Frameworkless Front-End Development

Do You Control Your Dependencies Or Are They Controlling You?

Francesco Strazzullo

Apress®

Frameworkless Front-End Development

Do You Control Your Dependencies Or Are They Controlling You?

Francesco Strazzullo

Apress®

Frameworkless Front-End Development

Francesco Strazzullo
TREVISO, Treviso, Italy

ISBN-13 (pbk): 978-1-4842-4966-6　　ISBN-13 (electronic): 978-1-4842-4967-3
https://doi.org/10.1007/978-1-4842-4967-3

Managing Director, Apress media LLC: Welmoed Spahr
Acquisitions Editor: Louise Corrigan
Development Editor: James Markham
Coordinating Editor: Nancy Chen

Cover designed by eStudioCalamar

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/9781484249666. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To Mom and Dad

Table of Contents

About the Author

Francesco Strazzullo is an experienced front-end engineer, JavaScript trainer, and the co-founder of the Marca User Group (MUG). He has presented at tech conferences and meet-ups around Europe, is a technical reviewer for multiple tech publishers, and writes technical articles on his blog. He is always enthusiastic about trying out new APIs, and he is a firm believer that the best way to learn something new is to explain and teach it to somebody else. He co-founded the Frameworkless Movement, which is a group for those interested in developing software without using frameworks.

About the Technical Reviewer

Alexander Chinedu Nnakwue studied mechanical engineering at the University of Ibadan, Nigeria. He has been a front-end developer for more than three years, working on both web and mobile technologies. He has experience as a technical author, writer, and reviewer. He enjoys programming for the web, and occasionally, you can find him playing soccer. He is currently based in Lagos, Nigeria.

Acknowledgments

This book would not have existed without my girlfriend, Lucia. She encouraged me and listened patiently to all my rumblings about JavaScript, frameworks, and decision-making for almost a year. I am very grateful for her support and for all the happiness that she brings to my life.

This book was born while I drank coffee with my dear friend Lorenzo Massacci. During a coffee break at the office, he asked, “What does it take to make an application to last forever?” We never answered that question, but we started thinking about the lifespan of an application and its relationship with frameworks, and we started talking about the frameworkless approach.

Another important element in the making of this book is Flowing, the company that I work for. Working at Flowing made me a better developer by believing in my skills and helping me to achieve my ambitions.

Finally, I really need to thank Avanscoperta, which helped me prepare for this book. Their workshop gave me the opportunity to get feedback about the Frameworkless approach from the attendees, which gave me the courage to write down my thoughts.

Frameworkless Movement

This book has two main topics: how to effectively work without frameworks and how to choose the right framework for the right project. To better explore these topics, I created—with my colleagues Antonio Dell'Ava, Lorenzo Massacci, and Alessandro Violini—the Frameworkless Movement. The manifesto of this movement is published on our official web site (`http://frameworklessmovement.org`).

Frameworkless
movement

The purpose of this movement is to create awareness of frameworkless topics and to create a community to discuss them. One of our main concerns is helping people understand that to work without a framework is a now a real possibility. This book is one effort to help people understand the importance of technical decision-making.

We are organizing various events around Europe. If you're interested and want to be involved, contact us at `info@frameworklessmovement.org`.

CHAPTER 1

Let's Talk About Frameworks

You don't need a framework. You need a painting, not a frame.

—Klaus Kinski

So, *why should you read a book about effectively developing front-end applications without frameworks?* Because, sometimes, a framework is just not enough to fulfill your tasks. This book helps you understand the strategies for developing *frameworkless* applications, and more importantly, it shows you how to choose the right tool for the right job.

This chapter begins with my opinions on frameworks and why I believe that it's important to learn to live without them. After this short introduction, you start learning how to work without frameworks. I show you examples of rendering, routing, state management, and so on. After you learn how to go *frameworkless,* you should be able to figure out if it is the right choice for your project.

The last chapter in this book helps you decide which tool is best for you. I also talk about technical decision making and how to evaluate the trade-offs in every decision.

Now that you have an idea of what you can expect from this book, let's talk about frameworks.

F. Strazzullo, *Frameworkless Front-End Development*,
https://doi.org/10.1007/978-1-4842-4967-3_1

What Is a Framework?

Before going deeper, let's find a definition of *framework* that will guide us through the entire book. This is how the Cambridge Dictionary defines it:

A supporting structure around which something can be built.

This definition is consistent with the general idea of a software framework. If you think about the structure of an Angular application, it matches this definition exactly. Angular offers this structure with out-of-the-box elements like services, components, and pipe, around which you build your application.

In real-life applications, a stack contains other elements. You can use Lodash to manipulate arrays or objects, or Moment.js to parse dates.

Are these framework tools? The JavaScript community tends to call them *libraries*.

What's the difference between a library and a framework? I often use often the following definition during my presentations:

A framework calls your code. Your code calls a library.

A framework could internally use one or more libraries, but that fact is usually hidden to the developer that sees the framework as a single unit or a bunch of modules if you choose a modular framework. The relationship between your codebase, a framework, and a library is condensed in Figure 1-1.

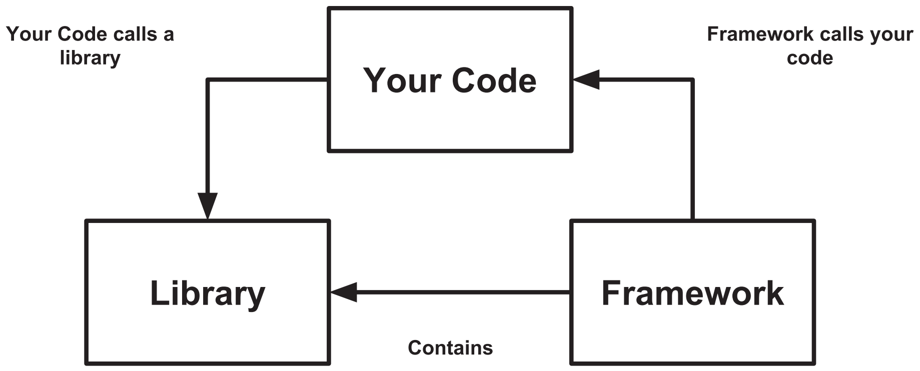


Figure 1-1. *Relationship between frameworks, libraries, and code*

Comparing Frameworks to Libraries

I am going to use some code snippets to show the difference between frameworks and libraries. For this comparison, I will use Angular and Moment.js.

Listing 1-1 and Listing 1-2 are basic examples of `Component` and `Service` in Angular.

Listing 1-1. Angular Service Example

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

const URL = 'http://example.api.com/';

@Injectable({
  providedIn: 'root',
})
export class PeopleService {
  constructor(private http: HttpClient) { }
  list() {
       return this.http.get(URL);
  }
}
```

Listing 1-2. Angular Component Example

```
import { Component, OnInit } from '@angular/core';
import { PeopleService } from '../people.service';

@Component({
  selector: 'people-list',
  templateUrl: './people-list.component.html'
})
export class PeopleListComponent implements OnInit {

  constructor(private peopleService: PeopleService) { }

  ngOnInit() {
    this.loadList();
  }

  loadList(): void {
    this.peopleService.getHeroes()
        .subscribe(people => this.people = people);
  }
}
```

Listing 1-3 is an example of using Moment.js to format the date.

Listing 1-3. Moment.js Example

```
import moment 'moment';

const DATE_FORMAT = 'DD/MM/YYYY';

export const formatDate = date => {
      return moment(date).format(DATE_FORMAT);
}
```

Given the previous definition, it's quite easy to understand that *Angular* is a framework, while Moment.*js* is a library (used to manipulate dates). In Angular, to let `PeopleListComponent` interact with `PeopleService`, you should use the `@Injectable` annotation and put the instance in the constructor. Angular offers a structure to fill with your code and a set of utilities (like HttpClient) to help with standard tasks.

Moment.js is completely unopinionated on how to structure your application code. You just import it and use it. As long as you respect the public API, you're good to go. Using this same definition, you can categorize a lot of your favorite *npm* packages. Frameworks include Angular, Vue.js, and Ember.js. A lot of libraries can be categorized by purpose, as seen in Table 1-1.

Table 1-1. *Some JavaScript Libraries*

Purpose	Libraries
Utilities	Lodash, Underscore.js
Date manipulation	Moment.js, date-fns
Data visualization	D3.js, Highcharts
Animations	Tween.js, Anime.js
HTTP requests	axios

I deliberately left out React—one of the most popular tools front-end developers—from this list. So, is React a library or a framework? Before answering this question, I want to introduce a new concept that will help shed some light on this topic: *the framework's way*.

The Framework's Way

As you saw in Listing 1-3, Moment.js has no opinion on how to integrate it into your code. Angular is very opinionated. You may see some of its strong ideas in the simple example shown in the previous section. The following sections discuss some of the constraints.

Language

Even if it is actually doable to build an Angular application with plain ECMAScript, TypeScript is the *de facto* standard in the Angular ecosystem. TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. Apart from type checking, it lets you use features that are not present in the original language, such as annotations.

TypeScript could be very useful if you are used to working with strongly typed languages. But if you use Angular, all of your code is written in a language that requires a transpiler.

Dependency Injection

To let elements communicate in an Angular application, you need to inject them with a dependency injection mechanism based on types. The old AngularJS had a dependency injection mechanism based on a service locator pattern. The same injection in AngularJS looks like Listing 1-4.

Listing 1-4. AngularJS Dependency Injection

```
const peopleListComponent = peopleService => {
       //Actual Code
};

angular.component('people-list',[
       'peopleService',
        peopleListComponent
]);
```

Later in the book, you will see how to create a very simple service locator to keep your code well organized.

Observables

Angular is heavily designed around RxJS, a library for reactive programming using observables. To get data from `PeopleListService`, you have to use the `subscribe` method of the `Observable` object. This approach is different from the other front-end frameworks, where HTTP requests are designed like promises. Promises are a standard way to represent the eventual completion (or failure) of an asynchronous operation. RxJS lets you easily transform an observable into a promise, and vice versa.

If you need to integrate a promise-based library in your Angular project, you need to do some extra work. Listing 1-5 and Listing 1-6 show how to use *axios*, a library for making HTTP requests based on promises.

Listing 1-5. Angular Service Without Observables

```
import axios from 'axios';
const URL = 'http://example.api.com/';

export default {
      list() {
             return axios.get(URL);
      }
}
```

Listing 1-6. Angular Component Without Observables

```
import people from 'people.js';

export class PeopleList {
      load(){
            people
                    .list()
                    .then(people => {
                           this.people = people
                    });
      }
}
```

Note `PeopleListComponent` is a class that uses the `people` service. It has no particular usefulness apart from showing you how to work with promises. I use to call all the constraints of a framework "the framework's way."

Apart from the constraints created by the core team of the framework, other constraints are part of the framework's way. For example, the *de facto* standards from the community are almost as important as the core ones.

In the AngularJS ecosystem, John Papa's style guide (`https://github.com/johnpapa/angular-styleguide/tree/master/a1`) was "the way" to write *AngularJS* applications. You were not enforced to use it, but most of the code that you read on the web was built that way.

Keep in mind that these constraints are neither bad nor good, but it's very important to analyze "the way" of the framework that a team chooses in order to assess if it's the right tool for the project.

Let's Talk About React

How is the framework's way related to knowing if React is a library or a framework? React is defined on its web site as "a JavaScript library for building user interfaces."

That sounds easy enough: React is a library. But the reality is far more complex than that. The main constraint of React is the usage of the declarative paradigm. You don't manipulate the DOM; instead, you modify the state of a component, and then React modifies the DOM for you. This way of programming is present in most of the libraries in the React ecosystem. Listing 1-7 is a very simple example of *Pose*, a library for animating React components. The purpose of this snippet is to show/hide a square by using an animation every time the user presses a Toggle button (see Figure 1-2).

Listing 1-7. React-Pose Animation Example

