t-----")
        for item in range(0, num_rows):
            print("\t{0}".format(", ".join(rows[item])))

    def set_up(self, mysql_x, user=None, passwd=None):
        """Setup functions"""
        pass

    def create(self):
        """Run Create test case"""
        pass

    def read_all(self):
        """Run Read(all) test case"""
        pass

```



```

def read_one(self):
    """Run Read(record) test case"""
    pass

def update(self):
    """Run Update test case"""
    pass

def delete(self):
    """Run Delete test case"""
    pass

def tear_down(self):
    """Tear down functions"""
    print("\nDisconnecting...")
    self.mygarage.disconnect()

```

Notice we perform the initialization, setup, and teardown steps in the base class. This way, we can ensure we execute those steps the same way for each class.

Notice also the methods we want to override are listed with “pass” as the body. This is essentially a “do nothing,” but legal method body. We will write the specifics for each method in the classes we use to create tests for the database classes.

For example, we migrate our test for the Vendor class by creating a new class named VendorTests derived from CRUDTest and stored in the file `unittests/vendor_test.py`. Listing 5-25 shows the code for the new class.

Listing 5-25. Code for the VendorTests Class

```

from __future__ import print_function

from unittests.crud_test import CRUDTest
from database.vendor import Vendor

class VendorTests(CRUDTest):
    """Test cases for the Vendor class"""

    def __init__(self):
        """Constructor"""
        CRUDTest.__init__(self)

```

```

self.vendor = None
self.last_id = None

def set_up(self, mysql_x, user=None, passwd=None):
    """Setup the test cases"""
    self.mygarage = self.begin(mysql_x, "Vendor", user, passwd)
    self.vendor = Vendor(self.mygarage)

def create(self):
    """Run Create test case"""
    print("\nCRUD: Create test case")
    vendor_data = {
        "Name": "ACME Bolt Company",
        "URL": "www.acme.org",
        "Sources": "looney toons"
    }
    self.vendor.create(vendor_data)
    self.last_id = self.mygarage.get_last_insert_id()[0]
    print("\tLast insert id = {0}".format(self.last_id))

def read_all(self):
    """Run Read(all) test case"""
    print("\nCRUD: Read (all) test case")
    rows = self.vendor.read()
    self.show_rows(rows, 5)

def read_one(self):
    """Run Read(record) test case"""
    print("\nCRUD: Read (row) test case")
    rows = self.vendor.read(self.last_id)
    print("\t{0}".format(", ".join(rows[0])))

def update(self):
    """Run Update test case"""
    print("\nCRUD: Update test case")
    vendor_data = {
        "VendorId": self.last_id,
        "Name": "ACME Nut Company",

```

```

        "URL": "www.acme.org",
        "Sources": "looney toons"
    }
    self.vendor.update(vendor_data)
def delete(self):
    """Run Delete test case"""
    print("\nCRUD: Delete test case")
    self.vendor.delete(self.last_id)
    rows = self.vendor.read(self.last_id)
    if not rows:
        print("\tNot found (deleted).")

```

What makes this technique powerful is we can go on to create new tests for each of the database classes named for the class and store them in the same `unittests` folder. We can then write a driver script that runs all the tests in a loop. Since the `Location` class has only a read all operation, we can code “no operation” for the other operations, which allows us to include the `LocationTests` in the loop. Listing 5-26 shows the code for a driver script named `run_all.py` also stored in the `unittests` folder.

Listing 5-26. Test Driver `run_all.py`

```

from __future__ import print_function
from getpass import getpass
from unittests.handtool_test import HandtoolTests
from unittests.location_test import LocationTests
from unittests.organizer_test import OrganizerTests
from unittests.place_test import PlaceTests
from unittests.powertool_test import PowertoolTests
from unittests.storage_test import StorageTests
from unittests.vendor_test import VendorTests
print("CRUD Tests for all classes...")
crud_tests = []
handtool = HandtoolTests()
crud_tests.append(handtool)
location = LocationTests()
crud_tests.append(location)
organizer = OrganizerTests()

```

```

crud_tests.append(organizer)
place = PlaceTests()
crud_tests.append(place)
powertool = PowertoolTests()
crud_tests.append(powertool)
storage = StorageTests()
crud_tests.append(storage)
vendor = VendorTests()
crud_tests.append(vendor)
# Get user, passwd
user = raw_input("User: ")
passwd = getpass("Password: ")
# Run the CRUD operations for all classes that support them
for test in crud_tests:
    test.set_up(mysqlx, user, passwd)
    test.create()
    test.read_one()
    test.read_all()
    test.update()
    test.read_one()
    test.delete()
    test.tear_down()

```

To execute this test, you can use the command shown in Listing 5-27 with the expected output. Here, we see only a portion of the output for brevity.

Listing 5-27. Executing the Test Driver

```

C:\Users\cbell\Documents\mygarage_v1>mysqlsh --py -f unittests/run_all.py
Running from MySQL Shell. Provide mysqlx in constructor.
CRUD Tests for all classes...
User: root
Password:
*** Handtool Class Unit test ***
Connecting...

```

CRUD: Create test case

 Last insert id = 2267

CRUD: Read (row) test case

 2267, 101, Plumpbus, Hammer, medium, 1001

CRUD: Read (all) test case

 First 5 rows:

 2050, Awl, Alloy Steel Scratch, 6-in, Kobalt 3000 Steel Rolling
Tool Cabinet (Black), Drawer, Left 3

 2048, Awl, Complex Hook, 3-in, Kobalt 3000 Steel Rolling Tool
Cabinet (Black), Drawer, Left 3

 2049, Awl, Curved Hook, 3-in, Kobalt 3000 Steel Rolling Tool
Cabinet (Black), Drawer, Left 3

 2047, Awl, Hook, 3-in, Kobalt 3000 Steel Rolling Tool Cabinet
(Black), Drawer, Left 3

 2046, Awl, Scratch, 3-in, Kobalt 3000 Steel Rolling Tool Cabinet
(Black), Drawer, Left 3

CRUD: Update test case

CRUD: Read (row) test case

 2267, 101, Plumpbus Pro, Screwdriver, grande, 1001

CRUD: Delete test case

 Not found (deleted).

Disconnecting...

*** Location Class Unit test ***

Connecting...

CRUD: Create test case (SKIPPED)

CRUD: Read (row) test case (SKIPPED)

CRUD: Read (all) test case

 First 5 rows:

 1007, Kobalt 3000 Steel Rolling Tool Cabinet (Black), Drawer, Bottom

 1001, Kobalt 3000 Steel Rolling Tool Cabinet (Black), Drawer, Left 1

 1002, Kobalt 3000 Steel Rolling Tool Cabinet (Black), Drawer, Left 2

 1003, Kobalt 3000 Steel Rolling Tool Cabinet (Black), Drawer, Left 3

 1004, Kobalt 3000 Steel Rolling Tool Cabinet (Black), Drawer, Right 1

CRUD: Update test case (SKIPPED)
CRUD: Read (row) test case (SKIPPED)
CRUD: Delete test case (SKIPPED)
...

Take some time to download the code from the book web site and test out the unit tests yourself. You should notice it is very easy to use this concept and you can develop others like it to test your database code. Just think; we did this all without having to write any user interface code, which allows to validate our database code before the first line of user interface code is written. Nice!

Summary

At first, some may be skeptical of the claim that MySQL Shell can be used as a development tool. This may be partly because it is new and partly because it isn't a typical code editor; rather, it is more like the Python interpreter.

However, you have seen for yourself how one can use the shell to not only test out code to see what methods work and how to work with the X DevAPI, but you also saw how easy it is to write code and execute it in the shell.

In fact, we kicked that up a notch further by demonstrating how to develop and test database code modules in Python all without having a user interface to support it. This is a huge benefit for developers because it is often left to the end to test modules like the database classes. This way, we test the database code before the user interface is written, thereby allowing us to concentrate on one piece of the application at a time.

In the next chapter, we will continue our journey in the X DevAPI using the shell by taking a deeper look at how to use the MySQL Document Store – a whole new, non-SQL (NOSQL) way of working with data.

CHAPTER 6

Using the Shell with a Document Store

Thus far in the book, we have discovered MySQL Shell and seen how we can use it to replace the older MySQL client (`mysql`) and how we can use the shell to administer our database using traditional SQL commands. We also learned how to use the shell to develop our relational database code using the X DevAPI and even test it!

Now it is time to learn more about what the document store is and how we can begin to work with it. And, yes, you can do that with the shell too. The core concept being JavaScript Object Notation (JSON) documents. We will learn more about what JSON is and how the MySQL Document Store works. We'll also see several examples of how to incorporate JSON with your relational database. In the next chapter, we'll tackle building a NoSQL solution with JSON documents.

Let's jump in with a brief foray into terminology and an overview of the technology.

Overview

The origins of the MySQL Document Store lie in several technologies that are leveraged together to form the document store. Specifically, Oracle has combined a key, value mechanism with a new data type, a new programming library, and a new access mechanism to create what is now the document store. Not only does this allow us to use MySQL with a NoSQL interface, it also allows us to build hybrid solutions that leverage the stability and structure of relational data while adding the flexibility of JSON documents.

In this chapter, we will learn about how MySQL supports JSON documents including how to add, find, update, and remove data (commonly referred to as create, read, update, and delete respectfully or simply CRUD). We begin with more information about the concepts and technologies you will encounter throughout this and the next chapter.

We will then move on to learning more about the JSON data type and the JSON functions in the MySQL server. While this chapter focuses on using JSON with relational data, a firm foundation on how to use JSON is required to master the MySQL Document Store NoSQL interface (the X DevAPI).

There are several new concepts and technologies and associated jargon we will encounter when working with the document store and JSON in MySQL. In this section, we will see how these concepts and technologies explain what comprises the JSON data type and document store interface. Let's begin with most basic concept that JSON uses: key, value mechanisms.

Origins: Key, Value Mechanisms

Like most things in this world, nothing is truly new in the sense that it is completely original without some form of something that came before and thus are typically built from existing technologies applied in novel ways. *Key, value* mechanisms are a prime example of a base technology. We use the term, mechanism, because the use of the key allows you to access the value.

When we say key, value we mean there exists some tag (normally a string) that forms the key and each key is associated with a value. For example, "name": "Charlie" is an example where key (name) has a value (Charlie). While the values in a key, value store are normally short strings, values can be complex; numeric, alphanumeric, lists, or even nested key, value sets.

Key, value mechanisms are best known for being easy to use programmatically while still retaining readability. That is, with diligent use of whitespace, a complex nested key, value data structure can be read by humans. The following shows one example formatted in a manner like how some developers would format code.¹ As you can see, it is very easy to see what this set of key, values are storing; name, address, and phone numbers.

```
{ "name": {
  "first": "Charlie",
  "last": "Harrington"
}
```

¹Debates over how to format code have been known to turn into a religious fervor concerning spacing, but most agree to disagree where that first curly brace should appear – on the first line by itself, on the same line, or on the next line.


```

},
"address": {
  "street": "123 Main Street",
  "city": "melborne",
  "state": "California",
  "zip": "90125"
}
"phone_numbers": [
  "800-555-1212",
  "888-212-1234"
]
}
}

```

One example of a key, value mechanism (or storage) is Extensible Markup Language (XML), which has been around for some time. The following is a simple example of XML using the preceding data. It is the result of a SQL SELECT query with the output (rows) shown in XML format. Notice how XML uses tags like HTML (because it is derived from HTML) along with the key, value storage of the data. Here, the keys are <row>, <field> and the values the contents between the start and end tag symbols (<field> </field>).

```

<?xml version="1.0"?>
<resultset statement="select * from thermostat_model limit 1;"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
    <field name="model_id">acme123</field>
    <field name="brand">WeMakeItSoCo</field>
  </row>
</resultset>

```

There are systems designed around key, value mechanisms (called key, value or relational stores) such as the Semantic Web. In short, the Semantic Web is an attempt to leverage associations of data to describe things, events, etc. Sometimes the terms “relation store” or “triple store” are used to describe the types of storage systems employed. There are several forms of key, value mechanisms used in the Semantic Web including Resource Description Framework (RDF), Web Ontology Language (OWL),

and Extensible Markup Language (XML). There are other examples of key, value mechanisms but the one most pertinent to the document store is JSON.

Now let's look at another key component of the document store – the NoSQL interface starting with the programming library.

Application Programming Interface

Recall, an application programming interface (API), sometimes called simply a library or programming library, is a set of classes and methods that support operations for one or more capability. These capabilities, through the classes and methods, allow a programmer to use the classes and methods to perform various tasks.

In the case of the MySQL Document Store, we use the X Developer API (X DevAPI) to access the server through a set of classes and methods that provide connectivity to the server, abstractions of concepts such as collections, tables, SQL operations, and more. These combine to allow a NoSQL interface to the MySQL server.

NoSQL Interface

There are several sometimes-conflicting definitions (if not examples) of NoSQL. For the purposes of this book and MySQL in general, a NoSQL interface is an API that does not require the use of SQL statements to access data. The API itself provides the connection to the server as well as classes and methods for creating, retrieving, updating, and deleting data. We saw this paradigm in action in the last chapter.

It is at this point that you're wondering about how MySQL handles the hybrid option of using JSON documents with relational data. Simply, MySQL has been designed to permit storing and retrieving JSON documents in the relational data (via the SQL interface). That is, the server has been modified to handle the JSON document. There are also a set of functions that allow you to do all manner of things with the JSON data making it easy to use JSON via the SQL interface.

However, you can also use JSON documents via the NoSQL X DevAPI either through an SQL command or as a pure document store using the special classes and methods of the X DevAPI. We will see an overview of using JSON both ways in this chapter with a dive into using JSON documents via the NoSQL interface in the next chapter.

Document Store

A document store (also known as a document-oriented database) is a storage and retrieval system for managing semi-structured data (hence documents). Modern document store systems support a key, value construct such as those found in XML and JSON. Document store systems are therefore sometimes considered a subclass of key, value storage systems.

Document store systems are also commonly accessed by a NoSQL interface implemented as a programming interface (API) that permits developers to incorporate the storage and retrieval of documents in their programs without need of a third-party access mechanism (the API implements the access mechanism).

Indeed, the metadata that describes the data is embedded with the data itself. Roughly, this means the keys and the layout (arrangement or nesting) of the keys form the metadata and the metadata becomes opaque to the storage mechanism. More specifically, how the data is arranged (how the document is formed or describes the data) is not reflected in or managed by the storage mechanism. Access to the semi-structured data requires accessing the mechanism designed to process the document itself using the NoSQL interface.

These two qualities, semi-structured data and NoSQL interfaces, are what separate document stores from relational data. Relational data requires structure that is not flexible, forcing all data to conform to a specific structure. Data is also grouped together with the same structure and there is often little allowance for data that can vary in content. Thus, we don't normally see document store accessible via traditional relational data mechanism. That is, until now.

One thing that is interesting about working with the document store is you don't need to be an expert on JavaScript or Python to learn how to work with the Document Store. Indeed, most of what you will do doesn't require mastery of any programming language. That is, there are plenty of examples of how to do things so you need not learn all that there is to know about the language to get started. In fact, you can pick up what you need very quickly and then learn more about the language as your needs mature.

JSON

JSON is a human and machine readable text-based data exchange format. It is also platform independent meaning there are no concepts of the format that prohibit it from being used in almost any programming language. In addition, JSON is a widely popular format used on the Internet.

JSON allows you to describe data in any way you want to without violating any structure. In fact, you can format (layout) your data any way you want to. The only real restriction is the proper use of the descriptors (curly braces, square brackets, quotes, commas, and the like) that must be aligned and in some cases paired correctly. The following is an example of a valid JSON string.

```
{
  "address": {
    "street": "123 First Street",
    "city": "Oxnard",
    "state": "CA",
    "zip": "90122"
  }
}
```

If you're thinking that looks a lot like the key, value example previously, you're right – it is! That is no mistake given how JSON was formed. However, we often use the term, string, to talk about JSON and indeed sometimes we see JSON represented without spaces and newlines shown as follows. It turns out most programming language JSON mechanisms can interpret the spaces and newlines correctly. We will see more about that in a later section.

```
{"address": {"street": "123 First Street", "city": "Oxnard", "state": "CA",
"zip": "90122"}}
```

When supported in programming languages, developers can easily read the data by accessing it via the keys. Better still, developers don't need to know what the keys are (but it helps!) because they can use the language support mechanisms to get the keys and iterate over them. In this way, like XML, the data is self-describing.

Now, let's dive into what JSON documents are and how we can use them with MySQL.

Introducing JSON Documents in MySQL

In MySQL 5.7.8 and beyond, we can use the JSON data type to store a JSON document in a field (columns) in a row stored in a traditional relational database table. Some may attempt (and succeed) at storing JSON in a blob or text field. While this is possible, there are several very good reasons not to do it. The most compelling reason is it requires the

application to do all the heavy lifting of reading and writing the JSON document thereby making it more complex and potentially error-prone. The JSON data type overcomes this problem in two big ways.

- *Validation*: The JSON data type provides document validation. That is, only valid JSON can be stored in a JSON column.
- *Efficient Access*: When a JSON document is stored in a table, the storage engine packs the data into a special optimized binary format allowing the server fast access to the data elements rather than parsing the data each time it is accessed.

This opens a whole new avenue for storing unstructured data in a structured form (relational data). However, Oracle didn't stop with simply adding a JSON data type to MySQL. Oracle also added a sophisticated programming interface as well as the concept of storing documents as collections in the database. We'll see more about these aspects in the next chapter. For this chapter, we will see how to use JSON with relational data.

Quick Start

If you have never worked with JSON before, this section will help get you started. There are only a few things you should learn about JSON and its use with MySQL, but first and foremost is the JSON formatting rules.

JSON is formed using strings bracketed or organized using certain symbols. While we have been discussing key, value mechanisms as they relate to JSON, there are two types of JSON attributes: arrays formed by a comma separated list and objects formed from a set of key, value pairs. You can also nest JSON attributes. For example, an array can contain objects and values in object keys can contain arrays or other objects. The combination of JSON arrays and objects is called a JSON document.

A JSON array contains a list of values separated by commas and enclosed within square brackets ([]). For example, the following are valid JSON arrays.

```
["Cub Cadet", "Troy-Bilt", "John Deere", "Craftsman"]
[33,67,1,55,909]
[True, True, False, False]
```

Notice we started and ended the array with square brackets and used a comma to separate the values. While we did not use whitespace, you can use whitespace and, depending on your programming language, you may be able to also use newlines, tabs, and carriage returns. For example, the following is still a valid JSON array.

```
[
True,
12,
False,
33
]
```

A JSON object is a set of key, value pairs where each key, value pair is enclosed within open and close curly braces (`{ }`) and separated by commas. For example, the following are valid JSON objects. Notice the key `address` has a JSON object as its value.

```
{"address": {
  "street": "123 First Street",
  "city": "Oxnard",
  "state": "CA",
  "zip": "90122"
}}
```

```
{"address": {
  "street": "4 Main Street",
  "city": "Melborne",
  "state": "California",
  "zip": "90125"
}}
```

JSON arrays are typically used to contain lists of related (well, sometimes) things, and JSON objects are used to describe complex data. JSON arrays and objects can contain scalar values such as strings or numbers, the `null` literal (just like in relational data), or Boolean literals `true` and `false`. Keep in mind that keys must always be strings and are commonly enclosed in quotes. Finally, JSON values can also contain time information (date, time, or datetime). For example, the following shows a JSON array with time values.

```
["15:10:22.021100", "2019-03-23", "2019-03-23 08:51:29.012310"]
```

The next section describes how we can use JSON in MySQL. In this case, we are referring to relational data but the formatting of JSON documents is the same in the document store.

Combining SQL and JSON

The use of JSON with relational data may seem a bit unusual or counter-intuitive. That is, why use unstructured data in a column? Doesn't that violate some relational database theory law or something?² While that may be true to some extent, the ability to add unstructured data within our relational data opens several doors previously closed to use.

For example, suppose you need to add more data to an existing table for an application that has been deployed for some time. If you add a new column, you run the risk of requiring all applications that use the data to be modified.³ Furthermore, suppose this data you need to add isn't available for every row. No big deal so far, yes? But what if this data varies in both type and scope, that is, for any given set of rows, the data added cannot be described in the same way or the data is different for each row? This is the nature of unstructured data – it has no pre-defined structure. Thus, you cannot easily extend the table or even create a new reference (child) table.

This is where having a JSON column can help. You simply add a new column and store the unstructured data as JSON. Of course, this does not alter the possibility that you must change the applications, but it does mean you won't have to retool the database itself (beyond adding the JSON column) or force-fit the data into a set of typed columns.

In this section, we will see how to work with JSON in MySQL including the mechanics of including JSON strings in our SQL statements, some of the special functions available for use with JSON in MySQL, how to access parts of a JSON document in your SQL statements, and prepare you for using JSON columns in your relational data. We will use the shell to demonstrate each of these topics. We save working with pure JSON documents for the next chapter.

²Some would say it does.

³There are clever ways to avoid this, but for this argument assume it isn't possible.

Tip MySQL Shell has several enhancements for working with JSON including the ability to display JSON in a human readable form. We can see results as either raw JSON (`json/raw`) or pretty-printed JSON (`json`). Use the `--result-format=json` command line option for pretty-printed JSON or the `--result-format=json/raw` command line option for unformatted JSON output.

Formatting JSON Strings in MySQL

When used in MySQL, JSON documents are written as strings. MySQL parses any string used in a JSON data type validating the document. If the document is not valid (not a properly formed JSON document), you will get an error. You can use JSON documents in any SQL statement where it is appropriate, such as `INSERT` and `UPDATE` statements as well as in clauses like the `WHERE` clause.

Tip Properly formatting JSON documents can be a bit of a challenge. The things to remember most is to balance your quotes, use commas correctly, and balance all curly braces and square brackets.

When you specify keys and values as strings, you must use the double quote character (`"`), not the single quote (`'`). Since MySQL expects JSON documents as strings, you can use the single quote around the entire JSON document but not within the document yourself. Fortunately, MySQL provides a host of special functions that you can use with JSON documents, one of which is the `JSON_VALID()` function that permits you to check a JSON document for validity. It returns a `1` if the document is valid and a `0` if it is not. The following shows the results of attempting to validate a JSON document with single quotes for the keys and values vs. a properly formatted JSON document with double quotes.

Note Henceforth, we will omit the shell SQL prompt for brevity.

```
> SELECT JSON_VALID('{vendor': {'name': 'Craftsman',
'URL': 'http://www.craftsman.com', 'sources': 'Lowe's'}}') AS IS_VALID \G
***** 1. ROW *****
IS_VALID: 0
1 row in set (0.0005 sec)

> SELECT JSON_VALID('{"vendor": {"name": "Craftsman",
"URL": "http://www.craftsman.com", "sources": "Lowe's"}}') AS IS_VALID \G
***** 1. ROW *****
IS_VALID: 1
1 row in set (0.0040 sec)
```

Notice the string with the double quotes inside is valid (the function returned a 1) but not the one with single quotes (the function returned a 0). This is what most people stumble over most often when working with JSON for the first time.

Using JSON Strings in SQL Statements

Let's look at how to use the JSON document in SQL statements. Suppose we wanted to store addresses in a table. For this example, we will keep it simple and insert the data in a very simple table. Listing 6-1 shows a transcript of the exercise starting with creating a test table then inserting the first two addresses.

Listing 6-1. Using JSON with SQL Statements

```
C:\Users\cbell> mysqlsh --uri root@localhost:33060 --sql
MySQL Shell 8.0.16
...
> CREATE DATABASE `testdb_6`;
Query OK, 1 row affected (0.0098 sec)

> USE `testdb_6`;
Query OK, 0 rows affected (0.0010 sec)

> CREATE TABLE `testdb_6`.`addresses` (`id` int(11) NOT NULL AUTO_
INCREMENT, `address` json DEFAULT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB
DEFAULT CHARSET=latin1;
Query OK, 0 rows affected (0.0310 sec)
```

```
> INSERT INTO `testdb_6`.`addresses` VALUES (NULL, '{"address": {"street":
"123 Second St", "city": "Maynard", "state": "CT", "zip": "19023"}}');
Query OK, 1 row affected (0.0042 sec)
```

```
> INSERT INTO `testdb_6`.`addresses` VALUES (NULL, '{"address":
{"street": "41 West Hanover", "city": "Frederick", "state": "Maryland", "z
ip": "20445"}}');
Query OK, 1 row affected (0.0030 sec)
```

```
> SELECT * FROM `testdb_6`.`addresses` \G
***** 1. ROW *****
      id: 1
address: {"address": {"zip": "19023", "city": "Maynard", "state": "CT",
"street": "123 Second St"}}
***** 2. ROW *****
      id: 2
address: {"address": {"zip": "20445", "city": "Frederick", "state":
"Maryland", "street": "41 West Hanover"}}
2 rows in set (0.0005 sec)
```

```
> DROP DATABASE `testdb_6`;
Query OK, 1 row affected (0.0132 sec)
```

Notice in the CREATE statement we used the data type JSON. This signals MySQL to allocate special storage mechanisms in the storage engine for handling JSON. Contrary to some reports, the JSON data type is not simply direct storage of a string. On the contrary, it is organized internally to optimize retrieval of the elements. Thus, it is very important that the JSON be formatted correctly. You can have multiple JSON columns in a table. However, the sum of the JSON documents in a table row is limited to the value of the variable `max_allowed_packet`.

Note JSON columns cannot have a default value like other columns.

Now, let's see what happens if we use an invalid JSON document (string) in the SQL statement. The following shows an attempt to insert the last address from the previous example only without the correct quotes around the keys. Notice the error thrown.

```
> INSERT INTO testdb_6.addresses VALUES (NULL, '{"address": {street:"173
Caroline Ave",city:"Monstrose",state:"Georgia",zip:31505}}');
ERROR: 3140: Invalid JSON text: "Missing a name for object member." at
position 13 in value for column 'addresses.address'.
```

You can expect to see errors like this and others for any JSON document that isn't formatted correctly. If you want to test your JSON first, use the `JSON_VALID()` function. However, there are two other functions that may also be helpful when building JSON documents; `JSON_ARRAY()` and `JSON_OBJECT()`.

The `JSON_ARRAY()` function takes a list of values and returns a valid formatted JSON array. The following shows an example. Notice it returned a correctly formatted JSON array complete with correct quotes (double instead of single) and the square brackets.

```
> SELECT JSON_ARRAY(1, true, 'test', 2.4);
+-----+
| JSON_ARRAY(1, true, 'test', 2.4) -----|
+-----+
| [1, true, "test", 2.4] -----          |
+-----+
1 row in set (0.00 sec)
```

The `JSON_OBJECT()` function takes a list of key, value pairs and returns a valid JSON object. The following shows an example. Notice here I used single quotes in calling the function. This is just one example where we can become confused with which quotes to use. In this case, the parameters for the function are not JSON documents; they're normal SQL strings, which can use single or double quotes.

```
> SELECT JSON_OBJECT("street","4 Main Street","city","Melborne",'state',
'California','zip',90125) \G
***** 1. ROW *****
JSON_OBJECT("street","4 Main Street","city","Melborne",'state','California',
'zip',90125): {"zip": 90125, "city": "Melborne", "state": "California",
"street": "4 Main Street"}
1 row in set (0.0040 sec)
```

Notice once again the automatic conversion of the quotes in the function result. This can be helpful if you need to build JSON on the fly (dynamically).

There is one other useful function for constructing JSON documents; the `JSON_TYPE()` function. This function takes a JSON document and parse it into a JSON value. It returns the value's JSON type if it is valid or throws an error if it is not valid. The following shows use of this function with the preceding statements.

```
> SELECT JSON_TYPE('[1, true, "test", 2.4]');
+-----+
| JSON_TYPE('[1, true, "test", 2.4]') |
+-----+
| ARRAY |
+-----+
1 row in set (0.00 sec)

> SELECT JSON_TYPE('{"zip": 90125, "city": "Melborne",
"state": "California", "street": "4 Main Street"}') \G
***** 1. ROW *****
JSON_TYPE('{"zip": 90125, "city": "Melborne", "state": "California",
"street": "4 Main Street"}'): OBJECT
1 row in set (0.00 sec)
```

There are more functions that MySQL provides to work with the JSON data type. We will see more about these in a later section.

This section has described only the basics for using JSON with MySQL in SQL statements. In fact, the formatting of the JSON document also applies to the document store. However, there is one thing we have't talked about yet – how to access the elements in a JSON document.

Path Expressions

To access an element – via its key – we use special notation called path expressions. The following shows a simple example. Notice the WHERE clause. This shows a path expression where I check to see if the address column includes the JSON key “city” referenced with the special notation `address->'$.address.city'`. We will see more details about path expressions in the next section.

```
> SELECT id, address->'$.address.city' FROM test.addresses
WHERE address->'$.address.zip' = '90125';
```

```
+-----+-----+
| id | address->'$.address.city' |
+-----+-----+
|  2 | "Melborne"                |
+-----+-----+
1 row in set (0.00 sec)
```

If you consider that a JSON document can be a complex set of semi-structured data and that at some point you will need to access certain elements in the document, you may also be wondering how to go about getting what you want from the JSON document. Fortunately, there is a mechanism to do this and it is called a path expression. More specifically, it is shortcut notation that you can use in your SQL commands to get an element without additional programming or scripting.

As you will see, it is a very specific syntax that, while not very expressive (it doesn't read well in English), the notation can get you what you need without a lot of extra typing. Path expressions are initiated with the dollar sign symbol (\$) enclosed in a string. But this notation must have a context. When using path expressions in SQL statements, you must use the `JSON_EXTRACT()` function, which allows you to use a path expression to extract data from a JSON document. This is because, unlike the X DevAPI classes and methods, path expressions are not directly supported in all SQL statements (but are for some as we will see). For example, if you wanted the third item in an array (in the example, the number 3), you would use the function as follows.

```
> SELECT JSON_EXTRACT('[1,2,3,4,5,6]', '$[2]') \G
***** 1. ROW *****
JSON_EXTRACT('[1,2,3,4,5,6]', '$[2]'): 3
1 row in set (0.0049 sec)
```

Notice this accesses data a JSON array. Here we use an array subscript with square brackets around the index (elements start at 0) as you would for an array in many programming languages.

Tip The use of path expressions in the SQL interface is limited to either one of the JSON functions or used only in specific clauses that have been modified to accept path expressions such as `SELECT` column lists or `WHERE`, `HAVING`, `ORDER BY`, or `GROUP BY` clauses.

Now suppose you wanted to access an element by key. You can do that too. In this case, we use the dollar sign followed by a period then the key name. The following shows how to retrieve the last name for a JSON object containing the name and address of an individual.

```
> SELECT JSON_EXTRACT('{ "name": { "first": "Billy-bob", "last": "Throckmutton" },
"address": { "street": "4 Main Street", "city": "Melborne", "state": "California",
"zip": "90125" } }', '$.name.first') AS Name \G
***** 1. ROW *****
Name: "Billy-bob"
1 row in set (0.0008 sec)
```

Notice I had to use two levels of access. That is, I wanted the value for the key named `first` from the object named `name`. Hence, I used `$.name.first`. This demonstrates how to use path expressions to drill down into the JSON document. This is also why we call this a path expression because the way we form the expression gives us the “path” to the element.

Now that we’ve seen a few examples, let’s review the entire syntax for path expressions; both for use in SQL and the NoSQL interfaces. Unless otherwise stated, the syntax aspects apply to both interfaces.

Once again, a path expression starts with the dollar sign and can optionally be followed by several forms of syntax called selectors that allow us to request a part of the document. These selectors include the following.

- A period followed by the name of a key name references the value for that key. The key name must be specified within double quotation marks if the name without quotes is not valid (it requires quotes to be a valid identifier such as a key name with a space).
- Use square brackets with an integer index (`[n]`) to select an element in an array. Indexes start at 0.
- Paths can contain the wildcards `*` or `**` as follows.
 - `.[*]` evaluates to the values of all members in a JSON object.
 - `[*]` evaluates to the values of all elements in a JSON array.
 - A sequence such as `prefix**suffix` evaluates to all paths that begin with the named prefix and end with the named suffix.

- Paths can be nested using a period as the separator. In this case, the path after the period is evaluated within the context of the parent path context. For example, \$.name.first limits the search for a key named first to the name JSON object.

If a path expression is evaluated as false or fails to locate a data item, the server will return null. For example, the following returns null because there are only 6 items in the array. Can you see why? Remember, counting starts at 0. This is a common mistake for those new to using path expressions (or arrays in programming languages).

```
> SELECT JSON_EXTRACT('[1,2,3,4,5,6]', '$[6]') \G
***** 1. ROW *****
JSON_EXTRACT('[1,2,3,4,5,6]', '$[6]'): NULL
1 row in set (0.0008 sec)
```

But wait, there's one more nifty option for path expressions. We can use a shortcut! That is, the dash and greater than symbol (->) can be used in place of the JSON_EXTRACT() function when accessing data in SQL statements by column. How cool is that? The use of the -> operation is sometimes called an “inline path expression”. For example, we could have written the preceding example to find the third item in a JSON array from a table as follows.

```
> CREATE TABLE testdb_6.ex1 (id int AUTO_INCREMENT PRIMARY KEY,
recorded_data JSON);
Query OK, 0 rows affected (0.0405 sec)
> INSERT INTO testdb_6.ex1 VALUES (NULL, JSON_ARRAY(1,2,3,4,5,6));
Query OK, 1 row affected (0.0052 sec)
> INSERT INTO testdb_6.ex1 VALUES (NULL, JSON_ARRAY(7,8,9));
> SELECT * FROM testdb_6.ex1 WHERE recorded_data->'$[2]' = 3 \G
***** 1. ROW *****
      id: 1
recorded_data: [1, 2, 3, 4, 5, 6]
1 row in set (0.0045 sec)
```

Notice I simply used the column name, recorded_data, and appended the -> to the end then listed the path expression. Brilliant!

But wait, there's more. There is one other form of this shortcut. If the result of the `->` operation (`JSON_EXTRACT`) evaluates to a quoted string, we can use the `->>` symbol (called the inline path operator) to retrieve the value without quotes. This is helpful when dealing with values that are numbers. The following shows two examples. One with the `->` operation and the same with the `->>` operation.

```
> INSERT INTO testdb_6.ex1 VALUES (NULL, '{"name":"will","age":"43"}');
Query OK, 1 row affected (0.00 sec)
> INSERT INTO testdb_6.ex1 VALUES (NULL, '{"name":"joseph","age":"11"}');
Query OK, 1 row affected (0.00 sec)
> SELECT * FROM testdb_6.ex1 WHERE recorded_data->'$.age' = 43 \G
***** 1. ROW *****
      id: 3
recorded_data: {"age": "43", "name": "will"}
1 row in set (0.0014 sec)
> SELECT * FROM testdb_6.ex1 WHERE recorded_data->'$.age' = '43' \G
***** 1. ROW *****
      id: 3
recorded_data: {"age": "43", "name": "will"}
1 row in set (0.0009 sec)
```

Notice the `recorded_data` values (age and name) were stored as a string. But what if the data were stored as an integer? Observe.

```
> INSERT INTO testdb_6.ex1 VALUES (NULL, '{"name":"amy","age":22}');
Query OK, 1 row affected (0.0075 sec)
> SELECT * FROM testdb_6.ex1 WHERE recorded_data->'$.age' = 22 \G
***** 1. ROW *****
      id: 5
recorded_data: {"age": 22, "name": "amy"}
1 row in set (0.0010 sec)
> SELECT * FROM testdb_6.ex1 WHERE recorded_data->>'$.age' = 22 \G
***** 1. ROW *****
      id: 5
recorded_data: {"age": 22, "name": "amy"}
1 row in set (0.0009 sec)
```


Aha! So, the `->>` operation is most useful when values must be unquoted. If they are already unquoted (such as an integer), the `->>` operation returns the same as the `->` operation.

Please note that the use of the shortcuts (inline path expressions) is not a direct replacement for the `JSON_EXTRACT()` function. The following summarizes the limitations.

- *Data Source*: When used in a SQL statement, the inline path expression uses the field (column) specified only. The function can use any JSON-typed value.
- *Path Expression String*: An inline path expression must use a plain string. The function can use any string-typed value.
- *Number of Expressions*: An inline path expression can use only one path expression against a single field (column). The function can use multiple path expressions against a JSON document.

Now let's look at the various JSON functions that we can use to work with JSON documents.

JSON Functions

There are many functions for working with JSON in MySQL. Rather than list all the functions and risk obsolescence (new ones seem to be added with every release), we will list some of the most frequently used functions to give you an idea of what is available. While we won't explore nuance of every function, we will see some of these in use in later sections. Table 6-1 lists the JSON functions available in MySQL 8.

Mastery of these functions is not essential to working with the document store but can help greatly when developing hybrid solutions where you use JSON in SQL statements.

These functions can be grouped into categories based on how they are used. We will see functions useful for adding data, those for retrieving (searching) data, and more. The following show how to use the functions using brief examples.

Table 6-1. *Commonly Used JSON Functions in MySQL*

Function	Description and Use
JSON_ARRAY()	Evaluates a list of values and returns a JSON array containing those values.
JSON_ARRAYAGG()	Aggregates a result set as a single JSON array whose elements consist of the rows.
JSON_ARRAY_APPEND()	Appends values to the end of the indicated arrays within a JSON document and returns the result.
JSON_ARRAY_INSERT()	Updates a JSON document, inserting into an array within the document and returning the modified document.
JSON_CONTAINS()	Returns 0 or 1 to indicate whether a specific value is contained in a target JSON document, or, if a path argument is given, at a specific path within the target document.
JSON_CONTAINS_PATH()	Returns 0 or 1 to indicate whether a JSON document contains data at a given path or paths.
JSON_DEPTH()	Returns the maximum depth of a JSON document.
JSON_EXTRACT()	Returns data from a JSON document, selected from the parts of the document matched by the path arguments.
JSON_INSERT()	Inserts data into a JSON document and returns the result.
JSON_KEYS()	Returns the keys from the top-level value of a JSON object as a JSON array, or, if a path argument is given, the top-level keys from the selected path.
JSON_LENGTH()	Returns the length of JSON document, or, if a path argument is given, the length of the value within the document identified by the path.
JSON_OBJECT()	Evaluates a list of key/value pairs and returns a JSON object containing those pairs.
JSON_OBJECTAGG()	Takes two column names or expressions as arguments, the first of these being used as a key and the second as a value, and returns a JSON object containing key/value pairs.
JSON_PRETTY()	Print a nicer looking layout of the JSON document.

(continued)

Table 6-1. (continued)

Function	Description and Use
JSON_QUOTE()	Quotes a string as a JSON value by wrapping it with double quote characters and escaping interior quote and other characters, then returning the result as a utf8mb4 string.
JSON_REMOVE()	Removes data from a JSON document and returns the result.
JSON_REPLACE()	Replaces existing values in a JSON document and returns the result.
JSON_SEARCH()	Returns the path to the given string within a JSON document.
JSON_SET()	Inserts or updates data in a JSON document and returns the result.
JSON_TABLE()	Extracts data from a JSON document and returns it as a relational table.
JSON_TYPE()	Returns a utf8mb4 string indicating the type of a JSON value.
JSON_VALID()	Returns 0 or 1 to indicate whether a value is a valid JSON document.

Creating JSON Data

There are several useful functions for creating JSON data. We have already seen two important functions: `JSON_ARRAY()` that builds a JSON array type and `JSON_OBJECT()` that builds a JSON object type. This section discusses some of the other functions that you can use to help create JSON documents including functions for aggregating, appending, and inserting data in JSON arrays.

The `JSON_ARRAYAGG()` function is used to create an array of JSON documents from several rows. It can be helpful when you want to summarize data or combine data from several rows. The function takes a column name and combines the JSON data from the rows into a new array. Listing 6-2 shows examples of using the function. This example takes the rows in the table and combines them to form a new array of JSON objects.

Listing 6-2. Using the `JSON_ARRAYAGG` Function

```
> CREATE TABLE testdb_6.favorites (id int(11) NOT NULL AUTO_INCREMENT,
preferences JSON, PRIMARY KEY (`id`));
> INSERT INTO testdb_6.favorites VALUES (NULL, '{"color": "red"}');
Query OK, 1 row affected (0.0077 sec)
```

```

> INSERT INTO testdb_6.favorites VALUES (NULL, '{"color": "blue"}');
Query OK, 1 row affected (0.0050 sec)
> INSERT INTO testdb_6.favorites VALUES (NULL, '{"color": "purple"}');
Query OK, 1 row affected (0.0034 sec)
> SELECT * FROM testdb_6.favorites \G
***** 1. ROW *****
      id: 1
preferences: {"color": "red"}
***** 2. ROW *****
      id: 2
preferences: {"color": "blue"}
***** 3. ROW *****
      id: 3
preferences: {"color": "purple"}
3 rows in set (0.0012 sec)
> SELECT JSON_ARRAYAGG(preferences) FROM testdb_6.favorites \G
***** 1. ROW *****
JSON_ARRAYAGG(preferences): [{"color": "red"}, {"color": "blue"},
{"color": "purple"}]
1 row in set (0.0049 sec)

```

The `JSON_ARRAY_APPEND()` is an interesting function that allows you to append data to a JSON array either at the end or immediately after a given path expression. The function takes as parameters a JSON array, a path expression, and the value (including a JSON document) to be inserted. Listing 6-3 shows several examples.

Listing 6-3. Using the `JSON_ARRAY_APPEND` Function

```

> SET @base = '["apple","pear",{"grape":"red"},"strawberry"]';
Query OK, 0 rows affected (0.0045 sec)
> SELECT JSON_ARRAY_APPEND(@base, '$', "banana") \G
***** 1. ROW *****
JSON_ARRAY_APPEND(@base, '$', "banana"): ["apple", "pear",
{"grape": "red"}, "strawberry", "banana"]
1 row in set (0.0009 sec)
> SELECT JSON_ARRAY_APPEND(@base, '$[2].grape', "green") \G

```

```

***** 1. IOW *****
JSON_ARRAY_APPEND(@base, '$[2].grape', "green"): ["apple", "pear",
{"grape": ["red", "green"]}, "strawberry"]
1 row in set (0.0012 sec)
> SET @base = '{"grape":"red"}';
Query OK, 0 rows affected (0.0004 sec)
> SELECT JSON_ARRAY_APPEND(@base, '$', '{"grape":"red"}') \G
***** 1. IOW *****
JSON_ARRAY_APPEND(@base, '$', '{"grape":"red"}'): [{"grape": "red"},
{"\grape\":"red\"}]
1 row in set (0.0007 sec)

```

Notice the first example simply adds a new value to the end of the array. The second example changes the value of the key in the JSON object in the third index to an array and adds a new value. This is an interesting by-product of this function. We see this again on the third example where we change a basic JSON object to a JSON array of JSON objects.

The `JSON_ARRAY_INSERT()` function is similar except it inserts the value before the path expression. The function takes as parameters a JSON array, a path expression, and the value (including a JSON document) to be inserted. When including multiple path expression and value pairs, the effect is cumulative where the function evaluates the first path expression and value applying the next pair to the result, and so on. Listing 6-4 shows some examples using the new function that are like the previous examples. Notice that the positions of the data inserted is before the path expression.

Listing 6-4. Using the `JSON_ARRAY_INSERT` Function

```

> SET @base = '["apple","pear",{"grape":["red","green"]},"strawberry"]';
Query OK, 0 rows affected (0.0007 sec)
> SELECT JSON_ARRAY_INSERT(@base, '$[0]', "banana") \G
***** 1. IOW *****
JSON_ARRAY_INSERT(@base, '$[0]', "banana"): ["banana", "apple", "pear",
{"grape": ["red", "green"]}, "strawberry"]
1 row in set (0.0008 sec)
> SELECT JSON_ARRAY_INSERT(@base, '$[2].grape[0]', "white") \G

```

```

***** 1. IOW *****
JSON_ARRAY_INSERT(@base, '$[2].grape[0]', "white"): ["apple", "pear",
{"grape": ["white", "red", "green"]}, "strawberry"]
1 row in set (0.0009 sec)
> SET @base = '["grape":"red"]';
Query OK, 0 rows affected (0.0004 sec)
> SELECT JSON_ARRAY_INSERT(@base, '$[0]', '{"grape":"red"}') \G
***** 1. IOW *****
JSON_ARRAY_INSERT(@base, '$[0]', '{"grape":"red"}'):
["{"grape\":\"red\"}", {"grape": "red"}]
1 row in set (0.0007 sec)

```

The `JSON_INSERT()` function is designed to take a JSON document and inserts one or more values at a specified path expression. That is, you can pass pairs of path expression and value at one time. But there is a catch. The path expression in this case must not evaluate to an element in the document. Like the last function, when including multiple path expressions, the effect is cumulative where the function evaluates the first path expression applying the next path expression to the result, and so on. Listing 6-5 shows an example. Notice the third path expression and value is not inserted because the path expression, `$$[0]`, evaluates to the first element, apple.

Listing 6-5. Using the `JSON_INSERT` Function

```

> SET @base = '["apple","pear",{"grape":["red","green"]},"strawberry"]';
Query OK, 0 rows affected (0.0007 sec)
> SELECT JSON_INSERT(@base, '$[9]', "banana", '$[2].grape[3]', "white",
'$[0]', "orange") \G
***** 1. IOW *****
JSON_INSERT(@base, '$[9]', "banana", '$[2].grape[3]', "white", '$[0]', "orange"):
["apple", "pear", {"grape": ["red", "green", "white"]}, "strawberry", "banana"]
1 row in set (0.0008 sec)

```

The `JSON_MERGE_PATCH()` and `JSON_MERGE_PRESERVE()` functions are designed to take two or more JSON documents and combine them. The `JSON_MERGE_PATH()` function replaces values for duplicate keys while the `JSON_MERGE_PRESERVE()` preserves the values for duplicate keys. Like the last function, you can include as many JSON documents as you want. Notice how I used this function to build the example JSON document from the earlier examples. Listing 6-6 shows an example using the methods.

Listing 6-6. Using the JSON_MERGE_PATCH and JSON_MERGE_PRESERVE Functions

```
> SELECT JSON_MERGE_PATCH('["apple","pear"]', '{"grape":["red","green"]}',
'["strawberry"]') \G
***** 1. ROW *****
JSON_MERGE_PATCH('["apple","pear"]', '{"grape":["red","green"]}',
'["strawberry"]'): ["strawberry"]
1 row in set (0.0041 sec)
> SELECT JSON_MERGE_PRESERVE('{"grape":["red","green"]}', '{"grape":["white"]}') \G
***** 1. ROW *****
JSON_MERGE_PRESERVE('{"grape":["red","green"]}', '{"grape":["white"]}'):
{"grape": ["red", "green", "white"]}
1 row in set (0.0008 sec)
```

If any JSON function is passed an invalid parameter, invalid JSON document, or the path expression does not find an element, some functions return null while others may return the original JSON document. Listing 6-7 shows an example. In this case, there is no element at position 8 because the array only has 4 elements.

Listing 6-7. Using the JSON_ARRAY_APPEND Function

```
> SET @base = '["apple","pear",{"grape":"red"},"strawberry"]' \G
Query OK, 0 rows affected (0.0007 sec)
> SELECT JSON_ARRAY_APPEND(@base, '$[7]', "flesh") \G
***** 1. ROW *****
JSON_ARRAY_APPEND(@base, '$[7]', "flesh"): ["apple", "pear", {"grape":
"red"}, "strawberry"]
1 row in set (0.0007 sec)
```

Now let's see functions that we can use to modify JSON data.

Modifying JSON Data

There are several useful functions for modifying JSON data. This section discusses functions that you can use to help modify JSON documents by removing, replacing, and updating elements in the JSON document.

The `JSON_REMOVE()` function is used to remove elements that match a path expression. You must provide the JSON document to operate on and one or more path expressions and the result will be the JSON document with the elements removed. When including multiple path expressions, the effect is cumulative where the function evaluates the first path expression applying the next path expression to the result, and so on. Listing 6-8 shows an example. Notice I had to imagine what the intermediate results would be – that is, I used `$$[0]` three times because the function removed the first element twice leaving the JSON object as the first element.

Listing 6-8. Using the `JSON_REMOVE` Function (single)

```
> SET @base = '["apple","pear",{"grape":["red","white"]},"strawberry"]';
Query OK, 0 rows affected (0.0008 sec)
> SELECT JSON_REMOVE(@base, '$[0]', '$[0]', '$[0].grape[1]') \G
***** 1. ROW *****
JSON_REMOVE(@base, '$[0]', '$[0]', '$[0].grape[1]'): [{"grape": ["red"]},
"strawberry"]
1 row in set (0.0009 sec)
```

This may take a little getting used to but you can use the function multiple times or nested as shown in the examples in Listing 6-9.

Listing 6-9. Using the `JSON_REMOVE` Function (nested)

```
> SET @base = '["apple","pear",{"grape":["red","white"]},"strawberry"]';
Query OK, 0 rows affected (0.0007 sec)
> SET @base = JSON_REMOVE(@base, '$[0]');
Query OK, 0 rows affected (0.0009 sec)
> SET @base = JSON_REMOVE(@base, '$[0]');
Query OK, 0 rows affected (0.0006 sec)
> SELECT JSON_REMOVE(@base, '$[0].grape[1]') \G
***** 1. ROW *****
JSON_REMOVE(@base, '$[0].grape[1]'): [{"grape": ["red"]}, "strawberry"]
1 row in set (0.0007 sec)
> SET @base = '["apple","pear",{"grape":["red","white"]},"strawberry"]';
Query OK, 0 rows affected (0.0004 sec)
> SELECT JSON_REMOVE(JSON_REMOVE(JSON_REMOVE(@base, '$[0]'), '$[0]'),
'$[0].grape[1]') \G
```



```
***** 1. IOW *****
JSON_REMOVE(JSON_REMOVE(JSON_REMOVE(@base, '$[0]'), '$[0]'), '$[0].
grape[1]'): [{"grape": ["red"]}, "strawberry"]
1 row in set (0.0005 sec)
```

The `JSON_REPLACE()` function takes a JSON document and pairs of path expression and value replacing the element that matches the path expression with the new value. Once again, the results are cumulative and work left to right in order. There is a catch with this function too. It ignores any new values or path expressions that evaluate to new values. Listing 6-10 shows an example. Notice the third pair was not removed because there is no tenth element.

Listing 6-10. Using the `JSON_REPLACE` Function

```
> SET @base = '["apple","pear",{"grape":["red","white"]},"strawberry"]';
Query OK, 0 rows affected (0.0008 sec)
> SELECT JSON_REPLACE(@base, '$[0]', "orange", '$[2].grape[0]', "green",
'$[9]', "waffles") \G
***** 1. IOW *****
JSON_REPLACE(@base, '$[0]', "orange", '$[2].grape[0]', "green", '$[9]',
"waffles"): ["orange", "pear", {"grape": ["green", "white"]}, "strawberry"]
1 row in set (0.0040 sec)
```

The `JSON_SET()` function is designed to modify JSON document elements. Like the other functions, you pass a JSON document as the first parameter and then one or more pairs of path expression and value to replace. However, this function also inserts any elements that are not in the document (the path expression is not found). Listing 6-11 shows an example. Notice the last element did not exist so it adds it to the documents.

Listing 6-11. Using the `JSON_SET` Function

```
> SET @base = '["apple","pear",{"grape":["red","white"]},"strawberry"]';
Query OK, 0 rows affected (0.0007 sec)
> SELECT JSON_SET(@base, '$[0]', "orange", '$[2].grape[1]', "green",
'$[9]', "123") \G
```

```
***** 1. IOW *****
JSON_SET(@base, '$[0]', "orange", '$[2].grape[1]', "green", '$[9]', "123"):
["orange", "pear", {"grape": ["red", "green"]}, "strawberry", "123"]
1 row in set (0.0009 sec)
```

Now let's look at the JSON functions you can use to find elements in the document.

Searching JSON Data

Another important operation for working with SQL and JSON data is searching for data in the JSON document. We discovered earlier in the chapter how to reference data in the document with the special notation (path expressions), and we learned there are JSON functions that we can use to search for the data. In fact, we saw these two concepts used together in the previous section. In this section, we review the JSON data searching mechanism since you are likely to use these functions more than any other, especially in your queries.

There are four JSON functions that allow you to search JSON documents. Like the previous functions, these operate on a JSON document with one or more parameters. I call them searching functions not because they allow you to search a database or table for JSON data, but rather they allow you to find things in JSON documents. The functions include those for checking to see if a value or element exists in the document, whether a path expression is valid (something can be found using it), and retrieving information from the document.

The `JSON_CONTAINS()` function has two options: you can use it to return whether a value exists anywhere in the document or if a value exists using a path expression (the path expression is an optional parameter). The function returns a 0 or 1 where a 0 means the value was not found. An error occurs if either document argument is not a valid JSON document or the path argument is not a valid path expression or contains a * or ** wildcard. There is another catch. The value you pass in must be a valid JSON string or document. Listing 6-12 shows several examples of using the function to search a JSON document.

Listing 6-12. Using the `JSON_CONTAINS` Function

```
> SET @base = '{"grapes":["red","white","green"],"berries":["strawberry",
"raspberry","boysenberry","blackberry"]}';
Query OK, 0 rows affected (0.0007 sec)
> SELECT JSON_CONTAINS(@base, '["red","white","green"]') \G
```

```

***** 1. ROW *****
JSON_CONTAINS(@base, ["red", "white", "green"]): 0
1 row in set (0.0010 sec)
> SELECT JSON_CONTAINS(@base, '{"grapes":["red", "white", "green"]}')
```

```

***** 1. ROW *****
JSON_CONTAINS(@base, '{"grapes":["red", "white", "green"]}'): 1
1 row in set (0.0006 sec)
> SELECT JSON_CONTAINS(@base, ["red", "white", "green"], '$.grapes') \G
```

```

***** 1. ROW *****
JSON_CONTAINS(@base, ["red", "white", "green"], '$.grapes'): 1
1 row in set (0.0004 sec)
> SELECT JSON_CONTAINS(@base, "blackberry", '$.berries') \G
```

```

***** 1. ROW *****
JSON_CONTAINS(@base, "blackberry", '$.berries'): 0
1 row in set (0.0004 sec)
> SELECT JSON_CONTAINS(@base, 'blackberry', '$.berries') \G
ERROR: 3141: Invalid JSON text in argument 2 to function json_contains:
"Invalid value." at position 0.
> SELECT JSON_CONTAINS(@base, "red", '$.grapes') \G
```

```

***** 1. ROW *****
JSON_CONTAINS(@base, "red", '$.grapes'): 1
1 row in set (0.0004 sec)
```

As you can see, this is a very useful function but it requires a bit of care to use properly. That is, you must make sure the value is a valid string. In all examples save one, I am searching the JSON document for either a JSON document (that makes searching for nested data easier) or a single value using a path expression. Remember, the function searches for values, not keys.

Notice the second to last example. This returns an error because the value is not a valid JSON string. You must use double quotes around it to correct it as shown in the following example.

The `JSON_CONTAINS_PATH()` function uses a parameter strategy that is a little different. The function searches a JSON document to see if a path expression exists but it also allows you to find the first occurrence or all occurrences. It can also take multiple paths and evaluate them either as an “or” or “and” condition depending on what value you pass as the second parameter as follows.

- If you pass one, the function will return 1 if at least one path expression is found (OR).
- If you pass all, the function will return 1 only if all path expressions are found (AND).

The function returns 0 or 1 to indicate whether a JSON document contains data at a given path or paths. Note that it can return null if any of the path expressions or the document is null. An error occurs if the JSON document, or any path expression is not valid, or the second parameter is not one or all. Listing 6-13 shows several examples of using the function.

Listing 6-13. Using the JSON_CONTAINS_PATH Function

```
> SET @base = '{"grapes":["red","white","green"],"berries":["strawberry",
"raspberry","boysenberry","blackberry"],"numbers":["1","2","3","4","5"]}';
Query OK, 0 rows affected (0.0004 sec)
> SELECT JSON_CONTAINS_PATH(@base,'one','$') \G
***** 1. ROW *****
JSON_CONTAINS_PATH(@base,'one','$'): 1
1 row in set (0.0005 sec)
> SELECT JSON_CONTAINS_PATH(@base,'all','$') \G
***** 1. ROW *****
JSON_CONTAINS_PATH(@base,'all','$'): 1
1 row in set (0.0005 sec)
> SELECT JSON_CONTAINS_PATH(@base,'all','$.grapes','$.berries') \G
***** 1. ROW *****
JSON_CONTAINS_PATH(@base,'all','$.grapes','$.berries'): 1
1 row in set (0.0004 sec)
> SELECT JSON_CONTAINS_PATH(@base,'all','$.grapes','$.berries','$.numbers')\G
***** 1. ROW *****
JSON_CONTAINS_PATH(@base,'all','$.grapes','$.berries','$.numbers'): 1
1 row in set (0.0004 sec)
> SELECT JSON_CONTAINS_PATH(@base,'all','$.grapes','$.berries','$.num') \G
***** 1. ROW *****
JSON_CONTAINS_PATH(@base,'all','$.grapes','$.berries','$.num'): 0
1 row in set (0.0005 sec)
```

```

> SELECT JSON_CONTAINS_PATH(@base,'one','$.grapes','$.berries','$.num') \G
***** 1. ROW *****
JSON_CONTAINS_PATH(@base,'one','$.grapes','$.berries','$.num'): 1
1 row in set (0.0005 sec)
> SELECT JSON_CONTAINS_PATH(@base,'one','$.grapes') \G
***** 1. ROW *****
JSON_CONTAINS_PATH(@base,'one','$.grapes'): 1
1 row in set (0.0004 sec)
> SELECT JSON_CONTAINS_PATH(@base,'all','$.grape') \G
***** 1. ROW *****
JSON_CONTAINS_PATH(@base,'all','$.grape'): 0
1 row in set (0.0004 sec)
> SELECT JSON_CONTAINS_PATH(@base,'one','$.berries') \G
***** 1. ROW *****
JSON_CONTAINS_PATH(@base,'one','$.berries'): 1
1 row in set (0.0004 sec)
> SELECT JSON_CONTAINS_PATH(@base,'all','$.berries') \G
***** 1. ROW *****
JSON_CONTAINS_PATH(@base,'all','$.berries'): 1
1 row in set (0.0005 sec)

```

Take some time to look through these examples so you can see how they work. Notice in the first two examples, I used a path expression of a single dollar sign. This is simply the path expression to the entire document so naturally, it exists. Notice also the differences in the use of one or all for the last two examples.

The `JSON_EXTRACT()` function is one of the most used functions. It allows you to extract a value or JSON array or JSON object, etc. from a JSON document using one or more path expressions. We have already seen a couple of examples. Recall the function returns the portion of the JSON document that matches the path expression. Listing 6-14 shows a few more examples using a complex path expressions.

Listing 6-14. Using the `JSON_EXTRACT` Function

```

> SET @base = '{"grapes":["red","white","green"],"berries":["strawberry",
"raspberry","boysenberry","blackberry"],"numbers":["1","2","3","4","5"]}';
Query OK, 0 rows affected (0.0004 sec)
> SELECT JSON_EXTRACT(@base,'$') \G

```

```

***** 1. IOW *****
JSON_EXTRACT(@base,'$'): {"grapes": ["red", "white", "green"], "berries":
["strawberry", "raspberry", "boysenberry", "blackberry"], "numbers": ["1",
"2", "3", "4", "5"]}
1 row in set (0.0006 sec)
> SELECT JSON_EXTRACT(@base,'$.grapes') \G
***** 1. IOW *****
JSON_EXTRACT(@base,'$.grapes'): ["red", "white", "green"]
1 row in set (0.0005 sec)
> SELECT JSON_EXTRACT(@base,'$.grapes[*]') \G
***** 1. IOW *****
JSON_EXTRACT(@base,'$.grapes[*]'): ["red", "white", "green"]
1 row in set (0.0005 sec)
> SELECT JSON_EXTRACT(@base,'$.grapes[1]') \G
***** 1. IOW *****
JSON_EXTRACT(@base,'$.grapes[1]'): "white"
1 row in set (0.0005 sec)
> SELECT JSON_EXTRACT(@base,'$.grapes[4]') \G
***** 1. IOW *****
JSON_EXTRACT(@base,'$.grapes[4]'): NULL
1 row in set (0.0006 sec)
> SELECT JSON_EXTRACT(@base,'$.berries') \G
***** 1. IOW *****
JSON_EXTRACT(@base,'$.berries'): ["strawberry", "raspberry", "boysenberry",
"blackberry"]
1 row in set (0.0009 sec)
> SELECT JSON_EXTRACT(@base,'$.berries[2]') \G
***** 1. IOW *****
JSON_EXTRACT(@base,'$.berries[2]'): "boysenberry"
1 row in set (0.0005 sec)
> SELECT JSON_EXTRACT(@base,'$.berries[2]','$.berries[3]') \G
***** 1. IOW *****
JSON_EXTRACT(@base,'$.berries[2]','$.berries[3]'): ["boysenberry", "blackberry"]
1 row in set (0.0006 sec)

```

Notice what happens when we use the single dollar sign. The function returns the entire document. Also, notice what happens when we use a path expression that, although its syntax is valid, it does not evaluate to an element in the document (see the fifth example).

Notice also the last example where we pass in two path expressions. Notice how it returns a JSON array whereas the example before it with only one path expression returns a JSON string value. This is one of the trickier aspects of the function. So long as you remember it returns a valid JSON string, array, or object, you will be able to use the function without issue.

The `JSON_SEARCH()` function is interesting because it is the opposite of the `JSON_EXTRACT()` function. More specifically, it takes one or more values and returns path expressions to the values if they are found in the document. This makes it easier to validate your path expressions or to build path expressions on-the-fly.

Like the `JSON_CONTAINS_PATH()` function, the `JSON_SEARCH()` function also allows you to find the first occurrence or all occurrences returning the path expressions depending on what value you pass as the second parameter as follows.

- If you pass one, the function will return the first match.
- If you pass all, the function will return all matches.

But there is a trick here too. The function takes a third parameter that forms a special search string that works like the `LIKE` operator in SQL statements. That is, search string argument can use the `%` and `_` characters the same way as the `LIKE` operator. Note that to use a `%` or `_` as a literal, you must precede it with the `\` (escape) character.

The function returns 0 or 1 to indicate whether a JSON document contains the values. Note that it can return null if any of the path expressions or the document is null. An error occurs if the JSON document, or any path expression is not valid, or the second parameter is not one or all. Listing 6-15 shows several examples of using the function.

Listing 6-15. Using the `JSON_SEARCH` Function

```
> SET @base = '{"grapes":["red","white","green"],"berries":["strawberry",
"raspberry","boysenberry","blackberry"],"numbers":["1","2","3","4","5"]}';
Query OK, 0 rows affected (0.0011 sec)
> SELECT JSON_SEARCH(@base,'all','red') \G
```

```

***** 1. ROW *****
JSON_SEARCH(@base,'all','red'): "$.grapes[0]"
1 row in set (0.0006 sec)
> SELECT JSON_SEARCH(@base,'all','gr_____') \G
***** 1. ROW *****
JSON_SEARCH(@base,'all','gr_____'): NULL
1 row in set (0.0004 sec)
> SELECT JSON_SEARCH(@base,'one','%berry') \G
***** 1. ROW *****
JSON_SEARCH(@base,'one','%berry'): "$.berries[0]"
1 row in set (0.0005 sec)
> SELECT JSON_SEARCH(@base,'all','%berry') \G
***** 1. ROW *****
JSON_SEARCH(@base,'all','%berry'): ["$.berries[0]", "$.berries[1]",
"$.berries[2]", "$.berries[3]"]
1 row in set (0.0006 sec)

```

Now let's look at the last group of JSON functions; those that are utilitarian in nature allowing you to get information about the JSON document and perform simple operations to help work with JSON documents.

Utility Functions

Lastly, there are several functions that can return information about the JSON document, help add or remove quotes, and even find the keys in a document. We have already seen several of the utility `JSON_TYPE()` and `JSON_VALID()` functions. The following are additional utility functions you may find useful when working with JSON documents.

The `JSON_DEPTH()` function returns the maximum depth of a JSON document. If the document is an empty array, object, or a scalar value, the function returns a depth of 1. An array containing only elements of depth 1 or nonempty objects containing only member values of depth 1 returns a depth of 2. Listing 6-16 shows several examples.

Listing 6-16. Using the JSON_DEPTH Function

```

> SELECT JSON_DEPTH('8') \G
***** 1. ROW *****
JSON_DEPTH('8'): 1
1 row in set (0.0017 sec)
> SELECT JSON_DEPTH('[]') \G
***** 1. ROW *****
JSON_DEPTH('[]'): 1
1 row in set (0.0007 sec)
> SELECT JSON_DEPTH('{}') \G
***** 1. ROW *****
JSON_DEPTH('{}'): 1
1 row in set (0.0007 sec)
> SELECT JSON_DEPTH('[12,3,4,5,6]') \G
***** 1. ROW *****
JSON_DEPTH('[12,3,4,5,6]'): 2
1 row in set (0.0008 sec)
> SELECT JSON_DEPTH('[[], {}]') \G
***** 1. ROW *****
JSON_DEPTH('[[], {}]'): 2
1 row in set (0.0004 sec)
> SET @base = '{"grapes":["red","white","green"],"berries":["strawberry",
"raspberry","boysenberry","blackberry"],"numbers":["1","2","3","4","5"]}';
Query OK, 0 rows affected (0.0004 sec)
> SELECT JSON_DEPTH(@base) \G
***** 1. ROW *****
JSON_DEPTH(@base): 3
1 row in set (0.0004 sec)
> SELECT JSON_DEPTH(JSON_EXTRACT(@base, '$.grapes')) \G
***** 1. ROW *****
JSON_DEPTH(JSON_EXTRACT(@base, '$.grapes')): 2
1 row in set (0.0005 sec)

```

The `JSON_KEYS()` function is used to return a list of keys from the top-level value of a JSON object as a JSON array. The function also allows you to pass a path expression, which results in a list of the top-level keys from the selected path expression value. An error occurs if the `json_doc` argument is not a valid JSON document or the path argument is not a valid path expression or contains a `*` or `**` wildcard. The resulting array is empty if the selected object is empty.

There is one limitation. If the top-level value has nested JSON objects, the array returned does not include keys from those nested objects. Listing 6-17 shows several examples of using this function.

Listing 6-17. Using the `JSON_KEYS` Function

```
> SET @base = '{"grapes":["red","white","green"],"berries":["strawberry",
"raspberry","boysenberry","blackberry"],"numbers":["1","2","3","4","5"]}';
Query OK, 0 rows affected (0.0004 sec)
> SELECT JSON_KEYS(@base) \G
***** 1. IOW *****
JSON_KEYS(@base): ["grapes", "berries", "numbers"]
1 row in set (0.0039 sec)
> SELECT JSON_KEYS(@base, '$') \G
***** 1. IOW *****
JSON_KEYS(@base, '$'): ["grapes", "berries", "numbers"]
1 row in set (0.0005 sec)
> SELECT JSON_KEYS('{"z":123,"x":{"albedo":50}}') \G
***** 1. IOW *****
JSON_KEYS('{"z":123,"x":{"albedo":50}}'): ["x", "z"]
1 row in set (0.0004 sec)
> SELECT JSON_KEYS('{"z":123,"x":{"albedo":50}}', '$.x') \G
***** 1. IOW *****
JSON_KEYS('{"z":123,"x":{"albedo":50}}', '$.x'): ["albedo"]
1 row in set (0.0004 sec)
```

The `JSON_LENGTH()` function returns the length of the JSON document passed. It also allows you to pass in a path expression and if provided, will return the length of the value that matches the path expression. An error occurs if the `json_doc` argument is not a valid JSON document or the path argument is not a valid path expression or contains a `*` or `**` wildcard. However, the value returned has several constraints as follows.

- A scalar has length 1.
- An array has a length equal to the number of array elements.
- An object has a length equal to the number of object members.

However, there is one surprising limitation: The length returned does not count the length of nested arrays or objects. Thus, you must use this function carefully using the path expression for nested documents. Listing 6-18 shows several examples of using the function.

Listing 6-18. Using the JSON_LENGTH Function

```
> SET @base = '{"grapes":["red","white","green"],"berries":["strawberry",
"raspberry","boysenberry","blackberry"],"numbers":["1","2","3","4","5"]}';
Query OK, 0 rows affected (0.0006 sec)
> SELECT JSON_LENGTH(@base,'$') \G
***** 1. ROW *****
JSON_LENGTH(@base,'$'): 3
1 row in set (0.0005 sec)
> SELECT JSON_LENGTH(@base,'$.grapes') \G
***** 1. ROW *****
JSON_LENGTH(@base,'$.grapes'): 3
1 row in set (0.0005 sec)
> SELECT JSON_LENGTH(@base,'$.grapes[1]') \G
***** 1. ROW *****
JSON_LENGTH(@base,'$.grapes[1]'): 1
1 row in set (0.0005 sec)
> SELECT JSON_LENGTH(@base,'$.grapes[4]') \G
***** 1. ROW *****
JSON_LENGTH(@base,'$.grapes[4]'): NULL
1 row in set (0.0004 sec)
> SELECT JSON_LENGTH(@base,'$.berries') \G
***** 1. ROW *****
JSON_LENGTH(@base,'$.berries'): 4
1 row in set (0.0004 sec)
> SELECT JSON_LENGTH(@base,'$.numbers') \G
```

```
***** 1. IOW *****
```

```
JSON_LENGTH(@base, '$.numbers'): 5
```

```
1 row in set (0.0005 sec)
```

Notice the fourth example returns null because the path expression, while valid syntax, does not evaluate to a value or nested JSON array or object.

The `JSON_QUOTE()` function is a handy function to use that will help you add quotes where they are appropriate. That is, the function quotes a string as a JSON string by wrapping it with double quote characters and escaping interior quote and other characters and returns the result. Note that this function does not operate on a JSON document, rather, only a string.

You can use this function to produce a valid JSON string literal for inclusion within a JSON document. Listing 6-19 shows a few short examples of using the function to quote JSON strings.

Listing 6-19. Using the `JSON_QUOTE` Function

```
> SELECT JSON_QUOTE("test") \G
```

```
***** 1. IOW *****
```

```
JSON_QUOTE("test"): "test"
```

```
1 row in set (0.0012 sec)
```

```
> SELECT JSON_QUOTE('[true]') \G
```

```
***** 1. IOW *****
```

```
JSON_QUOTE('[true]'): "[true]"
```

```
1 row in set (0.0007 sec)
```

```
> SELECT JSON_QUOTE('90125') \G
```

```
***** 1. IOW *****
```

```
JSON_QUOTE('90125'): "90125"
```

```
1 row in set (0.0008 sec)
```

```
> SELECT JSON_QUOTE('["red","white","green"]') \G
```

```
***** 1. IOW *****
```

```
JSON_QUOTE('["red","white","green"]'): "[\"red\", \"white\", \"green\"]"
```

```
1 row in set (0.0007 sec)
```

Notice the last example. Here the function adds the escape character (`\`) since the string passed contains quotes. Why is this happening? Remember, this function takes a string, not a JSON array as the parameter.

The `JSON_UNQUOTE()` function is the opposite of the `JSON_QUOTE()` function. The `JSON_UNQUOTE()` function removes quotes JSON value and returns the result as a `utf8mb4` string. The function is designed to recognize and not alter markup sequences as follows.

- `\"`: A double quote (") character
- `\b`: A backspace character
- `\f`: A form feed character
- `\n`: A newline (linefeed) character
- `\r`: A carriage return character
- `\t`: A tab character
- `\\`: A backslash (\) character

Listing 6-20 shows examples of using the function.

Listing 6-20. Using the `JSON_UNQUOTE` Function

```
> SELECT JSON_UNQUOTE("test 123") \G
***** 1. ROW *****
JSON_UNQUOTE("test 123"): test 123
1 row in set (0.0005 sec)
> SELECT JSON_UNQUOTE("true") \G
***** 1. ROW *****
JSON_UNQUOTE("true"): true
1 row in set (0.0007 sec)
> SELECT JSON_UNQUOTE("\true\") \G
***** 1. ROW *****
JSON_UNQUOTE("\true\"): true
1 row in set (0.0007 sec)
> SELECT JSON_UNQUOTE('9\t0\t125\\') \G
***** 1. ROW *****
JSON_UNQUOTE('9\t0\t125\\'): 9 0      125\
1 row in set (0.0006 sec)
```

The `JSON_PRETTY()` function formats a JSON document for easier viewing. You can use this to produce an output to send to users or to make the JSON look a bit nicer in the shell. Listing 6-21 shows an example without the function and the same with the function. Notice how much easier it is to read when using `JSON_PRETTY()`.

Listing 6-21. Using the `JSON_PRETTY` Function

```
> SET @base = '{"name": {"last": "Throckmutton", "first": "Billy-bob"},
"address": {"zip": "90125", "city": "Melborne", "state": "California",
"street": "4 Main Street"}}';
Query OK, 0 rows affected (0.0005 sec)
> SELECT @base \G
***** 1. ROW *****
@base: {"name": {"last": "Throckmutton", "first": "Billy-bob"},
"address": {"zip": "90125", "city": "Melborne", "state": "California",
"street": "4 Main Street"}}
1 row in set (0.0005 sec)
> SELECT JSON_PRETTY(@base) \G
***** 1. ROW *****
JSON_PRETTY(@base): {
  "name": {
    "last": "Throckmutton",
    "first": "Billy-bob"
  },
  "address": {
    "zip": "90125",
    "city": "Melborne",
    "state": "California",
    "street": "4 Main Street"
  }
}
1 row in set (0.0004 sec)
```

However, there is one thing the example did not cover. If the JSON data element is a string, you must use the `JSON_UNQUOTE()` function to remove the quotes from the string. Let's suppose we wanted to add a generated column for the color data element. If we add the column and index with the `ALTER TABLE` statements without remove the quotes, we will get some unusual results as shown in Listing 6-22.

Listing 6-22. Removing Quotes for Generated Columns on JSON Strings

```
> CREATE TABLE `testdb_6`.`thermostats` (`model_number` char(20) NOT
NULL,`manufacturer` char(30) DEFAULT NULL,`capabilities` json DEFAULT
NULL,PRIMARY KEY (`model_number`)) ENGINE=InnoDB DEFAULT CHARSET=latin1;
Query OK, 0 rows affected (0.0225 sec)
> INSERT INTO `testdb_6`.`thermostats` VALUES ('ODX-123','Genie','{"rpm":
3000, "color": "white", "modes": ["ac", "furnace"], "voltage": 220,
"capability": "fan"}') \G
Query OK, 1 row affected (0.0037 sec)
> INSERT INTO `testdb_6`.`thermostats` VALUES ('AB-90125-C1', 'Jasper',
'{"rpm": 1500, "color": "beige", "modes": ["ac"], "voltage": 110,
"capability": "auto fan"}') \G
Query OK, 1 row affected (0.0041 sec)
> ALTER TABLE `testdb_6`.`thermostats` ADD COLUMN color char(20) GENERATED
ALWAYS AS (capabilities->'$.color') VIRTUAL;
Query OK, 0 rows affected (0.0218 sec)
Records: 0 Duplicates: 0 Warnings: 0
> SELECT model_number, color FROM `testdb_6`.`thermostats` WHERE color =
"beige" \G
Empty set (0.0006 sec)
> SELECT model_number, color FROM `testdb_6`.`thermostats` LIMIT 2 \G
***** 1. IOW *****
model_number: AB-90125-C1
      color: "beige"
***** 2. IOW *****
model_number: ODX-123
      color: "white"
2 rows in set (0.0006 sec)
> ALTER TABLE `testdb_6`.`thermostats` DROP COLUMN color;
```

```

Query OK, 0 rows affected (0.0206 sec)
Records: 0 Duplicates: 0 Warnings: 0
> ALTER TABLE `testdb_6`.`thermostats` ADD COLUMN color char(20) GENERATED
ALWAYS AS (JSON_UNQUOTE(capabilities->'$.color')) VIRTUAL;
Query OK, 0 rows affected (0.0172 sec)
Records: 0 Duplicates: 0 Warnings: 0
> SELECT model_number, color FROM `testdb_6`.`thermostats` WHERE color =
'beige' LIMIT 1 \G
***** 1. ROW *****
model_number: AB-90125-C1
      color: beige
1 row in set (0.0006 sec)

```

Notice in the first SELECT statement, there is nothing returned. This is because the virtual generated column used the JSON string with the quotes. This is often a source of confusion when mixing SQL and JSON data. Notice in the second SELECT statement, we see there should have been several rows returned. Notice also after we drop the column and add it again with the `JSON_UNQUOTE()` function, the SELECT returns the correct data.

Tip For more information about using JSON functions, see the section, “JSON Functions” in the online MySQL reference manual (<https://dev.mysql.com/doc/refman/8.0/en/json-functions.html>).

Summary

The addition of the JSON data type to MySQL has ushered a paradigm shift for how we use MySQL. For the first time, we can store semi-structured data inside our relational data (tables). Not only does this give us far more flexibility that we ever had before, it also means we can leverage modern programming techniques to access the data in our applications without major efforts and complexity. JSON is a well-known format and used widely in many applications.

Understanding the JSON data type is key to understanding the Document Store. This is because the JSON data type, while designed to work with relational data, forms the pattern for how we store data in the document store – in JSON documents!

In this chapter, we explored the JSON data type in more detail. We saw examples of how to work with the JSON data in relational tables via the numerous built-in JSON functions provided in MySQL.

In the next chapter, we will explore the MySQL Document Store in more detail by taking the garage sample application and converting it to a document store. That is, we will see how to migrate a relational database solution to a document store all through using MySQL Shell to migrate the data and develop the code to build a NoSQL solution with JSON documents.

CHAPTER 7

Example: Document Store Development

Now that we have a firm foundation in JSON and we have seen how to use MySQL Shell to develop relational database code, we are ready to begin writing a NoSQL version of the sample application.

As you will see, the evolution of the application from a pure relational model to a document model demonstrates how we can avoid some of the messier aspects of using relational data. One element that may surprise you is the complexity of the code for the document store (NoSQL) version lower making the code easier to understand. What better reason to consider writing all your future applications using the MySQL Document Store?

In the last chapter, we explored working with JSON data in our relational databases by creating one or more JSON columns and using JSON functions to create, manipulate, search, and access data in a JSON document.

In this chapter, we take the garage sample application (relational database solution) and migrate it to a pure NoSQL application¹ using the MySQL Document Store. We can do this by using the many X DevAPI features through Python – all developed with the aide of MySQL Shell! Let's dive in.

Getting Started

Rather than leave the explanation of the document store with the last chapter, we back it up with examples that explain the benefits of a new way of working with data through code using a document store. We will demonstrate the concepts in an interactive manner so that the concepts are proven through the example rather than simply showing how it

¹The sample application presented here has no SQL statements and uses only Python and the X DevAPI.

is possible. In this section, we will learn about a sample application that strives to do just that – to prove how you can use the shell to develop your own document store.

However, in order to show the full capabilities of the shell in this manner, the sample must be sufficiently complex enough to have the depth (and breadth) to fulfill its role. Thus, for this chapter, we will use the garage sample application from Chapter 5 as the starting point and migrate it to a document store.

Recall, the sample application is a tool you can use to organize tools in your garage or workshop. However, unlike the relational database implementation, we will take a slightly different view of the data. Let's begin.

Sample Application Concept

Like the sample application we saw in Chapter 5, the sample application is all about organization. In Chapter 5, the data was organized in a typical manner that most relational database experts use – they construct tables to hold things that all have the same layout or, in plain terms, the tables represent the major categories of data. Recall, we had tables for hand tools, power tools, storage places, etc. – all things you would find in a garage or workshop.

For this version of the garage application, we will use change our focus from organizing like things into tables to organizing things by collections. For example, a toolchest contains tools, a storage unit contains bins, boxes, cases, etc. This may seem like a very subtle difference (and perhaps it is), but it changes the entire focus of the data. Rather than looking for a tool to find where it is stored, we can open a toolchest or look on a shelving unit to see what tools are stored there. In this way, we've created a much more user-friendly version. Now, we mimic what most people do when they look for a tool – they open drawers one at a time (you know you do it too).

This is one of the powers of the document store – the inherent flexibility in the data allows you choose the view you want to use (or require) and make the data and its access layers work. This is a challenge that some relational database applications fail miserably.²

Also, unlike the relational database version, we create collections for each of the major storage equipment and maintain the list of the contents of each (e.g., tools) in the document that represents the storage equipment. We will group all tools together in a

²I have personally witnessed several massive failures in design where the database forced an unnatural and often hostile workflow on the user. The application seemed to be written inside out making it difficult for users to learn much less use. Don't be like those developers.

single collection and reference them in the storage equipment collection by document id. Again, this may sound strange, but you will see how well it works as you read along.

Before that, let's take a moment to look at the modified user interface for this version of the sample application. We will name this application `mygarage_v2`. Since we focused the schema design on the storage equipment, our views in the user interface will be from that perspective. We retain the list view concept except we use major sections for each of the storage equipment. Figure 7-1 shows an example of the new interface showing a toolchest detail view.

MyGarage v2
Cabinets
Toolchests
Workbenches
Shelving Units
Organizers
Tools
Vendors

Toolchest - Detail

Description

Vendor

Physical Location

Width

Depth

Height

Tools in Toolchest

Drawer - Left 1

Action	Type	Description	Category	Size
Modify/View	Screwdriver	1/8-in X 1-1/2-in	Handtool	Slotted
Modify/View	Screwdriver	3/16-in X 4-in	Handtool	Slotted
Modify/View	Screwdriver	1/4-in X 1-1/2-in	Handtool	Slotted
Modify/View	Screwdriver	1/4-in X 4-in	Handtool	Slotted
Modify/View	Screwdriver	5/16-in X 6-in	Handtool	Slotted
Modify/View	Screwdriver	1-1/2-in	Handtool	Phillips #0
Modify/View	Screwdriver	3-in	Handtool	Phillips #1
Modify/View	Screwdriver	1-1/2-in	Handtool	Phillips #2
Modify/View	Screwdriver	4-in	Handtool	Phillips #2
Modify/View	Screwdriver	6-in	Handtool	Phillips #3
Modify/View	Screwdriver	offset	Handtool	Slotted and Phillips
Modify/View	Screwdriver	magnetizer/demagnetizer	Handtool	
Modify/View	Screwdriver	Ratcheting replaceable tip	Handtool	various

Figure 7-1. Toolchest Detail View

Note We use “schema” when working with a document store or NoSQL model and “database” for a relational database model.

Notice here we see the toolchest details along with each of the storage locations (drawers, shelves) and the list of tools in them. While we could have adopted a similar view for the relational database version,³ the document store makes this considerably easier to code.

Tip Rather than explain every nuance of the sample application, we will focus on the portions that are best used to prove the utility of using the shell to develop code – the schema collection code modules.

Unlike the relational database version, the code behind this version is easy to understand and in some ways much less complex. But before we see the schema design, let’s discuss the collections used in this version of the application in the following list. This will help you understand the differences from the relational database version (version 1).

- *Cabinets*: Storage equipment that has doors and may have one or more shelves – used to store a variety of tools and organizers
- *Toolchests*: Storage equipment that has zero or more shelves and one or more drawers – used for storing smaller tools
- *Workbench*: Storage equipment that has one or more shelves and zero or more drawers – used for storing larger tools
- *Shelving Unit*: Storage equipment that has no doors and one or more shelves – used for storing larger bins and similar organizers
- *Organizers*: Containers that can hold one or more tools but requires storing in storage equipment
- *Tools*: Hand and power tools
- *Vendors*: Manufacturers of tools and equipment

³I challenge you to do just that as an exercise!

Notice there is a bit of a vocabulary change here. In the first version of the application, hence first version or version 1, we used table names in the singular. Document stores typically use the plural since each collection often contains more than one item (document). Also, when working with a document store, we should always use the term, schema, rather than database. While some would argue they are synonyms, the X DevAPI makes a clear distinction, so we will adopt the same and use the term “schema”.

You may be wondering how we can get the association from toolchest to tools as shown in preceding text. This is accomplished by simply storing the document ids of the tools with each storage place (renamed to tool location) in the toolchest document.

Listing 7-1 is a sample listing that may help visualize how this is accomplished for the organizers collection. As you will see, we use the shell to connect to the schema then get the collections and follow the document id key values to fetch the list of tools (an array named `tool_ids`). This is similar how one would look up things in a relational database, but in this case, we don’t have to craft special SQL commands (or worse, evil SQL joins) to get the data. We’ll talk more about those `_id` fields later.

Listing 7-1. Sample Toolchest JSON Document

```
> garage_v2 = my_session.get_schema('garage_v2')
> tc = garage_v2.get_collection('toolchests')
> tc.find().execute().fetch_one()
{
  "_id": "00005cc33db4000000000000025f",
  "depth": 22,
  "description": "Kobalt 3000 Steel Rolling Tool Cabinet (Black)",
  "height": 54,
  "location": "Rear wall right of workbench",
  "tool_locations": [
    "00005cc33db400000000000000260",
    "00005cc33db400000000000000261",
    "00005cc33db400000000000000268"
  ],
  "vendorid": "00005cc33db400000000000000130",
  "width": 48
}
```

```

> locs = garage_v2.get_collection('locations')
> locs.find('_id = :param1').bind('param1',
'00005cc33db40000000000000260').execute().fetch_one()
{
  "_id": "00005cc33db40000000000000260",
  "depth": 17,
  "description": "Left 1",
  "height": 2,
  "tool_ids": [
    "00005cc33db40000000000000146",
    "00005cc33db40000000000000147",
    "00005cc33db40000000000000148",
    "00005cc33db40000000000000149",
    "00005cc33db4000000000000014a",
    "00005cc33db4000000000000015a"
  ],
  "type": "Drawer",
  "width": 21
}

```

At this point, you may be wondering what happened to the happy-go-lucky no format rules capabilities of JSON documents. In short, it's still there, but our code requires a certain set of attributes for each document. As you will see, the attributes we define will be used in the code directly to access the code in the document.

This does not preclude using additional attributes that can be added at any time, but it does require your code support such changes. This is what is meant by code-centric schema less embedded data design (or simply code-driven data). We can use our code to augment our documents (and collections) as we need to over the evolution of the application.

For example, if we need to add a new attribute in the future, we can add code to handle the new data presented, which must include how to handle documents that do not have the new attribute, but may also include code to add the attribute to the older documents as needed. Unlike relational database applications that require modification of the table(s), which can force the code to change (perhaps in unhappy ways), we can let the code for the document store effect the changes instead. It all comes down to the code.

Let's talk more about the schema design and how we can migrate our relational database to a document store.

Schema Design

You may be tempted to think we can use a tool like MySQL Workbench to create the schema and collections (and you can) like you would for a relational database, but you do not need to do that. You should use code to enact the creation events. More specifically, if you were to import the schema for a document store into a tool like MySQL Workbench, you won't see much that is of interest. This is because MySQL Shell (or Workbench) in SQL mode sees the collection like a database. For example, listing 7-2 shows the CREATE TABLE statement for the database (schema) for this version of the sample application (garage_v2).

Listing 7-2. Sample CREATE statement from SQL for garage_v2.

```
> EXPLAIN toolchests \G
***** 1. ROW *****
Field: doc
Type: json
Null: YES
Key:
Default: NULL
Extra:
***** 2. ROW *****
Field: _id
Type: varbinary(32)
Null: NO
Key: PRI
Default: NULL
Extra: STORED GENERATED
2 rows in set (0.0025 sec)
```

Clearly, that isn't going to work, and it has little more than a passing interest to use. Note the `_id` field. That's the document id.

When working with a document store, we use code to create the schema and collections. Recall, we want to name the schema `garage_v2` and create the collections (cabinets, toolchests, workbenches, shelving units, organizers, tools, and vendors). We also have a collection named `locations` that stores as a document each of the types of storage places (called tool locations) like shelves, drawers, etc. The

locations collection is not accessible from the main menu in the user interface since the focus is on the storage equipment collections. However, each storage equipment detail view allows you to modify the tool locations for that storage equipment fulfilling the create, read, update, and delete (CRUD) operations for that collection.

We will use the shell and the X DevAPI to create each of these objects as shown in Listing 7-3. Notice we first connect to the server, request the session, then create the schema and collections. That's it - our schema is done and we have the basis for our document store!

Listing 7-3. Creating the Document Store

```

from getpass import getpass
try:
    input = raw_input
except NameError:
    pass
# Get user id and password
userid = input("User Id: ")
passwd = getpass("Password: ")

user_info = {
    'host': 'localhost',
    'port': 33060,
    'user': userid,
    'password': passwd,
}
# Connect to the database garage_v1
my_session = mysqlx.get_session(user_info)
# Create the schema for garage_v2
my_session.drop_schema('garage_v2')
garage_v2 = my_session.create_schema('garage_v2')
# Create the collections
cabinets = garage_v2.create_collection('cabinets')
organizers = garage_v2.create_collection('organizers')
shelving_units = garage_v2.create_collection('shelving_units')
tools = garage_v2.create_collection('tools')

```

```

toolchests = garage_v2.create_collection('toolchests')
locations = garage_v2.create_collection('locations')
workbenches = garage_v2.create_collection('workbenches')
vendors = garage_v2.create_collection('vendors')
# Show the collections
print(garage_v2.get_collections())

```

When you run the code, you should see output like the following.

```

> mysqlsh --py -f listing7-3.py
User Id: root
Password: *****
[<Collection:cabinets>, <Collection:locations>, <Collection:organizers>,
<Collection:shelving_units>, <Collection:toolchests>, <Collection:tools>,
<Collection:vendors>, <Collection:workbenches>]

```

Wasn't that a lot easier than slogging through a bunch of SQL CREATE statements? You can run this on your machine if you'd like to create the schema and collections from scratch, but if you are converting the data we saw in Chapter 5 from a relation database to a document store, you may want to follow along with the example conversion in the *Setup and Configuration* section.

Before we do that, let's talk about the collections again. This time, we will see a sample document for each collection.

Note We use “tool location” in place of the version 1 concept of “storage place” to both distinguish and better describe the attribute in the JSON document.

Cabinets Collection

The cabinets collection stores documents that describe a cabinet, which can contain one or more shelves and one or more doors. As such, we want to store information about each cabinet in the collection to include its physical size and location as well as the tool locations (shelves) it contains. Listing 7-4 shows an example JSON document from this collection.

Listing 7-4. Cabinets Collection Example JSON Document

```

> garage_v2 = my_session.get_schema('garage_v2')
> cabinets = garage_v2.get_collection('cabinets')
> cabinets.find().execute().fetch_one()
{
  "_id": "00005cae74150000000000000161",
  "depth": 24,
  "description": "Kobalt Steel Freestanding Garage Cabinet",
  "height": 72,
  "location": "Right wall",
  "numdoors": 2,
  "tool_locations": [
    "00005cae74150000000000000162",
    "00005cae74150000000000000163",
    "00005cae74150000000000000164",
    "00005cae74150000000000000165",
    "00005cae74150000000000000166"
  ],
  "vendorid": "00005cae74150000000000000001",
  "width": 48
}

```

Notice we have an attribute named `_id`. When we created the document, we did not specify this attribute and, if you don't, MySQL will create a unique value for you. This is called a document id. You can specify your own values for `_id` if you require, but it is generally discouraged as the internal mechanism will ensure the documents are unique. Think of it as a primary key.⁴ For more information about document ids, see <https://dev.mysql.com/doc/x-devapi-userguide/en/working-with-document-ids.html>.

We also added attributes for the description, size (depth, height, width), its physical location, number of doors, and an array of tool locations. Finally, notice we have an attribute that contains the document id for the vendor. Notice that the `tool_locations` array does not impose any restrictions such as allowing only shelves. This is because in a document store, those types of constraints are added to the code. The document store simply stores the documents.

⁴But don't call it that. It's a document id.

Note We use lower case names for collections and attributes in the JSON documents. This is not strictly necessary but does follow a familiar pattern.

Locations Collection

The locations collection stores documents that describe the tool storage locations such as a shelf or drawer. That is, we want to store information about each location in the collection to include its physical size and location as well as the tool locations (shelves or drawers) it contains. Listing 7-5 shows an example JSON document from this collection.

Listing 7-5. Locations Collection Example JSON Document

```
> garage_v2 = my_session.get_schema('garage_v2')
> locations = garage_v2.get_collection('locations')
> locations.find().execute().fetch_one()
{
  "_id": "00005cae741500000000000000146",
  "depth": 17,
  "description": "Left 1",
  "height": 2,
  "tool_ids": [
    "00005cae741500000000000000141",
    "00005cae741500000000000000142",
    "00005cae741500000000000000139"
  ],
  "type": "Drawer",
  "width": 21
}
```

Notice we have an array of tool ids, which are the document ids for each of the tools stored in this location. Notice also we have a type attribute to store the type of tool location. Control of these values are also moved to the code, which allows you to change the type permitted through code rather than modifying the underlining data storage, which can cause additional headaches during development and release.

WAIT, WHY NOT LUMP THE TOOLS IN THE TOOLCHEST?

Some document store developers may tell you it is bad form to use a separate collection and reference documents by ids. They would say, “just dump all the tools into the storage equipment collection as an array and don’t mess with a locations collection.” That is one design choice and a valid one, but some rigor in your data such as extracting the mapping of tool locations to tools in this manner is equally as valid and in some cases may make the conversion of your data easier to accomplish and visualize. We will see such a case later in this chapter. Yes, JSON will let you lump everything together, but having a little rigor in how the data is organized does not validate the NoSQL objective whatsoever. Don’t be fooled into thinking no rigor equates to better NoSQL design.

Organizers Collection

The organizers collection stores documents that describe an organizer such as a bin, box, case, etc., which can contain one or more tools. As such, we want to store information about each organizer in the collection to include its physical size and location as well as the tools it contains. Listing 7-6 shows an example JSON document from this collection.

Listing 7-6. Organizers Collection Example JSON Document

```
> garage_v2 = my_session.get_schema('garage_v2')
> organizers = garage_v2.get_collection('organizers')
> organizers.find().execute().fetch_one()
{
  "_id": "00005cae7415000000000000013b",
  "depth": 14,
  "description": "SAE/Metric Socket Set",
  "height": 4,
  "tool_ids": [
    "00005cae741500000000000000b2",
    "00005cae741500000000000000b9",
    "00005cae741500000000000000c7"
  ],
  "type": "Case",
  "width": 12
}
```

Like the locations collection, we also have a type attribute that we control in code. Aside from that, we added attributes to describe its physical size and description. There is also an array for the tool ids. That's all we need.

One caveat here for the sample application. Since organizers can be small, they can be placed in a tool location. As such, the application needs additional code to handle lookups to distinguish between organizers and tools. This is an example of allow for flexibility in your application. A relational database design would never permit this because the types (organizer vs. tool location) aren't the same. Since we're controlling the interface to the data with code, all we need to do is write the code to handle such a condition! We'll see more about that in the next section on the code design.

Shelving Units Collection

The shelving_units collection stores documents that describe a shelving unit, which can contain one or more shelves. As such, we want to store information about each shelving unit in the collection to include its physical size and location as well as the tool locations (shelves) it contains. Listing 7-7 shows an example JSON document from this collection.

Listing 7-7. Shelving Units Collection Example JSON Document

```
> garage_v2 = my_session.get_schema('garage_v2')
> shelving_units = garage_v2.get_collection('shelving_units')
> shelving_units.find().execute().fetch_one()
{
  "_id": "00005cae7415000000000000014f",
  "depth": 24,
  "description": "Wire shelving #1",
  "height": 72,
  "location": "Right wall",
  "tool_locations": [
    "00005cae74150000000000000150",
    "00005cae74150000000000000152",
    "00005cae74150000000000000153"
  ],
  "vendorid": "00005cae7415000000000000015",
  "width": 48
}
```

Are you starting to see a trend here? Yes, the collections of storage equipment have a very similar set of attributes. That is by design. Why? So we can isolate the documents to discrete collections to make the collections shallow (fewer documents). Admittedly, this will result in a very small performance issue, but it is good practice to use. It also permits you to change the set of attributes in each collection as your application matures or evolves.

For example, if you add a new type of shelving unit that has doors, drawers, tool hangers, and so on, you can change the code to handle the new additions without redoing any of the documents in the collection and without forcing the change on the other collections.

Recall, we had a bit of that in the relational database version where we had the field for storing the number of doors, which was not needed by all storage equipment types. JSON documents are the answer to that SQL conundrum.

Toolchests Collection

The `toolchests` collection stores documents that describe a toolchest, which can contain zero or more shelves and one or more drawers. As such, we want to store information about each toolchest in the collection to include its physical size and location as well as the tool locations (shelves and drawers) it contains. Listing 7-8 shows an example JSON document from this collection.

Listing 7-8. Toolchests Collection Example JSON Document

```
> garage_v2 = my_session.get_schema('garage_v2')
> toolchests = garage_v2.get_collection('toolchests')
> toolchests.find().execute().fetch_one()
{
  "_id": "00005cae74150000000000000145",
  "depth": 22,
  "description": "Kobalt 3000 Steel Rolling Tool Cabinet (Black)",
  "height": 54,
  "location": "Rear wall right of workbench",
  "tool_locations": [
    "00005cae74150000000000000146",
    "00005cae74150000000000000147",
```

```

    "00005cae7415000000000000014e"
  ],
  "vendorid": "00005cae74150000000000000001",
  "width": 48
}

```

Tools Collection

The tools collection stores documents that describe a tool. We will include all tools; not just hand tools or power tools. We store the characteristics of each tool in the document as you expect such as a description, category, size, etc. Listing 7-9 shows an example JSON document from this collection.

Listing 7-9. Tools Collection Example JSON Document

```

> garage_v2 = my_session.get_schema('garage_v2')
> tools = garage_v2.get_collection('tools')
> tools.find().execute().fetch_one()
{
  "_id": "00005cae741500000000000000024",
  "category": "Handtool",
  "description": "1/8-in X 1-1/2-in",
  "size": "Slotted",
  "type": "Screwdriver",
  "vendorid": "00005cae741500000000000000002"
}

```

One note here about this collection. Recall we had two tables in the relational database for version 1 because of the unique enumerated values for some of the fields. Since JSON permits us to store the attributes we want, the documents in this collection may or may not have one or more of the attribute. For example, not all tools have a size attribute. Like the other collections, the values for the category and type are handled in the code.

Vendors Collection

The vendors collection stores documents that describe a vendor. We store the same characteristics from the relational database in version 1 including the name, sources, and the URL. Listing 7-10 shows an example JSON document from this collection.

Listing 7-10. Vendors Collection Example JSON Document

```
> garage_v2 = my_session.get_schema('garage_v2')
> vendors = garage_v2.get_collection('vendors')
MySQL Py > vendors.find().execute().fetch_one()
{
  "_id": "00005cae74150000000000000001",
  "name": "Kobalt",
  "sources": "Lowe's",
  "url": "https://www.kobalttools.com/"
}
```

Workbenches Collection

The workbenches collection stores documents that describe a workbench, which can contain zero or more shelves. As such, we want to store information about each workbench in the collection to include its physical size and location as well as the tool locations (shelves) it contains. Listing 7-11 shows an example JSON document from this collection.

Listing 7-11. Workbenches Collection Example JSON Document

```
> garage_v2 = my_session.get_schema('garage_v2')
> workbenches = garage_v2.get_collection('workbenches')
> workbenches.find().execute().fetch_one()
{
  "_id": "00005cae741500000000000000167",
  "depth": 24,
  "description": "Large metal workbench",
  "height": 42,
  "location": "Rear wall",
  "tool_locations": [
```