```
import React, { Component } from 'react';
import posed from 'react-pose';

const Box = posed.div({
  hidden: { opacity: 0 },
  visible: { opacity: 1 },
  transition: {
    ease: 'linear',
    duration: 500
  }
});

class PosedExample extends Component {
  constructor (props) {
    super(props)
    this.state = {
      isVisible: true
    }

    this.toggle = this.toggle.bind(this)
  }

  toggle () {
    this.setState({
      isVisible: !this.state.isVisible
    })
  }

  render () {
    const { isVisible } = this.state
    const pose = isVisible ? 'visible' : 'hidden'
    return (
      <div>
        <Box className='box' pose={pose} />
        <button onClick={this.toggle}>Toggle</button>
      </div>
    )
  }
}

export default PosedExample
```

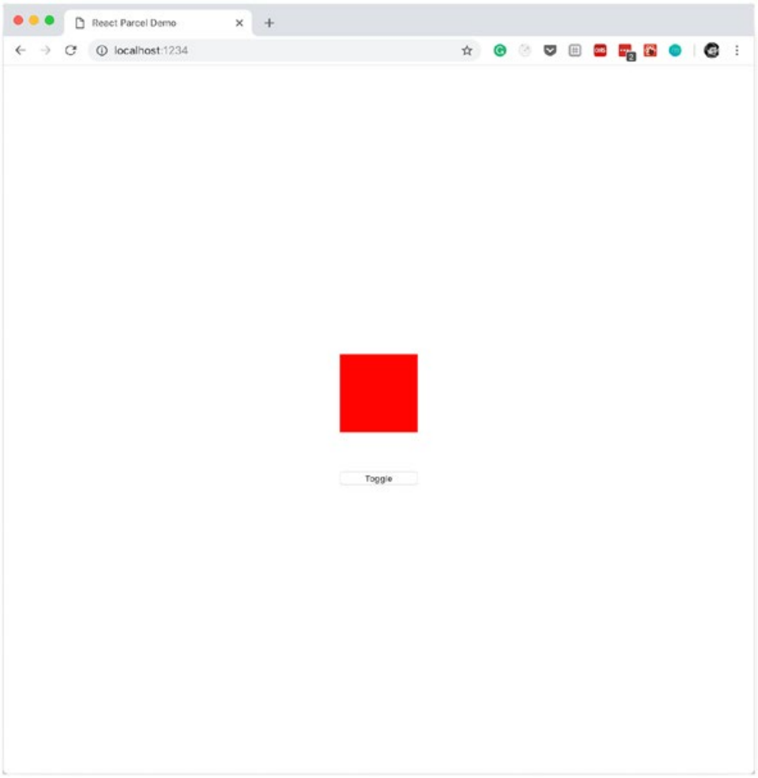


Figure 1-2. *Example of React animation using Pose*

As you can see in Figure 1-2, you don't directly animate the square. You just declare how to map the state with the animation (visible or hidden), and then change the state. This is the core of the *declarative pattern* used in *React*.

Listing 1-8 produces the same output, but it's based on a standard API called the Web Animations API (`https://developer.mozilla.org/en-US/docs/Web/API/Web_Animations_API`).

Listing 1-8. React Animation with Web Animations API

```
import React, { Component } from 'react'

const animationTiming = {
  duration: 500,
  ease: 'linear',
  fill: 'forwards'
}

const showKeyframes = [
  { opacity: 0 },
  { opacity: 1 }
]

const hideKeyframes = [
  ...showKeyframes
].reverse()

class PosedExample extends Component {
  constructor (props) {
    super(props)
    this.state = {
      isVisible: true
    }

    this.toggle = this.toggle.bind(this)
  }

  toggle () {
    this.setState({
      isVisible: !this.state.isVisible
    })
  }
  componentDidUpdate (prevProps, prevState) {
    const { isVisible } = this.state
    if (prevState.isVisible !== isVisible) {
      const animation = isVisible ? showKeyframes :
      hideKeyframes
      this.div.animate(animation, animationTiming)
    }
  }

  render () {
    return (
      <div>
        <div ref={div => { this.div = div }} className='box' />
        <button onClick={this.toggle}>Toggle</button>
      </div>
    )
  }
}

export default PosedExample
```

If you're a React developer, this second example may seem out of place. This is because you're moving the square with an imperative pattern. This "strangeness" is why I believe that React is a framework and not just a library. I believe this not because of its code, but because of the constraints that the community accepted by using it. In other words, the declarative pattern is a part of React's way.

Tip Where there is a "framework's way" of doing things, there is a framework.

Brief History of JavaScript Frameworks

This section is a very brief history of front-end frameworks. It's not meant to be comprehensive, but it's an opinionated view of the most important milestones in the front-end ecosystem.

jQuery

Created by John Resig in 2006, jQuery is the mother of all JavaScript frameworks. By far, it is the most-used framework in production, as you can see at `http://libscore.com/#libs`. The most important feature of jQuery is its famous selector syntax: `var element = $('.my-class')`.

It may seem useless today but you have to consider that in 2006, browsers were not aligned as they are today. This is the real value that jQuery brought to the front-end world. jQuery created a lingua franca between the browsers. It helped the community to grow around a common ground. In addition to the selector syntax, a lot of features were added to the core project, such as AJAX requests, animations, and other utilities. It rapidly became the Swiss Army knife of front-end development.

jQuery has an official UIKit called jQueryUI, which is easily pluggable, and so the web is full of plugins for every need. Today, front-end developers tend to joke about jQuery, but it's been a cornerstone of modern web development.

AngularJS

If jQuery can be seen as the invention of writing, AngularJS is probably the equivalent of Gutenberg's printing press. AngularJS was originally developed in 2009 by Miško Hevery as a side project; later, he became a Google employee. For this reason, AngularJS is actually maintained by Google engineers. The 1.0 version was released in May 2011. AngularJS was hugely successful in making single-page applications a mainstream pattern.

The most notable feature is two-way data binding. You can see an example of this characteristic in Listing 1-9; in this case, I used `ng-model`, which is probably the most famous AngularJS directive.

Listing 1-9. AngularJS Two-Way Data Binding Example

```
<div ng-app="app" ng-controller="ctrl">
    Value: <input ng-model="value">
    <h1>You entered: {{value}}</h1>
</div>
<script>
    angular
        .module('app', [])
        .controller('ctrl', [
            '$scope',
            $scope => {
                $scope.value = 'initial value'
            }
        ]);
</script>
```

The core of this mechanism is the `$scope` object. Every change to `$scope` is automatically applied to the DOM. Events from the input produces new values in the `$scope` object. You can see a schema of the two-way data binding system in Figure 1-3.

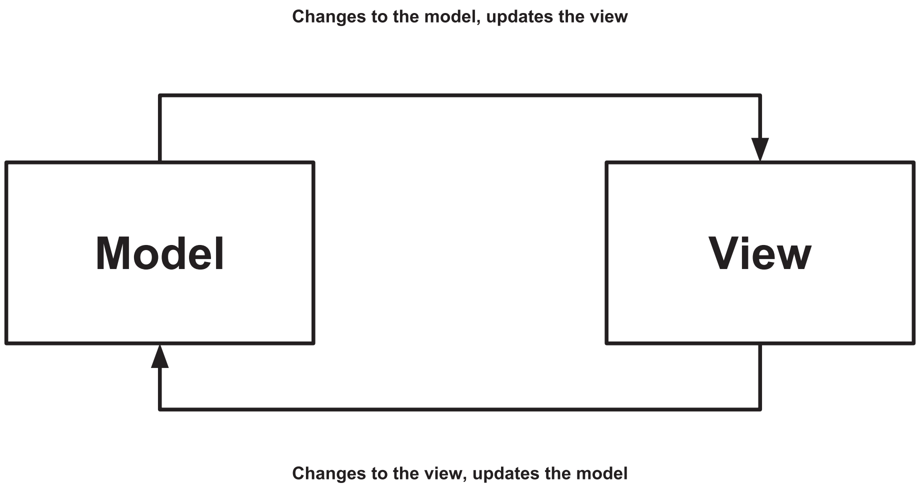

Changes to the model, updates the view
Model
View
Changes to the view, updates the model

Figure 1-3. *Two-way data binding schema*

Two-way data binding lets developers quickly create web applications. Over time, however, a lot of developers left AngularJS because two-way data binding is not suitable for large applications. In any case, AngularJS has the merit of introducing a lot of developers to the front-end ecosystem.

React

Created by Facebook in 2011 and open sourced in 2013, React is currently the most popular front-end library (or "framework"). Let's look at the code of a `Timer` component that simply renders the number of seconds elapsed from its first rendering (see Listing 1-10).

Listing 1-10. Basic React Component with Some Lifecycle Methods

```
import React, { Component } from 'react'
import { render } from 'react-dom'

class Timer extends Component {
    constructor(props){
        super(props)
        this.state = {
            seconds: 0
        }
    }

    componentDidMount() {
        this.interval = setInterval(() => {
            const { seconds } = this.state
            this.setState({
                seconds: seconds + 1
            })
        },1000)
    }

    componentWillUnmount() {
        clearInterval(this.interval)
    }

    render(){
        const { seconds } = this.state
        return (
            <div>
                Seconds Elapsed: {seconds}
            </div>
        )
    }
}

const mountNode = document.getElementById('app')

render(<Timer></Timer>, mountNode)
```

React works with a declarative paradigm. Usually, you don't modify the DOM directly; instead, you change the state with the `setState` method and let React do the rest.

Technically, React is a rendering library and not a framework. This fact allows the front-end community to fill in the gaps with a lot of very interesting ideas, especially for state management. I talk about some of these libraries, including Redux and MobX, in Chapter 7.

Angular

Angular was previously known as Angular2 because the project was intended to be a new version of AngularJS. The team behind the project took the semantic versioning very seriously, thus Angular2 became a completely different framework. Such a different approach between the two versions caused a period of panic around the project. After the first release of Angular2 in September 2016, the team decided to rename the project Angular, probably due to a planned release cycle featuring a new major version every six months.

Angular tried to appeal to the corporate world. A lot of corporations developed single-page applications with AngularJS, but the tool was not designed for very large applications. The fact that TypeScript is the de facto standard for working with Angular helped a lot of Java and C# developers start developing front-end applications.

Technical Debt

When you need to add a feature to a project, you always have a range of options. Some of them are quick and messy, while others are well designed but slower to put in production. In order to better understand the impact of this kind of decision, Ward Cunningham created the concept of *technical debt* (http://wiki.c2.com/?WardExplainsDebtMetaphor). The metaphor itself is quite simple: every time that you choose the dirty solution, you incur a debt.

As you can see in Figure 1-4, if you start to incur debt, the cost of a new feature or to change an existing feature increases exponentially over time, similar to financial debt, which, if not paid, increases due to interest.

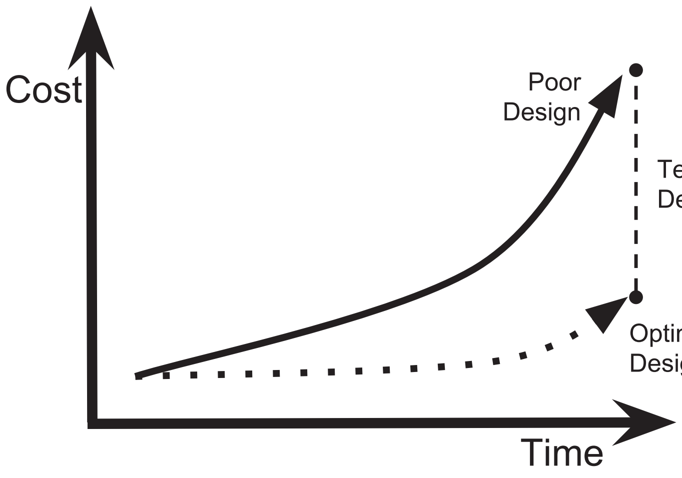


Figure 1-4. *Technical debt*

The Cost of Frameworks

Why did I dedicate a section to technical debt? Because I firmly think that every framework has technical debt. I know that this is a very strong statement, but let's think again about the debt metaphor. You start incurring debt when you choose a path that is not optimal in order to gain something else, usually velocity. My point is that someone else's code is not optimal for solving my exact problem. Of course, I'm talking about an ideal world where in every new project, a team has the time to build everything from scratch. I would be naive to think that I will work with plain JavaScript in every project. But I would be equally naive to think that frameworks are free of charge. Every framework has a cost in terms of the difficulty to change the code in the future. The cost is the fact that a framework imposes its architecture on your code. Over time, software needs change, because of the market or other factors, and our architecture should change too. But most of the time, a framework is a roadblock in this sort of change.

Technical Investment

At the beginning of this chapter, I stated that this book is not against frameworks. This statement seems in contrast with the idea that every framework has technical debt. You are probably wondering how frameworks can be a good thing if they always have technical debt. Technical debt is not always a bad thing. In the financial world, debt is not automatically a bad thing. For example, to buy a house, you usually need a loan, which is debt. But people tend to not consider a loan a bad thing, but an investment. On the other hand, if a friend without a stable job wants to go to a bank to get a loan to buy a Ferrari, you will probably try to stop him. The difference is not in the debt itself but in the reason behind the debt.

In software development, there is the same kind of mechanism. If we use a quick solution for a good reason, it is not technical debt, it is a *technical investment*. An investment is a type of debt, but it is not reckless. And a framework, when chosen for a good reason, is not a *cost* but an *asset*. In Chapter 8, I will show you some of the techniques that I use as a consultant to understand if a framework is an asset for a project, and how to choose the framework that "costs" less.

Summary

This chapter defined a "framework" and explained how it is different from a library. I discussed my personal history of JavaScript frameworks, and I pointed out values to the front-end ecosystem. Lastly, you learned the meaning of technical debt and its relationship with frameworks.

In the next chapter, I talk about rendering and the basic principles behind DOM manipulation.

CHAPTER 2

Rendering

One of the most important features of any web application is displaying data. On a more “close to the metal” level, displaying data means **rendering elements to the screen** or another output device. The way that the World Wide Web Consortium (W3C) defines rendering elements programmatically is with the *Document Object Model*, also known as DOM. The purpose of this chapter is to learn how to effectively manipulate DOMs without frameworks.

The Document Object Model

The DOM is an API that lets you manipulate the elements that make up a web application. You can read more about it at the W3C specification page (`https://www.w3.org/TR/1998/WD-DOM-19980720/introduction.html`).

To better understand what the DOM is, let’s go back to the basics. From a technical standpoint, every HTML page (or a fragment of it) is a tree. The HTML table in Listing 2-1 has its DOM representation shown in Figure 2-1.

Listing 2-1. Simple HTML Table