```

    "00005cae7415000000000000168",
    "00005cae7415000000000000169",
    "00005cae741500000000000016a"
  ],
  "vendorid": "00005cae74150000000000000002",
  "width": 72
}

```

Now that we've seen how the collections are formed, let's look at how we can use the shell to write the code we need to implement the CRUD operations for the collections.

Code Design

The code for the sample application is very similar to the code in version 1; it is a Python Flask application with the same choices for organizing the code modules. Like version 1, you may find things you would do differently, it should still be usable in its current form for demonstration purposes. The code design choices made for the sample application include the following.

- Use Flask framework for web-based interface.
- Use a class to represent each table in the database.
- Place a single class in its own code module.
- Place all schema code modules in its own folder (named schema).
- Use a class to encapsulate the connection to the database server.
- Use class modules to test each of the table/view classes.
- Use a code module run from the shell to test the class modules.

We will see most of these constraints in the demonstration. As mentioned previously, a description of the user interface is included in the appendix.

Fortunately, we've made it a little easier to write the code by using a document store. This is because collections allow you to specify the attributes, so aside from adding constraints for a list of valid values (enumerated columns in the relational database version) and checking for errors, the code for the create, read, update, and delete (CRUD) are the same. Thus, we will use an object hierarchy with a base class that contains the main code for the CRUD operations and allow sub classes to add the constraints code specific to each collection.

Let's take a moment to list the code modules we will use. In this sample application, we will create a folder named `schema` and place the following modules in the folder. We can then import them as needed in the application code. Table 7-1 shows the code modules needed.

Table 7-1. *Schema Code Modules*

Code Module	Class Name	Description
<code>garage_v2</code>	<code>MyGarage</code>	Implements connection to server and general server interface
<code>garage_collection</code>	<code>GarageCollection</code>	Base class for all collection classes
<code>cabinets.py</code>	<code>Cabinets</code>	Models the cabinets collection
<code>locations.py</code>	<code>Locations</code>	Models the locations collection
<code>organizers.py</code>	<code>Organizers</code>	Models the organizers collection
<code>shelving_units.py</code>	<code>ShelvingUnits</code>	Models the shelving_units collection
<code>toolchests.py</code>	<code>Toolchests</code>	Models the toolchests collection
<code>tools.py</code>	<code>Tools</code>	Models the tools collection
<code>vendors.py</code>	<code>Vendors</code>	Models the vendors collection
<code>workbenches.py</code>	<code>Workbenches</code>	Models the workbenches collection

When we write the code for the sample application to use these code modules, we will use the `MyGarage` class to make a connection to the database server and, when requested, use the class associated with each collection to call the CRUD operations on each.

Now that we understand the goals for the sample application and its design, let's get started with writing the schema code for the sample application.

Setup and Configuration

The setup for the following demonstration does not require installing anything or even using the sample application, rather, we need only load the sample schema because we will only be working with the schema code modules. While images are used to depict certain aspects of the sample application, you don't strictly need it for this chapter. Once again, see the appendix for how to setup and use the complete sample application.

Since we are using the same data from the sample application in Chapter 5, we'll look at how to convert the relational database data to a document store. After that, we will see how to write the code for the base and collection classes.

Converting Relational Data to a Document Store

Rather than start from scratch, we can convert the relational database to a document store. If you haven't run the sample application from Chapter 5 and you want to see how the conversion works, you can run the `Ch05/mygarage/database/garage_v1.sql` file found in the source code for this book first, then follow along as we see how to convert the database tables to collections and the data to documents. Even if you do not plan to use the sample application, this section can help you in the future convert other databases to document stores.

We will take a stepwise approach to forming a script to convert the data. This is because there are several challenges that can make the conversion code a little tricky. Fortunately, we will walk through each of these as we examine the code necessary for converting the data. The challenges for converting from a relational database to a document store concern how the data is massaged into the new collections. This means you must plan your collections and how you want to use them before attempting the conversion. Otherwise, you may find yourself redoing the conversion code as your schema design matures.⁵

The challenges for converting the sample application (version 1) to a document store include the following.

- We cannot use the auto increment values. Thus, we will need to create a mapping from the old auto increment values to the new document ids.
- We are combining the hand tools and power tools into one collection.
- We are splitting the storage table into separate collections.
- We must traverse the place table and preserve the tool locations in the new collections.

⁵Yep. Been there, done that more times than I'd like to admit. Planning solves a lot of problems!

Let's begin with the first step – the preamble or setup of the code for the conversion. If you plan to convert your own relational database to a document store, you can use a similar set of steps to write your own conversion code, but you may not need all of them.

Step 1: Conversion Setup Code

This step is straight forward. We simply import the modules we need, then connect to the server and get an instance of the `garage_v1` database and create the `garage_v2` schema and collections like we did earlier in Listing 7-3. Listing 7-12 shows the code for the setup of the conversion script.

Listing 7-12. Conversion Setup Code

```
import json
from getpass import getpass
try:
    input = raw_input
except NameError:
    pass

try:
    import mysqlx
except Exception:
    from mysqlsh import mysqlx

# Get user id and password
userid = input("User Id: ")
passwd = getpass("Password: ")

user_info = {
    'host': 'localhost',
    'port': 33060,
    'user': userid,
    'password': passwd,
}
```

```

# Connect to the database garage_v1
my_session = mysqlx.get_session(user_info)
garage_v1 = my_session.get_schema('garage_v1')
# Get the tables
handtool_tbl = garage_v1.get_table('handtool')
organizer_tbl = garage_v1.get_table('organizer')
place_tbl = garage_v1.get_table('place')
powertool_tbl = garage_v1.get_table('powertool')
storage_tbl = garage_v1.get_table('storage')
vendor_tbl = garage_v1.get_table('vendor')

# Create the schema for garage_v2
my_session.drop_schema('garage_v2')
garage_v2 = my_session.create_schema('garage_v2')
# Create the collections
cabinets = garage_v2.create_collection('cabinets')
organizers = garage_v2.create_collection('organizers')
shelving_units = garage_v2.create_collection('shelving_units')
tools = garage_v2.create_collection('tools')
toolchests = garage_v2.create_collection('toolchests')
locations = garage_v2.create_collection('locations')
workbenches = garage_v2.create_collection('workbenches')
vendors = garage_v2.create_collection('vendors')

```

Step 2: Helper Functions

The next step requires some explanation. It occurs next in the script (but could be placed in the code earlier). In this step, we create a several helper functions for working with the database tables and reconstructing the links between the original tables and the new document id mappings. Table 7-2 lists the new helper functions and a description of each. We will see the code for the functions as well.

Table 7-2. *Helper Functions for Conversion Script*

Name	Parameters	Description
<code>show_collection(col_object)</code>	collection object	Print the contents of a collection (for debugging).
<code>get_places(storage_id)</code>	auto increment id	Get the storage places that match this storage id.
<code>get_organizer_ids(place_id)</code>	auto increment id	Get the list of organizer ids at the storage place.
<code>get_handtool_ids(place_id)</code>	auto increment id	Get the list of handtool ids at the storage place.
<code>get_powertool_ids(place_id)</code>	auto increment id	Get the list of powertool ids at the storage place.
<code>get_mapping(old_id, mapping)</code>	auto increment id, map (array)	Get the new document id for the old vendor id.
<code>find_tool_in_organizers(tool_id)</code>	_id for the tool	Search the organizers collection for the tool.
<code>find_tool(collection_name, tool_id)</code>	collection name, _id for the tool	Search for a tool in a given collection.
<code>get_tool_location(tool_id)</code>	_id for the tool	Find the location _id for a tool.

The `get_*` functions are all used to query the relational database tables to find the row that matches the auto increment value and get a list of auto increment ids for power tools and hand tools. These are used to fetch a row so they can be converted to a JSON document. The `find_*` functions are used to search the collections for the JSON document that matches a tool id so that we can populate the locations collection.

Additionally, in order to associate tool storage locations to tools, we need a way to collect the tools. Instead of making a reference, join, or lookup table, we can store an array of the tool ids in the document. Thus, we create the link that way making it more intuitive – we open a toolbox and want to see what is inside.

Let's now look at the code for these functions. Listing 7-13 shows the code for the functions. Rather than explain what each does, we present the code and discuss them later in context. Don't worry if you do not see how they work or why they were written; they will make much more sense when you see them used in context.

Listing 7-13. Helper Functions

```

# Display the documents in a collection
def show_collection(col_object):
    print("\nCOLLECTION: {0}".format(col_object.name))
    results = col_object.find().execute().fetch_all()
    for document in results:
        print(json.dumps(json.loads(str(document)),
                          indent=4, sort_keys=True))

# Get the storage places that match this storage id
def get_places(storage_id):
    return place_tbl.select('Type', 'Description', 'Width', 'Depth',
                            'Height', 'Id')\
        .where("StorageId = {0}".format(storage_id)).execute()

# Get the list of organizer ids at the storage place
def get_organizer_ids(place_id):
    organizer_ids = []
    org_results = organizer_tbl.select('Id')\
        .where("PlaceId = {0}".format(place_id)).execute()
    for org in org_results.fetch_all():
        organizer_ids.append(get_mapping(org[0], organizer_place_map)[0])
    return organizer_ids

# Get the list of handtool ids at the storage place
def get_handtool_ids(place_id):
    handtool_ids = []
    ht_results = handtool_tbl.select('Id')\
        .where("PlaceId = {0}".format(place_id)).execute()
    for ht in ht_results.fetch_all():
        handtool_ids.append(ht[0])
    return handtool_ids

# Get the list of powertool ids at the storage place
def get_powertool_ids(place_id):
    powertool_ids = []
    pt_results = powertool_tbl.select('Id')\

```



```

        .where("PlaceId = {0}".format(place_id)).execute()
    for pt in pt_results.fetch_all():
        powertool_ids.append(pt[0])
    return powertool_ids

# Get the new docid for the old vendor id
def get_mapping(old_id, mapping):
    for item in mapping:
        if item[0] == old_id:
            return item
    return None

# Search the organizers collection for the tool
def find_tool_in_organizers(tool_id):
    # organizers contain no shelves or drawers so fetch only the tool ids
    organizers = garage_v2.get_collection('organizers')
    results = organizers.find().fields("_id", "tool_ids", "type",
                                       "description").execute().fetch_all()

    for result in results:
        if (result["tool_ids"]) and (tool_id in result["tool_ids"]):
            return ("{0}, {1}".format(result["type"], result["description"]),
                    'organizers', result["_id"])

    return None

# Search for a tool in a given collection
def find_tool(collection_name, tool_id):
    collection = garage_v2.get_collection(collection_name)
    storage_places = collection.find()\
        .fields("_id", "description", "tool_locations").execute().fetch_all()
    for storage_place in storage_places:
        if storage_place["tool_locations"]:
            for location in storage_place["tool_locations"]:
                loc_data = locations.find('_id = :param1') \
                    .bind('param1',
                          location).execute().fetch_all()
            if loc_data:
                loc_dict = dict(loc_data[0])

```

```

    tool_ids = loc_dict.get("tool_ids", [])
    if tool_id in tool_ids:
        return ("{0}, {1} - {2}"
                ".format(storage_place["description"],
                          loc_dict["description"],
                          loc_dict["type"]),
                collection_name,
                storage_place["_id"])

    return None
# Find the location document id for a tool.
def get_tool_location(tool_id):
    loc_found = find_tool_in_organizers(tool_id)
    if loc_found:
        return loc_found
    storage_collections = [
        'toolchests', 'shelving_units', 'workbenches', 'cabinets'
    ]
    for storage_collection in storage_collections:
        loc_found = find_tool(storage_collection, tool_id)
        if loc_found:
            return loc_found
    return None

```

Step 3: Populate Collections

The next step is to populate the collections. We can populate the collections with data, but we must do it in a specific order. For example, each of the documents in the collections representing storage equipment and tools is the document id for the vendor. Thus, we need to do the vendors collection first creating a mapping of the old id column from the table to the new document id in the vendors collection. Let's see how to do that. Listing 7-14 shows the code for converting the vendor table to the vendors collection. We will explore this code in more detail since it forms the template for working with the other tables and collections.

Listing 7-14. Populate the Vendors Collection

```
# Get the vendors
my_results = vendor_tbl.select('Id', 'Name', 'URL', 'Sources').execute()
vendor_id_map = []
for v_row in my_results.fetch_all():
    new_item = {
        'name': v_row[1],
        'url': v_row[2],
        'sources': v_row[3]
    }
    last_docid = vendors.add(new_item).execute().get_generated_ids()[0]
    vendor_id_map.append((v_row[0], last_docid))
show_collection(vendors)
```

Notice here we open the vendor table and read all the data. Then, we create an empty map (array) that we will use to record the auto increment id from the table to the new document id from the collection. This will allow us to replace the auto increment id in the other tables for the vendor column to the new document id for the vendor in the collection. This is a nifty way to preserve relational links during the conversion.

We use a loop to read through the results from the query and form a dictionary with the attributes (in lower case) using the data from the table row. We then use the vendors collection to add the vendor document.

We can chain the `add()` method with the `get_generated_ids()` call to get the last document id generated. We then add this to the new mapping named `vendor_id_map`, which we will use later to insert the correct document id for the vendor in the other documents in the other collections.

To help visualize the results, we use the `show_collection()` function to print the contents of the collection.

The next collection we convert is the `tools` collection. Recall, we are going to combine the `handtool` and `powertool` tables into the `tools` collection. Thus, we must read each of these tables inserting them into the `tools` collection. Listing 7-15 shows the code for this conversion. Take a moment and familiarize yourself with the code.

Listing 7-15. Populate the Tools Collection

```

# Get the tools combining the handtool and powertool tables
ht_results = handtool_tbl.select('Id', 'VendorId', 'Description', 'Type',
    'Toolsize', 'PlaceId').execute()
tool_place_map = []
for ht_row in ht_results.fetch_all():
    new_item = {
        'category': 'Handtool',
        'vendorid': get_mapping(ht_row[1], vendor_id_map)[1],
        'description': ht_row[2],
        'type': ht_row[3],
        'size': ht_row[4],
    }
    last_docid = tools.add(new_item).execute().get_generated_ids()[0]
    tool_place_map.append((ht_row[0], last_docid))
pt_results = powertool_tbl.select('Id', 'VendorId', 'Description', 'Type',
    'PlaceId').execute()
for pt_row in pt_results.fetch_all():
    new_item = {
        'category': 'Powertool',
        'vendorid': get_mapping(pt_row[1], vendor_id_map)[1],
        'description': pt_row[2],
        'type': pt_row[3],
    }
    last_docid = tools.add(new_item).execute().get_generated_ids()[0]
    tool_place_map.append((pt_row[0], last_docid))
show_collection(tools)

```

As you can see, this code follows the same pattern as the previous code creating a map for the auto increment ids for the tools to the new document id generated. Notice that power tools do not have a size attribute, but hand tools do. Thus, we add that attribute for hand tools but not power tools. This demonstrates in a small way how we can use documents with different attribute (keys) in the same collection.

To help visualize the results, we use the `show_collection()` function to print the contents of the collection.

The next collection we convert is the organizers collection. Like before, we simply read the rows in the table and insert them into the collection. Listing 7-16 shows the code for converting the organizer table to the organizers collection.

Listing 7-16. Populate the Organizers Collection

```
# Get organizers
org_results = organizer_tbl.select('Id', 'Description', 'Type', 'Width',
'Depth', 'Height', 'PlaceId').execute()
organizer_place_map = []
for org_row in org_results.fetch_all():
    tool_ids = get_handtool_ids(org_row[0])
    tool_ids.extend(get_powertool_ids(org_row[0]))
    tool_docids = [get_mapping(item, tool_place_map)[1] for item in tool_ids]
    new_item = {
        'description': org_row[1],
        'type': org_row[2],
        'width': org_row[3],
        'depth': org_row[4],
        'height': org_row[5],
    }
    if tool_docids:
        new_item.update({'tool_ids': tool_docids})
    last_docid = organizers.add(new_item).execute().get_generated_ids()[0]
    # We also need to save the mapping of organizers to storage places
    organizer_place_map.append((org_row[0], last_docid))
show_collection(organizers)
```

While this code also follows the same pattern as before, we create the map of organizer ids to new document ids. However, since the organizer table in the database had a reference to the tools via the place table, we use the helper functions to retrieve the tool id that matches this organizer from the table. We then build an array of tool ids and store that in the attribute `tool_ids`. Take a moment to see how this works.

To help visualize the results, we use the `show_collection()` function to print the contents of the collection.

The next collection we convert is the `toolchests` collection. This is the first of the collections we will break out of the storage table making a separate collection for each of the storage equipment. Since we have more than one storage equipment in the storage table, we will restrict the result to those with the type set to `toolchest`. Like before, we simply read the rows in the table and insert them into the collection. Listing 7-17 shows the code for converting the storage table to the `toolchests` collection.

Listing 7-17. Populate the Toolchests Collection

```
# Get the toolchests
tc_results = storage_tbl.select('Id', 'VendorId', 'Description', 'Width',
'Depth', 'Height', 'Location').where("Type = 'Toolchest'").execute()
# For each toolbox, get its storage places and insert into the collection
for tc_row in tc_results.fetch_all():
    new_tc = {
        'vendorid': get_mapping(tc_row[1], vendor_id_map)[1],
        'description': tc_row[2],
        'width': tc_row[3],
        'depth': tc_row[4],
        'height': tc_row[5],
        'location': tc_row[6],
    }
    _id = toolchests.add(new_tc).execute().get_generated_ids()[0]
# Now, generate the tool locations for this document
tool_locations = []
for pl_row in get_places(tc_row[0]).fetch_all():
    # Get all organizers and tools that are placed here
    tool_ids = get_handtool_ids(pl_row[5])
    tool_ids.extend(get_powertool_ids(pl_row[5]))
    tool_docids = []
    org_ids = get_organizer_ids(pl_row[5])
    if org_ids:
        for org_id in org_ids:
            map_found = get_mapping(org_id, organizer_place_map)
            if map_found:
                tool_docids.append(map_found[1])
```

```

for item in tool_ids:
    map_found = get_mapping(item, tool_place_map)
    if map_found:
        tool_docids.append(map_found[1])
if pl_row[0] == 'Shelf':
    new_item = {
        'type': 'Shelf',
        'description': pl_row[1],
        'width': pl_row[2],
        'depth': pl_row[3],
        'height': pl_row[4],
    }
    if tool_docids:
        new_item.update({'tool_ids': tool_docids})
    loc_id = locations.add(new_item).execute().get_generated_ids()[0]
    tool_locations.append(loc_id)
else: # drawer is the only other value for type
    new_item = {
        'type': 'Drawer',
        'description': pl_row[1],
        'width': pl_row[2],
        'depth': pl_row[3],
        'height': pl_row[4],
    }
    if tool_docids:
        new_item.update({'tool_ids': tool_docids})
    loc_id = locations.add(new_item).execute().get_generated_ids()[0]
    tool_locations.append(loc_id)
if len(tool_locations) > 0:
    toolchests.modify('_id = :param1') \
        .bind('param1', _id) \
        .set('tool_locations', tool_locations).execute()
show_collection(toolchests)

```

This code starts out the same way as before by getting the rows from the table and creating a new document for the collection. However, this becomes a bit more complicated because we must convert the places table entries to the `tool_locations` array. This requires using the helper functions to build a list of the ids from the database `handtool` and `powertool` tables as well as the ids from the `organizer` table because, from experience, we know an organizer can be placed in a toolchest.

However, we also need to check the places table to find the storage locations from the database and convert those to the new `locations` collection. We use the tool ids found to update the document in the collection with the new list of tool ids. This sounds complicated, but if you take a moment to study the code, you will see we do this with the helper functions more easily.

To help visualize the results, we use the `show_collection()` function to print the contents of the collection.

For brevity, we will omit the code for the other collections (`cabinets`, `shelving_units`, and `workbenches`) as they follow the same pattern as the `toolchests` conversion code. Like before, we simply read the rows in the table and insert them into the new collection.

Step 4: Add Locations

The last step is used to populate the location for each tool and organizer. Recall from the database tables, we used a table reference to find the location. However, since we have a document store, we can simply use a string that is built in the code. This saves a reference that we don't need to maintain, rather, we set it when the location is set on the create and update operations.

To perform this, we use another helper function to build a string for the location. We update all the documents in the tools and organizers collections. Listing 7-18 shows the code for building the location string.

Listing 7-18. Build Location String and Update the Tools and Organizer Collections

```
# Add the location for each tool
tool_results = tools.find().execute().fetch_all()
for tool in tool_results:
    _id = tool["_id"]
    try:
        location = get_tool_location(_id)
```



```

    if location:
        r = tools.modify('_id = :param1').bind('param1', _id).
            set('location', location[0]).execute()
    except Exception as err:
        print(err)
        exit(1)
show_collection(tools)
# Add the location for each organizer
org_results = organizers.find().execute().fetch_all()
for org in org_results:
    _id = org["_id"]
    try:
        location = get_tool_location(_id)
        if location:
            r = organizers.modify('_id = :param1').bind('param1', _id).
                set('location', location[0]).execute()
    except Exception as err:
        print(err)
show_collection(organizers)
show_collection(locations)

```

Notice we simply get all the documents in each collection and update the document with the new string. At the end, we print the documents in the collection (for debugging).

Now that we've seen all the steps, we can execute the code. Since this is a very long script, we will use Python to execute the code, but you could use the shell to execute the steps individually one at a time. In fact, that would be the preferred method if you have never written code like this before.

You may think this was a lot of work, but it can come in handy as you develop your application. Especially if you are replacing an older relational database application that is still in use. More specifically, you can run this script several times during development to refine it and the new application. Better still, you can use the script in the process of switching to the new application.

Fortunately, you can find the completed code in a file on the book web site named `convert_rdb.py`. Listing 7-19 shows an excerpt of running the script.

Listing 7-19. Executing the Conversion Script

```
C:\Users\cbell\MySQL Shell\source\Ch07> mysqlsh --py -f convert_rdb.py
User Id: root
Password: *****
COLLECTION: vendors
{
  "_id": "00005cae7415000000000000016e",
  "name": "Kobalt",
  "sources": "Lowe's",
  "url": "https://www.kobalttools.com/"
}
{
  "_id": "00005cae7415000000000000016f",
  "name": "Craftsman",
  "sources": "Lowe's, Ace",
  "url": "https://www.craftsman.com/"
}
{
  "_id": "00005cae74150000000000000170",
  "name": "Irwin",
  "sources": "Lowe's",
  "url": "https://www.irwin.com/"
}
...
{
  "_id": "00005cae74150000000000000e41",
  "depth": 12,
  "description": "Top",
  "height": 24,
  "tool_ids": [
    "00005cae74150000000000000df5",
    "00005cae74150000000000000e0d"
  ],
  "type": "Shelf",
  "width": 96
}
```

```
{
  "_id": "00005cae7415000000000000e42",
  "depth": 48,
  "description": "Bottom",
  "height": 42,
  "tool_ids": [
    "00005cae7415000000000000e0a"
  ],
  "type": "Shelf",
  "width": 96
}
```

This will fully populate the `garage_v2` schema and collections. However, if you're wondering if you need to create a script like this for every conversion or data generation. The answer is you might not.

Importing Data to a Document Store

There is a nifty utility available in MySQL Shell that helps importing JSON documents into your collections. The shell has a utility named the JSON import utility which allows you to import JSON documents directly into your collections. Thus, if you have data that is in JSON form or you can write a script to get it into JSON format, you can use the JSON import utility to import the documents in one pass. How cool is that?

For example, suppose you have data that you've read from a file or some other input stream and generated JSON documents. If you write those to a file (without commas between the documents), you can use the utility to import all documents in one pass. Let's see how this is done using our vendor data from the preceding text.

We begin with a file where each document is presented in a JSON string like the following. Notice there are no commas between the documents. Also notice we don't have the `_id` attribute (but you could add it if you wanted to generate the document id yourself).

```
{
  "name": "Kobalt",
  "sources": "Lowe's",
  "url": "https://www.kobalttools.com/"
}
```

```

{
  "name": "Craftsman",
  "sources": "Lowes, Ace",
  "url": "https://www.craftsman.com/"
}
{
  "name": "Irwin",
  "sources": "Lowes",
  "url": "https://www.irwin.com/"
}
...

```

To import the document, you can use the shell to connect to the server, then use the `util` built-in class and the `import_json()` method specifying the path to the file you want to import and a dictionary of options to include the schema and collection. Listing 7-20 demonstrates import a file with the JSON documents into the `vendors` collection in the `garage_v2` schema. Notice the import is a good convenience utility for importing large amounts of (JSON) data.

Listing 7-20. Running the JSON Import Utility in the Shell

```

> mysqlsh --py --uri root@localhost:33060
> options = {
    -> 'schema': 'garage_v2',
    -> 'collection': 'vendors',
    -> }
    ->
> util.import_json('vendors.json', options)
Importing from file "vendors.json" to collection `garage_v2`.`vendors` in
MySQL Server at localhost:33060

.. 35
Processed 4.68 KB in 35 documents in 0.0161 sec (2.17K documents/s)
Total successfully imported documents 35 (2.17K documents/s)

```

As you can see, the utility will read the documents and insert them into the schema and collection you specify. The utility will also validate the JSON document before inserting; so if there are errors, you will see them reported and the import will stop.

The utility also permits execution in command line mode by specifying the import parameters and connection on the command line as shown in the following text. There are also a few other options you can use including importing data to a JSON column in a relational table as well as support for importing binary JSON (BSON) data. Nice! See <https://dev.mysql.com/doc/mysql-shell/8.0/en/mysql-shell-utilities-json.html> for more information about the JSON import utility.

```
> mysqlsh --py --uri root@localhost:33060 --schema=garage_v2 \
    --import vendors.json vendors
Please provide the password for 'root@localhost:33060': ****
Importing from file "vendors.json" to collection `garage_v2`.`vendors` in
MySQL Server at localhost:33060

.. 35
Processed 4.68 KB in 35 documents in 0.0097 sec (3.61K documents/s)
Total successfully imported documents 35 (3.61K documents/s)
```

Ok, now that we have a full populated document store, we can look at how the code is written for the schema classes.

Demonstration

The sample application follows the same execution as the sample application in Chapter 5. The difference is we will be using a base class for the CRUD operations and a sub class for each collection to handle the validation unique to the collection. We can do this because we are extracting the data layout (set of attributes) out of the data and into the code. Thus, the base operations for the CRUD methods are the same for all collections. We'll see how this works later in this section.

More specifically, we will see demonstrations of how to create the base class first (GarageCollection) then move on to some of the other classes. As you will see, they follow the same design pattern/layout so once you've seen one or two, the others are easy to predict. Thus, we will see detailed walkthroughs using a couple of the classes and the rest will be demonstrated and presented with fewer details for brevity.

If you want to follow along, be sure to have the sample schema loaded and MySQL Shell ready to go. You may also want to use a code or text editor to write the code modules. More importantly, you should create a folder named schema and start the shell from the parent folder.

For example, you should create a folder named `mygarage_v2` and in that folder, create the schema folder. We would then execute the shell from `mygarage_v2`. Why? Because will use the Python import directive and name the path to the code module using the folder name (e.g., `from schema import Locations`). We will also be creating unit tests and thus will need a folder named `unittests` where we will store the test files.

Let's begin with the `MyGarage` class. Recall, this is the utility class that handles getting a connection to the server and fetching the schema.

MyGarage Class

This class is intended to make it easier to work with the MySQL server by providing a mechanism to login (connect) to the server and encapsulate some of the common operations such as getting the session, current database, checking to see if the connection to MySQL is active, disconnecting, etc. We also will include methods to convert an SQL result or select result to a Python list (array) for easier processing. Table 7-3 shows the complete list of methods we will create for this class including the parameters required (some methods do not require them).

Table 7-3. *MyGarage Class Methods*

Method	Parameters	Description
<code>__init__()</code>	<code>mysqlx_sh</code>	Constructor – provide <code>mysqlx</code> if running from MySQL Shell.
<code>connect()</code>	<code>username</code> , <code>passwd</code> , <code>host</code> , <code>port</code>	Connect to a MySQL server at <code>host</code> , <code>port</code> .
<code>get_session()</code>		Return the session for use in other classes.
<code>get_schema()</code>		Return the schema for use in other classes.
<code>is_connected()</code>		Check to see if connected to the server.
<code>disconnect()</code>		Disconnect from the server.
<code>get_locations()</code>		Return a Python array with all the locations where a tool or organizer can be placed.
<code>build_storage _contents()</code>	<code>tool_location</code>	Return a Python array for the tools in a tool location.
<code>vendor_in_use()</code>	<code>vendor document id</code>	Returns <code>True</code> if vendor is used in any of the collections.

Writing the Source Code

Most of these methods are the same as we saw in Chapter 5. However, the last three are different. We need these to manage the locations for selecting a place to put a tool, get a list of all the tools in a specific tool location, and implement referential integrity for delete operations on the vendors collection. That is, we use code to ensure no one deletes a vendor that is being referenced from another document.

Listing 7-21 shows the code for the MyGarage class. While this code may seem longer than version 1 (and it is), it is because we moved the location string handling and vendor referential integrity to code.

Note Comments and non-essential lines have been removed in the source code listings in this chapter for brevity.

Listing 7-21. MyGarage Class Code

```
class MyGarage(object):
    def __init__(self, mysqlx_sh=None):
        self.session = None
        if mysqlx_sh:
            self.mysqlx = mysqlx_sh
            self.using_shell = True
        else:
            self.mysqlx = mysqlx
            self.using_shell = False
        self.schema = None
    def connect(self, username, passwd, host, port):
        config = {
            'user': username,
            'password': passwd,
            'host': host,
            'port': port,
        }
        try:
            self.session = self.mysqlx.get_session(**config)
```

```

except Exception as err:
    print("CONNECTION ERROR:", err)
    self.session = None
    raise
self.schema = self.session.get_schema('garage_v2')
def get_session(self):
    return self.session
def get_schema(self):
    return self.schema
def is_connected(self):
    return self.session and (self.session.is_open())
def disconnect(self):
    try:
        self.session.close()
    except Exception as err:
        print("WARNING: {0}".format(err))
def get_locations(self, include_organizers=True):
    tool_locations = []
    if include_organizers:
        organizers = self.schema.get_collection('organizers').find().\
            fields("_id", "type", "description").execute().fetch_all()
        for organizer in organizers:
            list_item_str = "{0} - {1}"\
                .format(organizer["type"], organizer["description"])
            tool_locations.append((list_item_str, list_item_str))
    storage_collections = ['toolchests', 'shelving_units',
                          'workbenches', 'cabinets']
    for storage_collection in storage_collections:
        collection = self.schema.get_collection(storage_collection)
        items = collection.find().fields("_id", "description",
                                         "tool_locations").execute().fetch_all()
        for item in items:
            locations_found = item["tool_locations"]
            if locations_found:
                for tool_loc_id in locations_found:

```



```

        tool_location = self.schema\
            .get_collection("locations")\
            .find('_id = :param1')\
            .bind('param1', tool_loc_id).execute().fetch_all()
    if tool_location:
        list_item_str = "{0}, {1} - {2}"\
            .format(item["description"],
                    tool_location[0]["description"],
                    tool_location[0]["type"])
        tool_locations.append((list_item_str,
                               list_item_str))

    return tool_locations
def build_storage_contents(self, tool_locations):
    storage_places = []
    tools = self.schema.get_collection('tools')
    organizers = self.schema.get_collection('organizers')
    locations = self.schema.get_collection('locations')
    if not tool_locations:
        return storage_places
    list_of_tools = []
    for loc_id in tool_locations:
        tool_location = locations.find("_id = :param1")\
            .bind("param1", loc_id).execute().fetch_all()
        if not tool_location or tool_location == []:
            organizer = organizers.find("_id = :param1")\
                .bind("param1", loc_id).execute().fetch_all()
            if not organizer or organizer == []:
                continue # This is an error!
            description = organizer[0]['description']
            loc_type = organizer[0]['type']
            list_of_tools.append(('organizers', loc_type,
                                description, 'organizer', ' '))
        continue
    else:

```

```

try:
    tool_id_list = tool_location[0]['tool_ids']
except KeyError:
    tool_id_list = []
description = tool_location[0]['description']
loc_type = tool_location[0]['type']
tool_list_str = '_id in [{0}]'.format(
    ', '.join(['{0}'.format(t_id) for t_id in tool_id_list]))
found_tools = tools.find(tool_list_str).execute().fetch_all()
for tool in found_tools:
    size = dict(tool).get('size', ' ')
    list_of_tools.append(('tools', tool['type'],
                        tool['description'],
                        tool['category'], size))
storage_places.append((loc_type, description, list_of_tools))
list_of_tools = []
return storage_places

def vendor_in_use(self, vendor_id):
    collections = ['cabinets', 'shelving_units', 'toolchests',
                  'tools', 'workbenches']
    for collection_name in collections:
        collection = self.schema.get_collection(collection_name)
        res = collection.find('vendorid = :param1').\
            bind('param1', vendor_id).execute().fetch_all()
        if res:
            return True
    return False

```

This code module, `garage_v2`, also includes a helper function named `make_list()` that we can use to make a list of Python arrays from a read operation. Listing 7-22 shows the code for this function. Take a moment to read through it and you will see it is simple code for converting the result. We can use this method in the collection classes to help work with results from the schema.

Listing 7-22. Helper Function

```
def make_list(results, key_list):
    """Build list of Python arrays from results

    Return a Python array for the list of documents returned from a read
    operation.
    """
    result_list = []
    for result in results:
        item_values = []
        for key in key_list:
            try:
                item_values.append(result[key])
            except KeyError:
                # If key not found, create a placeholder
                item_values.append("")
        result_list.append(item_values)
    return result_list
```

Testing the Class

Before we embark on testing the class, we must set the Python path variable (PYTHONPATH) to include the folder from which we want to run our tests. This is because we are using modules that are not installed at the system level, rather, are in a folder relative to the code we're testing. In Windows, you can use the following command to add the path for the execution to the Python path.

```
C:\Users\cbell\Documents\my_garage_v1> set PYTHONPATH=%PYTHONPATH%;c:\
users\cbell\Documents\mygarage_v1
```

Tip If your path has spaces, make sure you use quotes around the path.

Or, on Linux and macOS, you can use this command to set the Python path.

```
export PYTHONPATH=$(pwd);$PYTHONPATH
```

Now we can run the shell. For this, we will start in Python mode using the `--py` option. Let's exercise some of the methods in the class. We can do try all of them out except the `make_rows()` methods. We'll see those later. Listing 7-23 shows how to import the class in the shell, initialize (create) a class instance named `mygarage`, then connect with `connect()` and execute some of the methods. We close with a call to `disconnect()` to shut down the connection to the server.

Listing 7-23. Testing MyGarage using MySQL Shell

```
C:\Users\cbell\Documents\mygarage_v2>mysqlsh --py
> from schema.garage_v2 import MyGarage
Running from MySQL Shell. Provide mysqlx in constructor.
> myg = MyGarage(mysqlx)
> myg.connect('root', 'root', 'localhost', 33060)
> schema = myg.get_schema()
> s = myg.get_session()
> myg.is_connected()
true
> myg.disconnect()
> myg.is_connected()
false
```

Next, we will create a unit test for this class in a similar way we did in Chapter 5. In fact, we will create a test named `garage_v2_test.py` in the `unittests` folder that will use nearly the same code from Chapter 5 changing only the database to `schema` and `v1` to `v2` occurrences in the import statements. Thus, we present the code without further explanation in listing 7-24.

Listing 7-24. `garage_v2_test.py`

```
from __future__ import print_function

from getpass import getpass
from schema.garage_v2 import MyGarage

print("MyGarage Class Unit test")
mygarage = MyGarage(mysqlx)
user = raw_input("User: ")
```

```

passwd = getpass("Password: ")
print("Connecting...")
mygarage.connect(user, passwd, 'localhost', 33060)
print("Getting the schema...")
schema = mygarage.get_schema()
print(schema)
print("Getting the session...")
session = mygarage.get_session()
print(session)
print("Connected?")
print(mygarage.is_connected())
print("Disconnecting...")
mygarage.disconnect()
print("Connected?")
print(mygarage.is_connected())

```

Executing the unit test is also like how we did it in Chapter 5. Listing 7-25 shows this test run from the shell.

Listing 7-25. Running the garage_v1_test Unit Test

```

> mysqlsh --py -f unittests\garage_v2_test.py
Running from MySQL Shell. Provide mysqlx in constructor.
MyGarage Class Unit test
User: root
Password:
Connecting...
Getting the schema...
<Schema:garage_v2>
Getting the session...
<Session:root@localhost:33060>
Connected?
True
Disconnecting...
Connected?
False

```

Now, let's look at the base class that forms the foundation for the collection classes.

Collection Base Class

As mentioned, we will create a base class that contains all the CRUD operations for a collection. The reason we can use the same methods for all our collections is because the format, layout, or simply the field list for the operations is governed by the JSON document itself. Thus, by simply using the methods for the Collection class, we can simplify our development by using a base class that does all the CRUD operations and use sub classes to handle any constraints on the data. Recall, the constraints we will impose have to do with required fields and in some cases referential integrity.

The methods in the base class should be familiar since they are the same we used in Chapter 5 with a few added for convenience and validation. Table 7-4 shows each of the methods in the base class including a short description of each.

Table 7-4. *Methods for the GarageCollection Base Class*

Method	Parameters	Description
<code>__init__()</code>	Schema, collection name	Constructor
<code>check_create_prerequisites()</code>	JSON document	Check data prior to CREATE
<code>check_upate_prerequisites()</code>	JSON document	Check data prior to UPATE
<code>create()</code>	JSON document	Perform the CREATE operation
<code>read()</code>	Document Id	Perform the READ operation
<code>update()</code>	JSON document	Perform the UPATE operation
<code>delete()</code>	Document Id	Perform the DELETE operation
<code>get_last_docid()</code>		Return the last document id generated.
<code>get_tool_locations()</code>	Document Id	Returns the list of tool locations.

Notice we have the expected CRUD operation methods, but we also see methods for checking the prerequisites for create and update operations. These are set to return True by default with the expectation that the sub classes that need these methods will populate them for collection-specific requirements.

Notice also we have helper functions for retrieving the last document id generated, which is helpful for create operations and a method to get the list of tool locations. This last method is not strictly necessary as you can access the array using path expressions, but it makes the class more tidy and easier to use (and read) in code.

Writing the Source Code

The code for the base class implements the methods in preceding text using the name of the collection passed in the constructor to get an instance of the collection from the schema. This allows us to use the same X DevAPI calls for each of the CRUD operation no matter which collection we're using. In fact, since we moved the structure of the JSON document to the user interface code, we don't even need to work directly with it except for those cases where we want to validate for required fields or referential integrity.

Thus, we will create the prerequisite functions as described in preceding text to return True by default so that if a sub class (a collection) doesn't need them, the code will not stop if the prerequisite functions are not overridden in the sub class.

For example, should we not need to validate during the update operation, we simply don't include that function in the sub class definition, which means when it is called from the delete or update operation, it will still work.

Also, since we are using a base class, the sub class inherits the methods of the base (parent) class, which once again means we need only write the code once for the CRUD operations. Let's look at the code and you can see how this works. Listing 7-26 shows the code for the new base class.

Listing 7-26. The GarageCollection Base Class Code

```
class GarageCollection(object):
    def __init__(self, mygarage, collection_name):
        self.mygarage = mygarage
        self.schema = mygarage.get_schema()
        self.collection_name = collection_name
        self.col = self.schema.get_collection(collection_name)
        self.docid = None
```

```

def check_create_prerequisites(self, doc_data):
    return True
def check_update_prerequisites(self, doc_data):
    return True
def create(self, doc_data):
    if not self.check_create_prerequisites(doc_data):
        return (False, "Required fields missing.")
    try:
        json_str = {}
        for key in doc_data.keys():
            json_str.update({key: doc_data[key]})
        self.docid = self.col.add(json_str).\
            execute().get_generated_ids()[0]
    except Exception as err:
        print("ERROR: Cannot add {0}: {1}"
              "".format(err, self.collection_name))
        return (False, err)
    return (True, None)
def read(self, _id=None):
    if not _id:
        res = self.col.find().execute().fetch_all()
    else:
        res = self.col.find('_id = :param1').\
            bind('param1', _id).execute().fetch_all()
    return res
def update(self, doc_data):
    _id = doc_data.get("_id", None)
    assert _id, "You must supply an Id to update the {0}." \
        "".format(self.collection_name.rstrip('s'))
    if not self.check_update_prerequisites(doc_data):
        return (False, "Required fields missing.")
    try:
        for key in doc_data.keys():
            # Skip the _id key
            if key != '_id':

```



```

        self.col.modify('_id = :param1') \
            .bind('param1', _id) \
            .set(key, doc_data[key]).execute()
    except Exception as err:
        print("ERROR: Cannot update {0}: {1}".format(
            self.collection_name.rstrip('s'), err))
        return (False, err)
    return (True, None)
def delete(self, _id=None):
    assert _id, "You must supply an Id to delete the {0}." \
        "".format(self.collection_name.rstrip('s'))
    try:
        self.col.remove('_id = :param1').bind('param1', _id).execute()
    except Exception as err:
        print("ERROR: Cannot delete {0}: {1}"
            "".format(self.collection_name.rstrip('s'), err))
        return (False, err)
    return (True, None)
def get_last_docid(self):
    docid = self.docid
    self.docid = None # Clear it after it was read
    return docid
def get_tool_locations(self, _id=None):
    assert _id, "You must supply an Id to get the tool locations."
    results = []
    if _id:
        places = self.col.find('_id = :param1').bind('param1', _id).\
            fields("tool_locations").execute().fetch_all()
        try:
            tool_locations = places[0]["tool_locations"]
            if tool_locations:
                locations = self.mygarage.get_schema().\
                    get_collection("locations")
                tool_ids = ', '.join(["{0}"
                    ".format(tool_id) for tool_id in tool_locations])

```

```

    tool_loc_str = '_id in [{0}]'.format(tool_ids)
    results = locations.find(tool_loc_str).\
        execute().fetch_all()
except KeyError:
    results = []
return results

```

Let's see one of the sub classes. In this example, we will see the `Vendors` class, which models the vendors collection in the `garage_v2` schema. We place this code in a code file in the `schema` folder with the name `vendors.py`. The following shows the code for the `Vendors` class. Notice it is considerably less code than the `Vendor` class we used in version 1 of the sample application from Chapter 5.

```

from schema.garage_collection import GarageCollection
class Vendors(GarageCollection):
    def __init__(self, mygarage):
        """Constructor - set collection name"""
        GarageCollection.__init__(self, mygarage, 'vendors')
    def check_create_prerequisites(self, doc_data):
        """Check prerequisites for the create operation."""
        vendor_name = doc_data.get("name", None)
        assert vendor_name, "You must supply a name for the vendor."
        return True

```

The remaining classes for the collections are similarly short and contain only the validation code methods applicable for the collection. Listing 7-27 shows a composition of the collection code modules (each is saved in a separate code module) with comments removed for brevity. Take a few moments to see how using the base class makes writing the collection classes easier by allowing you to place the collection-specific code in the sub class. The sections for each collection are highlighted in bold. Once again, comments and extra lines have been removed for brevity.

Note Rather than walk you through testing each of the class modules, we'll reuse the technique from Chapter 5 and test the class modules with unit tests.

Listing 7-27. Collection Classes for MyGarage V2**# Cabinets collection - cabinets.py**

```
class Cabinets(GarageCollection):
    ...
def check_create_prerequisites(self, doc_data):
    vendor_id = doc_data.get("vendorid", None)
    description = doc_data.get("description", None)
    location = doc_data.get("location", None)
    numdoors = doc_data.get("numdoors", None)
    assert vendor_id, "You must supply a vendor id for the cabinet."
    assert description, "You must supply a description for the cabinet."
    assert numdoors, "You must supply the number of doors "\
        "for the cabinet."
    assert location, "You must supply a location for the cabinet."
    return True
```

Locations collection - locations.py

```
class Locations(GarageCollection):
    ...
def check_create_prerequisites(self, doc_data):
    loc_type = doc_data.get("type", None)
    description = doc_data.get("description", None)
    assert loc_type, "You must supply a type for the location."
    assert description, "You must supply a description for the location."
    return True
def remove_tool(self, tool_id):
    location = self.col.find(':param1 in $.tool_ids').\
        bind('param1', tool_id).execute().fetch_all()
    if location:
        tool_locations = location[0]['tool_ids']
        tool_locations.remove(tool_id)
```

Organizers collection - organisers.py

```
ORGANIZER_TYPES = [
    ('Bag', 'Bag'), ('Basket', 'Basket'), ('Bin', 'Bin'),
    ('Box', 'Box'), ('Case', 'Case'), ('Crate', 'Crate')
]
```

```

class Organizers(GarageCollection):
...
    def check_create_prerequisites(self, doc_data):
        description = doc_data.get("description", None)
        org_type = doc_data.get("type", None)
        assert description, "You must supply a description for "\
            "the organizer."
        assert org_type, "You must supply type for the organizer."
        return True
    def remove_tool(self, tool_id):
        location = self.col.find(':param1 in $.tool_ids').\
            bind('param1', tool_id).execute().fetch_all()
        if location:
            tool_locations = location[0]['tool_ids']
            tool_locations.remove(tool_id)
# Shelving Units collection - shelving_units.py
class ShelvingUnits(GarageCollection):
...
    def check_create_prerequisites(self, doc_data):
        vendor_id = doc_data.get("vendorid", None)
        description = doc_data.get("description", None)
        location = doc_data.get("location", None)
        assert vendor_id, "You must supply a vendor id for "\
            "the shelving_unit."
        assert description, "You must supply a description for "\
            "the shelving_unit."
        assert location, "You must supply a location for the shelving_unit."
        return True
# Toolchests collection - toolchests.py
class Toolchests(GarageCollection):
...
    def check_create_prerequisites(self, doc_data):
        vendor_id = doc_data.get("vendorid", None)
        description = doc_data.get("description", None)
        location = doc_data.get("location", None)

```

```

    assert vendor_id, "You must supply a vendor id for the toolchest."
    assert description, "You must supply a description for "\
        "the toolchest."
    assert location, "You must supply a location for the toolchest."
    return True

```

Tools collection - tools.py

```

TOOL_TYPES = [
    ('Adjustable Wrench', 'Adjustable Wrench'), ('Awl', 'Awl'),
    ('Clamp', 'Clamp'), ('Crowbar', 'Crowbar'), ('Drill Bit', 'Drill Bit'),
    ('File', 'File'), ('Hammer', 'Hammer'), ('Knife', 'Knife'),
    ('Level', 'Level'), ('Nutdriver', 'Nutdriver'), ('Pliers', 'Pliers'),
    ('Prybar', 'Prybar'), ('Router Bit', 'Router Bit'), ('Ruler', 'Ruler'),
    ('Saw', 'Saw'), ('Screwdriver', 'Screwdriver'), ('Socket', 'Socket'),
    ('Socket Wrench', 'Socket Wrench'), ('Wrench', 'Wrench'),
    ('Corded', 'Corded'), ('Cordless', 'Cordless'), ('Air', 'Air')
]

```

```

class Tools(GarageCollection):
    ...
    def check_create_prerequisites(self, doc_data):
        vendor_id = doc_data.get("vendorid", None)
        description = doc_data.get("description", None)
        tool_type = doc_data.get("type", None)
        category = doc_data.get("category", None)
        assert vendor_id, "You must supply a vendor id for the tool."
        assert description, "You must supply a description for the tool."
        assert category, "You must supply the category of tool "\
            "(handtool or powertool) for the tool."
        assert tool_type, "You must supply category for the tool."
        return True

```

Workbenches collection - workbenches.py

```

class Workbenches(GarageCollection):
    ...
    def check_create_prerequisites(self, doc_data):
        vendor_id = doc_data.get("vendorid", None)
        description = doc_data.get("description", None)

```

```
location = doc_data.get("location", None)
assert vendor_id, "You must supply a vendor id for the workbench."
assert description, "You must supply a description for "\
                    "the workbench."
assert location, "You must supply a location for the workbench."
return True
```

Notice there are customizations for each of the collections mainly for the validation methods derived from the base class. But some of the classes also add additional methods to permit collection-specific options.

For example, we see the method `remove_tool()` in the `Locations` and `Organizers` classes. This method allows us to remove a tool by document id from the location or organizer. In this way, we can be sure to remove a tool when it is deleted from the collection.

Also notice we added arrays for the attributes in the collections that have a known set of values (enumerated values in the relational tables from version 1). In this case, they appear in the `organizers` and `tools` collections. Recall, we mentioned these are handled in code. In the sample application for this chapter, we use Python arrays to use in the drop down lists in the user interface. Thus, we use code to establish the set of valid values. Figure 7-2 shows one such example.

Handtool - Detail

Location

Vendor

Description

ToolSize

Type

- Adjustable Wrench
- Awl
- Clamp
- Crowbar
- Drill Bit
- File
- Hammer
- Knife
- Level
- Nutdriver
- Pliers
- Prybar
- Router Bit
- Ruler
- Saw
- Screwdriver
- Socket
- Socket Wrench
- Wrench
- Corded
- Cordless
- Air

Figure 7-2. Drop down list for Tool Types

Now, let's review how we can test the class modules before we write the rest of the application.

Testing the Class Modules

We will also be using the same unit test mechanism from Chapter 5. For brevity, we will only examine one of the unit test code modules to remind us of the code. We then execute the unit tests using the same `run_all.py` code module mechanism we used in Chapter 5.

Recall, we created a base class named `CRUDTest` in the `unittests/crud_test.py` code module that implements some methods for starting (or setup) of the test, a generic method to show the rows returned, and one each for the test cases we want to run. We then created a code module with a class to test one of the collection classes (or as we called them in Chapter 5, table classes).

For example, we create a test for the Vendors class by creating a new class named VendorTests derived from CRUDTest and stored in the file unittests/vendor_test.py. Listing 7-28 shows the code for the new class. As you will see, it is very similar to the test we wrote in Chapter 5, which also demonstrates how easy it is to develop code for both an SQL and NoSQL interface – the code is very similar.

Listing 7-28. Code for the VendorTests Class

```

from __future__ import print_function

from unittests.crud_test import CRUDTest
from schema.vendors import Vendors

class VendorTests(CRUDTest):
    """Test cases for the Vendors class"""

    def __init__(self):
        """Constructor"""
        CRUDTest.__init__(self)
        self.vendors = None
        self.last_id = None
        self.vendors = None

    def setup(self, mysql_x, user=None, passwd=None):
        """Setup the test cases"""
        self.mygarage = self.begin(mysql_x, "Vendors", user, passwd)
        self.vendors = Vendors(self.mygarage)

    def create(self):
        """Run Create test case"""
        print("\nCRUD: Create test case")
        vendor_data = {
            "name": "ACME Bolt Company",
            "url": "www.acme.org",
            "sources": "looney toons"
        }
        self.vendors.create(vendor_data)
        self.last_id = self.vendors.get_last_docid()
        print("\tlast insert id = {0}".format(self.last_id))

```



```

def read_all(self):
    """Run Read(all) test case"""
    print("\nCRUD: Read (all) test case")
    docs = self.vendors.read()
    self.show_docs(docs, 5)

def read_one(self):
    """Run Read(record) test case"""
    print("\nCRUD: Read (doc) test case")
    docs = self.vendors.read(self.last_id)
    self.show_docs(docs, 1)

def update(self):
    """Run Update test case"""
    print("\nCRUD: Update test case")
    vendor_data = {
        "_id": self.last_id,
        "name": "ACME Nut Company",
        "url": "www.weesayso.co",
    }
    self.vendors.update(vendor_data)
def delete(self):
    """Run Delete test case"""
    print("\nCRUD: Delete test case")
    self.vendors.delete(self.last_id)
    docs = self.vendors.read(self.last_id)
    if not docs:
        print("\tNot found (deleted).")

```

What makes this technique powerful is we can go on to create new tests for each of the database classes named for the class and store them in the same `unittests` folder. Since the tests use the same class signature (same methods) as those in Chapter 5, we can reuse them substituting schema for database and use the plural for the collection names in the import statements and then make minor changes to the data as needed to match the new set of attributes for the collection modules.

Once all the collection class test code modules are written, we can then write a driver script that runs all the tests in a loop. Recall, the driver script is named `run_all.py` also stored in the `unittests` folder. Listing 7-29 shows the code for this module.

Listing 7-29. Test Driver `run_all.py`

```

from __future__ import print_function

from getpass import getpass
from unittests.cabinet_test import CabinetTests
from unittests.location_test import LocationTests
from unittests.organizer_test import OrganizerTests
from unittests.shelving_unit_test import ShelvingUnitTests
from unittests.toolchest_test import ToolchestTests
from unittests.tool_test import ToolTests
from unittests.vendor_test import VendorTests
from unittests.workbench_test import WorkbenchTests

print("CRUD Tests for all classes...")
crud_tests = []
cabinets = CabinetTests()
crud_tests.append(cabinets)
locations = LocationTests()
crud_tests.append(locations)
shelving_units = ShelvingUnitTests()
crud_tests.append(shelving_units)
toolchests = ToolchestTests()
crud_tests.append(toolchests)
tools = ToolTests()
crud_tests.append(tools)
organizers = OrganizerTests()
crud_tests.append(organizers)
vendors = VendorTests()
crud_tests.append(vendors)
workbenches = WorkbenchTests()
crud_tests.append(workbenches)
user = raw_input("User: ")

```

```
passwd = getpass("Password: ")
for test in crud_tests:
    test.set_up(mysqlx, user, passwd)
    test.create()
    test.read_one()
    test.read_all()
    test.update()
    test.read_one()
    test.delete()
    test.tear_down()
```

To execute this test, you can use the command shown in Listing 7-30 with the expected output. Here, we see only a portion of the output for brevity.

Listing 7-30. Executing the test driver

```
C:\Users\cbell\Documents\mygarage_v2>mysqlsh --py -f unittests\run_all.py
Running from MySQL Shell. Provide mysqlx in constructor.
CRUD Tests for all classes...
User: root
Password: *****

*** Cabinets Class Unit test ***

Connecting...

CRUD: Create test case
      Last insert id = 00005cc4bca00000000000000001

CRUD: Read (doc) test case

First 1 docs:
-----
{
  "_id": "00005cc4bca00000000000000001",
  "depth": 11,
  "description": "Large freestanding cabinet",
  "height": 11,
  "location": "Read wall next to compressor",
```

```

    "numdoors": 2,
    "shelves": [
      {
        "depth": 20,
        "description": "Middle",
        "height": 18,
        "width": 48
      }
    ],
    "vendorid": "00005cae741500000000000000cd6",
    "width": 11
  }
}

```

...

CRUD: Update test case

CRUD: Read (doc) test case

First 1 docs:

```

{
  "_id": "00005cc4bca00000000000000001",
  "depth": 11,
  "description": "Cold Storage",
  "height": 11,
  "location": "3rd floor basement",
  "numdoors": 2,
  "shelves": [
    {
      "depth": 20,
      "description": "Top",
      "height": 18,
      "width": 48
    },
    {
      "depth": 20,
      "description": "Bottom",

```

```

        "height": 18,
        "tool_ids": [
            "00005cafa3eb00000000000007c5",
            "00005cafa3eb00000000000007c6",
            "00005cafa3eb00000000000007c7"
        ],
        "width": 48
    }
],
"vendorid": "00005cae74150000000000000cd6",
"width": 11
}

```

CRUD: Delete test case
 Not found (deleted).

Disconnecting...

...

Take some time to download the code from the book web site and test out the unit tests yourself. You are encouraged to download the source code for this sample application and test it out yourself. Dig into the code and see how it all works. Using this sample application as a guide may surprise you with the ease with which you can create your own document store applications.

You should notice it is very easy to use this concept and you can develop others like help test your database code. Just think; we did this all without having to write any user interface code, which allows to validate our database code before the first line of user interface code is written. Nice!

Summary

Creating and coding an application for the document store (NoSQL) may seem less intuitive than traditional relational database application, but now that you've seen the power behind placing control of the data directly into code, you can see that document store applications are easier to write. Indeed, they use similar CRUD operations as we saw in the relational database example from Chapter 5.

In this chapter, we saw how to use the shell to develop a NoSQL application. We discovered how to write the CRUD operations for working with the document store and how to write modular schema classes to manage the operations. We also saw how to write tests to exercise the schema classes. This demonstrated the utility of the shell for use in developing applications in Python with the X DevAPI.

In the next chapter, we will look at another major feature in MySQL that you can use the shell to manage – MySQL Group Replication. As you will see, Group Replication is a major step forward in high availability for MySQL.

CHAPTER 8

Using the Shell with Group Replication

One of the most advanced features built into MySQL is its ability to connect two or more servers together where all servers maintain a copy (replica) of the data for redundancy. Database administrators and systems architects who manage infrastructures understand the need for building in redundancy while keeping maintenance chores to a minimum. One of the tools used to achieve this is a class of features that make the server or service available as much as possible. We call this high availability (HA).

Not only is high availability a key factor in establishing robust, always ready infrastructures, it is also a quality of robust, enterprise-grade database systems. Oracle has continued to develop and improve the high availability features in MySQL. Indeed, they have matured to include detailed management and configuration, status reporting, and even automatic failover of the primary to ensure your data is available even if the primary server goes down. Best of all, Oracle has included these features in the community edition of MySQL so the whole world can use them.

In this chapter, we will get a glimpse of how we can setup, configure, and maintain high availability in MySQL. Since high availability is such a large topic with many facets to consider, we will first take a short tour of high availability – what it is and how it works followed by the high availability features of MySQL – and we will see how basic replication works. Then, in the next chapter, we will see a demonstration of how you can set up and configure MySQL Group Replication on your own systems with MySQL Shell.

Tip This chapter prepares you to begin working with MySQL Group Replication by studying classic replication. If you are familiar with MySQL Replication and know how to set up classic replication with binary log files and know what global transaction identifiers (GTIDs) are, you may want to skim this chapter.

Overview

MySQL high availability is a collection of components built upon the long-term stability of MySQL Replication. The components include modifications to the server and new features and components such as global transaction identifiers, numerous improvements to the core replication feature set, and the addition of MySQL Group Replication, hence Group Replication. Together, these components form a new paradigm in MySQL high availability.

As you will see, there is quite a lot of power and several options for using high availability in MySQL. Let's begin with a brief tutorial on high availability.

What is High Availability?

High availability is easiest to understand if you consider it loosely synonymous with reliability—making the solution as accessible as possible and tolerant to failures either planned or unplanned for an agreed upon period. That is, it's how much users can expect the system to be operational. The more reliable the system and thus the longer it is operational equates to a higher level of availability.

High availability can be accomplished in many ways, resulting in different levels of availability. The levels can be expressed as goals to achieving some higher state of reliability. Essentially, you use techniques and tools to boost reliability and make it possible for the solution to keep running and the data to be available as long as possible (also called uptime). Uptime is represented as a ratio or percentage of the amount of time the solution is operational.

You can achieve high availability by practicing the following engineering principles:

- *Eliminate single points of failure:* Design your solution so that there are as few components as possible that, should they fail, render the solution unusable.
- *Add recovery through redundancy:* Design your solution to permit multiple, active redundant mechanisms to allow rapid recovery from failures.
- *Implement fault tolerance:* Design your solution to actively detect failures and automatically recover by switching to a redundant or alternative mechanism.

These principles are building blocks or steps to take to reach higher levels of reliability and thus high availability. Even if you do not need to achieve maximum high availability (the solution is available nearly all the time), by implementing these principles, you will make your solution more reliable at the least, which is a good goal to achieve.

Now that you understand the goals or requirements that high availability (HA) can solve, let's now discuss some of the options for implementing HA in your MySQL solutions. There are four options for implementing goals of high availability. By implementing all of these, you will achieve a level of high availability. How much you achieve depends on not only how you implement these options but also how well you meet your goals for reliability.

- *Recovery*: The easiest implementation of reliability you can achieve is the ability to recover from failures. This could be a failure in a component, application server, database server, or any other part of the solution. Recovery therefore is how to get the solution back to operation in as little time and cost as possible.
- *Logical Backup*: A logical backup makes a copy of the data by traversing the data, making copies of the data row by row, and typically translating the data from its binary form to SQL statements. The advantage of a logical backup is the data is human readable and can even be used to make alterations or corrections to the data prior to restoring it. The downside is logical backups tend to be slow for larger amounts of data and can take more space to store than the actual data (depending on data types, number of indexes, and so on).
- *Physical Backup*: A physical backup makes a binary copy of the data from the disk storage layer. The backup is typically application specific; you must use the same application that made the backup to restore it. The advantage is the backup is much faster and smaller in size. Plus, applications that perform physical backups have advanced features such as incremental backups (only the data that has changed since the last backup) and other advanced features. For small solutions, a logical backup may be more than sufficient, but as your solution (your data) grows, you will need to use a physical backup solution.

- *Redundancy*: One of the more challenging implementations of reliability is redundancy – having two or more components serving the same role in the system. A goal for redundancy may be simply having a component in place in case you need to replace the primary. This could be a hot standby where the component is actively participating in parallel with the primary where your system automatically switches to the redundant component when a failure is detected. The most common target for redundancy is the database server.
- *Scaling*: Another reliability implementation has to do with performance. In this case, you want to minimize the time it takes to store and retrieve data. You do this by designing your solution to write (save) data to the master (primary) and read the data from the slave (secondary). As the application grows, you can add additional slaves to help minimize the time to read data. By splitting the writes and reads, you relieve the master of the burden of having to execute many statements. Given most applications have many more reads than writes, it makes sense to devote a different server (or several) to providing data from reading and leaving the writes to the one master server.
- *Fault Tolerance*: The last implementation of reliability and indeed what separates most high availability solutions regarding uptime is fault tolerance, which is the ability to detect failures and recover from the event. Fault tolerance is achieved by leveraging recovery and redundancy and adding the detection mechanism and active switchover. That is, when a component fails, another takes its place. The best form of fault tolerance implements automatic switching (failover) that permits applications to continue running even when a single component fails.

Note There are two forms of scale out: read and write. You can achieve read scale out using redundant readers such as multiple slaves in MySQL Replication, but achieving write scale out requires a solution that can negotiate and handle updates on two or more servers. Fortunately, we have MySQL Group Replication. We will see more about this feature in a later section.

MySQL High Availability Features

MySQL has several high availability features that meet the more challenging of these techniques and their goals. The following summarizes the features you can use for each of the preceding techniques.

- *Recovery*: MySQL Replication provides a mechanism through the replica to provide recovery. Should the master go down, a slave can be promoted to take on its role.
- *Logical Backup*: The client utility `mysqldump` (or the older `mysqldump`) can be used to make logical backups of your data for restoring later. Be aware that these tools have all the challenges with a logical backup. Specifically, they dump the database into a file constructed of SQL CREATE and INSERT statements, so they may not be best for large datasets.
- *Physical Backup*: The Enterprise Edition of MySQL provides a tool called MySQL Enterprise Backup, which can be used to create physical backups of your data and restore them.
- *Redundancy*: MySQL Replication through its use of multiple slaves can permit some redundancy with certain tips and tricks such as using a dedicated slave with the same hardware as the master (called a hot standby) or using multiple slaves to ensure you always have enough read access to your data.
- *Scaling*: Once again, MySQL Replication can provide read scale out for your applications. Write scale out is achieved using MySQL Group Replication.
- *Fault Tolerance*: You can use MySQL replication to achieve the switch. That is, when the master goes down, we use the replication commands in MySQL to switch the role of master to one of the slaves. There are two types of the master role change when working with MySQL: switchover, which is switching the role of master to a slave when the master is still operational, and failover, which is selecting a slave to take on the role of master when the master is no longer operational. That is, switchover is intentional and failover is a reactive event. However, you can use Group Replication, which permits automatic failover.

Note There is also MySQL Router, which is a connection router for MySQL that allows you to set up a specific set of servers to be used by the router such that the router automatically switches to another server should the current server go offline (become unreachable). Fortunately, both Group Replication and Router are part of InnoDB Cluster, which we will explore in Chapter 10.

Now that we know what high availability is and some of the techniques and goals for achieving it, let's dive into the fundamental high availability feature – MySQL Replication.

What is MySQL Replication?

MySQL Replication is an easy-to-use feature and yet a complex and major component of the MySQL server. This section presents a bird's-eye view of replication for the purpose of explaining its features. We will see an example of setting up MySQL Replication in the next section.

MySQL Replication requires two or more servers. One server must be designated as the primary or master. The master role means all data changes (writes) to the data are sent to the master and only the master. All other servers in the topology maintain a copy of the master data and are by design and requirement read-only (RO) servers called secondaries or slaves. Thus, when your applications send data for storage, they send it to the master. Applications you write to use the sensor data can read it from the slaves.

The copy mechanism works using a technology called the binary log that stores the changes in a special format, thereby keeping a record of all the changes. These changes are then shipped to the slaves where the changes are stored in a similar log file (called the relay log) then read and executed on the slave. Thus, once the slave executes the changes (called events), it has an exact copy of the data.

At the lowest level, the binary log exchanges between the master and slaves support three formats: statement-based replication (SBR), which replicates entire SQL statements; row-based replication (RBR), which replicates only the changed rows; and for certain scenarios mixed-based replication (MBR), which is a hybrid of RBR with some events recorded using SQL statements.

As you can imagine, there is a very slight delay from the time a change is made on the master to the time it is made on the slave. Fortunately, this delay is almost unnoticeable except in topologies with high traffic (lots of changes). For your purposes, it is likely

when you read the data from the slave, it is up to date. You can check the slave's progress using the command `SHOW SLAVE STATUS`; among many other things, it shows you how far behind the master the slave has become. You see this command in action in a later section.

MySQL Replication supports two methods of replication. The original (sometimes called classic replication, binary log file and position replication, or simply log file and position replication) method involves using a binary log file name and position to keep track of and execute events or apply changes to synchronize data between the master and slaves. A newer, transactional method uses global transaction identifiers (GTIDs) and therefore does not require working with log files or positions (but it still uses the same binary log and relay log files), which greatly simplifies many common replication tasks. Best of all, replication using GTIDs guarantees consistency between master and slave because when a slave connects to a master, the new protocol permits slaves to request the GTIDs that are missing. Thus, each slave can negotiate with the master to receive the missing events.

WHAT ARE GTIDS?

GTIDs enable servers to assign a unique identifier to each set or group of events thereby making it possible to know which events have been applied on each slave. To perform failover with GTIDs, one takes the best slave (the one with the least missing events and the hardware that matches the master best) and make it a slave of every other slave. We call this slave the candidate slave. The GTID mechanism will ensure only those events that have not been executed on the candidate slave are applied. In this way, the candidate slave becomes the most up to date and therefore a replacement for the master.

MySQL Replication also supports two types of synchronization. The original type, asynchronous, is one-way where events executed on the master are transmitted to the slaves and executed (or applied) as they arrive with no checks to ensure the slaves are all at the same synchronization point as the master (slave updates may be delayed when there are many transactions). The other type, semi-synchronous, where a commit performed on the master blocks before returning to the session that performed the transaction until at least one slave acknowledges that it has received and logged the events for the transaction.

Synchronous replication, where all nodes are guaranteed to have the same data in an all or none commit scenario is supported by MySQL NDB Cluster. See the MySQL NDB Cluster section in the online reference manual for information about synchronous replication.

Tip For more information about replication, see the online MySQL reference manual (<https://dev.mysql.com/doc/refman/8.0/en/replication.html>).

What is Group Replication?

MySQL Group Replication is an advanced form of MySQL Replication used to implement fault-tolerant systems. The replication group (topology) is a set of servers that interact with each other through message passing. The communication layer provides a set of guarantees such as atomic message and total order message delivery. These are powerful properties that translate into very useful abstractions that enable advanced database replication solutions.

Group Replication builds on top of the existing replication properties and abstractions and implements a multi-master, update everywhere replication protocol. One of the technologies that makes Group Replication possible is GTIDs. Thus, servers that participate in Group Replication must have GTIDs enabled.

Essentially, a group is formed by multiple servers and each server in the group may execute transactions independently. But all read-write (RW) transactions commit only after they have been approved by the group. Read-only (RO) transactions need no coordination within the group and thus commit immediately. In other words, for any RW transaction, the group needs to decide whether it commits or not, thus the commit operation is not a unilateral decision from the originating server.

To be precise, when a transaction is ready to commit at the originating server, the server atomically broadcasts the write values (rows changed) and the correspondent write set (unique identifiers of the rows that were updated). Then a global total order is established for that transaction. Ultimately, this means that all servers receive the same set of transactions in the same order. Since all servers apply the same set of changes in the same order, they remain consistent within the group.

Group Replication provides redundancy by replicating the system state among the replication group. Should one (or more) of the servers fail, the system is still available. While it is possible if enough servers fail that some performance or scalability will be impacted, the system will remain available.

This is made possible by a group membership service, which relies on a distributed failure detector that can signal when any servers leave the group, either through deliberate interaction or due to a failure. There is a distributed recovery procedure to ensure that when servers join the group, they are brought up-to-date automatically. There is no need for server failover, and the multi-master update everywhere nature ensures that not even updates are blocked in the event of a single server failure. Therefore, MySQL Group Replication guarantees that the database service is continuously available.

Group Replication makes the topology eventually synchronous replication (among the nodes belonging to the same group) a reality, while the existing MySQL Replication feature is asynchronous (or at most semi-synchronous). Therefore, better high availability guarantees can be provided, since transactions are delivered to all members in the same order (despite being applied at its own pace in each member after being accepted).

Group Replication does this via a distributed state machine with strong coordination among the servers assigned to a group. This communication allows the servers to coordinate replication automatically within the group. More specifically, groups maintain membership so that the data replication among the servers is always consistent at any point in time. Even if servers are removed from the group, when they are added, the consistency is initiated automatically. Further, there is also a failure detection mechanism for servers that go offline or become unreachable.

Group Replication can also be used with the MySQL Router to enable application-level routing and fault tolerance for operations (at the application level). Figure 8-1 shows how you would use Group Replication with our applications to achieve high availability.

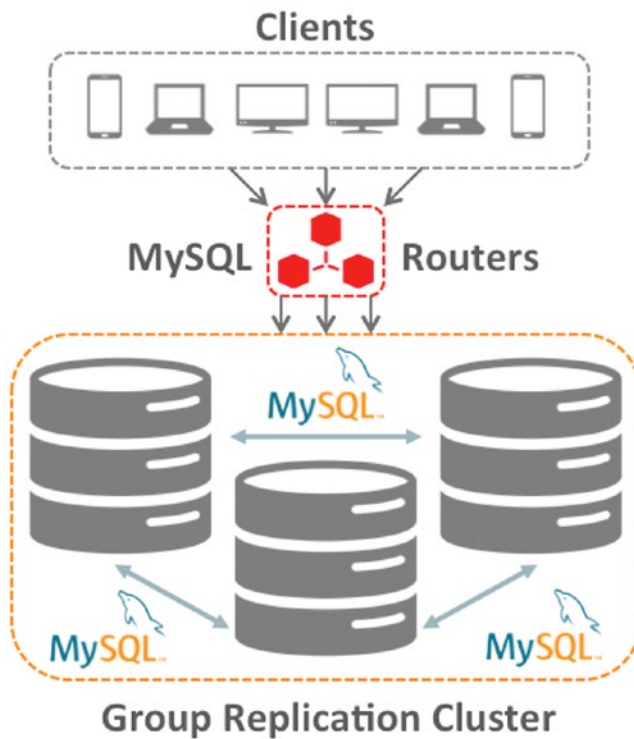


Figure 8-1. *Using Group Replication with Applications for High Availability (Courtesy of Oracle)*

Notice Group Replication can be used with the MySQL Router to allow your applications to have a layer of isolation from the cluster. We will see a bit about the router when we examine the InnoDB Cluster in Chapters 10 and 11.

Another important distinction between Group Replication and standard replication is that all the servers in the group can participate in updating the data with conflicts resolved automatically. Yes, you no longer must carefully craft your application to send writes (updates) to a specific server! However, you can configure Group Replication to allow updates by only one server (called the primary) with the other servers acting as secondary servers or as a backup (for failover).

These capabilities and more are made possible using three specific technologies built into Group Replication: group membership, failure detection, and fault tolerance. The following lists these technologies along with a short overview of each.

- *Group Membership*: Manages whether servers are active (online) and participating in the group. Also, ensures every server in the group has a consistent view of the membership set, that is, every server knows the complete list of servers in the group. When servers are added to the group, the group membership service reconfigures the membership automatically.
- *Failure Detection*: A mechanism that can find and report which servers are offline (unreachable) and assumed to be dead. The failure detector is a distributed service that allows all servers in the group to test the condition of the presumed dead server and in that way, the group decides if a server is unreachable (dead). This allows the group to reconfigure automatically by coordinating the process of excluding the failed server.
- *Fault Tolerance*: This service uses an implementation of the Paxos distributed algorithm to provide distributed coordination among the servers. In short, the algorithm allows for automatic promotion of roles within the group to ensure the group remains consistent (data is consistent and available) even if a server (or several) fails or leaves the group. Like similar fault tolerance mechanisms, the number of failures (servers that fail) is limited. Currently, Group Replication fault tolerance can be defined as $n = 2f + 1$ where n is the number of servers needed to tolerate f failures. For example, if you want to tolerate up to 5 servers failing, you need at least 11 servers in the group.

Tip For more information about Group Replication, see the online reference manual at <https://dev.mysql.com/doc/refman/8.0/en/group-replication.html>

Now that we know a bit more about MySQL Replication and Group Replication, let's see a short primer on how to set up and configure MySQL Replication. We will use this as a spring board into the next chapter where we will configure Group Replication using MySQL Shell.

Setup and Configuration

Now that you have a little knowledge of replication and how it works, let's see how to set it up. The next section discusses how to set up replication with one server as the master and another as the slave. We will see both types of replication used. As you will see, there are only a few differences in how you configure the servers and start replication.

Note The terms, “master” and “slave” are used exclusively in MySQL Replication and representative of the fact that only one server can be written to and thus has the “master” copy. The remaining servers are read-only containing a copy (replicant) of the data. These terms were changed to “primary” and “secondary” in Group Replication to better describe the roles in the new features.

But first, you will need to have two instances of MySQL running either on your system using different ports, on two additional systems, or on two virtual machines. For most cases of exploration, running more than one instance of MySQL on your local computer will work well.

There are some prerequisites for replication that you must set when you launch the instances. It is recommended that you create a separate configuration file (`my.cnf` or `my.ini` in Windows) for each so that you don't risk using the same directories for both instances. To do this, we should already have MySQL installed on our system.

Launching a new instance of MySQL is easy and requires only a few administrative tasks. The following is an outline of these tasks.

- *Data directory*: You must create a folder to contain the data directory.
- *Port*: You must choose a port to use for each instance.
- *Configuration file*: You must create a separate configuration file for each server.
- *Launch the instance manually*: To run the instance, you will launch MySQL (`mysqld`) from the command line (or via a batch file) specifying the correct configuration file.

We will see these steps and more demonstrated in the next section, which presents a tutorial of how to set up and run MySQL Replication on your local computer.

Tutorial

This section demonstrates how to set up replication from one server (the master) to another (a slave). The steps include preparing the master by enabling binary logging and creating a user account for reading the binary log, preparing the slave by connecting it to the master, and starting the slave processes. The section concludes with a test of the replication system.

Note The steps used to setup replication with binary log file and position are the same as those for using GTIDs, but the commands differ slightly in some of the steps. This tutorial will show both methods.

The steps to set up and configure MySQL Replication include the following. There may be other, equally as viable procedures to set up replication, but these can be done on any machine and will not affect any existing installations of MySQL. That said, it is recommended to perform these steps on a development machine to remove risk of disrupting production systems.

- *Initialize the Data Directories:* Create folders to store the data.
- *Configure the Master:* Configure the master with binary logging, new config file.
- *Configure the Slaves:* Configure the slaves with binary logging, new config file.
- *Start the MySQL Instances:* Launch the instances of MySQL server.
- *Create the Replication User Account:* Create the replication user on all servers.
- *Connect the Slaves to the Master:* Connect each slave to the master.
- *Start Replication:* Initiate replication.
- *Verify Replication Status:* Perform a short test to ensure data is being copied.

The following sections demonstrate each of these steps in greater detail. While the tutorial uses multiple, local instances to demonstrate how to use replication, the procedure would be the same for setting up replication in a development or production environment using discrete machines (or virtual machines). The details of the individual commands to use specific hosts, drives, folders, ports, etc. are the only things that would change to use the procedure in production.

Note The steps shown in this tutorial are run on the macOS platform. While there are platform-specific commands and a few platform-specific options, the tutorial can be run on Linux and Windows platforms with minor changes.

Initialize the Data Directories

The first step is to initialize a data directory for each of the machines used. In this case, we will create a folder on our local machine to contain all the data directories. We will use two instances of MySQL to represent a single master and a single slave. The following demonstrates creating the folders needed. Notice I create these in a local folder accessible to the user account I am using, not a system or administrative account. This is because we will be running the instances locally and do not need the additional privileges or access such accounts permit. If you were to create the folders with an administrative account, you would have to run the server (`mysqld`) as that administrator, which we do not want to do.

```
$ mkdir rpl
$ cd rpl
$ mkdir data
```

Now that we have a folder, `<user_home>/rpl/data`, we can use the initialization option of the MySQL server to set up our data directories. We do this using the special `--initialize-insecure` and `--datadir` options of the server executable. The `--initialize-insecure` option tells the server to create the data directory and populate it with the system data but to skip the use of any authentication. This is safe because there are no users created yet (there's no data directory!).

The `--datadir` option specifies the location of the data directory main folder. Since we are running this as a local user, we also need the `--user` option.

Tip Be sure to use your own user name for the `--user` option and all paths if you are copying the commands in this tutorial.

We also need to know the base directory (called `basedir`) from the MySQL server installed on the local machine. You can get that information from the server configuration file or by using the shell and issuing the `SHOW SQL` command. The following demonstrates how to do this. Here, we see the base directory is `/usr/local/mysql-8.0.16-macos10.14-x86_64`. We will use this value so that the `mysqld` executable can find its dependent libraries and files.

```
$ mysqlsh --uri root@localhost:33060 --sql -e "SHOW VARIABLES LIKE 'basedir'"
Variable_name      Value
basedir            /usr/local/mysql-8.0.16-macos10.14-x86_64/
```

The following shows the commands needed to initialize the data directories for the master and a slave. Notice I use “`slave1`” for the slave. This so that you can expand the tutorial to multiple slaves should you want to experiment with adding additional slaves.

```
$ mysqld --user=cbell --initialize-insecure --basedir=/usr/local/
mysql-8.0.16-macos10.14-x86_64/ --datadir=/Users/cbell/rpl/data/master
$ mysqld --user=cbell --initialize-insecure --basedir=/usr/local/
mysql-8.0.16-macos10.14-x86_64/ --datadir=/Users/cbell/rpl/data/master
```

When you run these commands, you will see several messages printed. You can safely ignore the warnings but notice the last one tells us the root user does not have a password assigned. This is okay for our tutorial, but something you never want to do for a production installation. Fortunately, we can fix that easily once we start the instance.

Now that we have the data directories created and populated, we can configure the master and slave(s).

Configure the Master

Replication requires the master to have binary logging enabled. It is not turned on by default, so you must add this option in the configuration file. In fact, we will need a configuration file for each of the instances we want to start. In this section, we concentrate on the master and in the next we will see the configuration file for a slave.

We also need to select the port for the instance. For this tutorial, we will use port numbers starting from 13001 for the master and 13002+ for the slaves. In addition, we will need to choose unique server identification numbers. We will use 1 for the master and 2+ for the slaves.

There are other settings we will need to make. Rather than list them, let's view a typical base configuration file for a master using replication with binary log and file position.

Listing 8-1 shows the configuration file we will use for the master in this tutorial.

Listing 8-1. Master Configuration File (Log File and Position)

```
[mysqld]
datadir="/Users/cbell/rpl/data/master"
basedir="/usr/local/mysql-8.0.16-macos10.14-x86_64/"
port=13001
socket="/Users/cbell/rpl/master.sock"
server_id=1
master_info_repository=TABLE
relay_log_info_repository=TABLE
log_bin=master_binlog
binlog_format=row
```

Notice the configuration file has one section named `mysqld`, which applies only to the MySQL server executable. That is, only the `mysqld` and related executables will read this section for values. Among those values are the common required settings for `datadir`, `basedir`, `port`, and `socket` (for *nix style platforms). Notice these values match the settings we've discussed previously.

The next section sets the server id, turns on the `TABLE` option for storing replication information, which makes replication recoverable from crashes, and turns on the binary log and sets its location. Finally, we use the `ROW` format for the binary log, which is a binary format and is the default for the latest versions of MySQL Replication.

If we wanted to use GTID-based replication, there are some additional options that must be set. For the master, there are only three; turn GTIDs on, set consistency enforcement, and log slave updates. Thus, the configuration file for a GTID-enabled master server is shown in Listing 8-2. Notice the first portion of the file is the same as the previous example. Only the last few lines are added to enable GTIDs.

Listing 8-2. Master Configuration File (GTIDs)

```
[mysqld]
datadir="/Users/cbell/rpl/data/master"
basedir="/usr/local/mysql-8.0.16-macos10.14-x86_64/"
port=13001
socket="/Users/cbell/rpl/master.sock"
server_id=1
master_info_repository=TABLE
relay_log_info_repository=TABLE
log_bin=master_binlog
binlog_format=row

# GTID VARIABLES
gtid_mode=on
enforce_gtid_consistency=on
log_slave_updates=on
```

For this tutorial, we will be using GTID-enabled replication, so you should create a file in the folder we created earlier named `master.cnf`; for example, `/Users/cbell/rpl/master.cnf`. We will use this file to start the instance for the master in a later step.

Tip Some platforms may fail to start MySQL if the configuration file is world readable. Check the log if your server does not start for messages regarding the permissions of files.

Now, let's look at the configuration files for the slaves.

Configure the Slaves

While log file and position replication require the master to have binary logging enabled, it is not required for the slaves. However, it is a good idea to turn on the binary log for the slaves if you want to use the slave to generate backups or for crash recovery. However, binary logging is also required if you want to use GTID-enabled replication. In this section, we will use binary logging on the slaves.

Like the master, we need to set several variables including the `datadir`, `basedir`, `port`, and `socket` (for *nix style platforms). Listing 8-3 shows the configuration file for the first slave (named `slave1`).

Listing 8-3. Slave Configuration File (Log File and Position)

```
[mysqld]
datadir="/Users/cbell/rpl/data/slave1"
basedir="/usr/local/mysql-8.0.16-macos10.14-x86_64/"
port=13002
socket="/Users/cbell/rpl/slave1.sock"
server_id=2
master_info_repository=TABLE
relay_log_info_repository=TABLE
log_bin=slave1_binlog
binlog_format=row
report-port=13002
report-host=localhost
```

Notice there are two additional variables set; `report-port` and `report-host`. These are necessary to ensure commands like `SHOW SLAVE HOSTS` report the correct information. That is, the information for that view is derived from these variables. Thus, it is always a good idea to set these correctly.

Notice also we set the data directory to one set aside for this slave and the server id is changed. Finally, we also change the name of the binary log to ensure we know from which server the log originated (if needed in the future).

If we wanted to use GTID-based replication, we would add the same set of variables we did for the master as shown in Listing 8-4.

Listing 8-4. Slave Configuration File (GTIDs)

```
[mysqld]
datadir="/Users/cbell/rpl/data/slave1"
basedir="/usr/local/mysql-8.0.16-macos10.14-x86_64/"
port=13002
socket="/Users/cbell/rpl/slave1.sock"
server_id=2
```



```

master_info_repository=TABLE
relay_log_info_repository=TABLE
log_bin=slave1_binlog
binlog_format=row
report-port=13002
report-host=localhost

# GTID VARIABLES
gtid_mode=on
enforce_gtid_consistency=on
log_slave_updates=on

```

For this tutorial, we will be using GTID-enabled replication, so you should create a file in the folder we created earlier named `slave1.cnf`; for example, `/Users/cbell/rpl/slave1.cnf`. If you want to add more slaves, create additional configuration files with the same data changing only the data directory, socket, port, server id, and binary log file.

Start the MySQL Instances

Now we are ready to start the MySQL instances. This is easy to do since we have already created the configuration file with all the parameters we need. We only need provide the configuration file with the `--defaults-file` option. The following shows the commands to start both server instances.

```

$ mysqld --defaults-file=master.cnf
$ mysqld --defaults-file=slave1.cnf

```

When you run these commands, you should run them from the folder that contains the configuration files; otherwise, you must provide the full path to the configuration file. It is also a good idea to either use a separate terminal window to launch each instance or redirect the output (logging of messages) to a file as shown in Listing 8-5. However, you may want to use a separate terminal the first time you start the server to ensure there are no errors. Listing 8-5 shows an excerpt of the messages printed when launching the master (without the return `&`).

Listing 8-5. Starting the Master Instance

```

297Z 0 [System] [MY-010116] [Server] /usr/local/mysql-8.0.16-
macos10.14-x86_64/bin/mysqld (mysqld 8.0.16) starting as process 2615
2019-05-03T23:14:46.081413Z 0 [Warning] [MY-010159] [Server] Setting
lower_case_table_names=2 because file system for /Users/cbell/rpl/data/
master/ is case insensitive
2019-05-03T23:14:46.341919Z 0 [Warning] [MY-010068] [Server] CA certificate
ca.pem is self signed.
2019-05-03T23:14:46.342661Z 0 [Warning] [MY-011810] [Server] Insecure
configuration for --pid-file: Location '/Users/cbell/rpl/data' in the path
is accessible to all OS users. Consider choosing a different directory.
2019-05-03T23:14:46.355855Z 0 [System] [MY-010931] [Server] /usr/local/
mysql-8.0.16-macos10.14-x86_64/bin/mysqld: ready for connections. Version:
'8.0.16' socket: '/Users/cbell/rpl/master.sock' port: 13001 MySQL
Community Server - GPL.
2019-05-03T23:14:46.569522Z 0 [System] [MY-011323] [Server] X Plugin ready
for connections. Socket: '/Users/cbell/rpl/masterx.sock' bind-address: '::'
port: 33061

```

If you plan to use a single terminal, it is recommended to redirect the output to a file named `master_log.txt` and use the option to start the application in another process (e.g., the `&` symbol). The log files are updated as the server generates messages, so you can refer to them if you encounter problems. It also helps to keep your terminal session clear of extra messages. The following shows how to structure the preceding command to start as a separate process and log messages to a file.

```
$ mysqld --defaults-file=master.cnf > master_output.txt 2>&1 &
```

If you haven't done so already, go ahead and start the slave. The following is the command I used to start the slave (slave1).

```
$ mysqld --defaults-file=slave1.cnf > slave1_output.txt 2>&1 &
```

Create the Replication User Account

After the MySQL instance has started, you must create a user to be used by the slave to connect to the master and read the binary log before you can setup replication. There is a special privilege for this named `REPLICATION SLAVE`. The following shows the correct `GRANT` statement to create the user and add the privilege. Remember the username and password you use here because you will need it for connecting the slave.

The following shows the commands needed to create the replication user. Execute these commands on all your servers. While it is not needed for the slaves, creating the user now will allow you to use the slaves for recovery, switchover, or failover without having to create the user. In fact, this step is required for permitting automatic failover.

```
SET SQL_LOG_BIN=0;
CREATE USER rpl_user@'localhost' IDENTIFIED BY 'rpl_pass';
GRANT REPLICATION SLAVE ON *.* TO rpl_user@'localhost';
FLUSH PRIVILEGES;
SET SQL_LOG_BIN=1;
```

Notice the first and last commands. These commands tell the server to temporarily disable logging of changes to the binary log. We do this whenever we do not want to replicate the commands on other machines in the topology. Specifically, maintenance and administrative commands like creating users should not be replicated. Turning off the binary log is a great way to ensure you do not accidentally issue transactions that cannot be executed on other machines.

The best way to execute these commands is to save them to a file named `create_rpl_user.sql` and use the source command of the `mysql` client to read the commands from the file and execute them. Thus, you can quickly create the replication user on all instances with the following commands.

```
$ mysqlsh --uri root@localhost:33061 --sql -f "create_rpl_user.sql"
$ mysqlsh --uri root@localhost:33062 --sql -f "create_rpl_user.sql"
```

Now we are ready to connect the slave to the master and start replicating data.

Connect the Slaves to the Master

The next step is to connect the slaves to the master. There are different ways to do this depending on which form of replication you are using. Specifically, the command to connect the slave to the master differs when using log file and position vs. GTID replication. There are also two steps: configuring the slave to connect and starting replication. Let's look at configuring the slave with log file and position first.

Connect with Log File and Position

To connect a slave to the master using log file and position, we need a bit of information. This information is needed to complete the `CHANGE MASTER` command that instructs the slave to make a connection to the master. Table 8-1 shows the complete list of information needed. The table includes one of the sources where the information can be found along with an example of the values used in this tutorial.

Table 8-1. *Information Needed for Connecting a Slave (Log File and Position)*

Item from Master	Source	Example
IP address or hostname	master.cnf	localhost
Port	master.cnf	13001
Binary log file	SHOW MASTER STATUS	master_binlog.000005
Binary log file position	SHOW MASTER STATUS	154
Replication user ID	create_rpl_user.sql	rpl_user
Replication user password	create_rpl_user.sql	rpl_pass

The information for the master binary log file can be found with the `SHOW MASTER STATUS` command. The following shows how to use the `mysql` client to execute the command and return.

```
$ mysqlsh --uri root@localhost:33061 --sql -e "SHOW MASTER STATUS"
File      Position      Binlog_Do_DB      Binlog_Ignore_DB      Executed_Gtid_Set
master_binlog.000005      155
```

Notice the command also displays any filters that are active as well as a GTID-specific value for the latest GTID-executed set. We won't need that for this tutorial, but it is a good idea to file that away should you need to recover a GTID-enabled topology.

Now that you have the master's binary log file name and position as well as the replication user and password, you can visit your slave and connect it to the master with the `CHANGE MASTER` command. The command can be constructed from the information in Table 8-1 as follows (formatted to make it easier to read – remove the `\` if you are following along with this tutorial).

```
CHANGE MASTER TO MASTER_USER='rpl_user', MASTER_PASSWORD='rpl_pass', \
  MASTER_HOST='localhost', MASTER_PORT=13001, \
  MASTER_LOG_FILE='master_binlog.000005', MASTER_LOG_POS=155;
```

You must run this command on all the slaves. It may be easier to save this to a file and execute it using the `mysql` client like we did for the replication user. For example, save this to a file named `change_master.sql` and execute it as following.

```
$ mysqlsh --uri root@localhost:33062 --sql -f "change_master.sql"
```

There is one more step to starting the slave, but let's first look at how to configure the `CHANGE MASTER` commands for GTID-enabled replication.

Connect with GTIDs

To connect a slave to the master using GTIDs, we need a bit of information. This information is needed to complete the `CHANGE MASTER` command that instructs the slave to make a connection to the master. Table 8-2 shows the complete list of information needed. The table includes one of the sources where the information can be found along with an example of the values used in this tutorial.

Table 8-2. *Information Needed for Connecting a Slave (GTIDs)*

Item from Master	Source	Example
IP address or hostname	master.cnf	localhost
Port	master.cnf	13001
Replication user ID	create_rpl_user.sql	rpl_user
Replication user password	create_rpl_user.sql	rpl_pass

Notice we need less information than log file and position replication. We don't need to know the master binary log file or position because the GTID handshake procedure will resolve that information for us. All we need then is the host connection information for the master and the replication user and password. For GTID-enabled replication, we use a special parameter, `MASTER_AUTO_POSITION` to instruct replication to negotiate the connection information automatically. The `CHANGE MASTER` command can be constructed from the information in Table 8-2 as follows (formatted to make it easier to read – you do not need to remove the `\` if you are following along with this tutorial).

```
CHANGE MASTER TO MASTER_USER='rpl_user', MASTER_PASSWORD='rpl_pass', \
    MASTER_HOST='localhost', MASTER_PORT=13001, MASTER_AUTO_POSITION = 1;
```

You must run this command on all the slaves. It may be easier to save this to a file and execute it using the `mysql` client like we did for the replication user. For example, save this to a file named `change_master.sql` and execute it as following.

```
mysqlsh --uri root@localhost:33062 --sql -f "change_master.sql"
```

If you want to be able to use the file for either form of replication, you can simply place both commands in the file and comment out one that you don't need. For example, the following shows an example file with both `CHANGE MASTER` commands. Notice the GTID variant is commented out with the `#` symbol.

```
CHANGE MASTER TO MASTER_USER='rpl_user', MASTER_PASSWORD='rpl_pass',
MASTER_HOST='localhost', MASTER_PORT=13001, MASTER_LOG_FILE='master_
binlog.000005', MASTER_LOG_POS=155;
# GTID option:
# CHANGE MASTER TO MASTER_USER='rpl_user', MASTER_PASSWORD='rpl_pass',
MASTER_HOST='localhost', MASTER_PORT=13001, MASTER_AUTO_POSITION = 1;
```

Now that we have our slaves configured to connect, we must finish the process by telling the slaves to initiate the connection and start replication.

Start Replication

The next step is to start the slave processes. This command is simply `START SLAVE`. We would run this command on all the slaves like we did for the `CHANGE MASTER` command. The following shows the commands for starting the slaves.

```
$ mysqlsh --uri root@localhost:33062 --sql -e "START SLAVE"
```

The `START SLAVE` command normally does not report any errors; you must use `SHOW SLAVE STATUS` to see them. Listing 8-6 shows the command in action. For safety as well as peace of mind, you may want to run this command on any slave you start.

Listing 8-6. Checking SLAVE STATUS

```
$ mysqlsh --uri root@localhost:33062 --sql -e "SHOW SLAVE STATUS\G"
***** 1. IOW *****
      Slave_IO_State: Waiting for master to send event
      Master_Host: localhost
      Master_User: rpl_user
      Master_Port: 13001
      Connect_Retry: 60
      Master_Log_File: master_binlog.000005
Read_Master_Log_Pos: 155
      Relay_Log_File: MacBook-Pro-2-relay-bin.000002
      Relay_Log_Pos: 377
Relay_Master_Log_File: master_binlog.000005
      Slave_IO_Running: Yes
      Slave_SQL_Running: Yes
      Replicate_Do_DB:
      Replicate_Ignore_DB:
      Replicate_Do_Table:
      Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
      Replicate_Wild_Ignore_Table:
      Last_Errno: 0
      Last_Error:
      Skip_Counter: 0
Exec_Master_Log_Pos: 155
      Relay_Log_Space: 593
      Until_Condition: None
      Until_Log_File:
      Until_Log_Pos: 0
      Master_SSL_Allowed: No
      Master_SSL_CA_File:
```

```
Master_SSL_CA_Path:
  Master_SSL_Cert:
  Master_SSL_Cipher:
  Master_SSL_Key:
Seconds_Behind_Master: 0
Master_SSL_Verify_Server_Cert: No
  Last_IO_Errno: 0
  Last_IO_Error:
  Last_SQL_Errno: 0
  Last_SQL_Error:
Replicate_Ignore_Server_Ids:
  Master_Server_Id: 1
    Master_UUID: b6632bf2-6df6-11e9-8bf5-cc9584d9b3f3
  Master_Info_File: mysql.slave_master_info
    SQL_Delay: 0
  SQL_Remaining_Delay: NULL
Slave_SQL_Running_State: Slave has read all relay log; waiting for
                        more updates
  Master_Retry_Count: 86400
    Master_Bind:
Last_IO_Error_Timestamp:
Last_SQL_Error_Timestamp:
  Master_SSL_Crl:
  Master_SSL_Crlpath:
Retrieved_Gtid_Set:
  Executed_Gtid_Set: ccc5263e-6df6-11e9-88d5-910b8477c67b:1
    Auto_Position: 1
Replicate_Rewrite_DB:
  Channel_Name:
  Master_TLS_Version:
Master_public_key_path:
Get_master_public_key: 0
  Network_Namespace:
```


Take a moment to slog through all these rows. There are several key fields you need to pay attention to. These include anything with *error* in the name, and the *state* columns. For example, the first row (*Slave_IO_State*) shows the textual message indicating the state of the slave's I/O thread. The I/O thread is responsible for reading events from the master's binary log. There is also a SQL thread that is responsible for reading events from the relay log and executing them.

For this example, you just need to ensure that both threads are running (YES) and there are no errors. For detailed explanations of all the fields in the `SHOW SLAVE STATUS` command, see the online MySQL reference manual in the section “SQL Statements for Controlling Slave Servers.”¹

Now that the slave is connected and running, let's check replication by checking the master and creating some data.

Verify Replication Status

Checking the slave status with the `SHOW SLAVE STATUS` command is the first step to verifying replication health. The next step is to check the master using the `SHOW SLAVE HOSTS` command. Listing 8-7 shows the output of the `SHOW SLAVE HOSTS` for the topology setup in this tutorial. This command shows the slaves that are attached to the master and their UUIDs. It should be noted that this information is a view and is not real-time. That is, it is possible for slave connections to fail and still be shown on the report until the processes time out and the server kills them. Thus, this command is best used as a sanity check.

Listing 8-7. SHOW SLAVE HOSTS Command (Master)

```
mysqlsh --uri root@localhost:33061 --sql -e "SHOW SLAVE HOSTS\G"
***** 1. IOW *****
Server_id: 2
  Host: localhost
  Port: 13002
Master_id: 1
Slave_UUID: ccc5263e-6df6-11e9-88d5-910b8477c67b
```

¹<https://dev.mysql.com/doc/refman/8.0/en/replication-slave-sql.html>

Here we see all the slave is connected and we know from the last section the slave status is good.

Next, let's create some simple data on the master then see if that data is replicated to the slave. In this case, we will simply create a database, a table, and a single row then run that on the master. Listing 8-8 shows the sample data as executed on the master.

Listing 8-8. Creating Sample Data for Testing Replication (Master)

```
$ mysqlsh --uri root@localhost:33061 --sql
MySQL Shell 8.0.16

Copyright (c) 2016, 2019, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.

Type '\help' or '\?' for help; '\quit' to exit.
Creating a session to 'root@localhost:33061'
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 18 (X protocol)
Server version: 8.0.16 MySQL Community Server - GPL
No default schema selected; type \use <schema> to set one.

MySQL localhost:33061+ ssl SQL > CREATE DATABASE test;
Query OK, 1 row affected (0.0029 sec)

MySQL localhost:33061+ ssl SQL > USE test;
Query OK, 0 rows affected (0.0002 sec)

MySQL localhost:33061+ ssl SQL > CREATE TABLE test.t1 (c1 INT PRIMARY
KEY, c2 TEXT NOT NULL);
Query OK, 0 rows affected (0.0054 sec)

MySQL localhost:33061+ ssl SQL > INSERT INTO test.t1 VALUES (1, 'Dr.
Charles Bell');
Query OK, 1 row affected (0.0092 sec)
```

To verify the data was replicated, all we need to do is issue a SELECT SQL command on the table on one of the slaves (or all of them if you are so inclined). The following shows an example of what we expect to see on each of the slaves.

```
$ mysqlsh --uri root@localhost:33062 --sql -e "SELECT * FROM test.t1"
c1    c2
1     Dr. Charles Bell
```

This concludes the short tutorial on setting up MySQL Replication. This section presented a brief look at MySQL Replication in its barest, simplest terms. Now, let's look at how we can script an example setup of MySQL Replication. However, there is one more step.

Shutting Down Replication

If you try this tutorial on your own machine, remember to shut down your instances in a precise order. On each slave, you first want to stop the slave with the following command.

```
$ mysqlsh --uri root@localhost:33062 --sql -e "STOP SLAVE"
```

Once all slaves are stopped, you can shut down the slaves first then the master with the following commands. Notice we used the old protocol ports (13001, 13002) for the master and slaves. This is because the X Protocol does not support the shutdown command. If you encounter an error when using the MySQL X ports, try the old protocol and rerun the command.