```
<html>
<body>
  <table>
    <tr>
      <th>Framework</th>
      <th>GitHub Stars</th>
    </tr>
    <tr>
      <td>Vue</td>
      <td>118917</td>
    </tr>
    <tr>
      <td>React</td>
      <td>115392</td>
    </tr>
  </table>
</body>
</html>
```

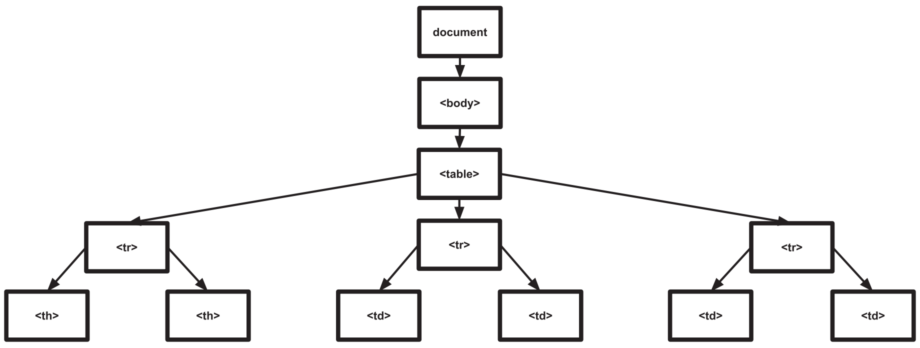


Figure 2-1. *DOM representation of a table*

This example makes it clear that the DOM is a way to manage the tree defined by your HTML elements. So if you want to change the background color of a React cell, you can write something like Listing 2-2.

Listing 2-2. Changing the Color of a React Cell

```
const SELECTOR = 'tr:nth-child(3) > td'
const cell =  document.querySelector(SELECTOR)
cell.style.backgroundColor = 'red'
```

The code is straightforward. You select the right cell with the `querySelector` method using a standard CSS selector, and then change the `style` property of the cell node. The `querySelector` method is a Node method. Node is the basic interface that represents a node in your HTML tree. You can read about all of its methods and properties on the Mozilla Developer Network page (`https://developer.mozilla.org/en-US/docs/Web/API/Node`).

Monitoring Rendering Performance

When designing a rendering engine for the web, you should keep in mind *readability* and *maintainability*. Rendering is a very important task in any web application; if you decide to write it from scratch, it should be very easy to understand and to evolve.

Another important factor in a rendering engine is performance. In the next section, I show you a bunch of tools to monitor the performance of your rendering engine.

Chrome Developer Tools

The first tool that you are going to use is a browser, specifically Chrome and its well-known developer tools. One of the features that you can use to monitor rendering performances is a handy frames-per-second (FPS) meter. Open Chrome DevTools and press Cmd/Ctrl+Shift+P to show the Command menu. Next, choose the “Show frame per seconds (FPS) meter” menu item. You can see the Command Menu in Figure 2-2.

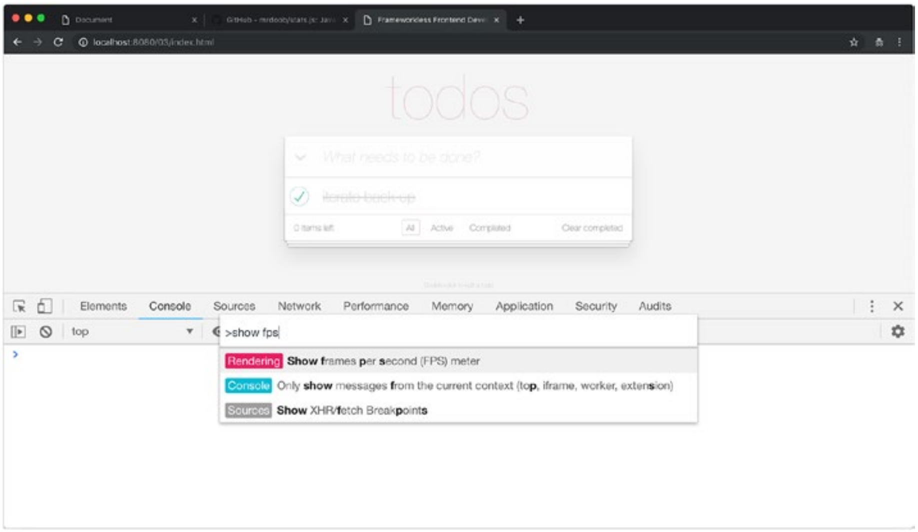

localhost:8080/03/index.html
todos
What needs to be done?
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Audits
top
>show fps
Rendering Show frames per second (FPS) meter
Console Only show messages from the current context (top, iframe, worker, extension)
Sources Show XHR/fetch Breakpoints

Figure 2-2. *Chrome’s Command menu*

The FPS meter shows up in the upper-right corner of the screen. It displays the amount of memory used by the GPU, as you can see in Figure 2-3.

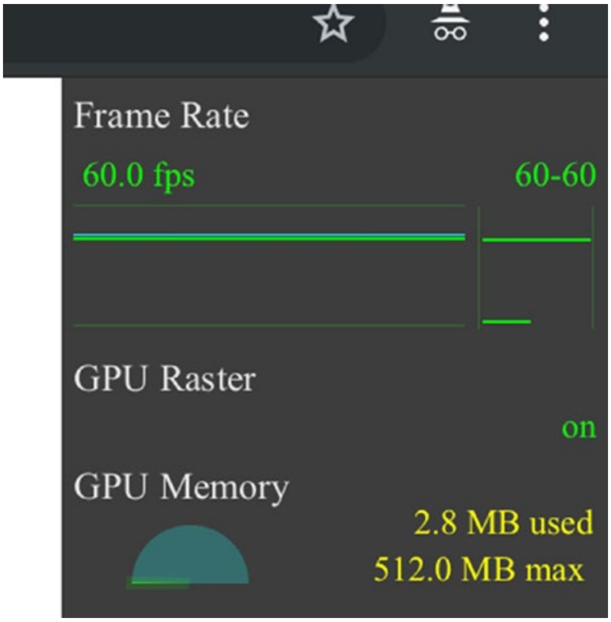


Figure 2-3. *Chrome FPS meter*

stats.js

Another way to monitor the FPS of your application is to use stats.js (`https://github.com/mrdoob/stats.js/`), a very simple library that is easy to embed in any web application. This tool can also display the milliseconds needed to render a frame and the megabytes of allocated memory. On the Readme page of the GitHub repository, you find a simple bookmarklet to attach the widget to *any* web site, like the one shown in Figure 2-4.

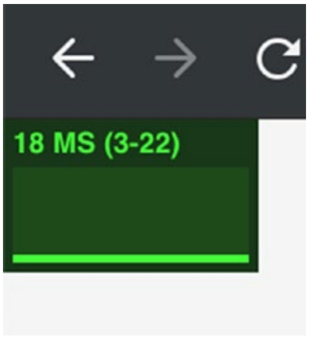


Figure 2-4. *stats.js widget showing milliseconds needed to render a frame*

Custom Performance Widget

Creating a widget that shows your application's FPS is easy. The main concept is to use the `requestAnimationFrame` callback to track the time between a render cycle and the next one, and to keep track of the number of times the callback is invoked in a second. You can see an example widget in Listing 2-3.

Listing 2-3. Custom Performance Monitor Widget

```
let panel
let start
let frames = 0

const create = () => {
  const div = document.createElement('div')

  div.style.position = 'fixed'
  div.style.left = '0px'
  div.style.top = '0px'
  div.style.width = '50px'
  div.style.height = '50px'
  div.style.backgroundColor = 'black'
  div.style.color = 'white'

  return div
}

const tick = () => {
  frames++
  const now = window.performance.now()
  if (now >= start + 1000) {
    panel.innerText = frames
    frames = 0
    start = now
  }
  window.requestAnimationFrame(tick)
}

const init = (parent = document.body) => {
  panel = create()

  window.requestAnimationFrame(() => {
    start = window.performance.now()
    parent.appendChild(panel)
    tick()
  })
}

export default {
  init
}
```

After you calculate the FPS, you can display the number on a widget, as in this case, or use a console to print the data.

Rendering Functions

We are going to analyze various ways to render elements to the DOM with pure functions. Rendering elements with pure functions means that the DOM elements depend exclusively on the state of the application. To grasp this concept from a more formal point of view, take a look at Figure 2-5.

$$view = f(state)$$

Figure 2-5. *A mathematical representation of pure functions rendering*

I better define what the "state" of your application is and explain how to manage it in Chapter 7.

Using pure functions has many advantages, such as testability or composability, but as you will see later in this chapter, there also some challenges.

TodoMVC

As a base for our example in this chapter, we are going to use a TodoMVC template. TodoMVC (`http://todomvc.com`) is a project that collects implementation of the same to-do list written with different frameworks. You can see a live demo of a TodoMVC implementation at `http://todomvc.com/examples/react/#/`. Figure 2-6 shows a standard TodoMVC application.

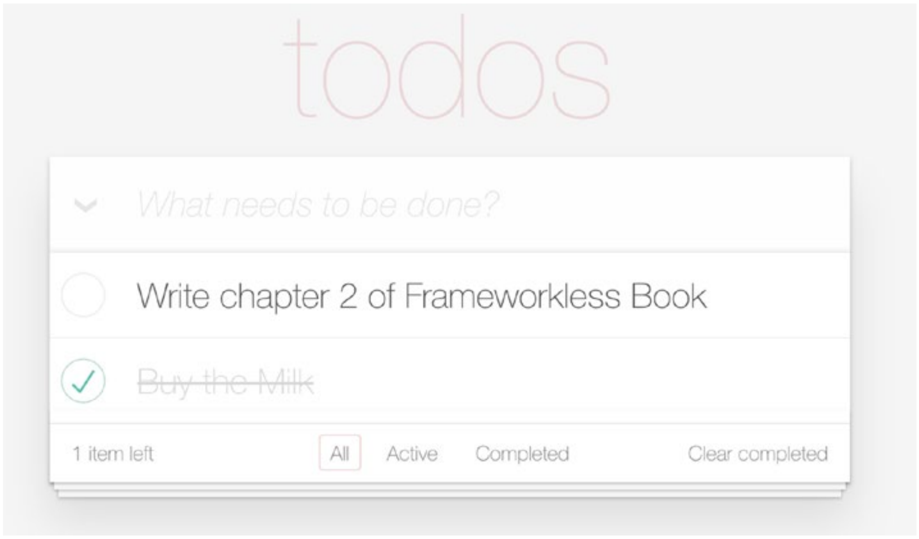


Figure 2-6. *TodoMVC example*

For now, we are going to concentrate on rendering. We are going to render the items and the toolbar. In later chapters, we will add other elements, such as HTTP requests, event handling, and so on, until we create a complete application.

Pure Functions Rendering

In our first example, we are going to use strings to render elements. You can see the skeleton of a TodoMVC application in the next snippet. You can see the complete code of this example at (`https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter02/01`). Listing 2-4 shows the content of our `index.html`.

Listing 2-4. Basic TodoMVC App Structure

```
<body>
  <section class="todoapp">
    <header class="header">
      <h1>todos</h1>
      <input
        class="new-todo"
        placeholder="What needs to be done?"
        autofocus="">
    </header>
    <section class="main">
       <input
         id="toggle-all"
         class="toggle-all"
         type="checkbox">
       <label for="toggle-all">
          Mark all as complete
       </label>
       <ul class="todo-list"></ul>
    </section>
    <footer class="footer">
      <span class="todo-count"></span>
      <ul class="filters">
        <li>
          <a href="#/">All</a>
        </li>
        <li>
          <a href="#/active">Active</a>
        </li>
        <li>
          <a href="#/completed">Completed</a>
        </li>
      </ul>
      <button class="clear-completed">
        Clear completed
      </button>
    </footer>
  </section>
  <footer class="info">
    <p>Double-click to edit a todo</p>
  </footer>
</body>
```

To make this application *dynamic*, we need to grab the to-do list data and update the following:

- the `ul` with the list of filtered todos
- the `span` with the number of not completed todos
- the links with filter types, adding the `'selected'` class to the right one

Listing 2-5 is our first attempt of functional rendering.

Listing 2-5. The First Version of a TodoMVC Rendering Function

```
const getTodoElement = todo => {
  const {
    text,
    completed
  } = todo
  return `
  <li ${completed ? 'class="completed"' : "}>
    <div class="view">
      <input
        ${completed ? 'checked' : "}
        class="toggle"
        type="checkbox">
      <label>${text}</label>
      <button class="destroy"></button>
    </div>
    <input class="edit" value="${text}">
  </li>`
}

const getTodoCount = todos => {
  const notCompleted = todos
    .filter(todo => !todo.completed)

  const { length } = notCompleted
  if (length === 1) {
    return '1 Item left'
  }

  return `${length} Items left`
}

export default (targetElement, state) => {
  const {
    currentFilter,
    todos
  } = state

  const element = targetElement.cloneNode(true)
  const list = element.querySelector('.todo-list')
  const counter = element.querySelector('.todo-count')
  const filters = element.querySelector('.filters')

  list.innerHTML = todos.map(getTodoElement).join('')
  counter.textContent = getTodoCount(todos)

  Array
    .from(filters.querySelectorAll('li a'))
    .forEach(a => {
      if (a.textContent === currentFilter) {
        a.classList.add('selected')
      } else {
        a.classList.remove('selected')
      }
    })

  return element
}
```

This view function takes a target DOM element used as a base. It then clones the original node and updates it using the state parameter. It then returns the new node. Notice that these DOM modifications are *virtual*; we are working with a *detached* element. To create a detached element, we clone an existing node with the `cloneNode` method. This newly created DOM element is an exact clone of a real DOM element, but completely unrelated from the body of the document.

No real modifications to the DOM were committed in Listing 2-5. Keep in mind that modifying a detached DOM element is performant. To connect this view function to the real DOM, you can use a simple controller, such as the one in Listing 2-6.

Listing 2-6. Basic Controller

```
import getTodos from './getTodos.js'
import view from './view.js'

const state = {
  todos: getTodos(),
  currentFilter: 'All'
}

const main = document.querySelector('.todoapp')

window.requestAnimationFrame(() => {
  const newMain = view(main, state)
  main.replaceWith(newMain)
})
```

Our simple "rendering engine" is based on `requestAnimationFrame` (`https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame`). Every DOM manipulation, or animation, should be based on this DOM API. Making DOM operations inside this callback makes everything more efficient; they don't block the main thread, and they are executed right before the next repaint is scheduled in the event loop. To better understand how the event loop works, I suggest watching the talk by Jake Archibald at `https://vimeo.com/254947206`.

Our data model is a random array generated with Faker.js (`https://github.com/marak/Faker.js/`), a small library useful for generating random data. In Figure 2-7, you can see the schema of our first rendering example.

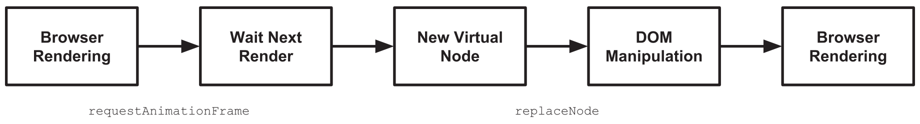


Figure 2-7. *Static Rendering schema*

Let's Review Our Code

Our rendering approach is performant enough using `requestAnimationFrame` and a virtual node manipulation. But our view function is not very readable. There are two major problems in the code.

- **It's a single, huge function**. We have only one function to manipulate different DOM elements. The situation can become messy very easily.
- **There are different approaches to do the same thing**. We create list items via strings. For the todo count element, we simply add the test to an existing element. For the filters, we manage `classList`.

In the next example, we will divide the view into smaller functions and try to address the consistency problem. You can find the complete code for this second version at `https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter02/02`.

Listing 2-7 shows the refactored version of our application, while Listings 2-8, 2-9, and 2-10 show the new functions for the counter, the filters, and the list.

Listing 2-7. App View Function with Smaller View Functions

```
import todosView from './todos.js'
import counterView from './counter.js'
import filtersView from './filters.js'

export default (targetElement, state) => {
  const element = targetElement.cloneNode(true)

  const list = element
    .querySelector('.todo-list')
  const counter = element
    .querySelector('.todo-count')
  const filters = element
    .querySelector('.filters')

  list.replaceWith(todosView(list, state))
  counter.replaceWith(counterView(counter, state))
  filters.replaceWith(filtersView(filters, state))

  return element
}
```

Listing 2-8. View Function to Show the todos Count

```
const getTodoCount = todos => {
  const notCompleted = todos
    .filter(todo => !todo.completed)

  const { length } = notCompleted
  if (length === 1) {
    return '1 Item left'
  }

  return `${length} Items left`
}

export default (targetElement, { todos }) => {
  const newCounter = targetElement.cloneNode(true)
  newCounter.textContent = getTodoCount(todos)
  return newCounter
}
```

Listing 2-9. View Function to Render the TodoMVC Filters

```
export default (targetElement, { currentFilter }) => {
  const newCounter = targetElement.cloneNode(true)
  Array
    .from(newCounter.querySelectorAll('li a'))
    .forEach(a => {
      if (a.textContent === currentFilter) {
        a.classList.add('selected')
      } else {
        a.classList.remove('selected')
      }
    })
  return newCounter
}
```

Listing 2-10. View Function to Render the List

```
const getTodoElement = todo => {
  const {
    text,
    completed
  } = todo

  return `
      <li ${completed ? 'class="completed"' : "}>
        <div class="view">
          <input
            ${completed ? 'checked' : "}
            class="toggle"
            type="checkbox">
          <label>${text}</label>
          <button class="destroy"></button>
        </div>
        <input class="edit" value="${text}">
      </li>`
}
```

```
export default (targetElement, { todos }) => {
  const newTodoList = targetElement.cloneNode(true)
  const todosElements = todos
    .map(getTodoElement)
    .join('')
  newTodoList.innerHTML = todosElements
  return newTodoList
}
```

Our code is way better now. We have three separate functions with the same signature. These functions are our first draft of a **component library**.

Component Functions

If you check the code of the app view (see Listing 2-7), we need to *manually* invoke the right function. If we want to create a component-based application, we should use a declarative way of interaction between the components. The system should automatically wire all the pieces.

The next application, which is hosted at https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter02/03, is an example of a rendering engine with a component registry. To achieve this goal, the first thing that should be done is to define how we can declare which component should be used in a particular use case. In our scenario, we have three different components: todos, counters, and filters. In Listing 2-11, you see how to determine which component should be used by using *data attributes* (https://developer.mozilla.org/en-US/docs/Learn/HTML/Howto/Use_data_attributes).

Listing 2-11. App Using Data Attributes to Use Components

```
<section class="todoapp">
    <header class="header">
        <h1>todos</h1>
    <input
        class="new-todo"
        placeholder="What needs to be done?"
        autofocus>
</header>
<section class="main">
    <input
        id="toggle-all"
        class="toggle-all"
        type="checkbox">
    <label for="toggle-all">
        Mark all as complete
    </label>
    <ul class="todo-list" data-component="todos">
    </ul>
</section>
<footer class="footer">
    <span
        class="todo-count"
        data-component="counter">
            1 Item Left
    </span>
    <ul class="filters" data-component="filters">
        <li>
            <a href="#/">All</a>
        </li>
        <li>
            <a href="#/active">Active</a>
        </li>
```

```
            <li>
                <a href="#/completed">Completed</a>
            </li>
        </ul>
        <button class="clear-completed">
            Clear completed
        </button>
    </footer>
</section>
```

In Listing 2-11, we put the “name” of the component in the `data-component` attribute. This attribute replaces the imperative invocation of view functions. Another requisite to create a component library is a registry, which is an index of all the components available in the application. The simplest registry that we can implement is a plain JavaScript object, like the one in Listing 2-12.

Listing 2-12. Simple Component Registry

```
const registry = {
  'todos': todosView,
  'counter': counterView,
  'filters': filtersView
}
```

The keys of our registry tally with the value of the `data-component` attribute. This is the key mechanism of our component-based rendering engine. This mechanism should be applied not only for the root container (our application view function) but also for every component that we will create. This way, every component can be used inside other components also. This kind of reusability is required for every component-based application.

To accomplish this task, every component should *inherit* from a base component that reads the values of the data-component attribute and automatically invokes the right function. Given that we are talking about pure functions, we can't really inherit from a base object. So we need to create a high-order function that wraps our components. An example of this high-order function is shown in Listing 2-13.

Listing 2-13. Rendering a High-Order Function

```
const renderWrapper = component => {
  return (targetElement, state) => {
    const element = component(targetElement, state)

    const childComponents = element
      .querySelectorAll('[data-component]')

    Array
      .from(childComponents)
      .forEach(target => {
        const name = target
          .dataset
          .component

        const child = registry[name]
        if (!child) {
          return
        }

        target.replaceWith(child(target, state))
      })

    return element
  }
}
```

This wrapper function takes the original component and returns a new component with the same signature. The two functions are identical in the system. The wrapper looks for every DOM element with the `data-component` attribute in the registry; when it finds something, it invokes the child component. But this child component is wrapped with the same function. This way, we can easily navigate all the way down to the last component, just like a recursive function.

To add a component to our registry, we need a simple function that wraps a component with the previous function, like the one in Listing 2-14.

Listing 2-14. Registry Accessor Method

```
const add = (name, component) => {
  registry[name] = renderWrapper(component)
}
```

We should also provide a method to render the root of our application to start the rendering from an initial DOM element. In our application, I called this method `renderRoot`, and you can see the code in Listing 2-15.

Listing 2-15. Boot Function of Component-Based Application

```
const renderRoot = (root, state) => {
  const cloneComponent = root => {
    return root.cloneNode(true)
  }

  return renderWrapper(cloneComponent)(root, state)
}
```

The `add` and `renderRoot` methods are the public interface of our component registry.

The last thing to do is mix all the elements together in the controller, which you can analyze in Listing 2-16.

Listing 2-16. A controller That Uses a Component Registry

```
import getTodos from './getTodos.js'
import todosView from './view/todos.js'
import counterView from './view/counter.js'
import filtersView from './view/filters.js'

import registry from './registry.js'

registry.add('todos', todosView)
registry.add('counter', counterView)
registry.add('filters', filtersView)

const state = {
  todos: getTodos(),
  currentFilter: 'All'
}

window.requestAnimationFrame(() => {
  const main = document.querySelector('.todoapp')
  const newMain = registry.renderRoot(main, state)
  main.replaceWith(newMain)
})
```

That's it! You just created your first frameworkless component-based application. You can consider it a *walking skeleton* (https://gojko.net/2014/06/09/forget-the-walking-skeleton-put-it-on-crutches/) of a real, component-based application. You can see a basic schema of the application in Figure 2-8.

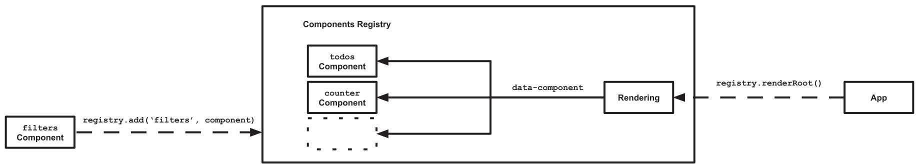


Figure 2-8. *Component registry schema*

Rendering Dynamic Data

In the previous examples, we used static data. But in a real-world application, data changes due to events from the user or the system. I talk about event listeners in the next chapter, but for now, let's change our state randomly every five seconds, as seen in Listing 2-17.

Listing 2-17. Rendering Random Data Every Five Seconds

```
const render = () => {
  window.requestAnimationFrame(() => {
    const main = document.querySelector('.todoapp')
    const newMain = registry.renderRoot(main, state)
    main.replaceWith(newMain)
  })
}

window.setInterval(() => {
  state.todos = getTodos()
  render()
}, 5000)

render()
```

Every time that we have new data, we create another virtual root element, and then replace the real one with the newly created one. This approach could be performant enough for a small application, but it could be a *performance killer* in a non-trivial project.

The Virtual DOM

The virtual DOM concept, made famous by React, is a way to make a declarative rendering engine a performant one. The representation of a UI is kept in memory and synced with the "real" DOM, which does the least number of operations possible. This process is called *reconciliation*. As an example, let's imagine that your "old" real DOM element is the following simple list.

```
<ul>
  <li>First Item</li>
</ul>
```

You want to replace it with a list with a new element, like the following one.