```
$ mysqlsh --uri root@localhost:13002 --sql -e "SHUTDOWN"
$ mysqlsh --uri root@localhost:13001 --sql -e "SHUTDOWN"
```

REFERENCES ON MYSQL HIGH AVAILABILITY

If you peruse some of the lengthier works on MySQL High Availability prior to InnoDB Cluster, you may find some additional, more complex challenges for specific use cases. If you would like to know more about MySQL Replication and MySQL High Availability features prior to InnoDB Cluster, the following are some excellent resources.

- Bell, Kindahl, Thalmann, *MySQL High Availability: Tools for Building Robust Data Centers*, 2nd Edition (O'Reilly, 2014)
- Das, *MySQL Replication Simplified: Easy step-by-step examples to establish, troubleshoot, and monitor replication* (Business Compass, LLC, 2014)

- Schwartz, Zaitsev, Tkachenko *High Performance MySQL: Optimization, Backups, Replication, and More* (O'Reilly, 2012)
- Davies, *MySQL High Availability Cookbook* (Packt, 2010)

The online reference manual has considerable documentation and should be your primary source (<https://dev.mysql.com/doc/refman/8.0/en/replication.html>).

Summary

Achieving high availability in MySQL is possible with MySQL Replication. Indeed, you can create robust data centers with replication. Better still, replication has been around for a long time and is considered very stable. Many organizations have and continue to have success using replication in production – from small installations to massive installations.

Even so, there are limitations to using MySQL Replication such as how to handle switching the master role to another machine (slave) if the master fails, how to perform this automatically, how to handle multiple write scenarios, and general troubleshooting and maintenance. Many of these have been improved in Group Replication. However, as we have seen, setup of replication requires some effort and maintenance, which may be a concern to planners and administrators alike.

In this chapter, we learned more what high availability is, an overview of some of the high availability features in MySQL, as well as how to implement MySQL Replication with one master and one slave. Even though the tutorial was limited to two machines, you can easily expand it to several machines.

You may feel MySQL Replication will meet your high availability needs (and that's great), but MySQL Group Replication takes high availability to another level by introducing fault tolerance and a host of features that make it more versatile for more demanding high availability requirements.

In the next chapter, we will see a complete walkthrough of how to set up Group Replication using MySQL Shell.

CHAPTER 9

Example: Group Replication Setup and Administration

As we learned in the last chapter, Group Replication is an evolution of MySQL Replication designed to make data replication more robust and reliable. Together with the modifications to the InnoDB storage engine (all under the hood and hidden away), Group Replication enables high availability capabilities that used to require specialized and sometimes customized products, middleware, and bespoke applications to achieve.

In this chapter, we take a more in-depth look at what makes up Group Replication. After that, we will take a guided tour of Group Replication in a hands-on walkthrough.

Getting Started

It may not surprise you that setting up MySQL Group Replication resembles the process of setting up MySQL Replication. After all, Group Replication is built upon the foundations of MySQL Replication. In the next section, we will see a demonstration of Group Replication but rather than concentrate on the same steps from the MySQL Replication tutorial, we will cover the same topics briefly and dive into the nuances specific to Group Replication.

But first, let's begin with a list of concepts and terms that make up the language of describing Group Replication.

Concepts, Terms, and Lingo

It is likely all but the most proficient or those with the latest knowledge of MySQL will fully understand all the terms and concepts thrown around describing Group Replication. In this section, we take a step back for a moment and discuss some of the terms and concepts you will encounter in this chapter and the next two chapters (or any book on MySQL High Availability). Thus, this section is a glossary of terms associated with Group Replication. Feel free to refer to this section from time to time.

- *Binary Log*: A file produced by the server that contains a binary form of all transactions executed. The binary log file is also used in replication to exchange transactions between two servers. When used on a primary (master), it forms a record of all changes, which can be sent to the secondary (slave) for execution to create a replicant.
- *Multi-Primary*: A group where writes may be sent to more than one primary and replicated among the group.
- *Failover*: An event that permits the group to recover from a fault on the primary automatically electing a new primary.
- *Fault Tolerant*: The ability to recover from a fault or error detected among the group without loss of data or functionality. Note that fault tolerance in Group Replication is limited by the number of servers in the group. See the online reference manual in the Group Replication section “Fault-tolerance”, for how to calculate the number of servers and how many faults the group can tolerate.¹
- *Group*: A set of MySQL servers participating in the same Group Replication communication setup.
- *Group Communication*: A special mechanism that uses a state machine and a communication protocol to keep the servers in the group coordinated including synchronization of transaction execution and selection/election of roles.

¹<https://dev.mysql.com/doc/refman/8.0/en/group-replication-fault-tolerance.html>

- *Instance*: A running MySQL server. Typically used to refer to one or more MySQL servers running on the same machine. This is not the same as “MySQL Server”, which often refers to the set of hardware and MySQL execution.
- *Primary*: A server in a group that is assigned the role of collecting all writes (updates) to data.
- *Relay Log*: A binary log file used on a secondary (slave) to record transactions read from the primary (master) binary log and saved for execution. It has the same format as the binary log.
- *Secondary*: A server in a group that is assigned the role of reader, which means applications can read data from a secondary but may not write to a secondary.
- *Single-Primary*: A group configured with a single primary and one or more secondary server. This is like the master/slave configuration in the older MySQL Replication feature.
- *Switchover*: A controlled maintenance event where an administrator actively changes the primary role removing it from one server and assigning it to another (making the new server the primary). This does not happen automatically and typically is not associated with a failure.
- *Transaction*: A set of data changes that must all succeed before the set is applied to the data. Failed transactions are not written to the database.
- *Topology*: A term describing the logical layout of servers in a replication group. Examples include single primary represented as a single server with radial connections to each slave, tiered represented as connections of single-primary groups where each secondary is the primary to another set of secondary servers, and multi-primary often represented where each primary connects to every other primary in the group as well as the secondary servers in the group.

There is one more aspect of Group Replication that you should understand; calculating the number of faults, a group can recover and continue to provide high availability.

Group Replication Fault Tolerance

Recall from Chapter 8, we learned Group Replication can successfully tolerate a certain number of server faults where the server goes offline (leaves the group). The number of faults that the group can tolerate depends on the number of servers in the group. The formula for determining how many failures simultaneous or consecutive, unrecovered failures, f , a set of servers, S can tolerate is as follows.

$$S = 2f + 1$$

For example, a set of 11 servers can tolerate, at most, 5 failures.

$$11 = 2f + 1$$

$$10 = 2f$$

$$2f = 10$$

$$f = 10/2$$

$$f = 5$$

If you want to know how many servers, s , it takes to tolerate a known number of failures, F , a little math application reveals the following. Note that you must round down any fractions. You can't have 1.5 servers fail.

$$s = 2F + 1$$

$$(s - 1) = 2F$$

$$2F = (s - 1)$$

$$F = (s - 1)/2$$

For example, a set of 5 servers can tolerate 2 failures.

$$F = (5 - 1)/2$$

$$F = 4/2$$

$$F = 2$$

Similarly, a set of 7 servers can tolerate 3 failures.

$$F = (7 - 1)/2$$

$$F = 6/2$$

$$F = 3$$

Setup and Configuration

Now that you have a little knowledge of Group Replication and how it works, let's see how to set it up. We will see how to set up Group Replication with four servers. Thus, you will need to have four instances of MySQL running either on your system using different ports, on four additional systems, or on four virtual machines. For most cases of exploration, running more than one instance of MySQL on your local computer will work well.

There are some prerequisites for Group Replication that you must set when you launch the instances. It is recommended that you create a separate configuration file (`my.cnf`) for each so that you don't risk using the same directories for both instances. To do this, we should already have MySQL installed on our system.

Launching a new instance of MySQL is easy and requires only a few administrative tasks. The following outline these tasks.

- *Data directory*: You must create a folder to contain the data directory.
- *Port*: You must choose a port to use for each instance.
- *Configuration file*: You must create a separate configuration file for each server.
- *Launch the instance manually*: To run the instance, you will launch MySQL (`mysqld`) from the command line (or via a batch file) specifying the correct configuration file.

We will see these steps and more demonstrated in the next section, which presents a tutorial of how to set up and run MySQL Group Replication on your local computer.

Tutorial

This section demonstrates how to set up Group Replication among a set of four MySQL instances running on a local computer. As mentioned previously, Group Replication uses different terms for the roles in the group. Specifically, there is a primary role and a secondary role. Unlike MySQL Replication where one server is designated as the master (primary), Group Replication can automatically change the roles of servers in the group as needed. Thus, while we will set up Group Replication identifying one of the servers as the primary, the end state of the group over time may result in one of the other servers becoming the primary.

If you would like to experience this tutorial on your own, you should prepare four servers. Like the last tutorial, we will use several instances running on the current machine. We need several instances to ensure the group has a viable set to enable redundancy and failover. In this case, the group can tolerate at most 1 failure; $(4-1)/2 = 1.5$ or 1 rounded down.

The steps to set up and configure Group Replication include the following. There may be other, equally as viable procedures to set up Group Replication, but these can be done on any machine and will not affect any existing installations of MySQL. That said, it is recommended to perform these steps on a development machine to remove risk of disrupting production systems.

Note The steps used to set up Group Replication is very similar to MySQL Replication. In fact, except for terminology (e.g., slave vs. secondary), the configuration files, and starting Group Replication on the master for the first time, the process is the same.

- *Initialize the Data Directories:* Create folders to store the data.
- *Configure the Primary:* Configure the primary with a new config file.
- *Configure the Secondaries:* configure the secondaries with a new config file.
- *Start the MySQL Instances:* Launch the instances of MySQL server.
- *Create the Replication User Account:* Create the replication user on all servers.
- *Start Group Replication on the Primary:* Initiate the primary and establish the group.
- *Connect Secondaries to the Primary:* Initiate replication.
- *Start Group Replication on the Secondaries:* Add secondary to group membership.
- *Verify Group Replication Status:* Perform a short test to ensure data is being copied.

The following sections demonstrate each of these steps in greater detail running on a macOS system with MySQL installed. The steps are the same for other platforms, but the paths may differ slightly. While the tutorial uses multiple, local instances to demonstrate how to use replication, the procedure would be the same for setting up replication in a production environment. The details of the individual commands to use specific hosts, drives, folders, ports, etc. are the only things that would change to use the procedure in production.

Initialize the Data Directories

The first step is to initialize a data directory for each of the machines used. In this case, we will create a folder on our local machine to contain all the data directories. We will use four instances of MySQL to represent a primary and three secondaries. The following demonstrates creating the folders needed. Notice we create these in a local folder accessible to the user account we're using, not a system or administrative account. This is because we will be running the instances locally and do not need the additional privileges or access such accounts permit.

```
$ mkdir gr
$ cd gr
$ mkdir data
```

Now that we have a folder, `<user_home>/gr/data`, we can use the initialization option of the MySQL server to set up our data directories. We do this using the special `--initialize-insecure` and `--datadir` options of the server executable. The `--initialize-insecure` option tells the server to create the data directory and populate it with the system data but to skip the use of any authentication. This is safe because there are no users created yet (there's no data directory!).

The `--datadir` option specifies the location of the data directory main folder. Since we are running this as a local user, we also need the `--user` option.

Tip Be sure to use your own user name for the `--user` option and all paths if you are copying the commands in this tutorial.

We also need to know the base directory (called `basedir`) from the MySQL server installed on the local machine. You can get that information from the server configuration file or by using the shell and issuing the `SHOW SQL` command. The following demonstrates how to do this. Here, we see the base directory is `/usr/local/mysql-8.0.16-macos10.14-x86_64`. We will use this value so that the `mysqld` executable can find its dependent libraries and files.

```
$ mysqlsh --uri root@localhost:33060 --sql -e "SHOW VARIABLES LIKE
'basedir'"
Variable_name  Value
basedir        /usr/local/mysql-8.0.16-macos10.14-x86_64/
```

The following shows the commands needed to initialize the data directories for the primary and secondaries.

```
$ mysqld --user=cbell --initialize-insecure \
--basedir=/usr/local/mysql-8.0.16-macos10.14-x86_64/ \
--datadir=/Users/cbell/gr/data/primary
$ mysqld --user=cbell --initialize-insecure \
--basedir=/usr/local/mysql-8.0.16-macos10.14-x86_64/ \
--datadir=/Users/cbell/gr/data/secondary1
$ mysqld --user=cbell --initialize-insecure \
--basedir=/usr/local/mysql-8.0.16-macos10.14-x86_64/ \
--datadir=/Users/cbell/gr/data/secondary2
$ mysqld --user=cbell --initialize-insecure \
--basedir=/usr/local/mysql-8.0.16-macos10.14-x86_64/ \
--datadir=/Users/cbell/gr/data/secondary3
```

Now that we have the data directories created and populated, we can configure the master and slave(s).

Configure the Primary

This step differs the most from MySQL Replication. In fact, the configuration file is quite a bit different. Specifically, we use the same variables from GTID-enabled replication in addition to several of the more common Group Replication variables that must be set. The following lists the Group Replication related variables and their uses. There are

additional variables for controlling Group Replication. See <https://dev.mysql.com/doc/refman/8.0/en/group-replication-options.html> in the online reference manual for a complete list.

- `transaction_write_set_extraction`: Defines the algorithm used to hash the writes extracted during a transaction. Group Replication must be set to `XXHASH64`.
- `group_replication_recovery_use_ssl`: Determines if Group Replication recovery connection should use SSL or not. Typically set to `ON`. The default is `OFF`.
- `group_replication_group_name`: The name of the group which this server instance belongs to. Must be a valid, unique UUID.
- `group_replication_start_on_boot`: Determines if the server should start Group Replication or not during server start.
- `group_replication_local_address`: The network address that the member provides for connections from other members, specified as a `host:port` formatted string. Note that this connection is for communication between the Group Replication members and not for client use.
- `group_replication_group_seeds`: A list of group members that is used to establish the connection from the new member to the group. The list consists of the seed member's `group_replication_local_address` network addresses specified as a comma separated list, such as `host1:port1,host2:port2`.
- `group_replication_bootstrap_group`: Configure this server to bootstrap the group. This option must only be set on one server and only when starting the group for the first time or restarting the entire group. After the group has been bootstrapped, set this option to `OFF`.

Notice the last variable, `group_replication_bootstrap_group`. This variable is something we will set to `OFF` in the configuration files but only after we have bootstrapped the group for the first time. This is one of the uses of the initial primary node – to start the group. We will see a special step that you must take the first time you boot the primary to start the group. After that, this variable must be set to `OFF`.

Primary Configuration File

To construct a configuration file for the primary, we need several things: the usual variables for data directory, base directory, port, etc. as well as the GTID variables and the Group Replication variables. It is also a good idea to add the plugin directory to ensure the server can find the Group Replication plugin (we will see this in a later step), and to turn on the binary log checksum.

Since the `group_replication_group_seeds` variable needs the list of servers initially participating in the group, we must decide on the ports each server will use. Group Replication setup requires two ports for each server: one for the normal connections and another for use with Group Replication. For this tutorial, we will use ports 24801+ for the server connections and ports 24901+ for the Group Replication ports. In addition, since we are using local instances, the hostname for all the members in the group will use the loopback address (127.0.0.1), but this would normally be the hostname of the server on which it is running. Finally, we also need to choose server ids, so we will use sequential values starting at 1. Listing 9-1 shows the configuration file we will use for the master in this tutorial.

Listing 9-1. Primary Configuration File (Group Replication)

```
[mysqld]
datadir="/Users/cbell/gr/data/primary"
basedir="/usr/local/mysql-8.0.16-macos10.14-x86_64/"
plugin_dir="/usr/local/mysql/lib/plugin/"
plugin-load=group_replication.so
port=24801
mysqlx-port=33061
mysqlx-socket="/Users/cbell/rpl/primaryx.sock"
socket="/Users/cbell/rpl/primary.sock"
server_id=101
master_info_repository=TABLE
relay_log_info_repository=TABLE
log_bin=master_binlog
binlog_format=row
```

```
# GTID VARIABLES
gtid_mode=ON
enforce_gtid_consistency=ON
log_slave_updates=ON
binlog_checksum=NONE

# GR VARIABLES
transaction_write_set_extraction=XXHASH64
group_replication_recovery_use_ssl=ON
group_replication_group_name="bbbbbbbb-bbbb-cccc-dddd-eeeeeeeeeeee"
group_replication_start_on_boot=OFF
group_replication_local_address="127.0.0.1:24801"
group_replication_group_seeds="127.0.0.1:24901,127.0.0.1:24902,
                                127.0.0.1:24903,127.0.0.1:24904"
group_replication_bootstrap_group=OFF
```

You may notice there is no `log-bin` variable set. The server will automatically enable the binary log when it encounters the variables for Group Replication as it is required. However, you can include the variable if you want to name the binary log files or located them in another folder, but that is an advanced configuration option that isn't necessary for a tutorial or even a development installation.

Note If you are running this tutorial on Windows and do not have SSL installed and MySQL configured for use with SSL connections, you must remove the `group_replication_recovery_use_ssl` option.

For this tutorial, you should create a file in the folder we created earlier named `primary.cnf`; for example, `/Users/cbell/gr/primary.cnf`. We will use this file to start the instance for the primary in a later step.

Now, let's look at the configuration files for the secondaries.

Secondary Configuration File

The configuration file for the secondaries is very similar to the one for the primary, with the only changes being to use the correct values for the instance-specific variables like port, data directory, socket, and server id. However, there are some differences beyond those settings. The `transaction_write_set_extraction` variable is set on the initial primary.

For the secondaries, we add `group_replication_recovery_get_public_key` and set it to `ON`. This variable determines if the secondary requests from the primary the public key required for RSA key pair-based password exchange. This variable applies to secondaries that authenticate with the `caching_sha2_password` authentication plugin.

We also include the `plugin_dir` for the path to the plugin executables, which you can use the `SHOW VARIABLES LIKE '%plugin%'` command. Another variable we include is the `plugin-load` variable with `group_replication.so` as the value. Note the `.so` file extension refers to shared object libraries on *nix computers. On Windows, you would use `.dll`.

Listing 9-2 shows the configuration file for the first secondary (named `secondary1.cnf`).

Listing 9-2. Secondary Configuration File (Group Replication)

```
[mysqld]
datadir="/Users/cbell/gr/data/secondary1"
basedir="/usr/local/mysql-8.0.16-macos10.14-x86_64/"
plugin_dir=/usr/local/mysql/lib/plugin/
plugin-load=group_replication.so
port=24901
mysqlx-port=33062
mysqlx-socket="/Users/cbell/rpl/secondary1x.sock"
socket="/Users/cbell/rpl/data/secondary1.sock"
server_id=102
master_info_repository=TABLE
relay_log_info_repository=TABLE
log_bin=slave1_binlog
binlog_format=row
report-port=24901
report-host=localhost

# GTID VARIABLES
gtid_mode=ON
enforce_gtid_consistency=ON
log_slave_updates=ON
binlog_checksum=NONE
```



```
# GR VARIABLES
group_replication_recovery_get_public_key=ON
group_replication_recovery_use_ssl=ON
group_replication_group_name="bbbbbbbb-bbbb-cccc-dddd-eeeeeeeeeeee"
group_replication_start_on_boot=OFF
group_replication_local_address="127.0.0.1:24901"
group_replication_group_see
ds="127.0.0.1:24901,127.0.0.1:24902,127.0.0.1:24903,127.0.0.1:24904"
group_replication_bootstrap_group=OFF
```

For this tutorial, we will be using three secondaries, so you should create a file for each in the folder we created earlier and name them `secondary1.cnf`, `secondary2.cnf`, and `secondary3.cnf`. Be sure to change the instance-specific variables such as the data directory, socket, port, server id, etc. You must change both ports; the port for the server and the port for Group Replication.

Start the MySQL Instances

Now we are ready to start the MySQL instances. This is easy to do since we have already created the configuration files with all the parameters we need. We only need to provide the configuration file with the `--defaults-file` option. The following shows the commands to start the server instances used in this tutorial. Notice a redirect is added to place the messages from the server in a log file.

```
$ mysqld --defaults-file=primary.cnf > primary_output.txt 2>&1 &
$ mysqld --defaults-file=secondary1.cnf > secondary1_output.txt 2>&1 &
$ mysqld --defaults-file=secondary2.cnf > secondary2_output.txt 2>&1 &
$ mysqld --defaults-file=secondary3.cnf > secondary3_output.txt 2>&1 &
```

When you run these commands, you should run them from the folder that contains the configuration files; otherwise, you will have to provide the full path to the configuration file. While the commands include the redirects, you may want to use a separate terminal the first time you start the server to ensure there are no errors. Listing 9-3 shows an excerpt of the messages printed when launching the primary.

Listing 9-3. Starting the Primary Instance

```

$ mysqld --defaults-file=primary.cnf
MacBook-Pro-2:gr cbell$ 2019-05-05T21:00:42.388965Z 0 [System] [MY-010116]
[Server] /usr/local/mysql-8.0.16-macos10.14-x86_64/bin/mysqld (mysqld
8.0.16) starting as process 935
2019-05-05T21:00:42.392064Z 0 [Warning] [MY-010159] [Server] Setting lower_
case_table_names=2 because file system for /Users/cbell/gr/data/primary/ is
case insensitive
2019-05-05T21:00:42.662123Z 0 [Warning] [MY-010068] [Server] CA certificate
ca.pem is self signed.
2019-05-05T21:00:42.676713Z 0 [System] [MY-010931] [Server] /usr/local/
mysql-8.0.16-macos10.14-x86_64/bin/mysqld: ready for connections. Version:
'8.0.16' socket: '/Users/cbell/rpl/primary.sock' port: 24801 MySQL
Community Server - GPL.
2019-05-05T21:00:42.895674Z 0 [System] [MY-011323] [Server] X Plugin ready
for connections. Socket: '/Users/cbell/rpl/primaryx.sock' bind-address:
':::' port: 33061

```

Once again, if you plan to use a single terminal, it is recommended to redirect the output to a file and use the option to start the application in another process (e.g., the `&` symbol).

If you are following along with this tutorial and haven't done so already, go ahead and start the secondaries. Use the command shown previously with the redirect to a file. Once all the server instances are started, we can move on to the next step.

INSTALLING THE GROUP REPLICATION PLUGIN

If you do not want to include the Group Replication plugin in the configuration file, you can install it manually with the `INSTALL PLUGIN` command. Once done, you do not have to run the command again – the server will reboot with the plugin enabled. The command requires the name of the plugin along with the name of the dynamically loadable executable. In this case, the name of the plugin is `group_replication` and the name of the loadable executable is `group_replication.so`. Thus, the command we would use is `INSTALL PLUGIN group_replication SONAME 'group_replication.so'`.

Notice the `.so` in the file name. This is the extension you would use for *nix platforms. On Windows, the file name extension is `.dll`. You can check the status of the plugins with the `SHOW PLUGINS` command.

You must run the command on all instances. Once the plugin is loaded on all instances, we can proceed to create the replication user on all the instances.

Create the Replication User Account

After the MySQL instances are started, you must create a user to be used by the servers to connect to each other. Recall, in Group Replication, the servers all “talk” to each other. Fortunately, the commands are the same as what we used in MySQL Replication. We need to create this user on all the server instances. The following shows the commands needed to create the replication user. Execute these commands on all your servers.

```
SET SQL_LOG_BIN=0;
CREATE USER rpl_user@'%' IDENTIFIED BY 'rpl_pass';
GRANT REPLICATION SLAVE ON *.* TO rpl_user@'%';
FLUSH PRIVILEGES;
SET SQL_LOG_BIN=1;
```

Notice the use of `%` in the hostname. This was done to ensure the replication user can connect from any server. You would not normally do this for a production environment, but for a tutorial or development testing, it makes things a bit easier.

Recall, these commands tell the server to temporarily disable logging of changes to the binary log. We do this whenever we do not want to replicate the commands on other machines in the topology. Specifically, maintenance and administrative commands like creating users should not be replicated. Turning off the binary log is a great way to ensure you do not accidentally issue transactions that cannot be executed on other machines.

The best way to execute these commands is to save them to a file named `create_rpl_user.sql` and use the `source` command of the `mysql` client to read the commands from the file and execute them. Thus, you can quickly create the replication user on all instances with the following commands.

```
$ mysqlsh --uri root@localhost:24801 --sql -f "create_rpl_user.sql"
$ mysqlsh --uri root@localhost:24802 --sql -f "create_rpl_user.sql"
$ mysqlsh --uri root@localhost:24803 --sql -f "create_rpl_user.sql"
$ mysqlsh --uri root@localhost:24804 --sql -f "create_rpl_user.sql"
```

Now we are ready to start Group Replication on the primary.

Start Group Replication on the Primary

The next step is start Group Replication on the primary for the first time. Recall from our discussion on the Group Replication variables, the variable `group_replication_bootstrap_group` is normally set to `OFF` except on the first start of the group. Since the group has never been started, we must do so on the primary.

Fortunately, the variable `group_replication_bootstrap_group` is dynamic, and we can turn it on and off on-the-fly. Thus, we can run the following commands on the primary to start Group Replication for the first time.

```
$ mysqlsh --uri root@localhost:24801 --sql
...
SET GLOBAL group_replication_bootstrap_group=ON;
START GROUP_REPLICATION;
SET GLOBAL group_replication_bootstrap_group=OFF;
```

If you recall, we set `group_replication_bootstrap_group` to `OFF` in the primary configuration file. This was so that if we restart the primary, the setting will be correct. You can set it `ON` if you like, but you would have to change it in the configuration file before you restart the primary. Setting it to `OFF` is much safer and less work.

If you are following along in this tutorial, go ahead and run those commands on the primary now. Once done, you are now ready to connect the secondaries to the primary.

Connect the Secondaries to the Primary

The next step is to connect the secondaries to the primary. We use the same `CHANGE MASTER` command as we saw in the last tutorial, however, we only need the replication user and password. We tell the server to connect to the special replication channel named `group_replication_recovery`. The following shows the command used for each of the secondaries to connect them to the primary.

```
CHANGE MASTER TO MASTER_USER='rpl_user', MASTER_PASSWORD='rpl_pass' FOR
CHANNEL 'group_replication_recovery';
```

Notice we need less information than even the GTID-enabled replication. Cool! You must run this command on all the secondaries. It may be easier to save this to a file and execute it using the mysql client like we did for the replication user. For example, save this to a file named `change_master.sql` and execute it as the following.

```
$ mysqlsh --uri root@localhost:24802 --sql -f "change_master.sql"
$ mysqlsh --uri root@localhost:24803 --sql -f "change_master.sql"
$ mysqlsh --uri root@localhost:24804 --sql -f "change_master.sql"
```

Now that we have our secondaries configured to connect primary, we must finish the process by starting Group Replication.

Start Group Replication on the Secondaries

The next step is to start Group Replication on the secondaries. Rather than use the `START SLAVE` command like MySQL Replication, Group Replication uses the command `START GROUP_REPLICATION`. Run this on each of the secondaries as the following.

```
$ mysqlsh --uri root@localhost:24802 --sql -e "START GROUP_REPLICATION"
$ mysqlsh --uri root@localhost:24803 --sql -e "START GROUP_REPLICATION"
$ mysqlsh --uri root@localhost:24804 --sql -e "START GROUP_REPLICATION"
```

The `START GROUP_REPLICATION` command normally does not report any errors, and it may take a bit longer to return. This is because there are a lot of things going on in the background when the secondary connects to and begins negotiating with the primary. However, unlike MySQL Replication, you cannot use `SHOW SLAVE STATUS` to check the status. In fact, issuing that command will get no results. So what do you do?

Verify Group Replication Status

Group Replication has redesigned how we monitor replication services. Group Replication adds several views to the `performance_schema` database that you can use to monitor Group Replication. There is a lot of information there and if you are interested, you can see <https://dev.mysql.com/doc/refman/8.0/en/group-replication-monitoring.html> to learn more about the views and what they contain.

Checking Group Replication status requires issuing queries against the `performance_schema` views. The `replication_group_members` view (table) is used for monitoring the status of the different server instances that are tracked in the current view or in other words are part of the group and as such are tracked by the membership service. The information is shared between all the server instances that are members of the replication group, so information on all the group members can be queried from any member. Listing 9-4 shows the command in action. You can save the commands in a file named `check_gr.sql` for later use.

Listing 9-4. Checking Group Replication Status

```
$ mysqlsh --uri root@localhost:24802 --sql
MySQL Shell 8.0.16
```

```
Copyright (c) 2016, 2019, Oracle and/or its affiliates. All rights reserved.
Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
Other names may be trademarks of their respective owners.
```

```
Type '\help' or '\?' for help; '\quit' to exit.
```

```
Creating a session to 'root@localhost:24802'
```

```
Fetching schema names for autocompletion... Press ^C to stop.
```

```
Your MySQL connection id is 28
```

```
Server version: 8.0.16 MySQL Community Server - GPL
```

```
No default schema selected; type \use <schema> to set one.
```

```
> SHOW SLAVE STATUS\G
```

```
Empty set (0.0003 sec)
```

```
> SELECT * FROM performance_schema.replication_group_members \G
```

```
***** 1. IOW *****
```

```
CHANNEL_NAME: group_replication_applier
MEMBER_ID: e0b8aeca-6f7e-11e9-bd22-533c3552fe03
MEMBER_HOST: MacBook-Pro-2.local
MEMBER_PORT: 24801
MEMBER_STATE: ONLINE
MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.16
```

```

***** 2. ROW *****
CHANNEL_NAME: group_replication_applier
  MEMBER_ID: e32085ac-6f7e-11e9-a081-cc47fe26c35d
  MEMBER_HOST: localhost
  MEMBER_PORT: 24901
  MEMBER_STATE: ONLINE
  MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.16
***** 3. ROW *****
CHANNEL_NAME: group_replication_applier
  MEMBER_ID: e5818c4c-6f7e-11e9-a41a-c85d7844eaca
  MEMBER_HOST: localhost
  MEMBER_PORT: 24902
  MEMBER_STATE: ONLINE
  MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.16
***** 4. ROW *****
CHANNEL_NAME: group_replication_applier
  MEMBER_ID: e7f23058-6f7e-11e9-a7c3-c576fe9a0754
  MEMBER_HOST: localhost
  MEMBER_PORT: 24903
  MEMBER_STATE: RECOVERING
  MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.16
4 rows in set (0.0013 sec)

```

Notice we run the `SHOW SLAVE STATUS` command but get nothing in return. Drat. However, when we query the view, we get a short list of information, including the current state of each member of the group. Interestingly, you can run this query on any member in the group. This shows how Group Replication propagates metadata to all the members in the group.

You can also narrow the output to get a more pleasing view including only the member host, port, state, and role shown here.

```
> SELECT MEMBER_HOST, MEMBER_PORT, MEMBER_STATE, MEMBER_ROLE FROM
performance_schema.replication_group_members;
```

```
+-----+-----+-----+-----+
| MEMBER_HOST          | MEMBER_PORT | MEMBER_STATE | MEMBER_ROLE |
+-----+-----+-----+-----+
| MacBook-Pro-2.local |      24801 | ONLINE      | PRIMARY     |
| localhost           |      24901 | ONLINE      | SECONDARY   |
| localhost           |      24902 | ONLINE      | SECONDARY   |
| localhost           |      24903 | ONLINE      | SECONDARY   |
+-----+-----+-----+-----+
4 rows in set (0.0006 sec)
```

If you want to just locate the primary, you can use the following query on any of the group members. When you execute this query on any one of the members in the group, you will see the UUID of the primary as the following.

```
> SELECT member_id, member_host, member_port FROM performance_schema.global_
status JOIN performance_schema.replication_group_members ON VARIABLE_
VALUE=member_id WHERE VARIABLE_NAME='group_replication_primary_member';
+-----+-----+-----+
| member_id          | member_host          | member_port |
+-----+-----+-----+
| e0b8aeca-6f7e-11e9-bd22-533c3552fe03 | MacBook-Pro-2.local |      24801 |
+-----+-----+-----+
1 row in set (0.0014 sec)
```

Now that we have Group Replication running, let’s create some data. We will use the same sample data as we did in the last tutorial. However, this time, we will execute the queries on one of the secondaries. What do you expect to happen? If you’re thinking in terms of MySQL Replication, you may expect the data to appear only on one of the secondaries. Let’s see what happens. The following shows executing the data queries on one of the secondaries.

```
> CREATE DATABASE test;
ERROR: 1290 (HY000): The MySQL server is running with the --super-read-only
option so it cannot execute this statement
```

Why did we get this error? It turns out, each secondary is started with super-read-only, which means that no one can submit writes, even those with the “super” privilege. So the common issue of writes sent to a slave (from MySQL Replication) is resolved.

Huzzah! Use of super-read-only also indicates we are running Group Replication in single-primary mode (which is the default). We will see other modes when we explore the nuances of InnoDB Cluster in later chapters.

Returning to our test of creating some data, let's run the same commands on the primary. The following shows the expected results.

```
$ mysqlsh --uri root@localhost:24801 --sql
...
> CREATE DATABASE test;
Query OK, 1 row affected (0.0042 sec)

> USE test;
Query OK, 0 rows affected (0.0003 sec)

> CREATE TABLE test.t1 (id INT PRIMARY KEY, message TEXT NOT NULL);
Query OK, 0 rows affected (0.0093 sec)

> INSERT INTO test.t1 VALUES (1, 'Chuck');
Query OK, 1 row affected (0.0103 sec)
```

Here, we see the data was created. Now, to check the secondary. The following shows the results of running a query on the secondary. As you can see, the data has been replicated.

```
$ mysqlsh --uri root@localhost:24803 --sql
...
> SELECT * FROM test.t1;
+----+-----+
| id | message |
+----+-----+
| 1 | Chuck   |
+----+-----+
1 row in set (0.0006 sec)
```

This concludes the short tutorial on setting up MySQL Group Replication. This section presented a brief look at MySQL Group Replication in its barest, simplest terms. Now, let's look at how we can script an example setup of MySQL Group Replication. However, there is one more step.

Shutting Down Group Replication

If you try this tutorial on your own machine, remember to shutdown your instances in a precise order. On each secondary, you first want to stop Group Replication with the following command.

```
$ mysqlsh --uri root@localhost:24802 --sql -e "STOP GROUP_REPLICATION"
```

Once all slaves are stopped, you can shutdown the slaves first then the master with the following commands. Notice we used the old protocol ports (13001, 13002) for the master and slaves. This is because the X Protocol does not support the shutdown command. If you encounter an error when using the MySQL X ports, try the old protocol and rerun the command.

```
$ mysqlsh --uri root@localhost:24804 --sql -e "SHUTDOWN"
$ mysqlsh --uri root@localhost:24803 --sql -e "SHUTDOWN"
$ mysqlsh --uri root@localhost:24802 --sql -e "SHUTDOWN"
$ mysqlsh --uri root@localhost:24801 --sql -e "SHUTDOWN"
```

Demonstration of Failover

Now that we have a working Group Replication setup, let's see how automatic failover works. If you haven't run the preceding tutorial and want to follow along, be sure to run the preceding steps first.

Automatic failover is a built-in feature of Group Replication. The communication mechanism ensures that the primary (in a single-primary configuration) is monitored for activity, and when it is no longer available or something serious has gone wrong, the group can decide to terminate the primary connection and elect a new primary.

Let's see how this works. Recall from the preceding tutorial, we have the initial primary running on port 24801. We can simulate a failure by killing the MySQL process for that server. Since we're running on Linux, we can determine the process id by inspecting the process id file, which MySQL creates with the name of the machine and a file extension of .pid in the data directory. For example, the file for the primary shown in the tutorial is in data/primary/oracle-pc.pid. The following demonstrates how to find the process id and stop it. Note that you may need super user privileges to kill the process.

```
$ more ./data/primary/oracle-pc.pid
18019
$ sudo kill -9 18019
```

Tip On Windows, you can use the task manager to kill the process.

Now that the primary is down, we can view the health of the group with the preceding queries. Recall, we use the `check_gr.sql` file that contains the queries (see Listing 9-4). Listing 9-5 shows the output from the queries.

Listing 9-5. Checking Group Health After Primary Goes Down

```
$ mysqlsh --uri root@localhost:24802 --sql -e "source check_gr.sql"
***** 1. ROW *****
CHANNEL_NAME: group_replication_applier
  MEMBER_ID: 2854aecd-4330-11e8-abb6-d4258b76e981
  MEMBER_HOST: oracle-pc
  MEMBER_PORT: 24802
  MEMBER_STATE: ONLINE
  MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.16
***** 2. ROW *****
CHANNEL_NAME: group_replication_applier
  MEMBER_ID: 2ecd9f66-4330-11e8-90fe-d4258b76e981
  MEMBER_HOST: oracle-pc
  MEMBER_PORT: 24803
  MEMBER_STATE: ONLINE
  MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.16
***** 3. ROW *****
CHANNEL_NAME: group_replication_applier
  MEMBER_ID: 3525b7be-4330-11e8-80b1-d4258b76e981
  MEMBER_HOST: oracle-pc
  MEMBER_PORT: 24804
  MEMBER_STATE: ONLINE
  MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.16
```

```

+-----+-----+-----+
| member_id          | member_host | member_port |
+-----+-----+-----+
| 2854aecd-4330-11e8-abb6-d4258b76e981 | oracle-pc  |          24802 |
+-----+-----+-----+
    
```

Notice we see the group has automatically chosen a new primary (on port 24802) and there are now only three servers in the group. So there is no loss of write capability. However, recall from an earlier discussion that the group can tolerate only so many failures and once that limit is reached, the group can no longer successfully failover, and in those cases, the group may not be fault tolerant. Listing 9-6 shows the state of this same group once the second and third primary machines have been stopped. Notice the state of the last primary is unknown.

Listing 9-6. State of the Group When No More Primary Servers Remain

```

$ mysqlsh --uri root@localhost:24804 --sql -e "source check_gr.sql"
***** 1. ROW *****
CHANNEL_NAME: group_replication_applier
MEMBER_ID: 2ecd9f66-4330-11e8-90fe-d4258b76e981
MEMBER_HOST: oracle-pc
MEMBER_PORT: 24803
MEMBER_STATE: UNREACHABLE
MEMBER_ROLE: PRIMARY
MEMBER_VERSION: 8.0.16
***** 2. ROW *****
CHANNEL_NAME: group_replication_applier
MEMBER_ID: 3525b7be-4330-11e8-80b1-d4258b76e981
MEMBER_HOST: oracle-pc
MEMBER_PORT: 24804
MEMBER_STATE: ONLINE
MEMBER_ROLE: SECONDARY
MEMBER_VERSION: 8.0.16
    
```

```

+-----+-----+-----+
| member_id          | member_host | member_port |
+-----+-----+-----+
| 2ecd9f66-4330-11e8-90fe-d4258b76e981 | oracle-pc   | 24803       |
+-----+-----+-----+

```

Here, we can see one of the secondaries has indeed taken over the primary role and the group has tolerated the failure. Note, however, that since we started with four servers, we can only tolerate one server failure. However, we can add new secondaries at any time to improve the number of faults the group can tolerate. For example, we can repair the failed secondary and add it back to the group.

Summary

There is no denying that Group Replication is a leap forward in MySQL high availability. However, as we have seen in the tutorial in this chapter, it is not simple to set up. While those familiar with working with MySQL Replication will see the process similar but with a few extra steps and slightly different commands, with the most changes in the configuration file, those new to MySQL and high availability may feel the learning curve is quite steep.

In this chapter, we took a ground floor view of Group Replication and experienced what it takes to set it up and maintain a group initially and during a failure or two. If you are among those thinking there must be a better way, there is and we are almost there!

In the next chapter, we will look at the latest and best high availability feature in MySQL – InnoDB Cluster and manage it with MySQL Shell.

CHAPTER 10

Using the Shell with InnoDB Cluster

Thus far, we have learned what high availability is and how to set up a basic high availability installation with MySQL Replication. We also learned how to configure Group Replication for even better high availability. However, we learned along the way that the setup and commands to make all this work are specific and require more manual setup and configuration steps than most would want. Fortunately, Oracle has listened to their customers and has been busy working on MySQL high availability by dramatically improving not only the features and capabilities but also the ease of use (or administration). This is where InnoDB Cluster really shines.

InnoDB Cluster represents a huge leap forward in high availability for MySQL. Best of all, it comes standard in all releases of MySQL 8.0. In this chapter, we will discover what makes InnoDB Cluster such an important feature for enterprises large and small. As you will see, InnoDB Cluster is made up of several components that all work together well and achieve the highest form of high availability for MySQL right out of the box!

Overview

One of the most exciting new features in MySQL 8.0 is the InnoDB Cluster. It is designed to make high availability easier to set up, use, and maintain. InnoDB Cluster works with the X DevAPI via MySQL Shell and the AdminAPI, Group Replication, and the MySQL Router to take high availability and read scalability to a new level.

That is, it combines new features in the InnoDB storage engine (internals of MySQL) for cloning data with Group Replication and the MySQL Router to provide a new way to set up and manage high availability at the application layer. The following list describes the components that make up InnoDB Cluster. We have seen most of these before, and

we will learn more about these in the next sections and details of how to use each in the next chapter.

- *Group Replication*: A new form of replication that builds upon MySQL Replication adding an active communication protocol (group membership) that permits higher levels of availability including fault tolerance with automatic failover.
- *MySQL Shell*: The new MySQL client and programming environment for JavaScript and Python and, as we shall see, the administration console for InnoDB Cluster.
- *XDevAPI*: A special application programming interface for applications to interact with data programmatically.
- *AdminAPI*: Is a special API available via the MySQL Shell for configuring and interacting with InnoDB Cluster. Thus, the AdminAPI has features designed to make working with InnoDB Cluster easier.
- *MySQL Router*: Lightweight middleware that provides transparent routing between your application and back-end MySQL Servers. We will learn more about MySQL Router in a later section.

You may be wondering how all these products and features can work together seamlessly. As you will see, working with InnoDB Cluster through MySQL Shell hides many of the details (and tedium) of working with the components individually. For example, we no longer must write specialized configuration files for replication.

Let's look at a conceptual configuration to get a sense of how the components interact. In this use case, a Cluster of three servers is set up with a single primary, which is the target for all writes (updates). Multiple secondary servers maintain replicas of the data, which can be read from and thus enable reading data without burdening the primary, thus enabling read-out scalability (but all servers participate in consensus and coordination).

The incorporation of Group Replication means the Cluster is fault tolerant and group membership is managed automatically. MySQL Router caches the metadata of the InnoDB Cluster and performs high availability routing to the MySQL Server instances making it easier to write applications to interact with the Cluster. Figure 10-1 shows how each of these components are arranged conceptually to form InnoDB Cluster.

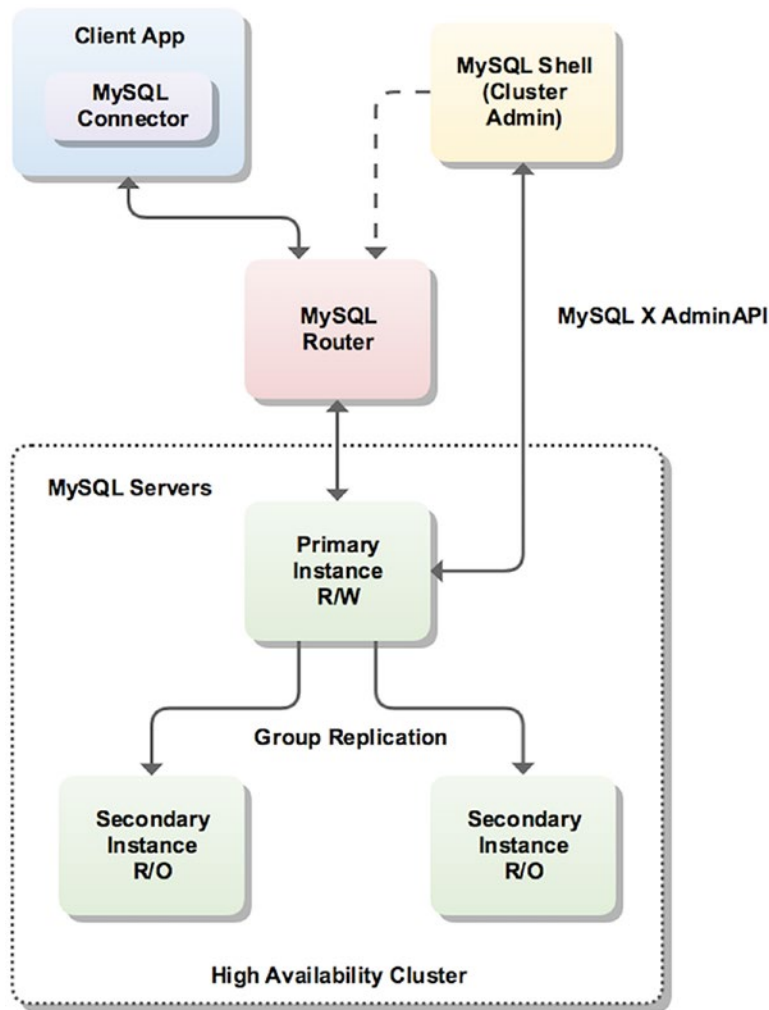


Figure 10-1. Typical Configuration of InnoDB Cluster (courtesy of Oracle)

You may be wondering what makes this different from a read-out scalability setup with standard replication. At a high level, it may seem that the solutions are solving the same use case. However, with InnoDB Cluster, you can create, deploy, and configure servers in your Cluster from MySQL Shell. That is, you can use the X AdminAPI (also called the AdminAPI) via the shell to create and administer an InnoDB Cluster programmatically using either JavaScript or Python.

Furthermore, you can deploy InnoDB Cluster in a sandbox through the MySQL Shell. More specifically, you can deploy a test Cluster running on your local computer and experiment with it prior to deploying it in production. Fortunately, all the steps are

the same with the only addition of the keyword `sandbox` in some of the function names and a few extra functions for creating the local MySQL instances. We'll see an in-depth coverage of running InnoDB Cluster in a sandbox in Chapter 11.

WHAT'S A SANDBOX?

A sandbox is a term used to describe a situation where data and metadata (configuration files, etc.) are organized in such a way that the data and metadata does not affect (replace) existing data or installations of a product. In the case of the MySQL AdminAPI, the sandbox it implements ensures any configuration of the servers in an InnoDB Cluster will not affect any existing installation of MySQL on the machine.

Deploying InnoDB Cluster in production would require setting up the servers individually then connecting to them from the shell and preparing them. Again, this step is the same as in the sandbox deployment. The only difference is you're using an existing, running MySQL server rather than another instance running on your local machine.

Now that we have an overview of what InnoDB Cluster is, let's look at the core components of InnoDB Cluster beginning with the InnoDB storage engine.

WHAT IS A STORAGE ENGINE?

A storage engine is a mechanism to store data in various ways. For example, there is a storage engine that allows you to interact with comma-separated values (text) files (CSV), another that is optimized for writing log files (Archive), one that stores data in memory only (Memory), and even one that doesn't store anything at all (Blackhole). Along with InnoDB, the MySQL server ships with several storage engines. The following sections describe some of the more commonly used alternative storage engines. Note that some storage engines have been dropped from support over the evolution of MySQL, including the Berkeley Database (BDB) storage engine.

InnoDB Storage Engine

The core component of InnoDB Cluster is the InnoDB storage engine. Since MySQL 5.5, InnoDB has been the flagship storage engine (and the default engine) for MySQL. Oracle has slowly evolved away from the multiple storage engine model focusing on what a modern database server should do – support transactional storage mechanisms. InnoDB is the answer to that requirement and much more.

InnoDB is a general-purpose storage engine that balances high reliability and high performance. The decision to use the InnoDB storage engine was made after several attempts to build a robust, high-performance storage engine for MySQL. Given the maturity and sophistication of InnoDB, it made much more sense to use what already existed. Plus, Oracle owned both MySQL and InnoDB.

The InnoDB storage engine is used when you need to use transactions. InnoDB supports traditional ACID transactions and foreign key constraints. All indexes in InnoDB are B-trees where the index records are stored in the leaf pages of the tree. InnoDB is the storage engine of choice for high reliability and transaction processing environments.

ACID stands for atomicity, consistency, isolation, and durability. Perhaps one of the most important concepts in database theory, it defines the behavior that database systems must exhibit to be considered reliable for transaction processing. The following briefly describes each aspect.

- *Atomicity* means that the database must allow modifications of data on an “all or nothing” basis for transactions that contain multiple commands. That is, each transaction is atomic. If a command fails, the entire transaction fails, and all changes up to that point in the transaction are discarded. This is especially important for systems that operate in highly transactional environments, such as the financial market.
- *Consistency* means that only valid data will be stored in the database. That is, if a command in a transaction violates one of the consistency rules, the entire transaction is discarded and the data is returned to the state they were in before the transaction began. Conversely, if a transaction completes successfully, it will alter the data in a manner that obeys the database consistency rules.

- *Isolation* means that multiple transactions executing at the same time will not interfere with one another. This is where the true challenge of concurrency is most evident. Database systems must handle situations in which transactions (alter, delete, etc.) cannot violate the data being used in another transaction. There are many ways to handle this. Most systems use a mechanism called locking that keeps the data from being used by another transaction until the first one is done. Although the isolation property does not dictate which transaction is executed first, it does ensure they will not interfere with one another.
- *Durability* means that no transaction will result in lost data nor will any data created or altered during the transaction be lost. Durability is usually provided by robust backup-and-restore maintenance functions. Some database systems use logging to ensure that any uncommitted data can be recovered on restart.

Perhaps the most important feature that sets InnoDB apart from the earlier storage engines in MySQL is its configurability. While some of the early storage engines were configurable, none were at the scale that exists for configuring InnoDB. There are dozens of parameters you can use to tune InnoDB to meet your unique storage needs.

Caution Use care when tinkering with InnoDB parameters. It is possible to degrade your system to the point of hurting performance. Like any tuning exercise, always consult the documentation (and experts) first, then plan to target specific parameters. Be sure to tune one parameter at a time, and test, confirm or revert before moving on.

While InnoDB works very well out of the box with well-chosen defaults and it is likely to not require much tuning for most, those that need to tune MySQL will find all they need and more to get their database systems running at peak efficiency.

See <https://dev.mysql.com/doc/refman/8.0/en/innodb-introduction.html> for more information about the InnoDB Storage engine including numerous configuration and tuning options.

Tip Another excellent source for tips and advice for configuring MySQL and InnoDB is the book *High Performance MySQL: Optimization, Backups, Replication* by Schwartz, Zaitsev, Tkachenko (O'Reilly, 2012).

To better understand how we arrived at InnoDB Cluster, let's take a short tour of the other storage engines available in MySQL 8.0 and earlier releases.

Tip If you want to see which storage engines are available for use on your MySQL server, you can use the `SHOW ENGINES` command. See <https://dev.mysql.com/doc/refman/8.0/en/create-table.html> to learn more how to specify a storage engine with the `CREATE TABLE` command.

Archive

The archive storage engine is designed for storing large amounts of data in a compressed format. The archive storage mechanism is best used for storing and retrieving large amounts of seldom-accessed archival or historical data. Such data include security-access-data logs. While not something that you would want to search or even use daily, it is something a database professional who is concerned about security would want to have should a security incident occur. No indexes are provided for the archive storage mechanism, and the only access method is via a table scan. Thus, the archive storage engine should not be used for normal database storage and retrieval.

Blackhole

The blackhole storage engine, an interesting feature with surprising utility, is designed to permit the system to write data, but the data are never saved. If binary logging is enabled, however, the SQL statements are written to the logs. This permits database professionals to temporarily disable data ingestion in the database by switching the table type. This can be handy in situations in which you want to test an application to ensure it is writing data that you don't want to store, such as when creating a relay slave for filtering replication.

CSV

The CSV storage engine is designed to create, read, and write comma-separated value (CSV) files as tables. While the CSV storage engine does not copy the data into another format, the sheet layout, or metadata, is stored along with the filename specified on the server in the database folder. This permits database professionals to rapidly export structured business data that is stored in spreadsheets. The CSV storage engine does not provide any indexing mechanisms, making it impractical for large amounts of data. It is intended to be used as a link between storing data and visualizing it in spreadsheet applications.

Federated

The federated storage engine is designed to create a single table reference from multiple database systems. The federated storage engine therefore works like the merge storage engine, but it allows you to link data (tables) together across database servers. This mechanism is similar in purpose to the linked data tables available in other database systems. The federated storage mechanism is best used in distributed or data mart environments.

The most interesting aspect of the federated storage engine is that it does not move data, nor does it require the remote tables to be the same storage engine. This illustrates the true power of the pluggable-storage-engine layer. Data are translated during storage and retrieval.

Memory

The memory storage engine (sometimes called HEAP tables) is an in-memory table that uses a hashing mechanism for fast retrieval of frequently used data. Thus, these tables are much faster than those that are stored and referenced from disk. They are accessed in the same manner as the other storage engines, but the data is stored in-memory and is valid only during the MySQL session. The data is flushed and deleted on shutdown (or a crash).

Memory storage engines are typically used in situations in which static data are accessed frequently and rarely ever altered. Examples of such situations include zip code, state, county, category, and other lookup tables. HEAP tables can also be used in databases that utilize snapshot techniques for distributed or historical data access.

MyISAM

The MyISAM storage engine was originally the default in MySQL and was used by most LAMP stacks, data warehousing, e-commerce, and enterprise applications. MyISAM files are an extension of ISAM built with additional optimizations, such as advanced caching and indexing mechanisms. These tables are built using compression features and index optimizations for speed.

Additionally, the MyISAM storage engine provides for concurrent operations by providing table-level locking. The MyISAM storage mechanism offers reliable storage for a wide variety of applications while providing fast retrieval of data. MyISAM is the storage engine of choice when read performance is a concern.

Merge (MyISAM)

The merge storage engine (sometimes named MRG_MYISAM) is built using a set of MyISAM tables with the same structure (tuple layout or schema) that can be referenced as a single table. Thus, the tables are partitioned by the location of the individual tables, but no additional partitioning mechanisms are used. All tables must reside on the same machine (accessed by the same server). Data is accessed using singular operations or statements, such as SELECT, UPDATE, INSERT, and DELETE. Fortunately, when a DROP is issued on a merge table, only the merge specification is removed. The original tables are not altered.

The biggest benefit of this table type is speed. It is possible to split a large table into several smaller tables on different disks, combine them using a merge-table specification, and access them simultaneously. Searches and sorts will execute more quickly because there is less data in each table to manipulate. For example, if you divide the data by a predicate, you can search only those specific portions that contain the category you are searching for. Similarly, repairs on tables are more efficient because it is faster and easier to repair several smaller individual files than a single large table. Presumably, most errors will be localized to an area within one or two of the files and thus will not require rebuilding and repair of all the data. Unfortunately, this configuration has several disadvantages:

- You can use only identical MyISAM tables, or schemas, to form a single merge table. This limits the application of the merge storage engine to MyISAM tables. If the merge storage engine were to accept any storage engine, the merge storage engine would be more versatile.

- The replace operation is not permitted.
- Indexed access has been shown to be less efficient than for a single table.

Merge storage mechanisms are best used in very large database (VLDB) applications, such as data warehousing where data resides in more than one table in one or more databases.

Performance Schema

The performance schema storage engine is a special reporting engine for use in monitoring MySQL Server execution at a low level. While it is shown in the list of available storage engines, it is not an available option for storing data.

Now that we understand more about what storage engines are and the InnoDB storage engine in particular, let's look at the other components of InnoDB Cluster.

Group Replication

If you have used MySQL replication, you are no doubt very familiar with how to leverage it when building high availability solutions. Indeed, it is likely you have discovered a host of ways to improve availability in your applications with MySQL Replication. And, if you've been following along in the book, you've also learned about Group Replication.

Group Replication was released as GA in December 2016 (starting with the 5.7.17 release) and is bundled with the MySQL server in the form of a plugin. Since Group Replication is implemented as a server plugin, you can install the plugin and start using Group Replication without having to reinstall your server, which makes experimenting with new functionality easy.

Recall that Group Replication also makes synchronous replication (among the nodes belonging to the same group) a reality, while the existing MySQL Replication feature is asynchronous (or at most semi-synchronous). Therefore, stronger data consistency is provided (data available on all members with no delay) at all times.

Another important distinction between Group Replication and standard replication we learned is that all the servers in the group can participate in updating the data with conflicts resolved automatically. However, you can configure Group Replication to allow updates by only one server (called the primary) with the other servers acting as secondary servers or as a backup (for failover).

It should come as no surprise then that Group Replication is a core component of InnoDB Cluster. We get all the benefits of Group Replication without the complexity of configuring and maintaining the group.

MySQL Shell

Recall the MySQL Shell is designed to use the new X Protocol for communicating with the server via the X Plugin, which allows the shell to communicate with the MySQL server and its components exposing new APIs for working with data and administration. We learned about the X DevAPI in Chapters 6 and 7. The administration API is called the X AdminAPI and allows the shell to communicate with InnoDB Cluster for setup and administration.

X DevAPI

Recall the X DevAPI is a library of classes and methods that implement a new NoSQL interface for MySQL. Specifically, the X DevAPI is designed to allow easy interaction with JSON documents and relational data. The X DevAPI has classes devoted to supporting both concepts allowing developers to use either (or both) in their applications. The X DevAPI allows us to work with MySQL using JavaScript or Python and, coupled with the AdminAPI, provides a powerful mechanism for managing InnoDB Cluster.

Tip See <https://dev.mysql.com/doc/x-devapi-userguide/en/> for more information about the X DevAPI.

AdminAPI

The Admin Application Programming Interface, hence, AdminAPI, is a library of classes and methods that implement a new management interface for InnoDB Cluster. Specifically, the AdminAPI is designed to allow easy interaction with InnoDB Cluster using a scripting language from the MySQL Shell. Thus, MySQL Shell includes the AdminAPI, which enables you to deploy, configure, and administer InnoDB Cluster. The AdminAPI contains two classes for accessing the InnoDB Cluster functionality as follows.

- `dba`: Enables you to administer InnoDB Clusters using the AdminAPI. The `dba` class enables you to administer the Cluster such as creating a new Cluster, working with a sandbox configuration (a way to experiment with InnoDB Cluster using several MySQL instances on the same machine, checking the status of instances and the Cluster).
- `cluster`: Management handle to an InnoDB Cluster. The `cluster` class enables you to work with the cluster to add instances, remove instances, get the status (health) of the cluster, and more.

We will see more of the AdminAPI in Chapter 11 as we explore how to set up and manage InnoDB Cluster.

Tip See https://dev.mysql.com/doc/dev/mysqlsh-api-python/8.0/group__admin__a__p__i.html to learn more about the AdminAPI.

MySQL Router

The MySQL Router is a relatively new component in MySQL. It was originally built for the now obsolete MySQL Fabric product and has been significantly improved and reworked for use with InnoDB Cluster. In fact, it is a vital part of InnoDB Cluster.

The MySQL Router is a lightweight middleware component providing transparent routing between your application and MySQL Servers. While it can be used for a wide variety of use cases, its primary purpose is to improve high availability and scalability by effectively routing database traffic to appropriate MySQL Servers.

Traditionally, for client applications to handle failover, they need to be aware of the InnoDB Cluster topology and know which MySQL instance is the primary (write) server. While it is possible for applications to implement that logic, MySQL Router can provide and handle this functionality for you.

Moreover, when used with InnoDB Cluster, MySQL Router acts as a proxy to hide the multiple MySQL instances on your network and map the data requests to one of the instances in the cluster. If there are enough online replicas, and communication between the components is intact, applications will be able to (re)connect to one of them. The MySQL Router also makes it possible for this to happen by simply repointing applications to connect to Router instead of directly to MySQL.

Tip See <https://dev.mysql.com/doc/mysql-router/8.0/en/> for more information about the MySQL Router.

INNODB CLUSTER AND NDB CLUSTER: WHAT IS THE DIFFERENCE?

If you peruse the MySQL web site, you will find another product with “cluster” in the name. It is enticingly named NDB Cluster. NDB Cluster is a separate product from the MySQL server employing a technology that enables clustering of in-memory databases in a shared-nothing system. The shared-nothing architecture enables the system to work with inexpensive hardware, and with a minimum of specific requirements for hardware or software.

NDB Cluster is designed not to have any single point of failure. In a shared-nothing system, each component is expected to have its own memory and disk, and the use of shared storage mechanisms such as network shares, network file systems, and SANs is not recommended or supported. See <https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster-compared.html> to learn more about NDB Cluster and how it relates to InnoDB.

To learn more about NDB Cluster, see the excellent book by Krogh and Okuno entitled, *Pro MySQL NDB Cluster* (Apress 2018). This book covers every aspect of NDB Cluster and is a must for anyone interested in deploying and managing NDB Cluster.

Using InnoDB with Applications

You may be wondering how InnoDB Cluster could be beneficial in your environment (or even how it is used). That is, we know the benefits of InnoDB Cluster is better form of high availability, but how does MySQL Router and our applications fit into the picture?

When used with MySQL InnoDB Cluster (through the underlying Group Replication) to replicate databases across multiple servers while performing automatic failover in the event of a server failure, the router acts as a proxy to hide the multiple MySQL instances on your network and map the data requests to one of the cluster instances.

Provided there are enough online replicas and reliable network communication is possible, your applications that use the router will be able to contact one of the remaining servers. The router makes this possible by having applications connect to MySQL Router instead of directly to a specific MySQL server. Figure 10-2 shows a logical view of where the router sits in relation to your application and the InnoDB Cluster.

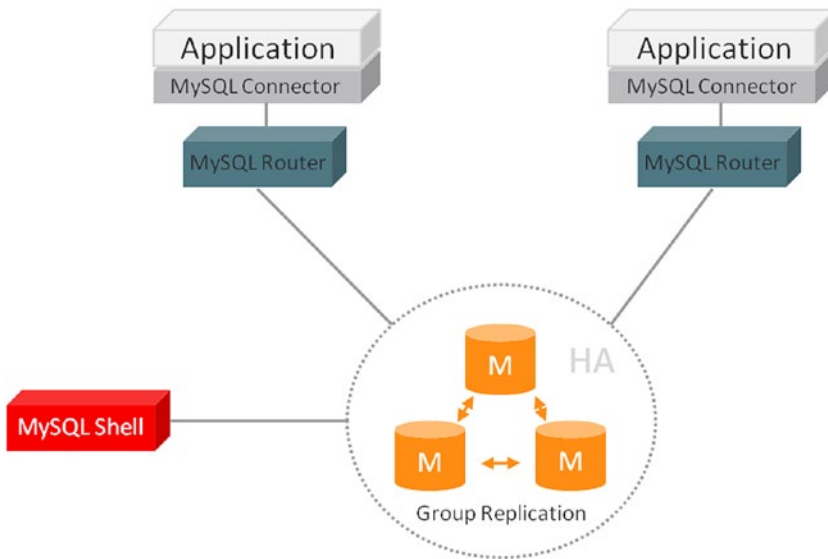


Figure 10-2. *Application Architecture with MySQL Router (courtesy of Oracle Corporation)*

Notice we have two applications depicted, each connecting to an instance of the MySQL Router. The router works by sitting in between applications and MySQL servers. When an application connects to the router, the router chooses a suitable MySQL server from the pool of candidates and connects forwarding all network traffic from the application to that server (and the returning responses from the server to the application).

Behind the scenes, the router stores a list of the servers from InnoDB Cluster along with their status. While this list (or cache) of servers is read initially from a configuration file, subsequent communication between the router and the cluster ensures it gets updated when the topology changes. This means when servers are lost, they become marked as offline by the router and the router skips them. Similarly, if new servers are added to the cluster, the router’s cache is updated to include them.

Thus, to keep the cache updated, the router keeps an open connection to one of the servers in the cluster querying the cluster metadata from the performance schema database. These tables (or views) are updated in real-time by Group Replication whenever a cluster state change is detected. For example, if one of the MySQL servers had an unexpected shutdown.

Finally, the router enables developers to extend MySQL Router using plugins for custom use cases. This means if your application requires a different routing strategy or you want to build packet inspection into your solution, you can extend the router with a custom plugin. While building custom plugins for the router is beyond the scope of this chapter and there are no examples (yet) to study, be sure to check the MySQL developer web site (<https://dev.mysql.com>) and the MySQL Engineering blogs (<https://mysqlserverteam.com/>) for the latest information and examples.

Now that we understand the components that make up InnoDB Cluster, let's briefly discuss what you need to do to setup and configure InnoDB Cluster. We leave a demonstration of InnoDB Cluster for the next chapter.

Setup and Configuration

Setup and configuration of InnoDB Cluster does not require any new and complex steps other than what we did in Chapter 2. If you recall during the installation of MySQL in Chapter 2, we did not select the sandbox option for MySQL Server and InnoDB Cluster. We also did not elect to configure MySQL Router automatically. Had you chosen these options, you would have InnoDB Cluster running in a sandbox with MySQL Router ready for use.

While you can do that by rerunning the installation (e.g., on Windows, run the MySQL Installer for Windows), we will use the more traditional manual setup so that you can learn how to setup InnoDB Cluster and MySQL Router, which will provide a better foundation for you when you deploy InnoDB Cluster and MySQL Router in your environment.

The process to prepare your system for running InnoDB Cluster as a test with several MySQL instances running locally and configure the router to run as well is not difficult. We will see an overview of the process in this chapter and save the demonstration for Chapter 11. But first, let's discuss a bit more about a topic that comes up often when planning new, enterprise-grade features like InnoDB Cluster – upgrading existing servers.

Upgrade Checker

There are a few prerequisites for using InnoDB Cluster. First and foremost is you must use a MySQL version that supports all its components including Group Replication. Fortunately, there is one more important step that you can take to ensure your MySQL servers are ready for InnoDB Cluster. You can use the MySQL Shell Upgrade Checker utility to check each server.

The upgrade checker is a function in MySQL Shell that allows you to verify that a server has been configured correctly for upgrading. If you are using older versions of MySQL or a set of servers that have different versions of MySQL, the upgrade checker utility can save you a lot of headaches when deploying features such as Group Replication or InnoDB Cluster.

The upgrade checker is intended for newer versions of MySQL, but can also work for MySQL 5.7 servers checking them for compatibility and errors in setup. Starting with release 8.0.16, the upgrade checker can also check configuration files for system variables that have non-default values.

To use the upgrade checker, simply run the shell and connect to the MySQL server you want to check. It is always best to run the shell on the same server, but it is not a requirement except for checking configuration files.

The upgrade checker function is included in the `util` library and is named `util.check_for_server_upgrade()`. The upgrade checker can take two optional parameters: 1) the connection information to the server (if there isn't one already established), and 2) a dictionary of options. There are many options including the following.

- `targetVersion`: The MySQL Server version to which you plan to upgrade
- `configPath`: The path to the configuration file for the MySQL server instance
- `outputFormat`: The format for displaying results – either `TEXT` or `JSON`

To best visualize using the upgrade checker, the following demonstrates running the utility on an older MySQL 5.7.22 server. We will check to see if it can be upgraded to 8.0.16 and we'll request `TEXT` output (the default). As you will see, there are quite a few things the utility checks. Listing 10-1 shows the complete output of the running the utility.

Listing 10-1. Demonstration of the Upgrade Checker Utility

```
> util.check_for_server_upgrade('root@localhost:13000',
{'targetVersion': '8.0.16', 'outputFormat': 'TEXT'})
```

```
The MySQL server at localhost:13000, version 5.7.22-log - MySQL Community
Server (GPL), will now be checked for compatibility issues for upgrade to MySQL
8.0.16...
```

- 1) Usage of old temporal type
No issues found
- 2) Usage of db objects with names conflicting with reserved keywords in 8.0
No issues found
- 3) Usage of utf8mb3 charset
No issues found
- 4) Table names in the mysql schema conflicting with new tables in 8.0
No issues found
- 5) Foreign key constraint names longer than 64 characters
No issues found
- 6) Usage of obsolete MAXDB sql_mode flag
No issues found
- 7) Usage of obsolete sql_mode flags
No issues found
- 8) ENUM/SET column definitions containing elements longer than 255 characters
No issues found
- 9) Usage of partitioned tables in shared tablespaces
No issues found
- 10) Usage of removed functions
No issues found
- 11) Usage of removed GROUP BY ASC/DESC syntax
No issues found
- 12) Removed system variables for error logging to the system log configuration
To run this check requires full path to MySQL server configuration file to be specified at 'configPath' key of options dictionary
More information:
<https://dev.mysql.com/doc/relnotes/mysql/8.0/en/news-8-0-13.html#mysqld-8-0-13-logging>

13) Removed system variables

To run this check requires full path to MySQL server configuration file to be specified at 'configPath' key of options dictionary

More information:

<https://dev.mysql.com/doc/refman/8.0/en/added-deprecated-removed.html#optvars-removed>

14) System variables with new default values

To run this check requires full path to MySQL server configuration file to be specified at 'configPath' key of options dictionary

More information:

<https://mysqlservertimeam.com/new-defaults-in-mysql-8-0/>

15) Schema inconsistencies resulting from file removal or corruption

No issues found

16) Issues reported by 'check table x for upgrade' command

No issues found

17) New default authentication plugin considerations

Warning: The new default authentication plugin 'caching_sha2_password' offers more secure password hashing than previously used 'mysql_native_password' (and consequent improved client connection authentication). However, it also has compatibility implications that may affect existing MySQL installations.

If your MySQL installation must serve pre-8.0 clients and you encounter compatibility issues after upgrading, the simplest way to address those issues are to reconfigure the server to revert to the previous default authentication plugin (mysql_native_password). For example, use these lines in the server option file:

```
[mysqld]
default_authentication_plugin=mysql_native_password
```

However, the setting should be viewed as temporary, not as a long term or permanent solution, because it causes new accounts created with the setting in effect to forego the improved authentication security.

If you are using replication please take time to understand how the authentication plugin changes may impact you.

More information:

<https://dev.mysql.com/doc/refman/8.0/en/upgrading-from-previous-series.html#upgrade-caching-sha2-password-compatibility-issues>

<https://dev.mysql.com/doc/refman/8.0/en/upgrading-from-previous-series.html#upgrade-caching-sha2-password-replication>

Errors: 0

Warnings: 1

Notices: 0

No fatal errors were found that would prevent an upgrade, but some potential issues were detected. Please ensure that the reported issues are not significant before upgrading.

Notice the utility found one warning. In this case, it was the use of the authentication plugin. The server tested was not using the newer, more secure plugin. Fortunately, the utility provided some hints for how we can overcome this issue.

Notice also we did not provide a path to the configuration file, so that step was skipped. It did not show as a warning because that is an optional parameter (check) that you can specify in the dictionary of options.

There are several other configurations you can use to check a server for compatibility. See the online reference manual for the utility at <https://dev.mysql.com/doc/mysql-shell/8.0/en/mysql-shell-utilities-upgrade.html> for more information about the upgrade checker utility.

Overview of Installing InnoDB Cluster

The process for installing InnoDB Cluster is straight forward and once you've done it a few times you will find it far easier than setting up MySQL Replication or MySQL Group Replication. The only step that may require a bit more work is configuring the router, but that is fortunately not a regular action (you set it up once and leave it running). The general outline of steps required is shown as follows. We will see each of these steps in action in Chapter 11.

- Choose the number of MySQL instances (for fault tolerance)
- Choose the port numbers for the MySQL instances and router

- Create the MySQL instances and configure them
- Create the cluster
- Add MySQL instances to the cluster
- Check status of the cluster
- Configure (bootstrap) the router

While configuration of the router and using it in your applications is not something related to MySQL Shell, a demonstration of InnoDB Cluster without it would be an error as the router provides application level failover, which may be critical for some use cases where high availability requires a high level of reliability.

Summary

We now have a complete high availability story. While MySQL InnoDB Cluster provides high availability for our data, it does not help us (directly) for achieving high availability at the application level. Yes, you can write your applications to query the cluster and get information that can help your applications “heal” should a server go offline, but practice shows this is a very brittle solution that relies far too much on known parameters. Should anything change in the cluster configuration, the application(s) can fail or require reworking to get restarted.

That’s not ideal for most organizations. What we need is the ability to quickly and easily make our applications resilient to changes in the cluster. More specifically, the application should not stop if a server in the cluster goes offline or is taken offline or its role changes. This is where MySQL Router shines.

MySQL Router takes the burden of connection routing out of the application placing it in its own lightweight, easily configured instance. Now, applications can be built to rely on the router for all connection routing including failover events or normal high availability recovery events.

In the next chapter, we will look at deploying MySQL InnoDB Cluster on a set of machines and configure the router with a simple application to show how you can complete your high availability goals for your own applications. We will be showing MySQL Shell as the administration console for working with InnoDB Cluster and will see a quick overview of the AdminAPI.

CHAPTER 11

Example: InnoDB Cluster Setup and Administration

Now that we have learned more about what InnoDB Cluster is and the components that make up the feature, we're almost to the point where we have enough information to begin working with a small, experimental InnoDB Cluster. There are just a few more things we need to learn including how to use the AdminAPI in the shell as well as become familiar with the steps required to set up not only InnoDB Cluster but also the MySQL Router.

In this chapter, we will see a demonstration of the easier form of deployment of InnoDB Cluster – running it in a sandbox deployment method via the MySQL Shell and the AdminAPI. We will create an InnoDB Cluster with four instances running on our local machine. We will see not only how to setup the cluster for use but also how the cluster handles failover and finally how to set up MySQL Router and demonstrate how it works with an application.

But first, let's begin with an overview of the AdminAPI.

Getting Started

The key component that permits us to set up our experimental InnoDB Cluster is called a sandbox. The AdminAPI has several methods to work with MySQL servers in a sandbox on a local machine. However, the AdminAPI also have classes with methods for working with InnoDB Clusters that use MySQL servers on remote machines. In this chapter, we will see an overview of the classes and methods available in the AdminAPI. We will use some of the methods discussed in this section in a demonstration of InnoDB Cluster in the next section.

There are two major classes in the AdminAPI: `dba` and `cluster`. Let's look at the details of each of these classes.

Note The following is a condensed version of the documentation available online written to provide an overview rather than specific use examples.

dba

The `dba` class enables you to administer InnoDB clusters using the AdminAPI. The `dba` class enables you to administer the cluster such as creating a new cluster, working with a sandbox configuration (a way to experiment with InnoDB Cluster using several MySQL instances on the same machine, checking the status of instances and the cluster).

Since this class is the setup and configuration arm of the API, it has methods for working with the sandbox as well as methods for working with remote servers. Table 11-1 shows the methods available¹ for working with instances in a sandbox (those with `sandbox` in the name).

Table 11-1. *Sandbox Methods (dba class)*

Returns	Function	Description
None	<code>delete_sandbox_instance</code> (int port, dict options)	Deletes an existing MySQL Server instance on localhost
Instance	<code>deploy_sandbox_instance</code> (int port, dict options)	Creates a new MySQL Server instance on localhost
None	<code>kill_sandbox_instance</code> (int port, dict options)	Kills a running MySQL Server instance on localhost
None	<code>start_sandbox_instance</code> (int port, dict options)	Starts an existing MySQL Server instance on localhost
None	<code>stop_sandbox_instance</code> (int port, dict options)	Stops a running MySQL Server instance on localhost

¹This table and subsequent tables in this chapter show the Python names for the methods. JavaScript names differ slightly due to the camelCase standard for JavaScript.

Notice that there are methods for deploying an instance in the sandbox as well as deleting an instance or killing an instance (delete removes it, kill stops the instance but leaves the data and metadata), and starting and stopping an instance (kill issues an uncontrolled shutdown). We will use most of these methods in the demonstration of the InnoDB Cluster in a sandbox in a later section.

Notice also these methods take a port number and a dictionary of options. The options you can use for these and other methods in the AdminAPI depend on the method itself as each method permits one or more options. Table 11-2 shows the options available for the methods in Table 11-1.

Table 11-2. *Options for the Sandbox Methods (dba class)*

Function	Option	Description
delete_sandbox_instance	sandboxDir	The path where the new instance location that will be deleted
deploy_sandbox_instance	portx	The port where the new instance will listen for X Protocol connections
	sandboxDir	The path where the new instance will be deployed
	password	The password for the MySQL root user on the new instance
	allowRootFrom	Create the remote root account, restricted to the given address pattern (%)
	ignoreSslError	Ignore errors when adding SSL support for the new instance, by default: true
kill_sandbox_instance	sandboxDir	The path where the new instance is located will be deployed
start_sandbox_instance	sandboxDir	The path where the new instance will be started
stop_sandbox_instance	password	The password for the MySQL root user on the deployed instance
	sandboxDir	The path where the new instance will be stopped

The options are specified in a dictionary in the form of a simple JSON document. For example, if you wanted to stop an instance on port 13004 and specify the sandbox directory and password, you would call the method as follows.

```
> stop_sandbox_instance(13004, {'sandboxDir': '/home/cbell/data1',
'password': 'secret'})
```

Caution If you use the `sandboxDir` option to change the location of the sandbox, you must ensure to use it for every method that permits it; otherwise some of your sandbox instances may be placed in the default location.

Table 11-3 shows the remaining methods in the class used for setup and configuration of MySQL instances and clusters.

Table 11-3. Instance and Cluster Methods (*dba class*)

Returns	Function	Description
JSON	<code>check_instance_configuration</code> (InstanceDef instance, dict options)	Validates an instance for MySQL InnoDB Cluster usage
None	<code>configure_local_instance</code> (InstanceDef instance, dict options)	Validates and configures a local instance for MySQL InnoDB Cluster usage
None	<code>configure_instance</code> (InstanceDef instance, dict options)	Validates and configures an instance for MySQL InnoDB Cluster usage
Cluster	<code>create_cluster(str name, dict options)</code>	Creates a MySQL InnoDB cluster
None	<code>drop_metadata_schema(dict options)</code>	Drops the Metadata Schema
Cluster	<code>get_cluster(str name, dict options)</code>	Retrieves a cluster from the Metadata Store
None	<code>reboot_cluster_from_complete_outage</code> (str clusterName, dict options)	Brings a cluster back ONLINE when all members are OFFLINE

The options for these methods are considerably more. In fact, some methods permit a long list of options. Rather than list each of the options for each of the methods, the following list summarizes the options in three categories. We will see some of these in action during the demonstration. More specific options are required for certain methods.

Table 11-4. *Options for dba Class Methods*

Area	Option	Description
General	Common options for most methods	
	verifyMyCnf	Optional path to the MySQL configuration file for the instance. If this option is given, the configuration file will be verified for the expected option values, in addition to the global MySQL system variables.
	outputMycnfPath	Alternative output path to write the MySQL configuration file of the instance
	password	The password to be used on the connection
	clusterAdmin	The name of the InnoDB cluster administrator user to be created. The supported format is the standard MySQL account name format.
	clusterAdminPassword	The password for the InnoDB cluster administrator account
	clearReadOnly	boolean value used to confirm that super_read_only must be disabled
URI or Dictionary	interactive	boolean value used to disable the wizards in the command execution, i.e, prompts are not provided to the user and confirmation prompts are not shown
	Options for secure connections	
	ssl-mode	the SSL mode to be used in the connection
	ssl-ca	the path to the X509 certificate authority in PEM format
	ssl-capath	the path to the directory that contains the X509 certificates authorities in PEM format

(continued)

Table 11-4. (continued)

Area	Option	Description
	ssl-cert	The path to the X509 certificate in PEM format
	ssl-key	The path to the X509 key in PEM format
	ssl-crl	The path to file that contains certificate revocation lists
	ssl-crlpath	The path of directory that contains certificate revocation list files
	ssl-cipher	SSL Cipher to use
	tls-version	List of protocols permitted for secure connections
	auth-method	Authentication method
	get-server-public-key	Request public key from the server required for RSA key pair-based password exchange. Use when connecting to MySQL 8.0 servers with classic MySQL sessions with SSL mode DISABLED.
	server-public-key-path	The path name to a file containing a client-side copy of the public key required by the server for RSA key pair-based password exchange
Connection Dictionary	Connection parameters	
	scheme	the protocol to be used on the connection
	user	the MySQL user name to be used on the connection
	dbUser	alias for user
	password	the password to be used on the connection
	dbPassword	same as password
	host	the hostname or IP address to be used on a TCP connection
	port	the port to be used in a TCP connection
	socket	the socket file name to be used on a connection through unix sockets
	schema	the schema to be selected once the connection is done.

cluster

The cluster class is a handle (think object instance) to an InnoDB Cluster. The cluster class enables you to work with the cluster to add instances, remove instances, get the status (health) of the cluster, and more.

Since this class is used to work directly with instances and the cluster, most of the methods are designed to work with a specific instance of the cluster retrieved via the dba class. Table 11-5 lists the methods in the cluster class.

Table 11-5. *Methods for the Cluster Class*

Returns	Function	Description
None	<code>add_instance(InstanceDef instance, dict options)</code>	Adds an Instance to the cluster
dict	<code>check_instance_state (InstanceDef instance, str password)</code>	Verifies the instance GTID state in relation with the cluster
str	<code>describe()</code>	Describe the structure of the cluster
None	<code>disconnect()</code>	Disconnects all internal sessions used by the cluster object
None	<code>dissolve(Dictionary options)</code>	Dissolves the cluster
None	<code>force_quorum_using_partition_of (InstanceDef instance, str password)</code>	Restores the cluster from quorum loss
str	<code>get_name()</code>	Retrieves the name of the cluster
None	<code>rejoin_instance (InstanceDef instance, dict options)</code>	Rejoins an Instance to the cluster
None	<code>remove_instance(InstanceDef instance, dict options)</code>	Removes an Instance from the cluster
None	<code>rescan()</code>	Rescans the cluster
str	<code>status()</code>	Describe the status of the cluster
None	<code>switch_to_single_primary_mode (InstanceDef instance)</code>	Switches the cluster to single-primary mode
None	<code>switch_to_multi_primary_mode()</code>	Switches the cluster to multi-primary mode More...

(continued)

Table 11-5. (continued)

Returns	Function	Description
None	<code>set_primary_instance</code> (InstanceDef instance)	Elects a specific cluster member as the new primary
str	<code>options</code> (dict options)	Lists the cluster configuration options
None	<code>set_option</code> (str option, str value)	Changes the value of a configuration option for the whole cluster More...
None	<code>set_instance_option</code> (InstanceDef instance, str option, str value)	Changes the value of a configuration option in a Cluster member More...
void	<code>invalidate</code> ()	Mark the cluster as invalid (e.g., dissolved)

Notice we have methods for adding, removing, and rejoining an instance. We will use these often in managing the instances in the cluster. There are also several methods for obtaining information, status, and forcing updates to the metadata such as `get_name()`, `status()` and `rescan()`.

We also have methods for switching the mode of an instance, which can come in handy if we want to manually change the primary role to a specific instance. We also have methods for displaying and setting general options for the cluster, which once again helps with maintenance.

Notice also like the `dba` class, some of the methods accept a dictionary of options. Such options are again unique to the method, but in general use the same options described in the previous section for connecting to an instance. And as mentioned, some permit options specific to the method.

The class has one property: the name of the cluster. The property is named simply, `name`, and can be set programmatically but is normally set when the cluster is created using the `dba` class.

Tip See https://dev.mysql.com/doc/dev/mysqlsh-api-python/8.0/group__admin__api.html to learn more about the AdminAPI.

Now that we've had a brief overview of the classes and methods in the AdminAPI, let's see it in action by setting up InnoDB Cluster in a sandbox.

Setup and Configuration

To prepare for using the sandbox, you merely need to decide on a few parameters and prepare an area on your system for working with the data for the cluster. There is one parameter that is required. We must decide on what port numbers we want to use for the experimental cluster. In this case, we will use ports 3311-3314 for the server listening ports.

We can also specify a directory to contain the sandbox data. While this is not required, it is recommended if you want to reinitialize the cluster later. There is no need to specify a directory otherwise because the AdminAPI uses a predetermined path for the sandbox. For example, on Windows, it is in the user directory named `MySQL\mysql-sandboxes`. This folder forms the root for storing all data and metadata for the sandbox. For example, when you deploy an instance to the sandbox using port 3312, you will see a folder with that name as follows.

```
C:\Users\cbell\MySQL\mysql-sandboxes\3312
```

If you plan to reuse or restart the cluster, you may want to specify a specific folder using the `sandboxDir` option. For example, you can specify the dictionary as `{'sandboxDir': 'c://idc_sandbox'}` in the AdminAPI. However, the folder must exist or you will get an error when you call the `deploy_sandbox_instance()` method. Listing 11-1 shows a custom sandbox directory on Windows with a single instance deployed on port 3311.

Listing 11-1. Creating a Directory for the Sandbox

```
C:\idc_sandbox>dir
Volume in drive C is Local Disk
Volume Serial Number is AAFC-6767

Directory of C:\idc_sandbox

05/09/2019  07:18 PM    <DIR>          .
05/09/2019  07:18 PM    <DIR>          ..
05/09/2019  07:18 PM    <DIR>          3311           0
File(s)
                0 bytes
                3 Dir(s)  172,731,768,832 bytes free
```

```
C:\idc_sandbox>dir 3311
Volume in drive C is Local Disk
Volume Serial Number is AAFC-6767

Directory of C:\idc_sandbox\3311

05/09/2019  07:19 PM    <DIR>          .
05/09/2019  07:19 PM    <DIR>          ..
05/09/2019  07:19 PM                6 3311.pid
05/09/2019  07:18 PM           726 my.cnf
05/09/2019  07:18 PM    <DIR>          mysql-files
05/09/2019  07:18 PM    <DIR>          sandboxdata
05/09/2019  07:18 PM           147 start.bat
05/09/2019  07:18 PM           207 stop.bat
             4 File(s)          1,086 bytes
             4 Dir(s)  172,557,893,632 bytes free
```

Note To reuse the instance data, you must start the instance. Attempting to redeploy it using the same port will generate an error because the directory is not empty.

Now, let's see a demonstration of setting up a cluster in the sandbox. There are several steps to create a sandbox deployment of InnoDB Cluster. They are as follows.

- *Create and Deploy Instances in the Sandbox:* Setup and configure our MySQL servers.
- *Create the Cluster:* Create an object instance of the cluster class.
- *Add the Instances to the Cluster:* Add the sandbox instances to the cluster.
- *Check the Status of the Cluster:* Check the cluster health.

We will also see a demonstration of how failover works within the cluster by killing one of the instances. Let's get started!

Create and Deploy Instances in the Sandbox

Let's begin by starting the shell and deploying four servers using the AdminAPI. In this case, we will use the ports 3311-3314 and `deploy_sandbox_instance()` method in the `dba` object to create new instances for each server. All of these will run on our localhost.

Note It is not necessary to import the `dba` class. The MySQL Shell makes it available whenever you switch to the Python mode.

The sandbox is created with the first call to the `deploy` method. Let's deploy the four servers now. The `deploy` method will ask you to provide a password for the root user. It is recommended that you use the same password for all four servers.

Note You must create the folder that you specify for the sandbox.

Listing 11-2 demonstrates how to deploy four servers. The commands used are highlighted in bold to help identify the commands from the messages. Notice I start the shell in Python mode. The four commands are shown in bold for easier reference.

Listing 11-2. Creating Local Server Instances

```
$ mysqlsh --py
MySQL Shell 8.0.16
...
Type '\help' or '\?' for help; '\quit' to exit.
> sandbox_options = {'sandboxDir': '/home/cbell/idc_sandbox'}
> dba.deploy_sandbox_instance(3311, sandbox_options)
A new MySQL sandbox instance will be created on this host in
/home/cbell/idc_cluster/3311
Warning: Sandbox instances are only suitable for deploying and
running on your local machine for testing purposes and are not
accessible from external networks.
Please enter a MySQL root password for the new instance: ****
Deploying new MySQL instance...
Instance localhost:3311 successfully deployed and started.
```

Use `shell.connect('root@localhost:3311');` to connect to the instance.

> dba.deploy_sandbox_instance(3312, sandbox_options)

A new MySQL sandbox instance will be created on this host in
`/home/cbell/idc_cluster/3312`

Warning: Sandbox instances are only suitable for deploying and running on your local machine for testing purposes and are not accessible from external networks.

Please enter a MySQL root password for the new instance: ****

Deploying new MySQL instance...

Instance `localhost:3312` successfully deployed and started.

Use `shell.connect('root@localhost:3312');` to connect to the instance.

> dba.deploy_sandbox_instance(3313, sandbox_options)

A new MySQL sandbox instance will be created on this host in
`/home/cbell/idc_cluster/3313`

Warning: Sandbox instances are only suitable for deploying and running on your local machine for testing purposes and are not accessible from external networks.

Please enter a MySQL root password for the new instance: ****

Deploying new MySQL instance...

Instance `localhost:3313` successfully deployed and started.

Use `shell.connect('root@localhost:3313');` to connect to the instance.

> dba.deploy_sandbox_instance(3314, sandbox_options)

A new MySQL sandbox instance will be created on this host in
`/home/cbell/idc_cluster/3314`

Warning: Sandbox instances are only suitable for deploying and running on your local machine for testing purposes and are not accessible from external networks.

Please enter a MySQL root password for the new instance: ****

Deploying new MySQL instance...

Instance `localhost:3314` successfully deployed and started.

Use `shell.connect('root@localhost:3314');` to connect to the instance.

Notice the `deploy_sandbox_instance()` method displays the location of the sandbox data and metadata (e.g., `c:\idc_sandbox\3314`) and prompts us for a password for the instance. Be sure to use a password that you will remember if you intend to restart or reuse the cluster. It is Ok to use the same password for all instances. Once you run all the commands, you will have four instances running on the local machine.

There is one feature in the `dba` class that often gets overlooked. The `check_instance_configuration()` method allows you to check to see if an instance is properly configured for use with InnoDB Cluster. You can run it on the sandbox instance (but they'll always be compatible) or, better, you can run it on your remote instances to check them before adding them to the cluster. Listing 11-3 demonstrates running the check. This is a recommended step prior to building your cluster. Note that you must connect to the instance first.

Listing 11-3. Checking an Instance for Configuration Compatibility

> \connect root@localhost:3311

Creating a session to 'root@localhost:3311'

Please provide the password for 'root@localhost:3311': ****

Save password for 'root@localhost:3311'? [Y]es/[N]o/Ne[v]er (default No): Y

Fetching schema names for autocompletion... Press ^C to stop.

Your MySQL connection id is 12

Server version: 8.0.16 MySQL Community Server - GPL

No default schema selected; type \use <schema> to set one.

> dba.check_instance_configuration()

Validating local MySQL instance listening at port 3311 for use in an InnoDB cluster...

Instance detected as a sandbox.

Please note that sandbox instances are only suitable for deploying test clusters for use within the same host.

This instance reports its own address as OPTIPLEX-7010

Clients and other cluster members will communicate with it through this address by default. If this is not correct, the `report_host` MySQL system variable should be changed.

Checking whether existing tables comply with Group Replication requirements...

No incompatible tables detected

Checking instance configuration...

Instance configuration is compatible with InnoDB cluster

The instance 'localhost:3311' is valid for InnoDB cluster usage.

```
{
  "status": "ok"
}
```

There is also a method to automatically configure a local instance. Use the `configure_local_instance()` method to make any changes needed to get the local instance configured properly for InnoDB Cluster. This can be very handy if you are working with existing MySQL servers that have been configured for MySQL Replication.

Create the Cluster

The next thing we need to do is set up a new cluster. We do this with the `create_cluster()` method in the `dba` object, which creates an object instance to the cluster class. But first, we must connect to the server we want to make our primary server. Note that this is a continuation of our shell session and demonstrates how to create a new cluster. Notice how this is done in Listing 11-4.

Listing 11-4. Creating a Cluster in InnoDB Cluster MySQL Py >

```
\connect root@localhost:3311
```

```
Creating a session to 'root@localhost:3311'
```

```
Fetching schema names for autocompletion... Press ^C to stop.
```

```
Your MySQL connection id is 9
```

```
Server version: 8.0.16 MySQL Community Server - GPL
```

```
No default schema selected; type \use <schema> to set one.
```

```
> my_cluster = dba.create_cluster('MyClusterSB')
```

```
A new InnoDB cluster will be created on instance 'root@localhost:3311'.
```

```
Validating instance at localhost:3311...
```

```
Instance detected as a sandbox.
```

```
Please note that sandbox instances are only suitable for deploying test clusters for use within the same host.
```

This instance reports its own address as localhost

Instance configuration is suitable.

Creating InnoDB cluster 'MyClusterSB' on 'root@localhost:3311'...

Adding Seed Instance...

Cluster successfully created. Use `Cluster.add_instance()` to add MySQL instances. At least 3 instances are needed for the cluster to be able to withstand up to one server failure.

Notice we name the cluster `MyClusterSB` and use a variable named `my_cluster` to store the object returned from the `create_cluster()` method. Notice also that the server we connected to first has become the primary and that the AdminAPI has detected we are running in a sandbox.

Failure to Create Cluster

If your local machine has an instance of MySQL running that not is part of a replication topology, you may get an error complaining about the host not being usable for Group Replication or that it resolves to `127.0.0.1`, you must stop each of your sandbox instances, edit the `my.cnf` adding the `report_host` option set to `localhost` and the `report_port` option set to the port for the instance. Then start the instance as shown in the following text. Repeat for each of the deployed instances. Notice also that there is a special start and stop script in each instance folder that you can use to quickly start and stop instance. Cool, eh?

```
$ cd ~/idc_sandbox
$ ./3311/stop.sh
Stopping MySQL sandbox using mysqladmin shutdown... Root password is required.
Enter password:
$ nano ./3311/my.cnf
...
report_host='localhost'
report_port=3311
$ ./3311/start.sh
Starting MySQL sandbox
```


Note that if you use the `dba.stop_sandbox_instance()` in the shell, this completely removes the instance from the sandbox. It is better to stop the instance using the helper script (`stop.sh` or `stop.bat`). You can, however, start the instance from inside the shell by using the `start_sandbox_instance()` method passing in the port for the instance.

```
> dba.start_sandbox_instance(3314, sandbox_options)
The MySQL sandbox instance on this host in
/home/cbell/idc_sandbox/3314 will be started
Starting MySQL instance...
Instance localhost:3314 successfully started.
```

Tip If you exit the shell, you can retrieve a running cluster with the `get_cluster()` method. For example, you can restart the shell, then issue the command `my_cluster = dba.get_cluster('MyClusterSB')`.

Add the Instances to the Cluster

Next, we add the other two server instances to complete the cluster. We are now using the cluster class instance saved in the variable `my_cluster` and using the `add_instance()`. We will add the three remaining instances to the cluster. These servers automatically become secondary servers in the group. Listing 11-5 shows how to add the instances to the cluster.

Listing 11-5. Adding Instances to the Cluster

```
> my_cluster.add_instance('root@localhost:3312')
```

A new instance will be added to the InnoDB cluster. Depending on the amount of data on the cluster this might take from a few seconds to several hours.

```
Adding instance to the cluster ...
```

```
Please provide the password for 'root@localhost:3312': ****
```

```
Save password for 'root@localhost:3312'? [Y]es/[N]o/[N]e[v]er (default No): Y
```

```
Validating instance at localhost:3312...
```

```
Instance detected as a sandbox.
```

Please note that sandbox instances are only suitable for deploying test clusters for use within the same host.

This instance reports its own address as localhost

Instance configuration is suitable.

The instance 'root@localhost:3312' was successfully added to the cluster.

> my_cluster.add_instance('root@localhost:3313')

A new instance will be added to the InnoDB cluster. Depending on the amount of data on the cluster this might take from a few seconds to several hours.

Adding instance to the cluster ...

Please provide the password for 'root@localhost:3313': ****

Save password for 'root@localhost:3313'? [Y]es/[N]o/[Ne[v]er (default No): Y

Validating instance at localhost:3313...

Instance detected as a sandbox.

Please note that sandbox instances are only suitable for deploying test clusters for use within the same host.

This instance reports its own address as localhost

Instance configuration is suitable.

The instance 'root@localhost:3313' was successfully added to the cluster.

> my_cluster.add_instance('root@localhost:3314')

A new instance will be added to the InnoDB cluster. Depending on the amount of data on the cluster this might take from a few seconds to several hours.

Adding instance to the cluster ...

Please provide the password for 'root@localhost:3314': ****

Save password for 'root@localhost:3314'? [Y]es/[N]o/[Ne[v]er (default No): Y

Validating instance at localhost:3314...

Instance detected as a sandbox.

Please note that sandbox instances are only suitable for deploying test clusters for use within the same host.

This instance reports its own address as localhost

Instance configuration is suitable.

The instance 'root@localhost:3314' was successfully added to the cluster.

Notice the `add_instance()` method takes a string with the URI connection information. In this case, it is simply the username, at sign (@), hostname, and port in the form `<user>@<host>:<port>`. Notice also the method prompts for the password for the instance.

At this point, we've seen how InnoDB Cluster can setup servers and add them to the group. Take a moment and think back to the Group Replication tutorial. What you do not see behind the scenes is all the Group Replication mechanisms – you get them for free! How cool is that?

Clearly, using the shell to set up and manage a cluster is a lot easier than setting up and managing a standard Group Replication setup. Specifically, you don't have to manually configure replication! Better still, should a server fail, you don't have to worry about reconfiguring your application or the topology to ensure the solution remains viable – InnoDB Cluster does this automatically for you.

Check the Status of the Cluster

Once the cluster is created and instances are added, we can get the status of the cluster using the `status()` method of our `my_cluster` object as shown in Listing 11-6. In this example, we also see how to retrieve the cluster from a running instance of one of the servers by connecting with the `\connect` command and using the `get_cluster()` method from the `dba` class. You can also connect to the server instance using the command line (`mysqlsh --uri root@localhost:3313`). Note that you do not have to connect to the first (or primary) server instance to retrieve the cluster. You can connect to any server to retrieve the cluster.

Listing 11-6. Getting the status of the cluster

```
> \connect root@localhost:3313
Creating a session to 'root@localhost:3313'
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 30
Server version: 8.0.16 MySQL Community Server - GPL
No default schema selected; type \use <schema> to set one.

> my_cluster = dba.get_cluster('MyClusterSB')
> my_cluster.status()
{
```

```

"clusterName": "MyClusterSB",
"defaultReplicaSet": {
  "name": "default",
  "primary": "localhost:3311",
  "ssl": "REQUIRED",
  "status": "OK",
  "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
  "topology": {
    "localhost:3311": {
      "address": "localhost:3311",
      "mode": "R/W",
      "readReplicas": {},
      "role": "HA",
      "status": "ONLINE",
      "version": "8.0.16"
    },
    "localhost:3312": {
      "address": "localhost:3312",
      "mode": "R/O",
      "readReplicas": {},
      "role": "HA",
      "status": "ONLINE",
      "version": "8.0.16"
    },
    "localhost:3313": {
      "address": "localhost:3313",
      "mode": "R/O",
      "readReplicas": {},
      "role": "HA",
      "status": "ONLINE",
      "version": "8.0.16"
    },
    "localhost:3314": {
      "address": "localhost:3314",
      "mode": "R/O",

```

```

        "readReplicas": {},
        "role": "HA",
        "status": "ONLINE",
        "version": "8.0.16"
    }
},
"topologyMode": "Single-Primary"
},
"groupInformationSourceMember": "localhost:3311"
}

```

Notice the output is in the form of a JSON document and contains the metadata about the cluster including all the instances, their roles, and status. You want to ensure all instances are online.

Now, let's see a demonstration of how the cluster can handle failover automatically.

Failover Demonstration

Now, let's a simulated failure scenario. In this case, we will purposefully kill one of the instances. Let's kill the one running on port 3311. We can do this in a variety of ways such as using the operating system to terminate the `mysqld` process, use the shutdown SQL command from the shell or MySQL client or the `dba` class. Listing 11-7 shows how to kill the instance and the results of the status after the instance stops.

Listing 11-7. Failover Demonstration

```

> sandbox_options = {'sandboxDir': '/home/cbell/idc_sandbox'}
MySQL Py > \connect root@localhost:3311
Creating a session to 'root@localhost:3311'
Fetching schema names for autocompletion... Press ^C to stop.
Your MySQL connection id is 55
Server version: 8.0.16 MySQL Community Server - GPL
No default schema selected; type \use <schema> to set one.
> dba.kill_sandbox_instance(3311, sandbox_options)
The MySQL sandbox instance on this host in
/home/cbell/idc_sandbox/3311 will be killed

```

Killing MySQL instance...

Instance localhost:3311 successfully killed.

> \connect root@localhost:3312

Creating a session to 'root@localhost:3312'

Fetching schema names for autocompletion... Press ^C to stop.

Your MySQL connection id is 38

Server version: 8.0.16 MySQL Community Server - GPL

No default schema selected; type \use <schema> to set one.

> my_cluster = dba.get_cluster('MyClusterSB')

> my_cluster.status()

```
{
  "clusterName": "MyClusterSB",
  "defaultReplicaSet": {
    "name": "default",
    "primary": "localhost:3313",
    "ssl": "REQUIRED",
    "status": "OK_PARTIAL",
    "statusText": "Cluster is ONLINE and can tolerate up to ONE
failure. 1 member is not active",
    "topology": {
      "localhost:3311": {
        "address": "localhost:3311",
        "mode": "n/a",
        "readReplicas": {},
        "role": "HA",
        "status": "(MISSING)"
      },
      "localhost:3312": {
        "address": "localhost:3312",
        "mode": "R/O",
        "readReplicas": {},
        "role": "HA",
        "status": "ONLINE",
        "version": "8.0.16"
      }
    }
  }
}
```

```

    },
    "localhost:3313": {
        "address": "localhost:3313",
        "mode": "R/W",
        "readReplicas": {},
        "role": "HA",
        "status": "ONLINE",
        "version": "8.0.16"
    },
    "localhost:3314": {
        "address": "localhost:3314",
        "mode": "R/O",
        "readReplicas": {},
        "role": "HA",
        "status": "ONLINE",
        "version": "8.0.16"
    }
},
"topologyMode": "Single-Primary"
},
"groupInformationSourceMember": "localhost:3313"
}

```

Notice we killed the server running on port 3311. However, when we went to get the cluster, we got an error. This is because we were already connected to that server we killed (3311). Thus, we need to connect to another server and retrieve the cluster again to refresh the data. Then, we can get the status and when we do, we see the server instance on port 3311 is listed as missing and the server on port 3312 has taken over the read-write capability.

At this point, we can try to recover the server instance on port 3311 or remove it from the cluster. Listing 11-8 demonstrates how to remove it from the cluster. Notice we use the `recover` method to remove the instance since we cannot connect to it (it is down).

Listing 11-8. Removing Downed Instance from Cluster

```
> my_cluster.rescan()
```

```
Rescanning the cluster...
```

```
Result of the rescanning operation for the 'default' ReplicaSet:
```

```
{
  "name": "default",
  "newTopologyMode": null,
  "newlyDiscoveredInstances": [],
  "unavailableInstances": [
    {
      "host": "localhost:3311",
      "label": "localhost:3311",
      "member_id": "27e8019b-8315-11e9-9f3e-5882a8945ac2"
    }
  ]
}
```

The instance 'localhost:3311' is no longer part of the ReplicaSet.

The instance is either offline or left the HA group. You can try to add it to the cluster again with the `cluster.rejoinInstance('localhost:3311')` command or you can remove it from the cluster configuration.

Would you like to remove it from the cluster metadata? [Y/n]: y

```
Removing instance from the cluster metadata...
```

The instance 'localhost:3311' was successfully removed from the cluster.

```
> my_cluster.status()
```

```
{
  "clusterName": "MyClusterSB",
  "defaultReplicaSet": {
    "name": "default",
    "primary": "localhost:3313",
    "ssl": "REQUIRED",
    "status": "OK",
    "statusText": "Cluster is ONLINE and can tolerate up to ONE failure.",
    "topology": {
```



```

    "localhost:3312": {
        "address": "localhost:3312",
        "mode": "R/O",
        "readReplicas": {},
        "role": "HA",
        "status": "ONLINE",
        "version": "8.0.16"
    },
    "localhost:3313": {
        "address": "localhost:3313",
        "mode": "R/W",
        "readReplicas": {},
        "role": "HA",
        "status": "ONLINE",
        "version": "8.0.16"
    },
    "localhost:3314": {
        "address": "localhost:3314",
        "mode": "R/O",
        "readReplicas": {},
        "role": "HA",
        "status": "ONLINE",
        "version": "8.0.16"
    }
},
"topologyMode": "Single-Primary"
},
"groupInformationSourceMember": "localhost:3313"
}

```

To add the missing instance back, simply restart it by running the start command located in the sandbox data directory (e.g., `idc_sandbox\3311\start.bat`) and add it back to the cluster using the `rescan()` method as follows. Use this method to add an instance that was previously part of the cluster. If you are replacing the instance altogether with a new one, you would use the `add_instance()` method instead.

> my_cluster.rescan()

Rescanning the cluster...

Result of the rescanning operation for the 'default' ReplicaSet:

```
{
  "name": "default",
  "newTopologyMode": null,
  "newlyDiscoveredInstances": [
    {
      "host": "localhost:3311",
      "member_id": "1e8f53e2-7335-11e9-9bb1-4ccc6ae8a7a4",
      "name": null,
      "version": "8.0.16"
    }
  ],
  "unavailableInstances": []
}
```

A new instance 'localhost:3311' was discovered in the ReplicaSet.

Would you like to add it to the cluster metadata? [Y/n]: y

Adding instance to the cluster metadata...

The instance 'localhost:3311' was successfully added to the cluster metadata.

Now that we've seen how to create our InnoDB Cluster, let's briefly discuss how to setup MySQL Router for use with the cluster and a test application.

Using MySQL Router

As mentioned previously, any coverage of InnoDB Cluster would be incomplete without a brief look at MySQL Router. In this section, we will see a brief tutorial for setting up the router and using it with applications. We will use a simple Python script to demonstrate application level failover.

Bootstrapping the Router

Recall that configuring the router was the step in creating an InnoDB Cluster that could require some additional setup. While that is true, for the most part the router can configure itself. It is only once your cluster becomes large or you have customized the

cluster for performance (read scaling) or if there are multiple applications or multiple clusters to which you want to route.

Tip If you have not installed MySQL Router, you must do so now. Some platforms, such as Linux, may require downloading the package separately.

Fortunately, we can use the special bootstrap option to connect to your cluster and read the metadata then update the configuration file automatically. If you're using a sandbox installation, this is the quickest and surest way to setup the router. Even if you are not using a sandbox installation, you can use this method to quickly set the base configuration, which you can later change to match your needs.

Let's see how to use the bootstrap option. To use the option, we need some other parameters. We need the following. In short, must provide the connection information and a user to use for securing the configuration file. We also add an optional parameter to supply a name for the configuration, which can be helpful if you're working with different clusters or configurations.

- `--bootstrap <server_url>`: Bootstrap and configure Router for operation with a MySQL InnoDB Cluster. You can also use the shortcut `-B`.
- `--name:` (optional) Gives a symbolic name for the router instance.
- `--user <username>`: Run the router as the user have the name specified (not available on Windows). You can also use the shortcut `-u`.

In this example, we provide the connection information with the bootstrap option in the form of a URI such as `<username>:<password>@<hostname>:<port>`. We will also use the local user to make it easier to run the router with the sandbox installation, which is also running under the current user. Thus, the command we will use is the following. We use elevated privileges because the default locations of the router files are protected.