```
<ul>
  <li>First Item</li>
  <li>Second Item</li>
</ul>
```

In our previous algorithm, we replaced the entire `ul`. With the virtual DOM method, the system should dynamically understand that the only operation that is needed on the real DOM is the addition of the last `li`. The core of the virtual DOM is a diff algorithm that easily understands the fastest way to turn the real DOM into an exact copy of the new DOM element that is detached (in other words, virtual) from the document. A visual explanation of this mechanism is shown in Figure 2-9.

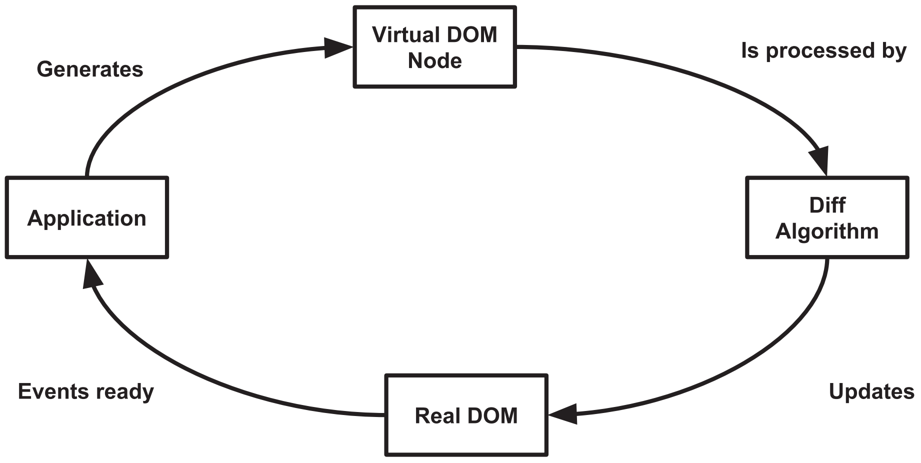


Figure 2-9. *Virtual DOM*

A Simple Virtual DOM Implementation

We are going to create a very simple diff algorithm, which we will use instead of `replaceWith` in our main controller (see Listing 2-18).

Listing 2-18. The Main Controller Using a diff Algorithm

```
const render = () => {
  window.requestAnimationFrame(() => {
    const main = document.querySelector('.todoapp')
    const newMain = registry.renderRoot(main, state)
    applyDiff(document.body, main, newMain)
  })
}
```

The `applyDiff` function parameters are the parent of the current real DOM node, the real DOM node, and the new virtual DOM node. Let's analyze what this function should do.

First, we need to remove the real node if the new node is not defined.

```
if (realNode && !virtualNode) {
  realNode.remove()
}
```

If the real node is not defined but the virtual one exists, we should add it to the parent node.

```
if (!realNode && virtualNode) {
  parentNode.appendChild(virtualNode)
}
```

If both nodes are defined, we need to check if there are any differences between them.

```
if (isNodeChanged(virtualNode, realNode)) {
  realNode.replaceWith(virtualNode)
}
```

We are going to analyze the isNodeChanged function code in a moment. But first, we need to apply the same diff algorithm for every child node.

```
const realChildren = Array.from(realNode.children)
const virtualChildren = Array.from(virtualNode.children)

const max = Math.max(
  realChildren.length,
  virtualChildren.length
)
for (let i = 0; i < max; i++) {
  applyDiff(
    realNode,
    realChildren[i],
    virtualChildren[i]
  )
}
```

You can see the complete code of the applyDiff function in Listing 2-19. Listing 2-20 shows the code for the isNodeChanged function.

Listing 2-19. applyDiff Function

```
const applyDiff = (
  parentNode,
  realNode,
  virtualNode) => {
  if (realNode && !virtualNode) {
    realNode.remove()
    return
  }
  if (!realNode && virtualNode) {
    parentNode.appendChild(virtualNode)
    return
  }

  if (isNodeChanged(virtualNode, realNode)) {
    realNode.replaceWith(virtualNode)
    return
  }

  const realChildren = Array.from(realNode.children)
  const virtualChildren = Array.from(virtualNode.children)

  const max = Math.max(
    realChildren.length,
    virtualChildren.length
  )
  for (let i = 0; i < max; i++) {
    applyDiff(
      realNode,
      realChildren[i],
      virtualChildren[i]
    )
  }
}
```

Listing 2-20. isNodeChanged Function

```
const isNodeChanged = (node1, node2) => {
  const n1Attributes = node1.attributes
  const n2Attributes = node2.attributes
  if (n1Attributes.length !== n2Attributes.length) {
    return true
  }
  const differentAttribute = Array
    .from(n1Attributes)
    .find(attribute => {
      const { name } = attribute
      const attribute1 = node1
        .getAttribute(name)
      const attribute2 = node2
        .getAttribute(name)

      return attribute1 !== attribute2
    })

  if (differentAttribute) {
    return true
  }

  if (node1.children.length === 0 &&
    node2.children.length === 0 &&
    node1.textContent !== node2.textContent) {
    return true
  }

  return false
}
```

In this implementation of a diff algorithm, we perform checks to determine if a node has changed by comparing it to another one.

- The number of attributes is different.
- At least one attribute has changed.
- The nodes have no children, and the `textContent` is different.

We can make refined checks to increase performances, but I suggest keeping the rendering engine as simple as possible. Keep an eye on performance with one of the tools mentioned at the beginning of this chapter. When a problem arises, try to adapt your algorithm to your situation. Donald Knuth once said, "Premature optimization is the root of all evil (or at least most of it) in programming."

Summary

In this chapter, you learned how to create a rendering engine for a frameworkless application. We also explored how to build a simple component registry and how to make our engine perform by using a virtual DOM algorithm.

In the next chapter, you learn how to manage events from the user and how to integrate these events with a rendering engine.

CHAPTER 3

Managing DOM Events

In the last chapter, we talked about rendering, or more generally, how to *draw* DOM elements that match with our data. **But a web application is not a painting; its contents change over time**. The cause of these changes is *events*.

Events, regardless that they are created by the user or by the system, are a very important aspect of the DOM API. The purpose of this chapter is to understand how to manage these events in a frameworkless application.

The first part of the chapter is an introduction to the DOM Events API. You will learn what an event handler is and how to properly attach it to DOM elements. In the second part of the chapter, you will add some interactivity managing events to the TodoMVC application.

The YAGNI Principle

In this chapter, you will modify the rendering engine from the previous chapter to add DOM events management. So, why did I decide to start showing you an engine that is incomplete—completely ignoring the events? Two reasons are readability and simplicity. But I would use the *same approach* for a real-world project. I would focus on the most important feature, and then I would iterate—evolving my architecture around new

© Francesco Strazzullo 2019
F. Strazzullo, *Frameworkless Front-End Development*,
https://doi.org/10.1007/978-1-4842-4967-3_3

needs. This is one of the principles of extreme programming (XP), which is called YAGNI (You aren't gonna need it.). To better explain the YAGNI principle, I often use this quote from Ron Jeffries, one of the founders of XP.

> *Always implement things when you actually need them. Never when you just foresee that you need them.*

This is a good principle to follow in any use case, but it's crucial for a frameworkless project. When I talk about the frameworkless approach, one of the criticisms that I often hear is, "You will just write another framework that no one will maintain." This is actually a risk if you overengineer your architecture. When you're creating your own architecture, you should apply YAGNI, and solve only the problems that you have at that moment. Don't try to foresee the future.

Look at the way I wrote the code from the last chapter as a reference for the YAGNI principle. I (tried to) write the best code possible for rendering, and only later did I add the events to the mix.

The DOM Events API

Events are actions that happen in a web application, which the browser tells you about so that you can *react* to them in some way. There is a wide variety of event types, and you can consult the Mozilla Developer Network for a comprehensive list (`https://developer.mozilla.org/en-US/docs/Web/Events`).

You can react to events triggered by the user, including mouse events (click, double click, etc.), keyboard events (keydown, keyup, etc.), and view events (resize, scroll, etc.). Furthermore, the system itself can emit events; or example, you can react to changes in your network status or when the DOM content is loaded (see Figure 3-1).

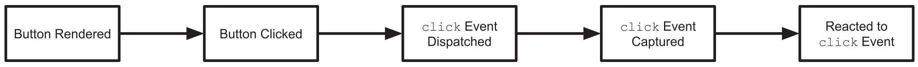


Figure 3-1. *Basic click event lifecycle*

To react to an event, you need to attach to the DOM element that triggered the event, which is a callback called an *event handler*.

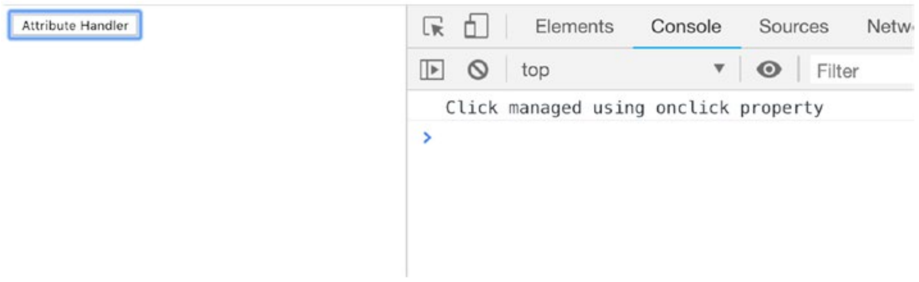
Tip For view or system events, you need to attach the event handler to the `window` object.

Attach Handlers with Properties

A quick-and-dirty way to attach an event handler to a DOM element is to use the `on*` properties. For every event type, there is a corresponding property on the DOM elements. A button has an `onclick` property, but also `ondblclick`, `onmouseover`, `onblur`, and `onfocus` properties as well. Properties make attaching a handler to a click event straightforward, as shown in Listing 3-1. The result of this listing is visible in Figure 3-2.

Listing 3-1. Click Handler with onclick Property

```
const button = document.querySelector('button')
button.onclick = () => {
  console.log('Click managed using onclick property')
}
```



```
Click managed using onclick property
```

Figure 3-2. *Example of onclick property handler*

I just said that this is a "quick and dirty" solution. It's easy to grasp why it's *quick*, but why is it also *dirty*? This kind of solution, even if it works, is usually considered a bad practice. The main reason is because with properties, you can attach only one handler at a time. So if a piece of code overwrites your `onclick` handler, your original handler is lost forever. In the next section, you learn a better, approach: the `addEventListener` method.

Attach Handlers with addEventListener

Every DOM node that handles events implements the `EventTarget` interface. Its `addEventListener` method adds event handlers to a DOM node. Listing 3-2 shows you how to add a simple button-click event handler by using this technique.

Listing 3-2. Click Handler with addEventListener

```
const button = document.querySelector('button')
button.addEventListener('click', () => {
  console.log('Clicked using addEventListener')
})
```

The first parameter is the event type. In the last example, we managed the click, but you can add listeners that handle any supported event type. The second parameter is the callback, which is invoked when the event is triggered.

In contrast to the `property` method, `addEventListener` can attach all the handlers that you need, as shown in Listing 3-3.

Listing 3-3. Multiple Click Event Handlers

```
const button = document.querySelector('button')
button.addEventListener('click', () => {
  console.log('First handler')
})
button.addEventListener('click', () => {
  console.log('Second handler')
})
```

Keep in mind that when an element is no longer present in the DOM, you should remove its event listeners as well, in order to prevent memory leaks. To do that, you use the `removeEventListener` method. Listing 3-4 is an example.

Listing 3-4. Removing Event Handlers

```
const button = document.querySelector('button')
const firstHandler = () => {
  console.log('First handler')
}

const secondHandler = () => {
  console.log('Second handler')
}

button.addEventListener('click', firstHandler)
button.addEventListener('click', secondHandler)
```