```
$ sudo mysqlrouter --bootstrap root:secret@localhost:3313 \
--name sandbox --user cbell
```

When we run this command, the router will contact the server we specified and retrieve all the metadata for the cluster creating the routes for us automatically. Listing 11-9 shows an example of the output from the command.

Listing 11-9. Configuration with the Bootstrap Option

```

$ sudo mysqlrouter --bootstrap root:secret@localhost:3313 --name sandbox
--user cbell
# Bootstrapping system MySQL Router instance...
Fetching Group Replication Members
disconnecting from mysql-server
trying to connecting to mysql-server at localhost:3313
- Checking for old Router accounts
  - No prior Router accounts found
- Creating mysql account mysql_router1_9twobjgwueud@%' for cluster management
- Storing account in keyring
- Adjusting permissions of generated files
- Creating configuration /etc/mysqlrouter/mysqlrouter.conf

# MySQL Router 'sandbox' configured for the InnoDB cluster 'MyClusterSB'

After this MySQL Router has been started with the generated configuration

    $ /etc/init.d/mysqlrouter restart
or
    $ systemctl start mysqlrouter
or
    $ mysqlrouter -c /etc/mysqlrouter/mysqlrouter.conf

the cluster 'MyClusterSB' can be reached by connecting to:

## MySQL Classic protocol

- Read/Write Connections: localhost:6446
- Read/Only Connections: localhost:6447

## MySQL X protocol

- Read/Write Connections: localhost:64460
- Read/Only Connections: localhost:64470

Existing configuration backed up to '/etc/mysqlrouter/mysqlrouter.conf.bak'

```

Notice we see the router has identified the read-write (RW) and read-only (RO) connections for us using the default ports of 6446 and 6447, respectively. We also see the bootstrap step creates routes for using the X Protocol on ports 64460 and 64470, respectively. Before we test the router, let's learn more about what the bootstrap method has done for us. Specifically, we will look at the modified configuration file.

Now, we can start the router.

Starting the Router

We can start the router with the following command. This launches the router, which will read the configuration file. Notice we're not using elevated privileges. This is because we provided a user option during the bootstrap step that permits the user to read the file. This can be important for securing your installation, which we will explore in a later chapter.

```
$ mysqlrouter &
Loading all plugins.
  plugin 'logger:' loading
  plugin 'metadata_cache:MyClusterSB' loading
  plugin 'routing:MyClusterSB_default_ro' loading
  plugin 'routing:MyClusterSB_default_rw' loading
  plugin 'routing:MyClusterSB_default_x_ro' loading
  plugin 'routing:MyClusterSB_default_x_rw' loading
Initializing all plugins.
  plugin 'logger' initializing
logging facility initialized, switching logging to loggers specified in
configuration
```

Now that we have the router configured, let's test it with the sample Python connection script.

Sample Application

Now that we have the router installed, configured for our InnoDB Cluster, and running, let's see how we can test the router using a simple Python script.

Listing 11-10 shows a simple Python script for connecting to InnoDB Cluster via the router. Recall that we installed the router on our machine and thus this script (for conformity if not practice) should be executed on the same machine. Take a moment to examine the code. If you are following along, you can save the file as `router_connection_test.py` on your machine.

Listing 11-10. Router Connection Test

```
#
# Introducing MySQL Shell
#
# This example shows how to use the MySQL Router to connect to
# the cluster. Notice how connecting via the router port 6446
# results in a seamless transport to one of the cluster servers,
# in this case, the server with the primary role.
#
# Dr. Charles Bell, 2019
#
import mysql.connector

# Simple function to display results from a cursor
def show_results(cur_obj):
    for row in cur:
        print(row)

my_cfg = {
    'user': 'root',
    'passwd': 'secret',
    'host': '127.0.0.1',
    'port': 6446 # <<<< Router port (R/W)
}

# Connecting to the server
conn = mysql.connector.connect(**my_cfg)

print("Listing the databases on the server.")
query = "SHOW DATABASES"
cur = conn.cursor()
```

```

cur.execute(query)
show_results(cur)

print("\Retrieve the port for the server to which we're connecting.")
query = "SELECT @@port"
cur = conn.cursor()
cur.execute(query)
show_results(cur)

# Close the cursor and connection
cur.close()
conn.close()

```

The first part of the code simply imports the connector and defines a dictionary of connection terms. In this case, the user, password, host, and port for the router. We are using the port number 6446 as shown during the configuration of the router.

Next, the code opens a connection, then runs two queries: one to get a list of the databases and display them (using a function defined as `show_results()`) and another to select the current port of the server. This second query result may surprise you as we will see.

To execute the code, save this file named `router_connect_test.py` (the extension identifies it as a Python script). Then, run the code using the following command.

```

$ python ./router_connection_test.py
Listing the databases on the server.
(u'information_schema',)
(u'mysql',)
(u'mysql_innodb_cluster_metadata',)
(u'performance_schema',)
(u'sys',)
Retrieve the port for the server to which we're connecting.
(3313,)

```

Wait! Why did the output show port 3313? Shouldn't it show port 6446? After all, that's the port we used in the code. Recall, the router simply routes communication to the appropriate server, it is not a server connection itself. Thus, the router successfully routed our connection to the machine on port 331. Recall that this machine is the primary (listed as read-write in the cluster).

So how do we connect to the read-only servers in the cluster? All we need to now is to modify the program to connect to the read-only servers (on port 6447). When we rerun the script, we will see the following output.

```
$ python ./router_connection_test.py
Listing the databases on the server.
(u'information_schema',)
(u'mysql',)
(u'mysql_innodb_cluster_metadata',)
(u'performance_schema',)
(u'sys',)
Retrieve the port for the server to which we're connecting.
(3312,)
```

Now we see we're connecting to a server other than one on port 3311. Recall from the sandbox setup, the machines on ports 3312, 3313, and 3314 are all read-only.

While this example is quite primitive, it does illustrate how the router redirects connections to other MySQL servers. It also helps to reinforce the concept that we must connect our applications to the router itself rather than machines in the cluster and allow the router to do all the heavy connection routing for us. As you can see, it is quite sophisticated and knows from its initial (and later cached) configuration which servers are requested based on the port listened by the router. In this case, we use 6446 for read-write connections and 6447 for read-only connections. Yes, it is that easy. No more elaborate hard-coded ports!

Application Failover Demonstration

Now, let's try failover only this time at the application layer. More specifically, we will kill the read-write (or primary) server in the cluster then start the script again. Recall, we want to connect to the read-write port that the router is using or in this case port 6446. Be sure to check the `my_cfg` dictionary in the script before running this test.

Take a moment to verify which instance in your cluster has the read-write role (primary). We can do this with the shell as follows. Here we see the server on port 3313 has the read-write mode.


```
$ mysqlsh --uri root@localhost:3312 --py
> dba.get_cluster('MyClusterSB').status()
...
    "localhost:3313": {
        "address": "localhost:3313",
        "mode": "R/W",
        "readReplicas": {},
        "role": "HA",
        "status": "ONLINE",
        "version": "8.0.16"
    },
```

Let's first connect to the cluster without killing any instances and see what port is returned for the read-write connection.

```
$ python ./router_connection_test.py
Listing the databases on the server.
(u'information_schema',)
(u'mysql',)
(u'mysql_innodb_cluster_metadata',)
(u'performance_schema',)
(u'sys',)
```

Retrieve the port for the server to which we're connecting.
(3313,)

Ok, so it mapped to port 3313. That's good. Now, let's unceremoniously kill that server instance.

```
more ~/idc_sandbox/3313/3313.pid
10264
$ sudo kill -9 10264
```

Then, run the script again and see what port is shown for the read-write server.

```
$ python ./router_connection_test.py
Listing the databases on the server.
(u'information_schema',)
(u'mysql',)
```

```
(u'mysql_innodb_cluster_metadata',)
(u'performance_schema',)
(u'sys',)
```

Retrieve the port for the server to which we're connecting.

```
(3311,)
```

Here we see we are indeed connecting to our InnoDB Cluster via the router and we see the port reported is 3311, which is the port of the new primary (read-write) server. Cool!

This demonstration shows how we can continue to run our applications even if there is a failover in the cluster. In this case, we were simply rerunning the application (script), but in production (or development), you would build your application simply to retry the connection for either the read-write port or the read-only port. That way, your application doesn't even need to restart – you just reconnect and keep going. How cool is that?

Tip See <https://dev.mysql.com/doc/mysql-router/8.0/en/mysql-router-configuration.html> to learn more about configuring the router for your application and environment.

Before we close out the discussion on InnoDB Cluster, let's briefly talk about some administrative tasks you may want to perform.

Administration

As you have discovered, setting up InnoDB Cluster is not difficult and except for learning how to use the MySQL Shell and a few of the classes and methods in the AdminAPI, the steps for configuring InnoDB Cluster are also equally easy. However, we all know from experience that setup and follow-in administration don't always compare in complexity.

In this section, we will take a high-level look at the administrative tasks you may need to perform on InnoDB Cluster. We will also see some specific tasks that you may want to perform on your InnoDB Cluster running in a sandbox, which may help you transition from a test environment to development and later production.

Common Tasks

The tool of choice for working with InnoDB Cluster is the MySQL Shell, which has been featured prominently in this book. It is especially helpful that you can write your own special scripts in either Java or Python to work with a cluster. The common administrative tasks for working with InnoDB Cluster are listed as follows. There are several others but these are the most common.

- *Getting the Cluster:* When working with the MySQL Shell to administer InnoDB Cluster, we must first request an instance of the Cluster class. Recall, we have seen this done several times throughout this book. To retrieve the Cluster instance, we use the `dba.get_cluster()` method.
- *Checking the Cluster Status:* Like the last task, we have seen how to retrieve the cluster status report. Recall, we use the `cluster.status()` method. We must connect to one of the servers in the cluster, retrieve it, then use the status method.
- *Describing the Cluster:* You can also get information about the cluster such as the hostnames of the machines in the cluster along with the ports used by each MySQL instance. The command we use is the `cluster.describe()` method.
- *Check an Instance for Suitability for Use with InnoDB Cluster:* We have seen the two methods you can use to prepare an instance for use with InnoDB Cluster. The first, `dba.configure_local_instance()`, is used to prepare the local machine for use in the cluster. The second, `dba.check_instance_configuration()` can be used to test the server for proper settings. Unlike the first method, the check instance configuration method can be run remotely.
- *Check and Instance for Cluster Status:* You can also check and instance for its current or last known state using the `dba.check_instance_state()` method. This method takes a server connection information as a parameter and returns its state.

- *Join an Instance to the Cluster:* We have seen how to join an instance to the cluster several times during this book. Recall, we use the `cluster.add_instance()` method passing in the connection information to connect to an instance to join the current cluster.
- *Remove an Instance from the Cluster:* If you need to perform maintenance on a physical machine or the MySQL instance running on the server, you should first remove it from the cluster. We can do this with the `cluster.remove_instance()` method.

Now, let's look at a couple of tasks that may be handy for exploring InnoDB Cluster and working with a sandbox deployment.

Example Tasks

The following are some specific tasks you may want to perform on your InnoDB Cluster such as shutting down and restarting the cluster or restarting the cluster.

Ordinarily, an InnoDB Cluster (or any high availability system) would never be completely shut down. In fact, it is the goal to keep the system running always. However, for our development cluster running sandbox, it is likely we will not want to allow the instances to run for an extended period. Not only would we want to shut down the cluster, but we also may want to restart it later. This section explains one method to safely shutdown the cluster and restart it.

Shutting Down the Cluster

Simply put, the InnoDB Cluster is not designed to be turned on and off at will. Rather, shutting down all the servers will cause the cluster to be in a complete loss of the cluster continuity. While this would be very bad for a production system, for our development cluster (or any one similar), it is not a grave concern. If you ever find yourself wanting to keep the cluster around because you've got data in it you don't want to lose or applications that rely on it, you are beyond the point of using a sandbox or similar small experimental installation.

So what do you do? The AdminAPI contains methods in the dba module to recover a cluster from complete loss. This will only work, however, if you perform a controlled shutdown of the servers in the cluster. The following outlines a process that you can use to power down your cluster.

- Get the cluster status and note the read-write server
- Connect to each of the read-only servers and shut them down
- Shut down the read-write server

Recall, we can connect to any machine in the cluster, fetch the cluster, and use the `status()` method to find the read-write server. Connecting to the read-only servers to shut them down should be done via the MySQL Shell or MySQL client issuing the `shutdown SQL` command as the following.

```
$ mysqlsh --uri root@localhost:3311 --sql -e "SHUTDOWN"
```

Repeat this command for the other read-only servers and then for the read-write server. Make note of which server is the read-write server.

Restarting the Cluster

While you may expect the cluster to reestablish itself simply by restarting all the servers in the cluster, but it doesn't work that way. When restarting the cluster from scratch, we must use a special method in the AdminAPI. This method works for clusters that have not suffered any errors – those that have been shut down successfully. This is known as recovering the cluster from total outage. However, this only works if all servers have been rebooted, MySQL has started on all of them, and they can access the network (and each other).

The following demonstrates how recover the cluster from total outage. Specifically, the servers have all been restarted (`mysqld` restarted) and you need to restart the cluster from the last known good position. We will use the `dba.reboot_cluster_from_complete_outage()` method to reboot the cluster. First, log into the read-write server as noted when you shut down the servers and run the command as shown in Listing 11-11.

Listing 11-11. Restarting a Cluster from Complete Outage**\$./3311/start.sh**

Starting MySQL sandbox

\$./3312/start.sh

Starting MySQL sandbox

\$./3313/start.sh

Starting MySQL sandbox

\$./3314/start.sh

Starting MySQL sandbox

\$ mysqlsh --py --uri root@localhost:3313

MySQL Shell 8.0.16

Copyright (c) 2016, 2019, Oracle and/or its affiliates. All rights reserved.
 Oracle is a registered trademark of Oracle Corporation and/or its affiliates.
 Other names may be trademarks of their respective owners.

Type '\help' or '\?' for help; '\quit' to exit.

Creating a session to 'root@localhost:3313'

Fetching schema names for autocompletion... Press ^C to stop.

Your MySQL connection id is 57

Server version: 8.0.16 MySQL Community Server - GPL

No default schema selected; type \use <schema> to set one.

> my_cluster = dba.reboot_cluster_from_complete_outage('MyClusterSB')

Reconfiguring the cluster 'MyClusterSB' from complete outage...

The instance 'localhost:3312' was part of the cluster configuration.

Would you like to rejoin it to the cluster? [y/N]: y

The instance 'localhost:3314' was part of the cluster configuration.

Would you like to rejoin it to the cluster? [y/N]: y

The instance 'localhost:3311' was part of the cluster configuration.

Would you like to rejoin it to the cluster? [y/N]: y

The cluster was successfully rebooted.

Notice the command reads the cluster metadata and attempts to reconnect (rejoin) all the servers. If this works, you will see messages indicating the cluster was rebooted. If you encounter errors, make sure all the servers are running and can access the network, correct any issues, and retry the command.

Tip For more information about InnoDB Cluster and a more in-depth coverage of how to work with InnoDB Cluster including how to adapt your applications, see my book *Introducing InnoDB Cluster*, Bell (Apress 2018).

Summary

What may have sounded like hype that the MySQL Shell is a game changer for MySQL should, by now, start to look more like the truth. We explored how to use the shell in building applications either through its SQL interface or the NoSQL interface. In doing so, we explored the X DevAPI by using a sample application. In this chapter, we looked at the AdminAPI for use in creating an InnoDB Cluster in a sandbox.

Now that you have seen how to setup MySQL Replication and later Group Replication, you should now have an appreciation for the huge step forward in user friendliness that the MySQL Shell and AdminAPI provide. In short, InnoDB Cluster makes working with Group Replication a simple matter of learning a few API classes and methods. The best part is it opens the door for DevOps and automation of the InnoDB Cluster – something up until now required expensive, custom tools. Yes, InnoDB Cluster is easier to manage and easier to automate with MySQL Shell.

This concludes our tour of MySQL Shell. By now, you should be itching at the chance to start using it for all your MySQL needs, from simple SQL commands to administration of InnoDB Cluster. The shell does it all.

CHAPTER 12

Appendix

The example applications in this book are written as web applications using Python and the Flask framework. If you want to implement the example applications, you will need a few things installed on your computer to get going.

This appendix will help you prepare your computer with the tools needed; what you need to install and how to configure your environment. We will also see a short primer on the Flask framework as well as a walkthrough to get the sample application running. Let's begin with a look at how to set up our computers to run the example applications.

Setup Your Environment

The changes to your environment are not difficult nor are they lengthy. We will be installing Flask and a few extensions, which are needed for the application user interface. These web libraries make developing web applications with Python much easier than using raw HTML code and writing your own handlers and code for the requests. Plus, Flask is not difficult to learn. The libraries we need to install are shown in Table A-1. The table lists the name of the library/extension, a short description, and the URL for the product documentation.

Table A-1. *List of Libraries Required*

Library	Description	Documentation
Flask	Python Web API	http://flask.pocoo.org/docs/0.12/installation/
Flask-Script	Scripting support for Flask	https://flask-script.readthedocs.io/en/latest/
Flask-Bootstrap	User interface improvements and enhancements	https://pythonhosted.org/Flask-Bootstrap/
Flask-WTF	WTForms integration	https://flask-wtf.readthedocs.io/en/latest/
WTForms	Forms validation and rendering	https://wtforms.readthedocs.io/en/latest/

Note Depending on how your system is configured, you may see additional or fewer components installed for the components installed in this section.

Of course, you should already have Python installed on your system. If you do not, be sure to download and install the latest version of either the 2.X or 3.X edition. The example code in this chapter was tested with Python 2.7.10 and Python 3.6.0.

To install the libraries, we can use the Python package manager, pip, to install the libraries from the command line. The pip utility is included in most Python distributions, but if you need to install it, you can see the installation documentation at <https://pip.pypa.io/en/latest/installing/>.

If you need to install pip on Windows, you will need to download an installer, get-pip.py (<https://pip.pypa.io/en/stable/installing/#installing-with-get-pip-py>), then add the path to the installed directory to the *PATH* environment variable. There are several articles that document this process in more detail. You can Google for “installing pip on Windows 10” and find several including <https://matthewhorne.me/how-to-install-python-and-pip-on-windows-10/>, which is among the most accurate.

Note If you have multiple versions of Python installed on your system, the pip command will install into whichever Python version environment is the default. To use pip to install to a specific version, use `pipN` where N is the version. For example, `pip3` installs packages in the Python 3 environment.

The pip command is very handy because it makes installing registered Python packages – those packages registered in the Python Package Index, abbreviated as PyPI (<https://pypi.python.org/pypi>) – very easy. The pip command will download, unpack, and install using a single command. Let’s discover how to install each of the packages we need.

Tip Some systems may require running pip with elevated privileges such as sudo (Linux, macOS), or in a command window run as an administrator user (Windows 10). You will know if you need elevated privileges if the install fails to copy files due to permission issues.

Installing Flask

Listing A-1 demonstrates how to install Flask using the command, `pip install flask`. Notice the command downloads the necessary components, extracts them, then runs the setup for each. In this case, we see Flask is composed of several components including Werkzeug, MarkupSafe, and Jinja2. We will learn more about some of these in the Flask Primer section.

Listing A-1. Installing Flask

```
$ pip3 install flask
Collecting flask
  Using cached Flask-0.12.2-py2.py3-none-any.whl
Collecting Werkzeug>=0.7 (from flask)
  Downloading Werkzeug-0.14.1-py2.py3-none-any.whl (322kB)
  100% |████████████████████████████████████████████████████████████████████████████████| 327kB 442kB/s
Collecting Jinja2>=2.4 (from flask)
```

```

Using cached Jinja2-2.10-py2.py3-none-any.whl
Collecting itsdangerous>=0.21 (from flask)
  Using cached itsdangerous-0.24.tar.gz
Collecting click>=2.0 (from flask)
  Downloading click-6.7-py2.py3-none-any.whl (71kB)
    100% |████████████████████████████████████████████████████████████████████████████████| 71kB 9.4MB/s
Collecting MarkupSafe>=0.23 (from Jinja2>=2.4->flask)
  Using cached MarkupSafe-1.0.tar.gz
Installing collected packages: Werkzeug, MarkupSafe, Jinja2, itsdangerous,
click, flask
  Running setup.py install for MarkupSafe ... done
  Running setup.py install for itsdangerous ... done
Successfully installed Jinja2-2.10 MarkupSafe-1.0 Werkzeug-0.14.1 click-6.7
flask-0.12.2 itsdangerous-0.24

```

Installing Flask-Script

Listing A-2 demonstrates how to install Flask-Script using the command, `pip install flask-script`. Notice in this case, we see the installation checking for prerequisites and their versions.

Listing A-2. Installing Flask-Script

```

$ pip3 install flask-script
Collecting flask-script
  Using cached Flask-Script-2.0.6.tar.gz
Requirement already satisfied: Flask in /Library/Frameworks/Python.
framework/Versions/3.6/lib/python3.6/site-packages (from flask-script)
Requirement already satisfied: click>=2.0 in /Library/Frameworks/Python.
framework/Versions/3.6/lib/python3.6/site-packages (from Flask->flask-script)
Requirement already satisfied: Jinja2>=2.4 in /Library/Frameworks/Python.
framework/Versions/3.6/lib/python3.6/site-packages (from Flask->flask-script)
Requirement already satisfied: Werkzeug>=0.7 in /Library/Frameworks/Python.
framework/Versions/3.6/lib/python3.6/site-packages (from Flask->flask-script)

```

```

Requirement already satisfied: itsdangerous>=0.21 in /Library/Frameworks/
Python.framework/Versions/3.6/lib/python3.6/site-packages (from Flask-
>flask-script)
Requirement already satisfied: MarkupSafe>=0.23 in /Library/Frameworks/
Python.framework/Versions/3.6/lib/python3.6/site-packages (from
Jinja2>=2.4->Flask->flask-script)
Installing collected packages: flask-script
  Running setup.py install for flask-script ... done
Successfully installed flask-script-2.0.6

```

Installing Flask-Bootstrap

Listing A-3 demonstrates how to install Flask-Bootstrap using the command, `pip install flask-bootstrap`. Once again, we see the installation checking for prerequisites and their versions as well as installation of dependent components.

Listing A-3. Installing Flask-Bootstrap

```

$ pip3 install flask-bootstrap
Collecting flask-bootstrap
  Downloading Flask-Bootstrap-3.3.7.1.tar.gz (456kB)
    100% |████████████████████████████████████████████████████████████████████████████████| 460kB 267kB/s
Requirement already satisfied: Flask>=0.8 in /Library/Frameworks/Python.
framework/Versions/3.6/lib/python3.6/site-packages (from flask-bootstrap)
Collecting dominate (from flask-bootstrap)
  Downloading dominate-2.3.1.tar.gz
Collecting visitor (from flask-bootstrap)
  Downloading visitor-0.1.3.tar.gz
Requirement already satisfied: click>=2.0 in /Library/Frameworks/Python.
framework/Versions/3.6/lib/python3.6/site-packages (from Flask>=0.8->flask-
bootstrap)
Requirement already satisfied: Jinja2>=2.4 in /Library/Frameworks/Python.
framework/Versions/3.6/lib/python3.6/site-packages (from Flask>=0.8->flask-
bootstrap)

```

```
Requirement already satisfied: Werkzeug>=0.7 in /Library/Frameworks/Python.  
framework/Versions/3.6/lib/python3.6/site-packages (from Flask>=0.8->flask-  
bootstrap)
```

```
Requirement already satisfied: itsdangerous>=0.21 in /Library/Frameworks/  
Python.framework/Versions/3.6/lib/python3.6/site-packages (from Flask>=0.8-  
>flask-bootstrap)
```

```
Requirement already satisfied: MarkupSafe>=0.23 in /Library/Frameworks/  
Python.framework/Versions/3.6/lib/python3.6/site-packages  
(from Jinja2>=2.4->Flask>=0.8->flask-bootstrap)
```

```
Installing collected packages: dominate, visitor, flask-bootstrap
```

```
  Running setup.py install for dominate ... done
```

```
  Running setup.py install for visitor ... done
```

```
  Running setup.py install for flask-bootstrap ... done
```

```
Successfully installed dominate-2.3.1 flask-bootstrap-3.3.7.1 visitor-0.1.3
```

Installing Flask-WTF

Listing A-4 demonstrates how to install Flask-WTF using the command, `pip install flask-wtf`.

Listing A-4. Installing Flask-WTF

```
$ pip3 install flask-wtf
```

```
Collecting flask-wtf
```

```
  Downloading Flask_WTF-0.14.2-py2.py3-none-any.whl
```

```
Requirement already satisfied: WTForms in /Library/Frameworks/Python.  
framework/Versions/3.6/lib/python3.6/site-packages (from flask-wtf)
```

```
Requirement already satisfied: Flask in /Library/Frameworks/Python.  
framework/Versions/3.6/lib/python3.6/site-packages (from flask-wtf)
```

```
Requirement already satisfied: Jinja2>=2.4 in /Library/Frameworks/Python.  
framework/Versions/3.6/lib/python3.6/site-packages (from Flask->flask-wtf)
```

```
Requirement already satisfied: click>=2.0 in /Library/Frameworks/Python.  
framework/Versions/3.6/lib/python3.6/site-packages (from Flask->flask-wtf)
```

```
Requirement already satisfied: Werkzeug>=0.7 in /Library/Frameworks/Python.  
framework/Versions/3.6/lib/python3.6/site-packages (from Flask->flask-wtf)
```

```
Requirement already satisfied: itsdangerous>=0.21 in /Library/Frameworks/
Python.framework/Versions/3.6/lib/python3.6/site-packages (from Flask-
>flask-wtf)
Requirement already satisfied: MarkupSafe>=0.23 in /Library/Frameworks/
Python.framework/Versions/3.6/lib/python3.6/site-packages (from
Jinja2>=2.4->Flask->flask-wtf)
Installing collected packages: flask-wtf
Successfully installed flask-wtf-0.14.2
```

Installing WTForms

The following demonstrates how to install WTForms using the command, `pip install wtforms`. In this case, the installation is simple since we only need the one package.

```
$ pip3 install wtforms
Collecting wtforms
  Using cached WTForms-2.1.zip
Installing collected packages: wtforms
  Running setup.py install for wtforms ... done
Successfully installed wtforms-2.1
```

Installing Connector/Python

You should also have the MySQL Connector/Python 8.0.16 or later database connector installed. If you do not, download it from <https://dev.mysql.com/downloads/connector/python/> and install it. If you have multiple versions of Python installed, be sure to install it in all Python environments you want to use. Otherwise, you may see an error like the following when starting the code.

```
$ python3 ./mygarage_v1.py runserver -p 5001
Traceback (most recent call last):
  File "./mygarage_v1.py", line 18, in <module>
    from database.mygarage import Databases
  File "../Ch06/database/mygarage.py", line 15, in <module>
    import mysql.connector
ModuleNotFoundError: No module named 'mysql'
```

Pip can also be used to install MySQL Connector/Python. The following shows how to use PIP to install the connector.

```
$ pip3 install mysql-connector-python
Collecting mysql-connector-python
  Downloading mysql_connector_python-8.0.16-cp36-cp36m-macosx_10_12_x86_64.whl (3.2MB)
    100% | ████████████████████████████████████████████████████████████████████████████████ | 3.2MB 16.9MB/s
Installing collected packages: mysql-connector-python
Successfully installed mysql-connector-python-8.0.16
```

Now that our computer is setup, let's take a crash course on Flask and its associated extensions. The following will not teach you every nuance of Flask; rather, the goal is to get you familiar with the layout of the application and how the pieces fit together.

Flask Primer

Flask is one of several web application libraries (sometimes called frameworks or application programming interfaces) for use with Python. Flask is unique among the choices in that it is small and, once you are familiar with how it works, easy to use. That is, once you write the initialization code, most of your work with Flask will be limited to creating web pages, redirecting responses, and writing your feature code.

Flask is considered a micro-framework because it is small and lightweight, and it doesn't force you into a box writing code specifically to interact with the framework. It provides everything you need and nothing you don't, leaving the choice of what to use in your code up to you.

Flask is made up of two major components providing the basic functionality: a Web Server Gateway Interface (WSGI) that handles all the work-hosting web pages and a template library for easier web page development that reduces the need to learn HTML, removes repetitive constructs, and provides a scripting capability for HTML code. The WSGI component is named Werkzeug, which loosely translated from German means, "work" "stuff" (<https://palletsprojects.com/p/werkzeug/>). The template component is named Jinja2 and is modeled after Django (<http://jinja.pocoo.org/docs/2.10/>). Both were developed and are maintained by the originators of Flask.

Flask is also an extensible library allowing other developers to create additions (extensions) to the basic library to add functionality. We saw how to install some of the extensions available for Flask in the previous section. We will be using the scripting, bootstrap, and WTForms extensions in sample applications.

One of the components that you may consider “missing” from Flask is the ability to interact with other services like database systems. This was a purposeful design and functionality like this can be achieved through extensions. We have already installed the extensions we need along with Connector/Python, which we need to interact with MySQL.

Flask, together with the extensions described earlier, provides all the wiring and plumbing you need to make a web application in Python. It removes almost all the burdens required to write web applications such as interpreting client response packets, routing, HTML form handling, and more. If you’ve ever written a web application in Python, you will appreciate the ability to create robust web pages without the complexity of writing HTML and style sheets.

Now, let’s get started learning Flask! If you take your time and try the sample application, your first Flask application will work on the first try. The hardest part of learning Flask is already past – installing Flask and its extensions. The rest is learning the concepts of writing applications in Flask. Before we do that, let’s learn more about the terminology in Flask as well as how to set up the base code we will use to initialize the application instance that we will be using in this chapter.

Terminology

Flask is designed to make a lot of the tedium of writing web applications easier. In Flask parlance, a web page is rendered using two parts of your code: a view, which is defined in the HTML file(s), and a route, which processes the requests from a client. Recall, we can see one of two requests: a GET request that requests loading of a web page (read from the client’s perspective), and a POST request that sends data from the client via the web page to the server (write from the client’s perspective). Both requests are handled in Flask using functions you define.

These functions then render the web page to send back to the client to satisfy the request. Flask calls the functions view functions (or views for short). The way Flask knows which method to call is using decorators that identify the URL path (called a route in Flask). You can decorate a function with one or more routes making it possible

to provide multiple ways to reach the view. The decorator used is `@app.route(<path>)`. The following shows an example of multiple routes for a view function. This shows a small excerpt of the function for brevity.

```
@app.route('/handtool', methods=['GET', 'POST'])
@app.route('/handtool/<int:handtool_id>', methods=['GET', 'POST'])
def handtool(handtool_id=None):
    """Manage handtool CRUD operations."""
    handtool_table = Handtool(mygarage)
    form = HandtoolForm()
    # Get data from the form if present
    form_handtoolid = form.handtoolid.data
    # Handtool type choices
    form.handtooltype.choices = HANDTOOL_TYPES
    vendor_list = Vendor(mygarage)
    vendors = vendor_list.read()
    vendor_list = []
    ...
    return render_template("handtool.html", form=form)
```

Notice there are multiple decorators. The first is `handtool`, which allows us to use a URL like `localhost:5000/handtool`, which causes Flask to route execution to the `handtool()` function. The second is `handtool/<handtool_id>`, which demonstrates how to use variables to pass information to the view. In this case, if the user (the application) uses the URL `localhost:5000/handtool/4842`, which Flask places the value, `4842`, in the `handtool_id` variable. In this way, we can pass information dynamically to our views.

At the end of the function, we return with a call to the `render_template()` function (imported from the flask module) which tells flask to return (refresh) the web page with data we've acquired or assigned. The web page, `handtool.html`, while part of the view is called a form in Flask. It is this concept that we will use to retrieve information from the database and send it to the user. We can return a simple HTML string (or an entire file) or what is called a form. Since we are using the Flask-WTF and WTForms extensions, we can return a template rendered as a form class. We will discuss forms, form classes, and other routes and views in a later section. As you will see, templates are another powerful feature making it easy to create web pages.

Flask builds a list of all the routes in the application making it easy for the application to route execution to the correct function when requested. But what happens when a route is requested but doesn't exist in the application? By default, you will get a generic error message like "Not Found. The requested URL was not found on the server." We will see how to add our own custom error handling routes in a later section.

Now that we know more about the terminology used in Flask and how it is structured to work with web pages, let's look at how a typical Flask application with the extensions we need is constructed.

Initialization and the Application Instance

Flask and its extensions provide the entry point for your web application. Instead of writing all that onerous code yourself, Flask does it for you! The Flask extensions we will be using in this chapter include Flask-Script, Flask-Bootstrap, Flask-WTF, and WTForms. The following sections briefly describe each.

Flask-Script

Flask-Script enables scripting in Flask applications by adding a command-line parser (manifested as `manager`) that you can use to link to functions you've written. This is enabled by decorating the function with `@manager.command`. The best way to understand what this does for us is through an example.

The following is a basic, raw Flask application that does nothing. It's not even a "hello, world" example because nothing is shown and there are no web pages hosted - it's just the raw Flask application.

```
from flask import Flask      # import the Flask framework
app = Flask(__name__)      # initialize the application
if __name__ == "__main__": # guard for running the code
    app.run()               # launch the application
```

Notice the `app.run()` call. This is called the server startup and is executed when we load the script using the Python interpreter. When we run this code, all we see is the default message from Flask as shown in the following. Notice we don't have any way to see help as there are no such options. We also see the code launches using defaults for the web server (which we can change in the code if we desire). For example, we can change the port that the server is listening.

```
$ python ./sample-code.py --help
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

With Flask-Script, we add not only a help option but options to control the server. The following code shows how easy it is to add the statements to enable Flask-Script. The new statements are highlighted in bold.

```
from flask import Flask                # import the Flask framework
from flask_script import Manager       # import the flask script manager class

app = Flask(__name__)                 # initialize the application
manager = Manager(app)                # initialize the script manager class

# Sample method linked as a command-line option
@manager.command
def hello_world():
    """Print 'Hello, world!'"""
    print("Hello, world!")

if __name__ == "__main__":           # guard for running the code
    manager.run()                     # launch the application via manager class
```

When this code is run, we can see there are additional options available. Notice that the documentation string (immediately following the method definition) is shown as the help text for the command added.

```
$ python ./flask-script-ex.py --help
usage: flask-script-ex.py [-?] {hello_world,shell,runserver} ...

positional arguments:
  {hello_world,shell,runserver}
    hello_world      Print 'Hello, world!'
    shell            Runs a Python shell inside Flask application
context.
  runserver          Runs the Flask development server i.e. app.run()

optional arguments:
  -?, --help        show this help message and exit
```

Notice we see the command line arguments (commands) we added, `hello_world`, but we also see two new ones supplied by Flask-Script: `shell` and `runserver`. You must choose one of these commands when launching the server. The `shell` command allows you to use the code in a Python interpreter or similar tool and the `runserver` executes the code starting the web server.

Not only can we get help about the commands and options, Flask-Script also provides more control over the server from the command line. In fact, we can see all the options for each command by appending the `--help` option as shown in the following.

```
$ python ./flask-script-ex.py runserver --help
usage: flask-script-ex.py runserver [-?] [-h HOST] [-p PORT] [--threaded]
                                     [--processes PROCESSES]
                                     [--passthrough-errors] [-d] [-D] [-r]
[-R]
                                     [--ssl-crt SSL_CERT] [--ssl-key SSL_KEY]
```

Runs the Flask development server i.e. `app.run()`

optional arguments:

```
-?, --help          show this help message and exit
-h HOST, --host HOST
-p PORT, --port PORT
--threaded
--processes PROCESSES
--passthrough-errors
-d, --debug         enable the Werkzeug debugger (DO NOT use in production
                    code)
-D, --no-debug     disable the Werkzeug debugger
-r, --reload       monitor Python files for changes (not 100% safe for
                    production use)
-R, --no-reload    do not monitor Python files for changes
--ssl-crt SSL_CERT Path to ssl certificate
--ssl-key SSL_KEY  Path to ssl key
```

Notice here we see we can control all manner of things about the server including the port, host, and even how it executes.

Finally, we can execute the method we've decorated as a command-line option as shown in the following.

```
$ python ./flask-script-ex.py hello_world
Hello, world!
```

Thus, Flask-Script provides some very powerful features with only a few lines of code. You've got to love that!

Flask-Bootstrap

Flask-Bootstrap was originally developed by Twitter for making uniform, nice-looking web clients. Fortunately, they've made it a Flask extension so that everyone can take advantage of its features. Flask-Bootstrap is a framework on its own and provides even more command-line control as well as user interface components for clean, attractive web pages. It is also compatible with the newest web browsers.

The framework does its magic behind the scenes as a client library of cascading style sheets (CSS) and scripts that are invoked from the HTML templates (commonly referred to as either HTML files or template files) in Flask. We will learn more about templates in a later section. Since it is client-side, we won't see much by initializing it in the main application. Regardless, the following shows how to add Flask-bootstrap to our application code. Here, we see we have a skeleton with Flask-Script and Flask-Bootstrap initialized and configured.

```
from flask import Flask          # import the Flask framework
from flask_script import Manager # import the flask script manager class
from flask_bootstrap import Bootstrap # import the flask bootstrap extension

app = Flask(__name__)           # initialize the application
manager = Manager(app)          # initialize the script manager class
bootstrap = Bootstrap(app)      # initialize the bootstrap extension

if __name__ == "__main__":     # guard for running the code
    manager.run()                # launch the application via manager class
```

WTForms

WTForms is a component we need to support the Flask-WTF extension. It provides much of the functionality that the Flask-WTF component provides (because the Flask-WTF component is a Flask-specific wrapper for WTForms). Thus, we need only install it as a prerequisite for Flask-WTF and we will discuss it in the context of Flask-WTF.

Flask-WTF

The Flask-WTF extension is an interesting component providing several very useful additions, most notably for our use, integration with WTForms (a framework agnostic component) that permits the creation of form classes, and additional web security in the form of cross-site request forgery (CSRF) protection. These two features allow you to take your web application to a higher level of sophistication.

Form Classes

Form classes provide a hierarchy of classes that make defining web pages more logical. With Flask-WTF, you can define your form using two pieces of code; a special class derived from `FormForm` class (imported from the Flask framework) that you use to define fields using one or more additional classes that provide programmatic access to data, and an HTML file (or template) for rendering the web page. In this way, we see an abstraction layer (form classes) over the HTML files. We will see more about the HTML files in the next section.

Using form classes, you can define one or more fields such as `TextField` for text, `StringField` for a string, and more. Better still, you can define validators that allow you to programmatically describe the data. For example, you can define a minimum and maximum number of characters for a text field. If the number of characters submitted is outside of the range, an error message is generated. And, yes, you can define error message! The following lists some of the validators available. See <http://wtforms.readthedocs.io/en/latest/validators.html> for a complete list of validators.

To form classes, we must import the class and any field classes we want to use in the preamble of the application. The following shows an example of importing the form class and form field classes. In this example, we also import some validators that we will use for validating the data automatically.

```

from flask_wtf import FlaskForm
from wtforms import (HiddenField, TextField, TextAreaField, SelectField,
                    SelectMultipleField, IntegerField, SubmitField)
from wtforms.validators import Required, Length

```

To define a form class, we must derive a new class from `FlaskForm`. From there, we can construct the class however we want, but it is intended to allow you to define the fields. The `FlaskForm` parent class includes all the necessary code that Flask needs to instantiate and use the form class.

Let's look at a simple example. The following shows the form class for the handtool web page. The handtool table, which we will link to this code via the view function, contains several fields. We add a class for each field we want to put on the page using one of the available field classes. Since the Id field is not something users need to see, we make that field a hidden field and the other fields derivatives of the `TextField()` class. Notice how these were defined in the listing with names (labels) as the first parameter.

```

class HandtoolForm(FlaskForm):
    handtoolid = HiddenField('Id')
    vendor = NewSelectField(
        'Vendor', validators=[Required(message=REQUIRED.format("Vendor"))]
    )
    description = TextField(
        'Description',
        validators=[Required(message=REQUIRED.format("Description")),
                    Length(min=1, max=125,
                           message=RANGE.format("Description", 1, 125))]
    )
    handtooltype = NewSelectField(
        'Handtool Type',
        validators=[Required(message=REQUIRED.format("Handtool Type"))]
    )
    toolsize = TextField('ToolSize')
    place = NewSelectField(
        'Location',
        validators=[Required(message=REQUIRED.format("Location"))]
    )

```

```
create_button = SubmitField('Add')
del_button = SubmitField('Delete')
close_button = SubmitField('Close')
```

Notice also we have defined an array of validators in the form of function calls imported from the WTForms component for the fields. In each case, we use strings for the messages to make the code easier to read and more uniform.

We use the `Required()` validator that indicates the field must have a value. We augment the default error message with the name of the field to make it easier for the user to understand. We also use a `Length()` validator function that defines the minimal and maximum length of the field data. Once again, we augment the default error message. Validators are applied only on POST operations (when a submit event has occurred).

Next, we see there are three `SubmitField()` instances: one for a create (add) button, another for a delete button, and a close button. As you may surmise, in HTML parlance, these fields are rendered as `<input>` fields with a type of “submit.”

There are several field classes available for use. Table A-2 shows a sample of the most commonly used field classes (also called HTML fields). You can also derive from these fields to create custom field classes and provide text for the label that you can display next to the field (or as the button text for example). We will see an example of this in a later section.

Table A-2. *WTForms Field Classes*

Field class	Description
BooleanField	A checkbox with True and False values
DateField	Accepts date values
DateTimeField	Accepts date-time values
DecimalField	Accepts decimal values
FileField	File upload field
FloatField	Accepts a floating-point value
HiddenField	Hidden text field
IntegerField	Accepts integer values
MultipleFileField	Allows choosing multiple files
PasswordField	A password (masked) text field
RadioField	A list of radio buttons
SelectField	A drop-down list (choose one)
SelectMultipleField	A drop-down list of choices (choose one or more)
StringField	Accepts simple text
SubmitField	Form submit button
TextAreaField	Multiline text field

Cross-Site Request Forgery (CSRF) Protection

Cross-Site Request Forgery (CSRF) Protection is a technique that permits developers to sign web pages with an encrypted key making it much more difficult for hackers to spoof a GET or POST request. This is accomplished by first placing a special key in the application code and then referencing the key in each of our HTML files. The following shows an example of the preamble of an application. Notice all we need to do is assign the SECRET_KEY index of the app.config array with a phrase. This should be a phrase that is not easily guessed.

```
from flask import Flask           # import the Flask framework
from flask_script import Manager # import the flask script manager class
```

```

from flask_bootstrap import Bootstrap # import the flask bootstrap extension

app = Flask(__name__)                # initialize the application
app.config['SECRET_KEY'] = "He says, he's already got one!"
manager = Manager(app)               # initialize the script manager class
bootstrap = Bootstrap(app)           # initialize the bootstrap extension

if __name__ == "__main__":          # guard for running the code
    manager.run()                    # launch the application via manager class

```

To activate the CSRF in our web pages, we merely add the `form.csrf_token` to the HTML file. This is a special hidden field that Flask uses to validate the requests. We will see more about where to place this in a later section. But first, let's see a cool feature of Flask called flash.

Message Flashing

There are many cool features in Flask. The creators and the creators of the Flask extensions seem to have thought of everything – even error messaging. Consider a typical web application. How do you communicate errors to the user? Do you redirect to a new page, issue a popup, or perhaps display the error on the page? Flask has a solution for this called message flashing.

Message flashing is accomplished using the `flash()` method from the Flask framework. We simply import it in the preamble of our code; then when we want to display a message, we call the `flash()` function passing in the error message we want to see. Flask will present the error in a nicely formatted box presented at the top of the form. It doesn't replace the form and isn't a popup, but it does allow the user to dismiss the message. You can use flash messaging to communicate errors, warnings, and even state changes to the user. Figure A-1 shows an example of a flash message. In this example, we see two flash messages demonstrating you can display multiple messages at the same time. Notice the small X to the right of the message used to dismiss the image.

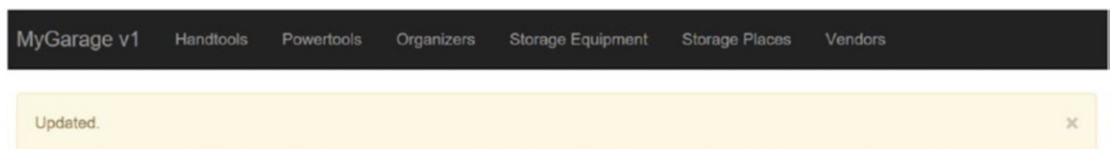


Figure A-1. Example Flash Message

HTML Files and Templates

Let's review our tour so far. We have discovered how to initialize an application with the various components and learned how Flask uses routes via the decorators to create a set of URLs for the application; these routes are directed to a view function, which instantiates the form class. The next piece of the puzzle is how to link the HTML web page to the form class.

Recall, this is done via the `render_template()` function where we pass in the name of a HTML file for processing. The reason `template` is in the name is because we can use the Jinja2 template component to make writing web pages easier. More specifically, the HTML file contains both HTML tags and Jinja2 template constructs.

Note All HTML files (templates) must be stored in the `templates` folder in the same location as the main application code. If you place them anywhere else, Flask won't be able to find the HTML files.

Templates together with form classes are where the user interface is designed. In short, templates are used to contain presentation logic and HTML files are used to contain the presentation data. These topics are likely to be the areas where some may need to spend some time experimenting with how to use them. The following sections give you a brief overview of Jinja2 templates and how to use them in our HTML files through demonstration of working examples. See the online documentation noted for more details.

Jinja2 Templates Overview

Jinja2 templates, hence templates, are used to contain any presentation logic like looping through data arrays, making decisions on what to display, and even formatting and presentation settings. If you are familiar with other web development environments, you may have seen this encapsulated in scripts or enabled through embedded scripting such as JavaScript.

Recall we rendered our web pages in our main code. This function tells Flask to read the file specified and convert the template constructs (render them) into HTML. That is, Flask will expand and compile the template constructs into HTML that the web server can present to the client.

There are several template constructs you can use to control the flow of execution, loops, and even comments. Whenever you want to use a template construct (think scripting language), you enclose it with `{% %}` prefix and suffix. This is so that the Flask framework recognizes the construct as a template operation rather than HTML.

However, it is not unusual and quite normal to see the template constructs intermixed with HTML tags. In fact, that is exactly how you should do it. After all, the files you will create are named `.html`. They just happen to contain template constructs. Does that mean you can only use templates when working with Flask? No, certainly not. If you want, you can render a pure HTML file!

At first, looking at templates can be quite daunting. But it isn't that difficult. Just look at all the lines with the `{% and %}` as the "code" portions. You may also see comments in the form of `{# #}` prefix and suffix.

If you look at the template, you will see the constructs and tags and formatted using indentation of two spaces. Indentation and whitespace in general don't matter outside the tags and constructs. However, most developers will use some form of indentation to make the file easier to read. In fact, most coding guidelines require indentation.

One of the cool features of templates beyond the constructs (think code) is the ability to create a hierarchy of templates. This allows you to create a "base" template that your other templates can use. For example, you can create a boilerplate of template constructs and HTML tags so that all your web pages look the same.

Recall from our look at Flask-Bootstrap, bootstrap provides several nice formatting features. One of those features is creating a pleasant looking navigation bar. Naturally, we would want this to appear on all our web pages. We can do this by defining it in the base template and extending it in our other template (HTML) files. Let's look at a base template for the sample application. Listing A-5 shows the base template for the library application.

Listing A-5. Sample Base Template

```
{% extends "bootstrap/base.html" %}
{% block title %}MyGarage{% endblock %}
{% block navbar %}
<div class="navbar navbar-inverse" role="navigation">
  <div class="container">
    <div class="navbar-header">
```

```

    <button type="button" class="navbar-toggle" data-toggle=
      "collapse" data-target=".navbar-collapse">
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
    </button>
    <a class="navbar-brand" href="/">MyGarage v1</a>
  </div>
  <div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
      <li><a href="/list/handtool">Handtools</a></li>
    </ul>
    <ul class="nav navbar-nav">
      <li><a href="/list/powertool">Powertools</a></li>
    </ul>
    <ul class="nav navbar-nav">
      <li><a href="/list/organizer">Organizers</a></li>
    </ul>
    <ul class="nav navbar-nav">
      <li><a href="/list/storage">Storage Equipment</a></li>
    </ul>
    <ul class="nav navbar-nav">
      <li><a href="/list/place">Storage Places</a></li>
    </ul>
    <ul class="nav navbar-nav">
      <li><a href="/list/vendor">Vendors</a></li>
    </ul>
  </div>
</div>
{% endblock %}

{% block content %}
<div class="container">
  {% for message in get_flashed_messages() %}
  <div class="alert alert-warning">

```

```

        <button type="button" class="close" data-dismiss="alert">&times;
        </button>
        {{ message }}
    </div>
    {% endfor %}

    {% block page_content %}{% endblock %}
</div>
{% endblock %}

```

Wow, there is a lot going on here! Take some time and read through it. While it may seem like something from the ship that crashed in Roswell, it really isn't that difficult to understand. You can find a full explanation of templates and Jinja2 at <http://jinja.pocoo.org/docs/2.10/>.

HTML Files Using Templates

Now that we know the template file generates the HTML for the page, we are ready to see how to manifest the field classes we defined in our form classes. Let's begin the discussion with a walkthrough of how to present data for the vendor data in the sample application. We begin with the form class and the field classes defined to the view function, which renders the template and finally the template itself.