```
window.setTimeout(() => {
  button.removeEventListener('click', firstHandler)
  button.removeEventListener('click', secondHandler)
  console.log('Removed Event Handlers')
}, 1000)
```

The most important thing to notice in the previous snippet is that to remove an event handler, you should keep a reference to it in order to pass it as a parameter in the `removeEventListener` method.

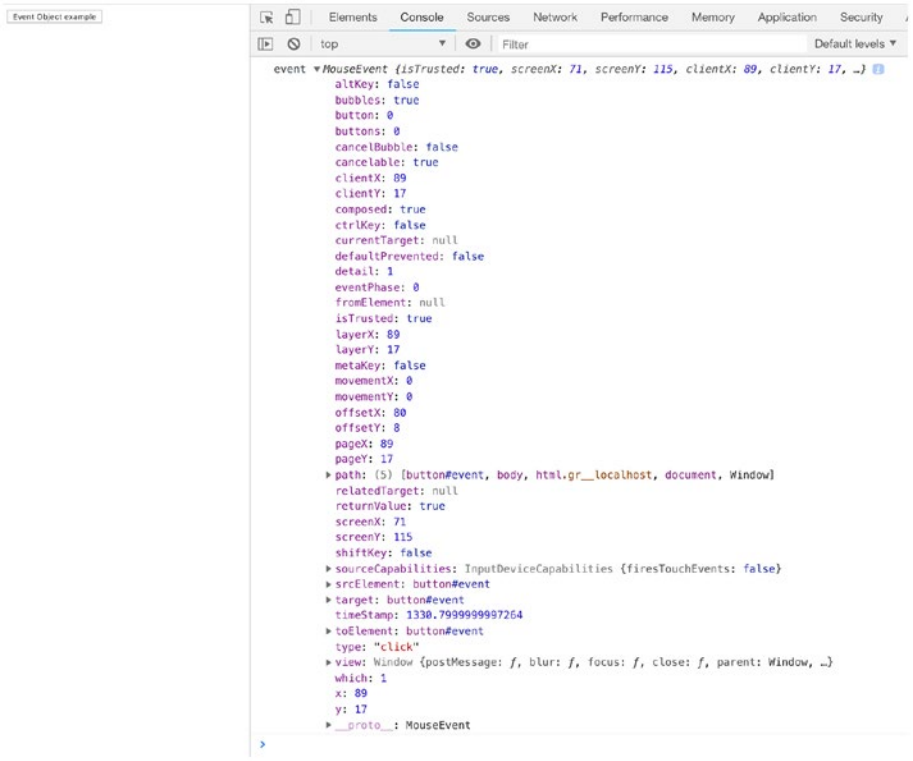
The Event Object

In all the code that we have analyzed so far, event handlers were created without parameters. But the signature of an event handler can contain a parameter that represents the event emitted by the DOM node or the system. In Listing 3-5, we simply print this event in the console.

As you can see in Figure 3-3, the event contains a lot of useful information, such as the coordinates of the pointer, the type of event, and the element that triggered the event.

Listing 3-5. Printing the Event Object to the Console

```
const button = document.querySelector('button')
button.addEventListener('click', e => {
  console.log('event', e)
})
```



```
event ▾MouseEvent {isTrusted: true, screenX: 71, screenY: 115, clientX: 89, clientY: 17, …}
          altKey: false
          bubbles: true
          button: 0
          buttons: 0
          cancelBubble: false
          cancelable: true
          clientX: 89
          clientY: 17
          composed: true
          ctrlKey: false
          currentTarget: null
          defaultPrevented: false
          detail: 1
          eventPhase: 0
          fromElement: null
          isTrusted: true
          layerX: 89
          layerY: 17
          metaKey: false
          movementX: 0
          movementY: 0
          offsetX: 80
          offsetY: 8
          pageX: 89
          pageY: 17
        ▸ path: (5) [button#event, body, html.gr__localhost, document, Window]
          relatedTarget: null
          returnValue: true
          screenX: 71
          screenY: 115
          shiftKey: false
        ▸ sourceCapabilities: InputDeviceCapabilities {firesTouchEvents: false}
        ▸ srcElement: button#event
        ▸ target: button#event
          timeStamp: 1330.7999999997264
        ▸ toElement: button#event
          type: "click"
        ▸ view: Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}
          which: 1
          x: 89
          y: 17
        ▸ __proto__: MouseEvent
>
```

Figure 3-3. *Printing the event object to the console*

Any event dispatched in a web application implements the `Event` interface. Based on its type, the event object can implement a more specific interface that extends the `Event` interface.

A `click` event (but also `dblclick`, `mouseup`, and `mousedown`) implements the `MouseEvent` interface. This interface contains information about the coordinates or the movement of the pointer during the event, and other useful data. The `MouseEvent` interface hierarchy is shown in Figure 3-4.

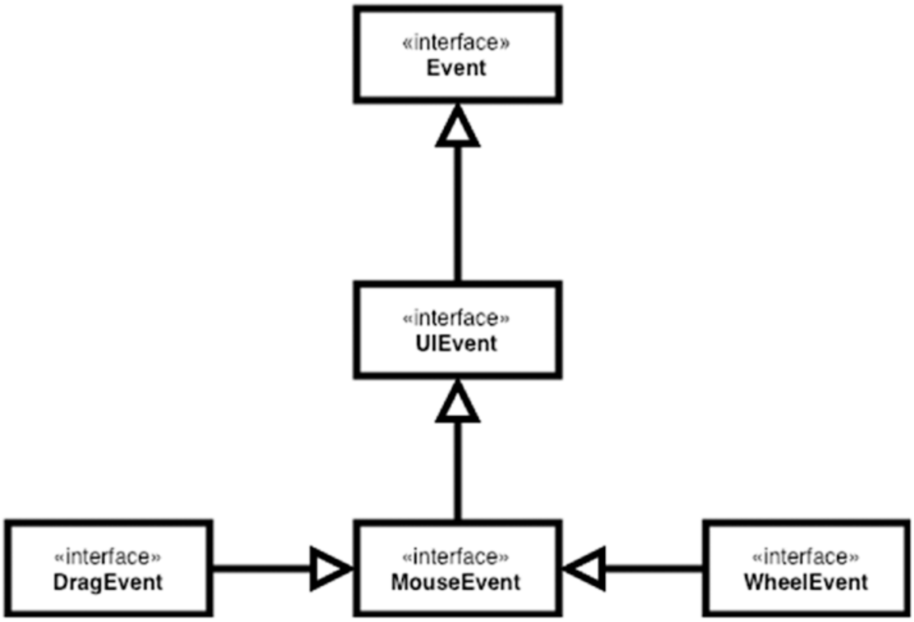


Figure 3-4. *MouseEvent interface hierarchy*

For a complete reference of the `Event` interface and the other interfaces, read the MDN guide at `https://developer.mozilla.org/en-US/docs/Web/API/Event`.

The DOM Event Lifecycle

When you read code that uses the `addEventListener` method to add a handler, you usually see something like this:

```
button.addEventListener('click', handler, false)
```

The third parameter is called `useCapture`, and its default value is false. This parameter has not always been optional. Ideally, you should include it to get the widest possible browser compatibility. But what does it mean to capture an event? And what happens if we set `useCapture` to true? Let's try to figure it out with an example. Consider the HTML structure in Listing 3-6.

Listing 3-6. A Simple, Nested HTML Structure

```
<body>
    <div>
        This is a container
        <button>Click Here</button>
    </div>
</body>
```

In Listing 3-7, event handlers are attached to both DOM elements: `div` and `button`.

Listing 3-7. Showing the Bubble Phase Mechanism

```
const button = document.querySelector('button')
const div = document.querySelector('div')

div.addEventListener('click', () => {
  console.log('Div Clicked')
}, false)

button.addEventListener('click', () => {
  console.log('Button Clicked')
}, false)
```

What happens if we click the button? Given that `button` is inside `div`, both handlers are invoked, starting with `button`. So the event object starts from the DOM node that triggered it (in this case, `button`) and goes up to all its ancestors. This mechanism it's called *bubble phase* or *event bubbling*. We can stop the bubble chain with the `stopPropagation` method from the `Event` interface. In Listing 3-8, this method is used in the `button` handler to stop the `div` handler.

Listing 3-8. Stopping the Bubble Chain

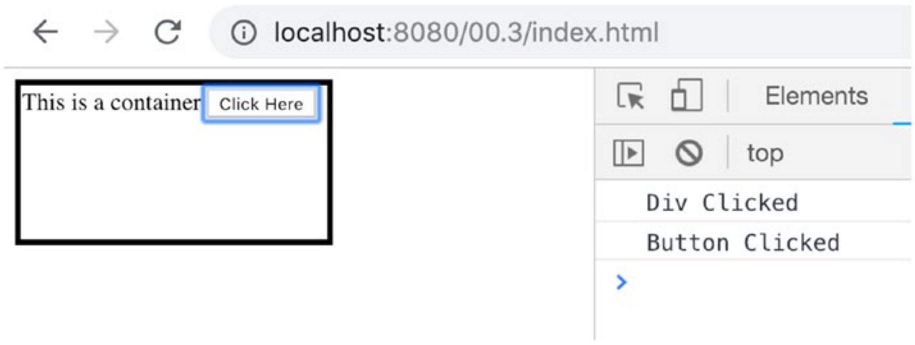
```
const button = document.querySelector('button')
const div = document.querySelector('div')

div.addEventListener('click', () => {
  console.log('Div Clicked')
}, false)

button.addEventListener('click', e => {
  e.stopPropagation()
  console.log('Button Clicked')
}, false)
```

In this case, the `div` handler is not invoked. This technique could be useful when you have a complex layout, but if you often rely on the order of the handlers, your code could become very hard to maintain. In these cases, the event delegation pattern could be useful. I talk about event delegation in greater detail at the end of the chapter.

You can use the `useCapture` parameter to reverse the handlers' order of execution. In Listing 3-9, the `div` handler is invoked before the `button` handler, as you can see in Figure 3-5.

Listing 3-9. Using useCapture to Reverse the Order of the Events Handlers

```
const button = document.querySelector('button')
const div = document.querySelector('div')

div.addEventListener('click', e => {
  console.log('Div Clicked')
}, true)

button.addEventListener('click', e => {
  console.log('Button Clicked')
}, true)
```



```
Div Clicked
Button Clicked
```

Figure 3-5. *Using capture phase*

In other words, using `true` for the `useCapture` parameter during the invocation of `addEventListener` means that we want to add the event handler to the capture phase instead of the bubble phase. While in the bubble phase, the handlers with a bottom-up process, in the capture phase it's the opposite. The system starts managing handlers from the `<html>` tag and goes deeper until the element that triggered the event is managed. It's important to remember that, for every DOM event that is generated, browsers run the capture phase (top-down) and then the bubble phase (bottom-up). There is also a third phase, called the *target phase*. This special phase occurs when the event reaches the target element—`button` in our case. The following summarizes most DOM events' lifecycle.

1. Capture phase: The event travels from `html` to target element.
2. Target phase: The event reaches the target element.
3. Bubble phase: The event travels from target element to `html`.

A more detailed version of this lifecycle is shown in Figure 3-6.

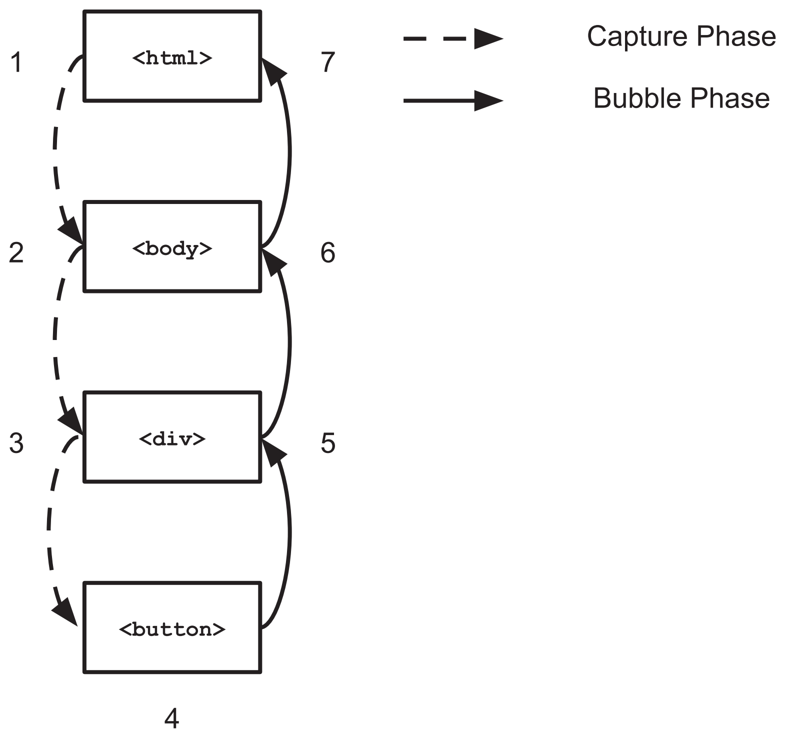


Figure 3-6. *Event lifecycle*

These phases exist for historical reasons. In the dark days, some browsers only managed the capture phase, while others only managed the bubble phase. Generally, it's OK to only use bubble phase handlers, but it's important to know about the capture phase to manage complex situations.

Using Custom Events

The only event that we have handled so far is a button click. In a similar way, we can handle a lot of different kind of events, like the one that we talked about at the beginning of the chapter. But the DOM event API is far more powerful. We can define custom event types and handle them like any other event.

This is a really important part of the DOM Events API because we can create DOM events that are bounded to our *domain* and only on what happened in the system itself. We can create an event handler for login or logout, or for something that happened to our dataset, such as the creation of a new record in a list.

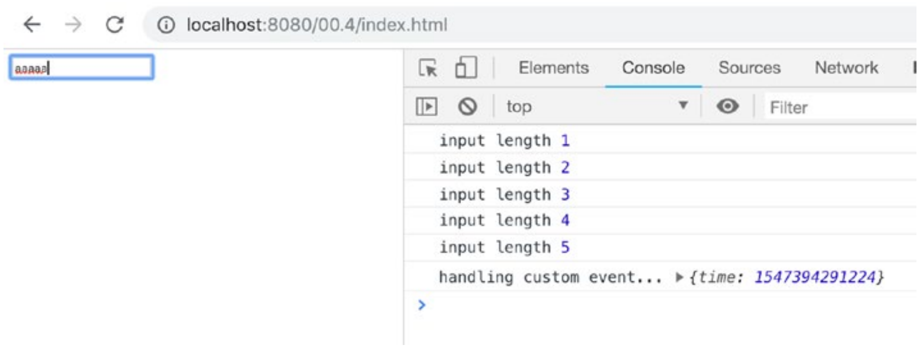
As you can see in Listing 3-10, to create a custom event, you have to use the `CustomEvent` constructor function.

Listing 3-10. Firing Custom Events

```
const EVENT_NAME = 'FiveCharInputValue'
const input = document.querySelector('input')

input.addEventListener('input', () => {
  const { length } = input.value
  console.log('input length', length)
  if (length === 5) {
    const time = (new Date()).getTime()
    const event = new CustomEvent(EVENT_NAME, {
      detail: {
        time
      }
    })

    input.dispatchEvent(event)
  }
})

input.addEventListener(EVENT_NAME, e => {
  console.log('handling custom event...', e.detail)
})
```

While managing the `input` event, we check for the length of the value itself. If the length is exactly five, we fire a special event called

`FiveCharInputValue`. We add a standard event listener with the usual `addEventListener` method to handle the custom event. Notice how we can use the same API for both a standard (`input`) and a custom event. We can also pass additional data to the handlers with the `detail` object that we used in the constructor (in this case, a timestamp). The result of Listing 3-10 is shown in Figure 3-7.