Recall, the form class is where we define one or more form fields. We will use these field class instances to access the data in our view functions and in the template. Listing A-6 shows the form class.

Listing A-6. Vendor Form Class

```

class VendorForm(FlaskForm):
    """Vendor form class"""
    vendorid = HiddenField('VendorId')
    name = TextField(
        'Name',
        validators=[
            Required(message=REQUIRED.format("Name")),
            Length(min=1, max=50, message=RANGE.format("Name", 1, 50))]
    )
    url = TextField(

```

```

        'URL', validators=[
            Required(message=REQUIRED.format("URL")),
            Length(min=0, max=125, message=RANGE.format("URL", 0, 125))]
    )
sources = TextField(
    'Sources',
    validators=[
        Required(message=REQUIRED.format("Sources")),
        Length(min=0, max=40, message=RANGE.format("Sources", 0, 40))]
    )
create_button = SubmitField('Add')
del_button = SubmitField('Delete')
close_button = SubmitField('Close')

```

Notice the form class creates four fields; one for the vendor Id, which is a hidden field and one each for the name, URL, and source columns in the database table. We also see three submit fields (buttons); one for creating new data (`create_button`), one for deleting vendor data (`del_button`), and another to close the form (`close_button`).

We pass the form data to the template when it is rendered after instantiating it in the view function. Listing A-7 shows the view function for the vendor data with the database code removed for clarity with placeholders shown in bold. Here, we instantiate the vendor form class first, then pass it to the template.

Listing A-7. Vendor View Function (no database access)

```

@app.route('/vendor', methods=['GET', 'POST'])
@app.route('/vendor/<int:vendor_id>', methods=['GET', 'POST'])
def vendor(vendor_id=None):
    """Manage vendor CRUD operations."""
    form = VendorForm()
    if vendor_id:
        # Read operation goes here
        form.create_button.label.text = "Update"
    else:
        del form.del_button

```

```

if request.method == 'POST':
    operation = "Create"
    if form.close_button.data:
        operation = "Close"
    if form.create_button.data:
        if form.create_button.label.text == "Update":
            operation = "Update"
    if form.del_button and form.del_button.data:
        operation = "Delete"
        # Delete operation goes here
    if form.validate_on_submit():
        # Get the data from the form here
        if operation == "Close":
            return redirect('/list/vendor')
        elif operation == "Create":
            # Create operation goes here
        elif operation == "Update":
            # Delete operation goes here
    else:
        flash_errors(form)
return render_template("vendor.html", form=form)

```

Notice here we see the routes we've defined for the view. Notice also we have set the methods for requests to include both GET and POST. Notice that we can check if the request is a POST (submission of data). It is in this condition that we can retrieve data from the form class instance and save it to the database.

Finally, notice we instantiate an instance of the vendor form class (form) and later pass that as a parameter to the `render_template("vendor.html", form=form)` call. In this case, we now render the `vendor.html` template stored in the templates folder.

Ok, now we have our form class and view function. The focus now is what happens when we render the HTML template file. Listing A-8 shows the HTML file (template) for the vendor data.

Listing A-8. Vendor HTML File

```

{% extends "base.html" %}
{% block title %}MyGarage Search{% endblock %}
{% block page_content %}
  <form method=post> {{ form.csrf_token }}
    <fieldset>
      <legend>Vendor - Detail</legend>
      {{ form.hidden_tag() }}
      <div style=font-size:20px; font-weight:bold; margin-left:150px;s>
        {{ form.name.label }} <br>
        {{ form.name(size=50) }} <br>
        {{ form.url.label }} <br>
        {{ form.url(size=100) }} <br>
        {{ form.sources.label }} <br>
        {{ form.sources(size=40) }} <br>
        <br>
        {{ form.create_button }}
        {% if form.del_button %}
          {{ form.del_button }}
        {% endif %}
        {{ form.close_button }}
      </div>
    </fieldset>
  </form>
{% endblock %}

```

Notice the template begins with extending (inheriting) the `base.html` template file that we discussed earlier. We see a block defining the title and another block defining the page content. In that block, we see how to define the fields on the page referencing the field class instances from the form class instance (`form`). Indeed, notice we reference the label of the field as well as the data. The label is defined when you declare the field class and the data is where the values are stored. When we want to populate a form (GET), we set the data element to the value, and when we want to read the data (POST), we reference the data element.

Notice also we added the CSRF token for security, rendered the hidden fields with the `form.hidden_tag()` function, and included the submit fields conditionally including the delete submit field (`del_button`).

Whew! That's how Flask works to present a web page. Once you're used to it, it is a nifty way to separate several layers of functionality and make it easy to get data from the user or present it to the user.

Now, let's look at how to build custom error handlers into our application and later how to redirect control in our application to the correct view functions.

Error Handlers

Recall we mentioned it is possible to create your own error handling mechanisms for errors in your application. There are two such error mechanisms you should consider making; one for the 404 (not found) error, and another for 500 (application errors). To define each, we first make a view function decorated with `@app.errorhandler(num)`, a view function, and an HTML file. Let's look at each example.

Not Found (404) Errors

To handle 404 (not found) errors, we create a view function with the special error handler routing function, which renders the HTML file. Flask will automatically direct all not found error conditions to this view. The following shows the view function for the 404 not found error handler. As you can see, it is simple.

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404
```

The associated error handler HTML code is in the file named `404.html` as shown in the following. Notice we inherit from the `base.html` file so the resulting web page looks the same as any other in the application complete with the menu from the bootstrap component. Notice we can also define the text for the error message and a title. Feel free to embellish your own error handlers to make things more interesting for your users.

```
{% extends "base.html" %}
{% block title %}MyGarage ERROR: Page Not Found{% endblock %}
{% block page_content %}
```

```
<div class="page-header">
  <h1>Page not found.</h1>
</div>
{% endblock %}
```

Application (500) Errors

To handle 500 (application) errors, we follow the same pattern as before. The following is the error handler for the application errors.

```
@app.errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

The associated error handler HTML code is in the file named 500.html as shown in the following. Notice we inherit from the base.html file, so the resulting web page looks the same as any other in the application complete with the menu from the bootstrap component.

```
{% extends "base.html" %}
{% block title %}MyGarage ERROR{% endblock %}
{% block page_content %}
<div class="page-header">
  <h1>OOPS! Application error.</h1>
</div>
{% endblock %}
```

Creating these basic error handlers is highly recommended for all Flask applications. You may find the application error handler most helpful when developing your application. You can even augment the code to provide debug information to be displayed in the web page.

Redirects

At this point, you may be wondering how a Flask application can programmatically direct execution from one view to another. The answer is another simple construct in Flask: redirects. We use the `redirect()` function (imported from the `flask` module) with a URL to redirect control to another view. For example, suppose you had a list form and, depending on which button the user clicks (submitting the form via POST), you want to display a different web page. The following demonstrates how to use the `redirect()` function to do this.

```
if kind == 'handtool':
    form.form_name.label = 'Handtools'
    if request.method == 'POST':
        return redirect('handtool')
...
elif kind == 'organizer':
    form.form_name.label = 'Organizers'
    if request.method == 'POST':
        return redirect('organizer')
...
elif kind == 'powertool':
    form.form_name.label = 'Powertools'
    if request.method == 'POST':
        return redirect('powertool')
...
```

Here, we see there are three redirects after a POST request. In each case, we are using one of the routes defined in our application to tell Flask to call the associated view function. In this way, we can create a menu or a series of submit fields to allow the user to move from one page to another.

The `redirect()` function requires a valid route, and for most cases, it is simply the text you supplied in the decorator. However, if you need for form a complex URL path, you can use the `url_for()` function to validate the route before you redirect. The function also helps avoid broken links if you reorganize or change your routes. For example, you can use `redirect(url_for('vendor'))` to validate the route and form a URL for it.

Additional Features

There is much more to Flask than what we've seen in this crash course. Some of the things not discussed that you may be interested in learning more about include the following (these are just a few of them). If these interest you, consider looking them up in the online documentation.

- *Application and Request Context*: there are variables you can use to capture application context such as session, global, request, and more. For more information, see <http://flask.pocoo.org/docs/0.12/appcontext/>.
- *Cookies*: You can work with cookies if you require. For more information, see <http://flask.pocoo.org/docs/0.12/quickstart/#cookies>.
- *Flask-Moment - Localization of Dates and Times*: If you need to work with localization of date and time, see the Flask-Moment extension at <https://github.com/miguelgrinberg/Flask-Moment>.

Tip For more information about Flask and how to use it and its associated packages, the following book is an excellent reference on the topic: *Flask Web Development: Developing Web Applications with Python* (O'Reilly Media 2014), Miguel Grinberg.

Flask Review: Sample Application

Now that we've had a brief primer on Flask, let's see how all this works with one of our sample applications. In this section, we will review how to set up the sample applications and how to start them. In this section, we will see the sample application from Chapter 5. The sample applications from later chapters work in a very similar manner. Once you are familiar with how to launch and interact with the examples here, you should be able to run the other sample applications. We begin with how to download and copy the files.

Preparing Your PC

The first thing you should do is to download the source code for this book from the book web site <https://www.apress.com/us/book/97814NNNNNNNNN>. You should see folders representing the source code for each chapter. Just download the folder that matches the chapter you want. Once you've downloaded the source code and extracted it, locate the folder for the sample application you want to use. For example, the sample application in Chapter 5 is in a folder by that name at <https://github.com/apress/introducing-mysql-8-shell>.

At this point, you must choose where you want to run the application. If you want to run it from the location where you downloaded and extracted the files, you can. However, it is best to move the code to another location.

For example, on Linux or macOS, you can place it in a folder named source in your home folder. Or, on Windows 10, you can place it in your Documents folder. Once you decide on the location, you can then create a folder to contain the sample application (mygarage_v1). Next, copy the files created when you extracted the code for the chapter into the mygarage_v1 folder including the subdirectories. If you then show the list of files in the folder, you should have the main executable file and two subfolders as shown in the following.

```
C:\Users\cbell\Documents\mygarage_v1>dir
Volume in drive C is Local Disk
Volume Serial Number is AAFC-6767

Directory of C:\Users\cbell\Documents\mygarage_v1

03/14/2019  02:20 PM    <DIR>          .
03/14/2019  02:20 PM    <DIR>          ..
03/14/2019  02:20 PM    <DIR>          database
03/08/2019  11:02 PM                38,045 mygarage_v1.py
03/14/2019  02:20 PM    <DIR>          templates
                1 File(s)        38,045 bytes
                4 Dir(s)  124,419,055,616 bytes free
```

The two subfolders, database and templates, are used to store the code modules and assorted files we need. The database folder is where we place the database code modules and the templates folder is where we place the .html files. You can explore the contents of those folders if you'd like.

By now, you should have MySQL installed and working on your computer. You will need the user account and password that you want to use to connect to MySQL to enter on the command to start the application. Before we do that, let's ensure we have the database created and populated.

Recall from the discussion in Chapter 5, the sample source code contains a file named `database/garage_v1.sql`, which contains the SQL statements for creating the sample database and populating it with sample data. If you haven't already done so, let's do that now.

Change to the database folder and issue the following command to tell the shell to open the file and execute the statements. It won't take but a minute to run and, since we're running in batch mode, will exit the shell when complete. Listing A-9 shows the results of running these commands.

Listing A-9. Populating the Example Database (Windows 10)

```
C:\Users\cbell\Documents\mygarage_v1>cd database
C:\Users\cbell\Documents\mygarage_v1\database>mysqlsh --uri root@
localhost:3306 --sql -f garage_v1.sql
Records: 31 Duplicates: 0 Warnings: 0
Records: 6 Duplicates: 0 Warnings: 0
Records: 250 Duplicates: 0 Warnings: 0
Records: 3 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 22 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 22 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 0 Duplicates: 0 Warnings: 0
Records: 2 Duplicates: 0 Warnings: 0
Records: 3 Duplicates: 0 Warnings: 0
```

Now that we have the database created and populated, we're ready to launch the sample application for the first time.

Running the Sample Application

Change back to the `mygarage_v1` folder and run the application with the following command shown in bold. Notice, you will be prompted for the MySQL user Id (user account) and password.

```
C:\Users\cbell\Documents\mygarage_v1>python mygarage_v1.py runserver
User Id: root
Password:
* Serving Flask app "mygarage_v1" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

If you are running with a newer version of Flask and Python, you may see one or more deprecation warnings such "DeprecationWarning: Required is going away in WTForms 3.0, use DataRequired". If that occurs, you can suppress the warnings with the option `-W ignore::DeprecationWarning` as shown in the following.

```
C:\Users\cbell\Documents\mygarage_v1>python -W ignore::DeprecationWarning
mygarage_v1.py runserver
```

Notice the line at the end of the output. This shows you the URL to use in your browser to connect to and use the application. It is running at this point, we just need to connect to it. Go ahead and copy that URL into your browser. When you press *ENTER* (or click Go, etc.), you will see the landing page for the application as shown in [Figure A-2](#).

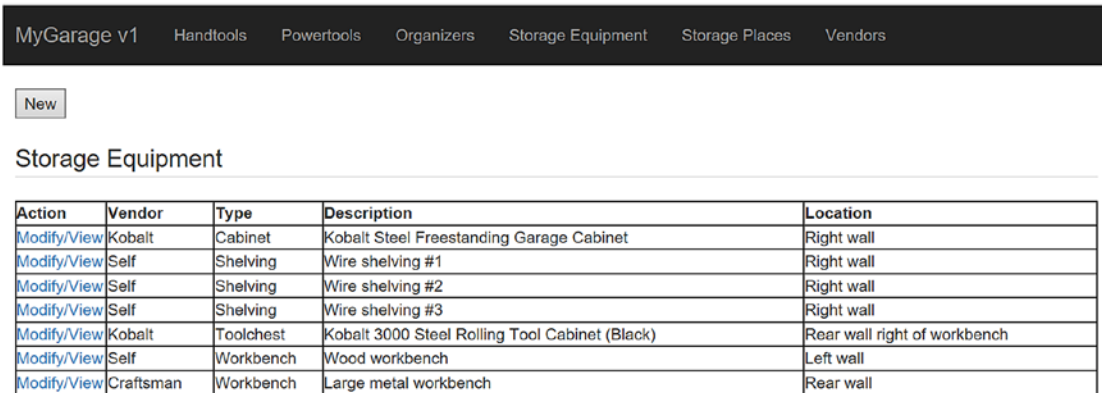


Figure A-2. Landing Page (*mygarage_v1*) – Storage Equipment List View

Here, we see the landing or default page is the storage equipment list. This is shown as the default as it is the highest level of view of a garage storage solution – all the containers in the garage.

Notice also across the top of the application we see the Flask banner, which has links (buttons) for each of the views in the application including handtools, powertools, organizers, storage equipment, storage places, and vendors. Each of these links displays a list of the items in the view. Thus, we will call them list views.

Notice each row in the view has a modify/view link, which we can use to see more detail about an item, or we can edit (or delete) the item. The *New* button at the top allows us to create a new item for the active list view.

As we learned in Chapter 5, each of these represents a table in the database. The handtools, powertools, and vendors represent the tools and their manufacturers in our garage. The other views are the containers we use in our garage to organize the tools.

As you may surmise from reading Chapter 5, each of the views in presented in list form. Thus, we will see a list of all the items (records) in each view (table). Let’s briefly look at each of these in order as shown on the banner. Figure A-3 shows the handtools view and Figure A-4 shows the handtool detail view.

MyGarage v1 Handtools Powertools Organizers Storage Equipment Storage Places Vendors

New

Handtools

Action	ToolType	Description	Tool Size/Application	Storage Equipment	Location Type	Location
Modify/View	Awl	Alloy Steel Scratch	6-in	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Left 3
Modify/View	Awl	Complex Hook	3-in	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Left 3
Modify/View	Awl	Curved Hook	3-in	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Left 3
Modify/View	Awl	Hook	3-in	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Left 3

Figure A-3. Handtools List View

In this view, we see the tool type, description, size, which storage equipment it's stored in, the location type, and location. Thus, with a glance we can see our 8" C clamp is stored in the rolling tool cabinet in the bottom drawer. Notice also we have the *modify/view* link for each item, which allows us to edit the item (row) or to see more details about the handtool.

MyGarage v1 Handtools Powertools Organizers Storage Equipment Storage Places Vendors

Handtool - Detail

Vendor

Description

ToolSize

Handtool Type

Location

Figure A-4. Handtool View

Here, we see that there are buttons at the bottom for update, delete, or close operations. If we want to make changes, we use the *Update* button. If we want to delete the item (record/row), we use the *Delete* button. The *Close* button simply closes the form and returns to the list view. These buttons are present on all the detailed view forms. Figure A-5 shows the powertools list view, which has a similar layout as the handtools list view.

Action	ToolType	Description	Storage Equipment	Location Type	Location
Modify/View	Air	0.5-in 1000-ft Air Impact Wrench	Wood workbench	Shelf	Top
Modify/View	Air	1.625-in 18-Gauge Finish Nail Gun	Wood workbench	Shelf	Top
Modify/View	Air	23-Gauge Headless Pin Pneumatic Nailer	Wood workbench	Shelf	Top
Modify/View	Air	6-Gallon Portable Electric Pancake Air Compressor	Wood workbench	Shelf	Bottom
Modify/View	Corded	120-Volt 2-Amp Sheet Sander 4.5" sheets	Wood workbench	Shelf	Top
Modify/View	Corded	4-1/2-in 5.5-Amp Corded Circular Saw with Aluminum Shoe	Wood workbench	Shelf	Top
Modify/View	Corded	5-Amp Keyless T or U Shank Variable Speed Corded Jigsaw	Wood workbench	Shelf	Top

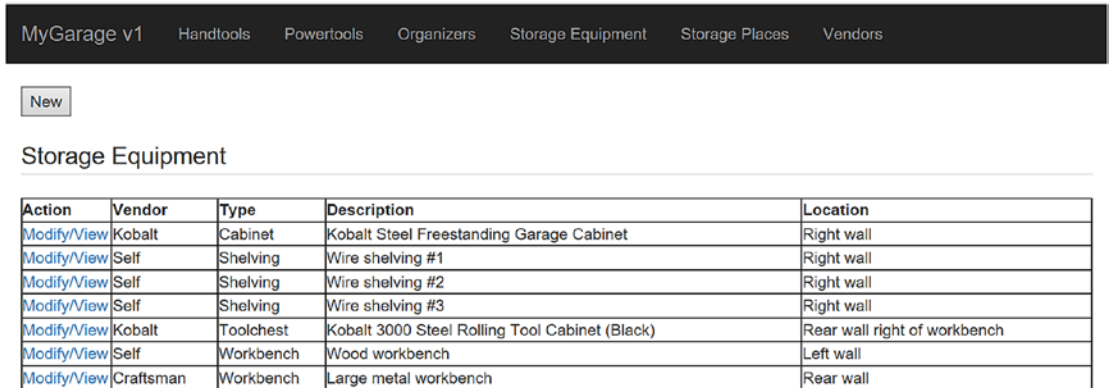
Figure A-5. Powertools List View

Figure A-6 shows the organizers list view.

Action	Type	Description	Storage Equipment	Location Type	Location
Modify/View	Case	Metric Socket Set	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Shelf	Top
Modify/View	Case	Metric Socket Set	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Shelf	Top
Modify/View	Case	SAE Socket Set	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Shelf	Top
Modify/View	Case	SAE Socket Set	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Shelf	Top
Modify/View	Case	SAE/Metric Socket Set	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Shelf	Top
Modify/View	Case	SAE/Metric Socket Set	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Shelf	Top

Figure A-6. Organizers List View

Figure A-7 shows the storage equipment list view, which is also the starting or landing view.

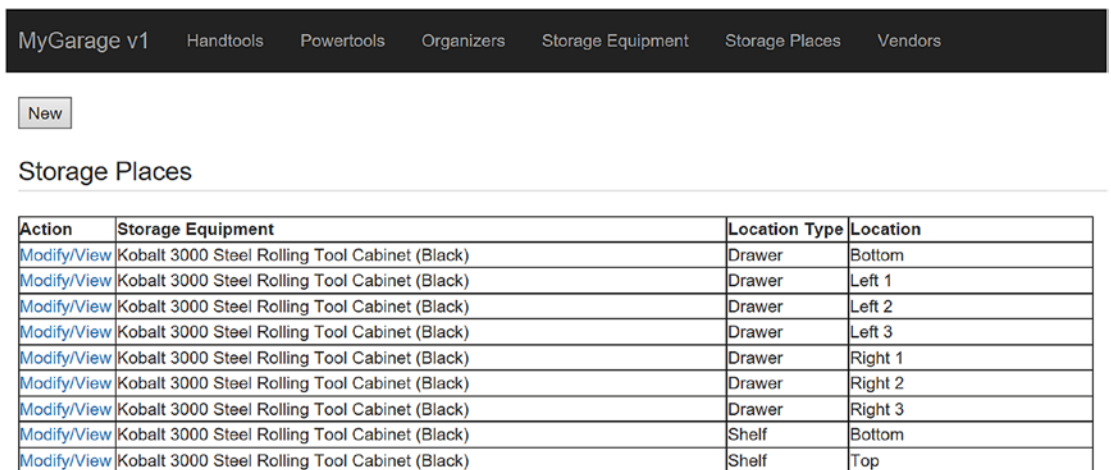


Action	Vendor	Type	Description	Location
Modify/View	Kobalt	Cabinet	Kobalt Steel Freestanding Garage Cabinet	Right wall
Modify/View	Self	Shelving	Wire shelving #1	Right wall
Modify/View	Self	Shelving	Wire shelving #2	Right wall
Modify/View	Self	Shelving	Wire shelving #3	Right wall
Modify/View	Kobalt	Toolchest	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Rear wall right of workbench
Modify/View	Self	Workbench	Wood workbench	Left wall
Modify/View	Craftsman	Workbench	Large metal workbench	Rear wall

Figure A-7. Storage Equipment List View

By now, you are starting to realize all the list views have a similar layout. The difference is in the columns as some list views use different columns to display the items. Let's complete the survey of the list views.

Figure A-8 shows the storage places list view. Here, we see slightly different columns that match the data we've stored in the table.



Action	Storage Equipment	Location Type	Location
Modify/View	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Bottom
Modify/View	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Left 1
Modify/View	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Left 2
Modify/View	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Left 3
Modify/View	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Right 1
Modify/View	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Right 2
Modify/View	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Drawer	Right 3
Modify/View	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Shelf	Bottom
Modify/View	Kobalt 3000 Steel Rolling Tool Cabinet (Black)	Shelf	Top

Figure A-8. Storage Places List View

Finally Figure A-9 shows the vendors list view.

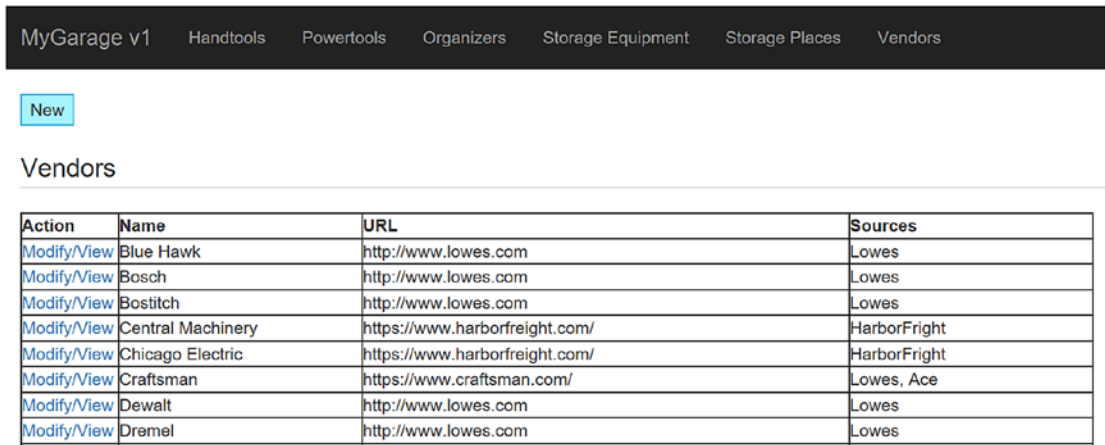


Figure A-9. Vendors List View

At this point, if you’ve launched the application, you should see several messages appear in the terminal (command window). Each of these tells us what the application responded to as you clicked through the views. The following shows an excerpt. This is completely normal and can help you diagnose any problems should something go wrong.

```
127.0.0.1 - - [17/Mar/2019 17:06:48] "GET /list/handtool HTTP/1.1" 200 -
127.0.0.1 - - [17/Mar/2019 17:07:22] "GET /list/powertool HTTP/1.1" 200 -
127.0.0.1 - - [17/Mar/2019 17:07:42] "GET /list/organizer HTTP/1.1" 200 -
127.0.0.1 - - [17/Mar/2019 17:08:03] "GET /list/storage HTTP/1.1" 200 -
127.0.0.1 - - [17/Mar/2019 17:08:26] "GET /list/place HTTP/1.1" 200 -
127.0.0.1 - - [17/Mar/2019 17:08:55] "GET /list/vendor HTTP/1.1" 200 -
127.0.0.1 - - [17/Mar/2019 17:09:17] "GET /list/place HTTP/1.1" 200 -
127.0.0.1 - - [17/Mar/2019 17:09:18] "GET /list/storage HTTP/1.1" 200 -
127.0.0.1 - - [17/Mar/2019 17:09:20] "GET /list/organizer HTTP/1.1" 200 -
127.0.0.1 - - [17/Mar/2019 17:09:22] "GET /list/powertool HTTP/1.1" 200 -
127.0.0.1 - - [17/Mar/2019 17:09:23] "GET /list/handtool HTTP/1.1" 200 -
```

Now, let’s review how to use the application to store and locate tools in our garage including how the create, read, update, and delete (CRUD) operations represented.

How to Use the Application

Let's begin by explaining how to use the sample application. While most sample applications you find in books are meant to simply demonstrate a concept,¹ seldom is there a complete, usable application. The applications in Chapters 5 and 6 are meant to be usable by anyone not just to demonstrate the concepts, but also to use!

We have already seen how to get a list of the various items in the tables (database) and we've learned that we can edit any item we want to make changes, but what do you do when you want to add a new item?

For example, when you acquire a new tool or maybe loose or given one away, you can edit the data by adding a new tool identifying where it is stored. That is, you identify the organizer, storage place, and storage equipment. For example, if you acquire a new hammer, you can click on the *New* button in the handtools list view, fill in the data, and use the drop-down lists to choose where it is stored. Note that when you open a new detail view, the *Delete* button is now shown (not relevant for a create operation). Figure A-10 shows an example.

The screenshot shows a web application interface for 'MyGarage v1'. At the top, there is a navigation bar with tabs for 'Handtools', 'Powertools', 'Organizers', 'Storage Equipment', 'Storage Places', and 'Vendors'. Below this, the 'Handtool - Detail' form is displayed. The form contains the following fields:

- Vendor:** A dropdown menu with 'Craftsman' selected.
- Description:** A text input field containing 'Sledge'.
- ToolSize:** A text input field containing '5 lb'.
- Handtool Type:** A dropdown menu with 'Hammer' selected.
- Location:** A dropdown menu with 'Wire shelving #1 - Shelf, Bottom' selected.

At the bottom of the form, there are two buttons: 'Add' and 'Close'.

Figure A-10. Adding a new hammer

Notice here we have drop-down lists for the vendor, tool type, and location. Each of these is represented in the database. Recall, the tool type is an enumeration

¹Most do an excellent job of that!

in the handtools table. The vendor is simply a look-up of the vendor names in the vendors table. The location drop down is populated a bit more sophisticatedly as it is a combination (join) of the storage equipment and storage place tables so that you can choose a single entry vs. having to choose from several drop-down lists.

So, how does that help you organize things? Recall, we can also add new organizers, storage places, and storage equipment. For example, if you want to add a bin to store things, you can create a new record for it in the organizers view, then identify where it is – perhaps you place it on a shelf on a shelving unit. In this way, you can build up the storage capacity of your garage or workshop all without losing track of where things are. That is, of course, you abide by the stock room moto: things go where things go per the thing that tells you where things go, not by where you think they should go.²

CRUD Operations in the Application

To complete our tour of the sample application, let's now see how the application implements the CRUD operations. The following briefly explains how each of the CRUD operations is represented.

- *Create*: The *New* button on each list view allows you to create a new record (row) in the table.
- *Read*: This is represented in two ways; you can view all rows for each table using the list views, and you can view the complete details of a record by clicking the *modify/view* link on the row in the list view.
- *Update*: The *Update* button on the detail view for each item allows you to update the record (row) in the table.
- *Delete*: The *Delete* button on the detail view for each item allows you to delete the record (row) in the table.

Now that we have seen all the views available in the application (except for each detail view), we now have the knowledge needed to install and use the sample application. Once again, the other sample applications in the book will work in a similar manner even though some of the views may differ.

There is just one more thing to learn: how to turn it off.

²As humans, we often fail this creed. Hence the need for inventory reconciliation.

Shutting Down the Sample Application

To shut down the application, simply return to the terminal window where you launched the application and *CTRL-C*. You can close the browser at any time, but if you try to use the application after stopping it, you may see errors such as can't reach this page or unable to connect as shown in Figures A-11 and A-12.

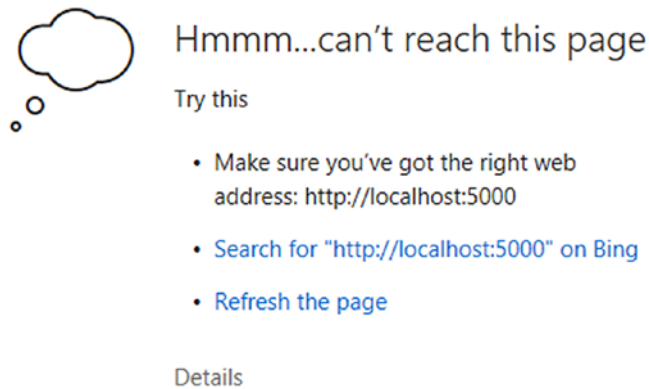


Figure A-11. Not Found Error (Windows 10 – Edge)

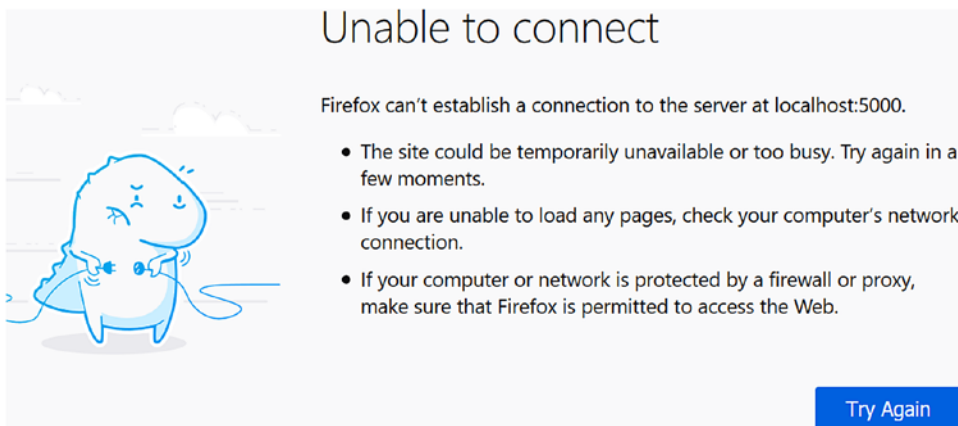


Figure A-12. Not Found Error (Windows 10 – Firefox)

Index

A

- Account management
 - locking, 21
 - passwords, 21
 - roles, 20
 - user limits, 20
- add_instance() method, 430, 436
- Admin Application Programming Interface (AdminAPI), 146, 395, 403–404
- Application programming
 - interface (API), 4, 234, 458
- APT repository
 - configuring packages, 67
 - download, 63
 - installation, 65, 66
 - server, 66, 67
 - shell, 70, 71
 - password encryption
 - dialog, 68
 - root user password, 68
 - save file, 65
 - select, 63
 - skipping the login, 64
- Archive storage engine, 399
- Authentication method dialog, 41, 42

B

- Binary log, 342, 349, 368
- Blackhole storage engine, 399

C

- CHANGE MASTER command, 358–360
- check_instance_configuration() method, 425
- Class modules, test
 - CRUDTest, 222, 223
 - run_all.py, 227, 228
 - Test Driver, 228, 230
- Comma-separated value (CSV) files, 16, 396, 400
- configure_local_instance() method, 426
- Cookies, 480
- create_cluster() method, 426, 427
- create() method, 209, 210
- Create, read, update, and delete (CRUD), 147, 191, 209, 282, 488
- CREATE TABLE command, 127, 192, 399
- Cross-Site Request Forgery (CSRF)
 - Protection, 465, 468–469
- CRUD operations, Python
 - creating data, 161
 - delete data, 164, 165
 - document store/relational data, 154
 - reading data, 162, 163
 - schema class, 155
 - table class, 156–158
 - updating data, 163, 164
 - working with results, 165, 167–170
- CRUDTest class
 - code, 223–225
 - methods, 223
- VendorTests, 225–227

D

Database design

code design

CRUD, 191

modules, 191

ERD, 181, 182

handtool table, 187, 188

location view, 189

organizer table, 184, 185

powertool table, 188, 189

storage equipment table, 186, 187

storage place table, 185

surrogate key, 182

vendor table, 183, 184

Database management, Python

built-in modules, 146

CRUD operations (*see* CRUD operations, Python)mysqlx module (*see* Mysqlx module)

script, 170–174

X DevAPI, 146, 147

Data definition language [DDL], 24, 119, 120

--datadir option, 350, 373

Data manipulation language [DML], 24, 119, 120

delete() method, 216, 218

deploy_sandbox_instance() method, 421, 423, 425

Development computer option, 41

Document-oriented database, 235

Document store development

base class

CRUD operations, 319

GarageCollection method, 319

collection classes, 324–327

CRUDTest, 328

drop down lists, 327, 328

GarageCollection Base Class code, 320, 321, 323

MyGarage class

code, 312–315

garage_v1_test unit test, 318

garage_v2_test.py, 317, 318

helper function, 316

methods, 311

Python path, 316

unit test, 317

test driver, 332–334

Test Driver run_all.py, 331, 332

vendor class, 323

VendorTests Class, 329, 330

Document store, schema design

cabinets collection, 283, 284

CREATE TABLE statement, 281

creation, 282

locations collection, 285

organizers collection, 286, 287

shelving_units collection, 287, 288

toolchests collection, 288, 289

tools collection, 289

vendors collection, 290

workbenches collection, 290, 291

Document store set up

import data, 308–310

relational data (*see* Relational data to document store)Document store systems, *See also*

Document-oriented database

API, 234

JSON, 235, 236

key, value mechanisms, 232, 233

metadata, 235

NoSQL interface, 234

Quick start, 237, 238

E

Entity-relationship diagram (ERD), 181–183

Error handlers

- application (500) Errors, 478
- not Found (404) errors, 477–478

F

Fault tolerance, 368, 370

Features

APIs

- adminAPI, 6
- X DevAPI, 6

auto completion, 5

batch code execution, 6

command history, 6

customize prompt, 6

global variables, 6

interactive code execution, 7

JSON import, 6

logging, 7

multi-line support, 7

new

- adminAPI, 8
- SQL mode execution, 8
- user defined reports, 8

output formats, 7

scripting languages, 7

sessions, 7

startup scripts, 7

upgrade checker, 8

user credentials, 8

Federated storage engine, 400

find_* functions, 296

Flash message, 469

Flask-Bootstrap, 464

Flask-Moment extension, 480

Flask primer

application and request

context, 480

cookies, 480

error handlers (*see* Error handlers)

HTML files, templates

vendor form class, 473, 474

vendor HTML File, 476, 477

vendor view function, 474, 475

initialization and

application instance

CSRF protection, 468, 469

Flask-Bootstrap, 464

Flask-Script, 461–464

Flask-WTF extension, 465

Form classes, 465–468

message flashing, 469

WTForms, 465

Jinja2 templates, 470, 471, 473

localization of date and time, 480

redirects, 479

sample application

CRUD operations, 490

database creation, 481–483

handtools list view, 489

run, 483–488

shutting down, 491

template library, 458

terminology, 459, 460

web application libraries, 458

Flask-Script, 461–464

Flask-WTF extension, 465

Form classes

Flask-WTF, 465

POST operations, 467

TextField, 465, 466

WTForms, 468

G

Garage application

- code design, 291
- code modules, 292
- document store, 276, 279
- storage equipment, 276
- toolchest detail view, 277
- Toolchest JSON Document, 279, 280

`get_classic_session()`, 91

`get_*` functions, 296

`get_last_insert_id()`, 210

`get_session()` method, 91, 148

Global transaction

- identifiers (GTIDs), 337, 343

GRANT commands, 125, 357

Group communication, 368

Group replication

- concepts, 368, 369
- failover, 388–391
- failure detection, 345, 347
- fault tolerance, 347
- group membership, 347
- GTIDs, 344
- high availability, 345
- recovery procedure, 345
- RW transaction, 344
- setup and configuration, 371
- setup tutorial (*see* Group replication setup)

`group_replication_bootstrap_group`, 382

Group Replication plugin, 376, 380

`group_replication_recovery_get_public_key`, 378

Group replication setup

- data directories, 373, 374
- MySQL instances, 379–381

online reference, 375

primary configuration, 376, 377

primary, start, 382

replication status, 383, 385, 386

secondaries, start, 383

secondaries to the primary, 382, 383

secondary configuration, 377–379

shutting down, 388

user account, 381, 382

GTID-based replication, 352, 354

H

Handtool class

SELECT query, 219

`sql()` method, 220

vendor class, 218

High availability (HA), 337

engineering principles, 338

fault tolerance, 340

features, 341

implementation, 339

recovery, 339

redundancy, 340

reliability, 338

scaling, 340

uptime, 338

I

`import_json()` method, 309

`--initialize-insecure` option, 350, 373

Inline path operator, 248

InnoDB, 16, 17

InnoDB Cluster

AdminAPI, 403

application architecture, 405–407

cluster class, 419, 420

- dba class, [417, 418](#)
- MySQL Router, [404](#)
- MySQL Shell, [403](#)
- overview, [393–396](#)
- sandbox, [396](#)
- storage engine
 - ACID, [397, 398](#)
 - archive, [399](#)
 - blackhole, [399](#)
 - CSV, [400](#)
 - federated, [400](#)
 - group replication, [402](#)
 - memory, [400](#)
 - merge storage engine/MRG_
 - MYISAM, [401, 402](#)
 - MyISAM, [401](#)
 - performance schema, [402](#)
- typical configuration, [395](#)
- upgrade checker, [407–412](#)
- X DevAPI, [403](#)

Installation

- community edition, [30, 31](#)
- Connector/Python, [457, 458](#)
- Flask, [453, 454](#)
- Flask-Bootstrap, [455, 456](#)
- Flask-Script, [454, 455](#)
- Flask-WTF, [456, 457](#)
- Linux (Ubuntu) (*see* APT repository)
- macOS (*see* macOS, install)
- MySQL Installer (*see* MySQL Installer, Windows)
- prerequisites, [29, 30](#)
- for Windows, [32](#)
- WTForms, [457](#)

J

- JavaScript Object Notation (JSON), [231](#)
 - functions, [249, 250](#)
 - creation, [251](#)
 - modification, [255, 257, 258](#)
 - searching, [258](#)
 - utility, [264](#)
 - MySQL, [236, 237](#)
 - and SQL, [239](#)
 - strings
 - MySQL, [240, 241](#)
 - SQL statements, [241–244](#)
- Jinja2 templates, [470–473](#)
- JOIN clause, [143](#)
- JSON_ARRAY() function, [243, 250](#)
- JSON_ARRAYAGG() function, [250, 251](#)
- JSON_ARRAY_APPEND() function, [250, 252](#)
- JSON_ARRAY_INSERT() function, [250, 253](#)
- JSON_CONTAINS() function, [250, 258](#)
- JSON_CONTAINS_PATH()
 - function, [259, 263](#)
- JSON data type
 - querying rows, [14, 15](#)
 - relational database, [11–13](#)
 - structures, [10](#)
- JSON_DEPTH() function, [250, 264](#)
- JSON_EXTRACT() function, [97](#)
- JSON_EXTRACT() function, [103, 245, 247, 249, 261, 263](#)
- JSON_INSERT() function, [250, 254](#)
- JSON_KEYS() function, [250, 266](#)
- JSON_LENGTH() function, [250, 266](#)
- JSON_MERGE_PATCH() function, [254](#)
- JSON_MERGE_PRESERVE()
 - function, [102, 254](#)

INDEX

JSON_OBJECT() function, [243, 250, 251](#)
JSON_PRETTY() function, [13, 101, 270, 271](#)
JSON_QUOTE() function, [251, 268](#)
JSON_REMOVE() function, [251, 256](#)
JSON_REPLACE() function, [251, 257](#)
JSON_SEARCH() function, [251, 263, 264](#)
JSON_SET() function, [251, 257, 258](#)
JSON_TYPE() function, [244, 251, 264](#)
JSON_UNQUOTE() function, [269–272](#)
JSON_VALID() function, [100, 243, 264](#)

K

Key, value mechanisms, [232–234](#)

L

Length() validator function, [467](#)

Location class

- class modules, [206, 207](#)
- CRUD operations, [205](#)
- limit() method, [203](#)
- MyGarage class, [203](#)
- mysqlx object, [204](#)
- primitive code, [203](#)
- testing, [207, 208](#)

M

macOS, install

- download server, [50](#)
- license dialog, [53](#)
- password dialog, [55, 56](#)
- root user password/start server dialog, [56](#)
- summary dialog, [57](#)
- type dialog, [54](#)
- welcome dialog, [52](#)
- download shell, [51](#)

- accept license dialog, [59](#)
- summary dialog, [61](#)
- type dialog, [60](#)
- welcome dialog, [58](#)

make_list() function, [315](#)

Memory storage engine, [400](#)

merge storage engine/MRG_MYISAM, [401](#)

Message flashing, [469](#)

MyGarage class

- garage_v1 Code, [196, 198](#)
- methods, [195](#)
- mysqlx module, [196, 199](#)
- testing
 - garage_v1_test.py, [201](#)
 - print() statements, [201](#)
 - py option, [200](#)
 - Python path, [199](#)

MyISAM storage engine, [401](#)

MySQL commands/functions

- create indexes, [139, 140](#)
- database creation, [126, 127](#)
- deleting data, [131, 132](#)
- selecting data (*see* [Selecting data](#), [MYSQL](#))
- simple join, [141, 143](#)
- stored procedure, [144, 145](#)
- storing data, [129, 130](#)
- table creation, [127](#)
 - AUTO INCREMENT, [128](#)
 - name, [127](#)
 - PRIMARY KEY, [128](#)
 - SHOW TABLES command, [129](#)
 - TIMESTAMP column, [128](#)
- terminology, [124](#)
- updating data, [130, 131](#)
- user accounts and granting
 - access, [125, 126](#)
- view creation, [140, 141](#)

- MySQL Installer, Windows, [32, 33](#)
 - accounts and roles, [43](#)
 - authentication method, [42](#)
 - components, [34, 36, 37](#)
 - connect to server, [47](#)
 - group replication, [40](#)
 - installation complete, [38, 49](#)
 - installation (in progress), [38](#)
 - installation (staging), [37](#)
 - product configuration, [39](#)
 - products and features, [35, 36](#)
 - router, [46](#)
 - samples and examples, [48](#)
 - setup type, [34](#)
 - type and networking, [41](#)
 - welcome panel/license agreement, [33, 34](#)
 - Windows service, [44](#)
 - MySQL replication
 - asynchronous, [343](#)
 - binary log, [342](#)
 - GTIDs, [343](#)
 - master, [342](#)
 - secondaries or slaves, [342](#)
 - semi-synchronous, [343](#)
 - setup and configuration, [348](#)
 - SHOW SLAVE STATUS, [343](#)
 - MySQL router, [342, 345](#)
 - MySQL Shell
 - command line options, [83, 84](#)
 - commands, [75, 77, 78](#)
 - using connections
 - individual options, [90](#)
 - scripts, [91](#)
 - SSL, [91, 92](#)
 - URI, [88, 89](#)
 - with database
 - create table, [97](#)
 - JSON_EXTRACT() function, [97, 99](#)
 - JSON_EXTRACT() function, [103](#)
 - JSON_PRETTY() function, [101](#)
 - JSON_VALID(), [100](#)
 - SELECT statement, [102](#)
 - selecting rows, [96, 97](#)
 - UPDATE command, [100](#)
 - installation, [76](#)
 - modes, [87, 88](#)
 - new features
 - adminAPI, [8](#)
 - changes, [22](#)
 - data dictionary, [18, 19](#)
 - SQL mode execution, [8](#)
 - user defined reports, [8](#)
 - options, [81, 82](#)
 - overview (*see* Overview, MySQL Shell)
 - pluggable password store, [109, 111](#)
 - sample database, [93–95](#)
 - session objects, [85–87](#)
 - sessions, [84, 85](#)
 - working
 - changing prompt, [116, 117](#)
 - code/command history, [108, 109](#)
 - option command, [111, 112](#)
 - Pluggable Password store, [115, 116](#)
 - set_perist() method, [113, 114](#)
 - shell.option object, [113](#)
 - using configuration file, [114](#)
 - using format, [104, 105, 107, 108](#)
- Mysqlx module
 - classes, [148, 149](#)
 - connection methods, [153](#)
 - get_session() method, [148](#)
 - miscellaneous methods, [153, 154](#)
 - schema class, [150, 151](#)
 - session class, [149, 150](#)
 - transaction methods, [151–153](#)

N

NoSQL, [24](#), [234](#)

O

Organizer class, [220](#)

Overview, MySQL Shell

API, [4](#)

JSON, [2](#)

sample commands, [5](#)

snapshot, [3](#)

SQL commands, [4](#)

P, Q

Paradigm shifting features

document store, [23](#), [24](#)

InnoDB Cluster, [25](#), [26](#)

replication, [24](#), [25](#)

Path expression

inline path operator, [248](#)

JSON array, [245](#), [247](#)

JSON document, [245](#), [246](#)

selectors, [246](#), [247](#)

X DevAPI classes, [245](#)

pip install flask-bootstrap command, [455](#)

pip install flask-script command, [454](#)

pip install wtforms command, [457](#)

Place class, [221](#)

Pluggable password store, [109](#), [115](#)

powertool table, [188–189](#), [221](#), [300](#)

Primary key, [128](#)

R

Read-only (RO) transactions, [344](#)

Read-write (RW) transactions, [344](#)

redirect() function, [479](#)

Relational database management system (RDBMS), [123](#)

Relational databases

commands, [122](#), [123](#)

DDL statements, [120–122](#)

DML statements, [120–122](#)

RDBMS, [123](#)

Relational data to document store

challenges, [293](#)

helper functions, [295–299](#)

populate collections

add() method, [300](#)

organizers collection, [302](#)

toolchests collection, [303–306](#)

tools collection, [300](#)

vendors collection, [299](#)

setup code, [294](#), [295](#)

Relay log, [342](#), [369](#)

render_template() function, [460](#), [470](#)

Replication, tutorial

data directory, [350](#), [351](#)

master configuration, [351–353](#)

sample data, [364](#)

server instances, [355](#), [356](#)

set up and configure, [349](#)

SHOW SLAVE HOSTS, [363](#)

SHOW SLAVE STATUS, [361](#), [363](#)

shut down, [365](#)

slaves configuration, [353–355](#)

slaves to master (*see* Slaves to master connection)

START SLAVE command, [360](#)

user account, [357](#)

Required() validator, [467](#)

rescan() method, [8](#), [436](#)

Resource Description Framework (RDF), [233](#)

Row-based replication (RBR), [342](#)

S

- Sandbox, 396
 - AdminAPI, 413
 - administration
 - restarting cluster, 448, 450
 - shutting down cluster, 448
 - tasks, 445–447
 - cluster class, 420
 - dba class, 414–416
 - MySQL Router
 - bootstrapping, 437, 438, 440
 - configuration, 440
 - connection test, 441–443
 - failover demonstration, 443, 444
 - setup and configuration
 - add_instance(), 428–430
 - check_instance_configuration()
 - method, 425
 - configure_local_instance()
 - method, 426
 - create_cluster() method, 426, 427
 - creating directory, 421, 422
 - dba.stop_sandbox_instance(), 427, 428
 - deploy_sandbox_instance()
 - method, 423, 425
 - failover demonstration, 432–435
 - rescan() method, 436
 - status() method, 430, 432
- Secure socket layer (SSL), 82
- SELECT command, 127
- SELECT SQL command, 364
- Selecting data, MYSQL
 - COUNT() function, 135
 - FROM clause, 132
 - functions, 137
 - GROUP BY clause, 135
 - JOIN operator, 132
 - ORDER BY clause, 135–137
 - SELECT statement, 132, 134
 - sensor values, 138
 - WHERE clause, 135
- Sessions, 84–87
- set_perist() method, 113
- shell.delete_all_credentials() method, 116
- shell.store_credential() method, 116
- show_collection() function, 300–302, 305
- SHOW MASTER STATUS command, 358
- SHOW SLAVE STATUS command, 343, 385
- SHOW SQL command, 351, 374
- Slaves to master connection
 - GTIDs
 - information, 359
 - MASTER_AUTO_POSITION, 360
 - log file and position
 - CHANGE MASTER command, 359
 - information, 358
- SQL database development
 - MyGarage, sample application, 179
 - handtool record, 179
 - Kobalt tool chest, 180, 181
 - storage portion, 180
 - Python, 195
 - sample database design (*see* Database design)
 - setup and configuration, 192, 193
- START GROUP_REPLICATION command, 383
- Statement-based replication (SBR), 342
- status() method, 430, 448
- Storage class, 222
- Storage engine, 16, 17
- Store_inventory database, 126
- Strong password encryption option, 41, 42
- Structured Query Language (SQL), 24, 119
- SubmitField() instances, 467
- Synchronous replication, 344, 345, 402

INDEX

T

Topology, [369](#)

U

update() method, [164](#), [214](#), [215](#)

Uptime, [338](#), [340](#)

url_for() function, [479](#)

USE <database> command, [126](#)

V

Vendor class

 create operation, [209-211](#)

 CRUD operations, [209](#)

 read operation

 create(), [213](#)

 delete operation, [216](#), [218](#)

 read() Method, [211-213](#)

 try...except block, [211](#)

 update operation, [214](#), [215](#)

 vendor_id, [211](#)

W

Web Ontology Language (OWL), [233](#)

Web Server Gateway Interface (WSGI), [458](#)

Windows firewall option, [41](#)

WTForms, [452](#), [457](#), [465](#)

 Field classes, [468](#)

X, Y, Z

X Developer API (X DevAPI), [15](#), [16](#), [234](#),
 [275](#), [279](#), [282](#), [403](#)

X Plugin, [15](#), [16](#)

X Protocol, [15](#), [16](#)