```
input length 1
input length 2
input length 3
input length 4
input length 5
handling custom event... ▸{time: 1547394291224}
```

Figure 3-7. *Using custom events*

In Chapter 4, I show you how to use custom events to let components communicate with each other.

Adding Events to TodoMVC

Now that you have learned the basic concepts of the DOM Events API, let's add event handling to our TodoMVC application. Let's take another look at a complete TodoMVC application (a screenshot is shown in Figure 3-8) to learn which events need to be handled.

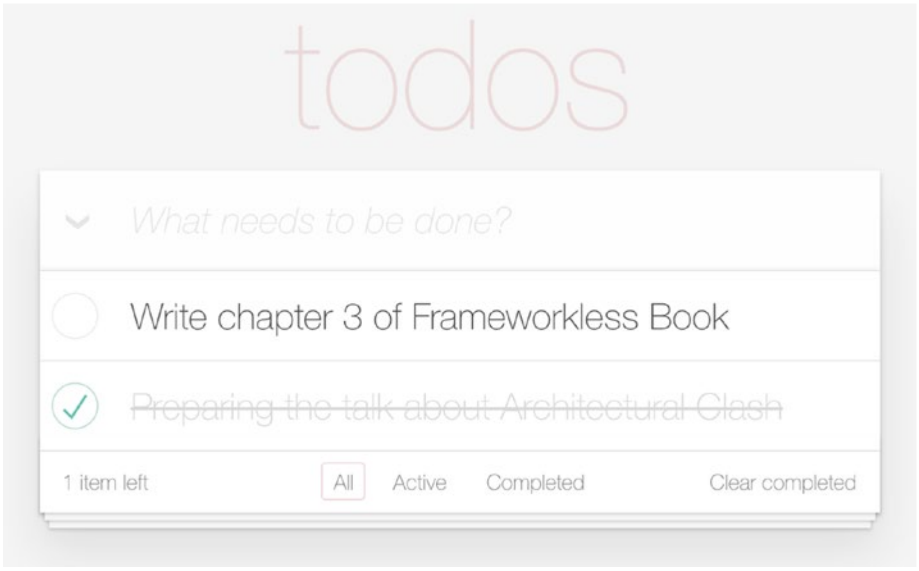

todos
What needs to be done?
Write chapter 3 of Frameworkless Book
~~Preparing the talk about Architectural Clash~~
1 item left
All
Active
Completed
Clear completed

Figure 3-8. *Complete TodoMVC application*

The following are the events that we need to manage.

- **Delete an item.** Click the cross to the right of every row.
- **Toggle an item as complete or not.** Click the circle to the left of every row.
- **Change the filter.** Click the filter name on the bottom.
- **Create a new item.** Input a value in the top input text and press Enter on the keyboard.
- **Remove all completed items.** Click the “Clear completed” label.
- **Toggle all items as completed or not.** Click the chevron in the top-left corner.
- **Edit an item.** Double-click the row, change the value, and press Enter on the keyboard.

Let's Review Our Rendering Engine

Before adding event handlers to the TodoMVC application, we need to change some parts of our rendering engine. The problem with the last implementation that we developed in Chapter 2 is that some parts worked with strings instead of DOM elements. In Listing 3-11, you can see the "todos" component from Chapter 2. You can find the complete code for this example at https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter02/05.

Listing 3-11. The Todos Component

```
const getTodoElement = todo => {
  const {
    text,
    completed
  } = todo

  return `
      <li ${completed ? 'class="completed"' : ''}>
        <div class="view">
          <input
            ${completed ? 'checked' : ''}
            class="toggle"
            type="checkbox">
          <label>${text}</label>
          <button class="destroy"></button>
        </div>
        <input class="edit" value="${text}">
      </li>`
}
```

```
export default (targetElement, { todos }) => {
  const newTodoList = targetElement.cloneNode(true)
  const todosElements = todos
    .map(getTodoElement)
    .join('')
  newTodoList.innerHTML = todosElements
  return newTodoList
}
```

Every todo element in our list is created with a string, joined together, and then added to the parent list with `innerHTML`. But we cannot add event handlers to strings; we need DOM nodes to invoke `addEventListener`.

The Template Element

There are many different techniques for creating DOM nodes programmatically. One of them is to use `document.createElement`, an API that lets developers create new empty DOM nodes. You can see an example of the usage of this method in Listing 3-12.

Listing 3-12. document.createElement Examples

```
const newDiv = document.createElement('div')
if(!condition){
  newDiv.classList.add('disabled')
}

const newSpan = document.createElement('span')
newSpan.textContent = 'Hello World!'

newDiv.appendChild(newSpan)
```

We could use this API to create an empty `li`, and then add the various `div` handlers, `input` handlers, and so on. But our code would be very hard to read and maintain. Another (better) option is to keep the markup of the todo element inside a `template` tag in the `index.html` file. A `template` tag is just what its name suggests: an invisible tag that you can use as a "stamp" for our rendering engine. Listing 3-13 has a `template` example of the todo-item.

Listing 3-13. todo-item template Element

```
<template id="todo-item">
  <li>
    <div class="view">
      <input class="toggle" type="checkbox">
      <label></label>
      <button class="destroy"></button>
    </div>
    <input class="edit">
  </li>
</template>
```

In Listing 3-14, this `template` is used in the todos component as a "stamp" to create a new `li` DOM node.

Listing 3-14. Using template to Generate todo Items

```
let template

const createNewTodoNode = () => {
  if (!template) {
    template = document.getElementById('todo-item')
  }

  return template
    .content
    .firstElementChild
    .cloneNode(true)
}
```

```
const getTodoElement = todo => {
  const {
    text,
    completed
  } = todo

  const element = createNewTodoNode()

  element.querySelector('input.edit').value = text
  element.querySelector('label').textContent = text

  if (completed) {
    element
      .classList
      .add('completed')

    element
      .querySelector('input.toggle')
      .checked = true
  }

  return element
}

export default (targetElement, { todos }) => {
  const newTodoList = targetElement.cloneNode(true)

  newTodoList.innerHTML = ''

  todos
    .map(getTodoElement)
    .forEach(element => {
      newTodoList.appendChild(element)
    })

  return newTodoList
}
```

We can then extend the template technique to all the applications by creating an app component. The first step is to wrap the markup of our todo-list in a `template` element, as shown in Listing 3-15.

Listing 3-15. Using template for the Entire App

```
<body>
    <template id="todo-item">
        <!-- Put here todo item content-->
    </template>
    <template id="todo-app">
        <section class="todoapp">
            <!-- Put here app content-->
        </section>
    </template>
    <div id="root">
        <div data-component="app"></div>
    </div>
</body>
```

In Listing 3-16, a new component called *app* is created. This component utilizes the newly created template to generate its content. This is the last part of the template porting of the TodoMVC application. This new version of the application will be the base of our event handlers' architecture. The application's complete code is hosted at `https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter03/01.1`.

Listing 3-16. App Component with Template

```
let template

const createAppElement = () => {
  if (!template) {
    template = document.getElementById('todo-app')
  }

  return template
    .content
    .firstElementChild
    .cloneNode(true)
}

export default (targetElement) => {
  const newApp = targetElement.cloneNode(true)
  newApp.innerHTML = ''
  newApp.appendChild(createAppElement())
  return newApp
}
```

A Basic Event Handling Architecture

Now that we have a new rendering engine that works with DOM elements instead of strings, we are ready to attach event handlers to our application. Let's start with a high-level overview and then look at a working example. Our rendering engine it's based on pure functions that get a state and generate a DOM tree.

For every new state, we can generate a new DOM tree and apply a virtual DOM algorithm. In this scenario, we can easily inject our event handlers in this "loop." After every event, we will manipulate the state and then invoke the main render function with this new state. Figure 3-9 is a schema of this *state-render-event* loop.

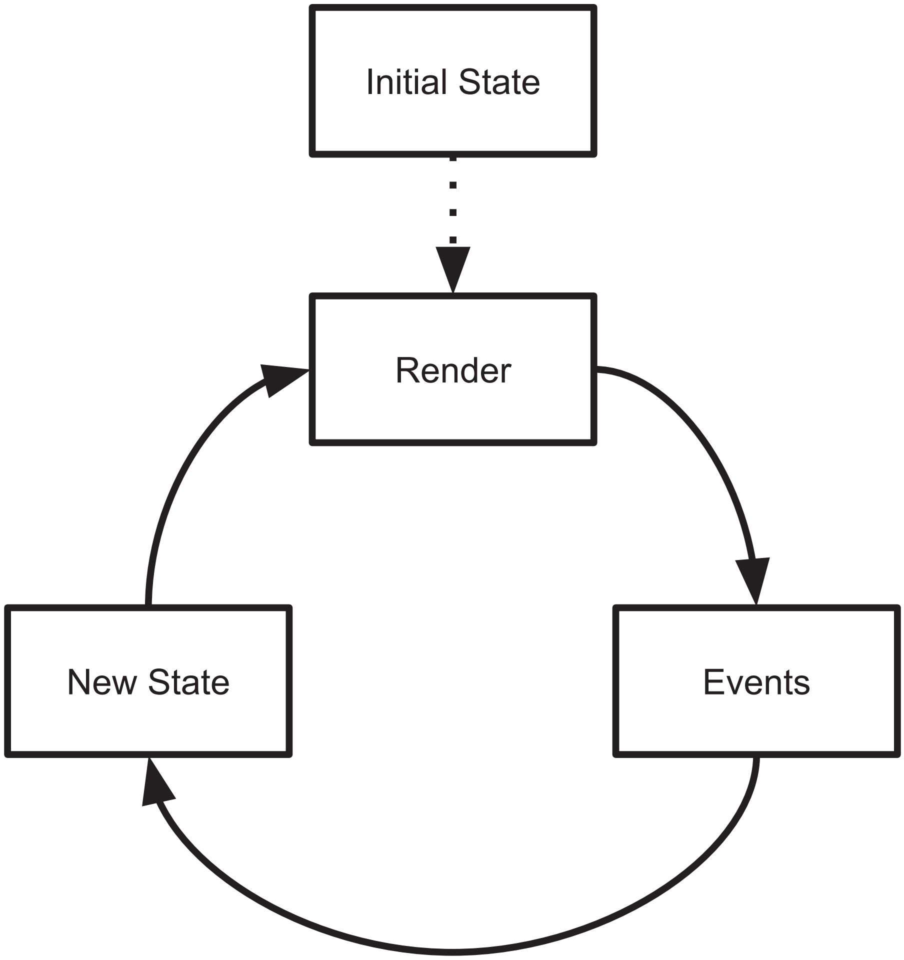


Figure 3-9. *Event handling high-level architecture*

We can test our *state-render-event* loop by enumerating the steps of a simple use case for our application. Let's try to imagine a user that adds and deletes an item from the list.

- **Initial state**: empty todo list
- **Render**: shows the user an empty list
- **Event**: the user creates a new item named "dummy item"

- **New state**: todo list with one item
- **Render**: shows the user a list with one item
- **Event**: the user deletes the item
- **New state**: empty todo list
- **Render**: showing the user an empty list

Now that we defined our high-level architecture, it's time to implement it. Listing 3-17 defines these events and the related state modification in our controller.

Listing 3-17. A Controller with Events

```
const state = {
  todos: [],
  currentFilter: 'All'
}

const events = {
  deleteItem: (index) => {
    state.todos.splice(index, 1)
    render()
  },
  addItem: text => {
    state.todos.push({
      text,
      completed: false
    })
    render()
  }
}
```

```
const render = () => {
  window.requestAnimationFrame(() => {
    const main = document.querySelector('#root')

    const newMain = registry.renderRoot(
      main,
      state,
      events)

    applyDiff(document.body, main, newMain)
  })
}

render()
```

The entry point of our rendering engine, the `renderRoot` function, now takes a third parameter that contains the events. In a moment, you will see that this new parameter is accessible to all of our components. Our events are very simple functions that modify the state and manually invoke a new render. In a real-world application, I suggest creating some kind of "event registry" that helps developers quickly add handlers and automatically invoke a new render cycle. For now, this implementation is good enough.

In Listing 3-18, the `addItem` handler is used by the app component to add a new item to the list.

Listing 3-18. App Component with addItem Event

```
let template

const getTemplate = () => {
  if (!template) {
    template = document.getElementById('todo-app')
  }

  return template.content.firstElementChild.cloneNode(true)
}
```

```
const addEvents = (targetElement, events) => {
  targetElement
    .querySelector('.new-todo')
    .addEventListener('keypress', e => {
      if (e.key === 'Enter') {
        events.addItem(e.target.value)
        e.target.value = ''
      }
    })
}

export default (targetElement, state, events) => {
  const newApp = targetElement.cloneNode(true)

  newApp.innerHTML = ''
  newApp.appendChild(getTemplate())

  addEvents(newApp, events)

  return newApp
}
```

For every render cycle, we generate a new DOM element and attach an event handler to the `input` handler used to insert the value of the new item. When the user presses Enter, the `addItem` function is fired and then the `input` handler is cleared.

In Listing 3-18, something may have seemed out of place. We cleared the value of the input inside the event itself. *Why the value of the input it's not part of the state like the list of todos or the current filter?* I address this topic in Chapter 7, so for now, we can ignore this problem.

The other action that the user can do in this example is delete an item. So, the component that needs access to events is todos, as you can see in Listing 3-19.

Listing 3-19. The todos Component with Events

```
const getTodoElement = (todo, index, events) => {
  const {
    text,
    completed
  } = todo

  const element = createNewTodoNode()

  element.querySelector('input.edit').value = text
  element.querySelector('label').textContent = text

  if (completed) {
    element.classList.add('completed')
    element
      .querySelector('input.toggle')
      .checked = true
  }

  const handler = e => events.deleteItem(index)

  element
    .querySelector('button.destroy')
    .addEventListener('click', handler)

  return element
}

export default (targetElement, { todos }, events) => {
  const newTodoList = targetElement.cloneNode(true)

  newTodoList.innerHTML = ''

  todos
    .map((todo, index) => getTodoElement(todo, index, events))
    .forEach(element => {
      newTodoList.appendChild(element)
    })

  return newTodoList
}
```

The Listing 3-19 example is very similar to Listing 3-18, but this time I created a different handler for every todo item. You may read the code of the complete application with all the events at `https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter03/01.3`.

Event Delegation

Event delegation is a feature provided with most front-end frameworks. It's usually well-hidden under the hood. To better understand what event delegation is, let's look at an example. Listing 3-20 is a revised version of Listing 3-19 and based on event delegation.

Listing 3-20. The todos Component with Event Delegation

```
const getTodoElement = (todo, index) => {
  const {
    text,
    completed
  } = todo

  const element = createNewTodoNode()

  element.querySelector('input.edit').value = text
  element.querySelector('label').textContent = text

  if (completed) {
    element.classList.add('completed')
    element
```

```
      .querySelector('input.toggle')
      .checked = true
  }

  element
    .querySelector('button.destroy')
    .dataset
    .index = index

  return element
}

export default (targetElement, state, events) => {
  const { todos } = state
  const { deleteItem } = events
  const newTodoList = targetElement.cloneNode(true)

  newTodoList.innerHTML = ''

  todos
    .map((todo, index) => getTodoElement(todo, index))
    .forEach(element => {
      newTodoList.appendChild(element)
    })

  newTodoList.addEventListener('click', e => {
    if (e.target.matches('button.destroy')) {
      deleteItem(e.target.dataset.index)
    }
  })

  return newTodoList
}
```

This is different from the previous component because here we have only one event handler, which is attached to the list itself. There is no separate event handler for every row. If you have a very long list, this approach could improve performance and memory usage.

Notice the usage of the `matches` API (`https://developer.mozilla.org/en-US/docs/Web/API/Element/matches`) to check if an element is our "real" event target. By using this approach on a larger scale, you can achieve only one event handler on the body of a web page. Building an event delegation library is beyond the scope of this book, but there are several libraries that you can use in your projects. One of these libraries is gator.js (`https://craig.is/riding/gators`), which is very easy to use. Listing 3-21 is a simple example of a handler attached to this library.

Listing 3-21. gator.js Example

```
Gator(document).on('click', 'button.destroy', e => {
  deleteItem(e.target.dataset.index)
})
```

I want to share the same advice that I used to close the last chapter. Don't add any kind of optimization, such as event delegation, until you really need it. Remember the YAGNI principle and that adding an event delegation library like gator.js to an existing project can be done in an iterative way for the most critical parts.

Summary

This chapter covered some basic concepts of the DOM Events API. You learned how to attach and remove event handlers, the difference between the bubble phase and the capture phase, and how to create custom events. We updated our TodoMVC application by adding events to add and remove an item.

Finally, I introduced the concept of event delegation, an important pattern to keep frameworkless applications performant enough for in nontrivial contexts.

In the next chapter, you learn how to work effectively with web components, particularly, a standard way to create components in web applications.

CHAPTER 4

Web Components

All the major front-end frameworks that developers use today have something in common. They all use components as basic blocks for building the UI. In Chapter 2, you saw how to create a component registry based on pure functions. On (almost) all modern browsers, it's possible to create components for your web applications with a suite of native APIs known as *web components*.

The APIs

Web components consist of three main technologies that let developers build and publish reusable UI components.

- **HTML templates**. The `<template>` tag is useful if you want to keep content that is not rendered, but may be used by JavaScript code as a "stamp" to create dynamic content.
- **Custom elements**. This API lets developers create their own fully featured DOM elements.
- **Shadow DOM**: This technique is useful if the web components should not be affected by the DOM outside the component itself. It's very useful if you're creating a component library or a widget that you want to share with the world.

F. Strazzullo, *Frameworkless Front-End Development*,
https://doi.org/10.1007/978-1-4842-4967-3_4

Caution The shadow DOM and the virtual DOM solve two completely different problems. The shadow DOM is about encapsulation, whereas the virtual DOM is about performances. For more information, I suggest reading the post at https://developer.com/shadow-dom-virtual-dom-889bf78ce701.

Can I Use It?

As I write this chapter in early 2019, all three API are supported by all browsers except Internet Explorer and Edge (see Table 4-1). But the team behind Edge is developing the feature, and they plan to ship it by the end of 2019. In any case, is it easily polyfillable with this package (https://github.com/webcomponents/custom-elements). You have to add a lot of polyfills if you have to support IE, however, so I strongly suggest that you do not start developing web components.

Table 4-1. *Status of Web Components Adoption (early 2019)*

API Supported	Chrome	Firefox	Safari	Edge	Internet Explorer
HTML templates	Yes	Yes	Yes	Yes	No
Shadow DOM	Yes	Yes	Yes	Developing	No
Custom elements	Yes	Yes	Yes	Developing	No

I talked about HTML templates in Chapter 3. We used it in the last implementation of our rendering engine. Shadow DOM is beyond the scope of this chapter. I suggest reading the MDN tutorial about it at https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM.

Custom Elements

The Custom Elements API is the core factor of the web components suite. In a nutshell, it permits you to create custom HTML tags, like this one:

```
<app-calendar/>
```

It is no coincidence that I used the name `app-calendar`. When you create a custom tag with the Custom Elements API, you have to use at least two words separated by a dash. Every one-word tag is for the sole use of the World Wide Web Consortium (W3C). In Listing 4-1, you see the simplest custom element possible: a "Hello World!" label.

Note A custom element is just a JavaScript class that extends HTML elements.

Listing 4-1. HelloWorld Custom Element

```
export default class HelloWorld extends HTMLElement {
  connectedCallback () {
    window.requestAnimationFrame(() => {
      this.innerHTML = '<div>Hello World!</div>'
    })
  }
}
```

`connectedCallback` is one of the lifecycle methods of a custom element. This method is invoked when the component is attached to the DOM. It's very similar to the `componentDidMount` method in React. It's a good place to render the content of the component, like in our case, or to start timers or fetch data from the network. Similarly, the `disconnectedCallback` is invoked when the component is removed from the DOM, which is a useful method in any cleanup operation.

To use this newly created component, we need to add it to the browser component registry. To achieve this goal, we need to use the `define` method of the `window.customElements` property, as shown in Listing 4-2.

Listing 4-2. Adding HelloWorld to Custom Elements Registry

```
import HelloWorld from './components/HelloWorld.js'

window
  .customElements
  .define('hello-world', HelloWorld)
```

To add a component to the browser component registry means connecting a tag name (`'hello-world'` in our case) to a custom element class. After that, you can simply use the component with the custom tag that you created (`<hello-world/>`).

Managing Attributes

The most important feature of web components is that developers can make new components that are compatible with any framework; not just with React or Angular, but any web application, including legacy applications built with JavaServer Pages or some other older tool. But, to achieve this goal, the components need to have the same public API as any other standard HTML element. So if we want to add an attribute to a custom element, we need to be sure that we can manage this attribute in the same way as any other attribute. For a standard element like `<input>`, we can set an attribute in three ways.

The most intuitive way is to add the attribute directly to the HTML markup.

```
<input type="text" value="Frameworkless">
```

In JavaScript, you can manipulate the `value` attribute with a setter.

```
input.value = 'Frameworkless'
```

Alternatively, it's possible to use the `setAttribute` method.

```
input.setAttribute('value', 'Frameworkless')
```

Each of these three methods accomplishes the same result: it changes the `value` attribute of the input element. They are also synchronized. If I input the value via the markup, I will read the same value with the getter or the `getAttribute` method. If I change the value with the setter or the `setAttribute` method, the markup will synchronize with the new attribute.

If we want to create an attribute for a custom element, we need to remember this characteristic of HTML elements. Listing 4-3 adds a `color` attribute to the `HelloWorld` component, which is used to change the color of the label's content.

Listing 4-3. HelloWorld with an Attribute

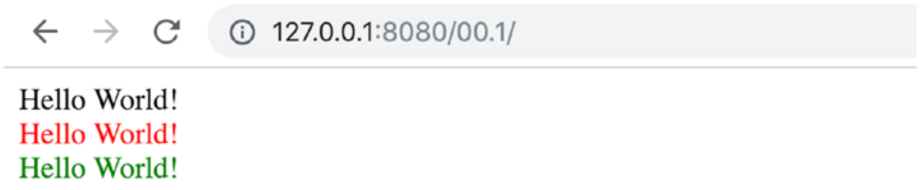
```
const DEFAULT_COLOR = 'black'

export default class HelloWorld extends HTMLElement {
  get color () {
    return this.getAttribute('color') || DEFAULT_COLOR
  }

  set color (value) {
    this.setAttribute('color', value)
  }

  connectedCallback () {
    window.requestAnimationFrame(() => {
      const div = document.createElement('div')
      div.textContent = 'Hello World!'
      div.style.color = this.color

      this.appendChild(div)
    })
  }
}
```

As you can see, the color getter/setter is just a wrapper about `getAttribute/setAttribute`. So, the three ways to set an attribute are automatically synchronized.

To set the color of the component, you can use the setter (or `setAttribute`), or you can set the color via markup. An example using the `color` attribute is shown in Listing 4-4, and the related result in Figure 4-1.

Listing 4-4. Using the color Attribute for the HelloWorld Component

```
<hello-world></hello-world>
<hello-world color="red"></hello-world>
<hello-world color="green"></hello-world>
```



Hello World!
Hello World!
Hello World!

Figure 4-1. *HelloWorld component*

Using this approach when designing attributes makes it easy for other developers to release the component. We only need to release the code of the component in a CDN, and then everyone could use it without any specific instructions. We just defined an attribute in the same way that the W3C did for standard components.

Nevertheless, this approach comes with a drawback: HTML attributes are strings. So when you need an attribute that is not a string, you have to first convert the attribute.

But, this strong constraint is really useful just for components that need to be published to other developers. In a real-world application based on web components, you may have a lot of components that are not meant to be published. They are "private" to your application. In these cases, you can just use a setter without converting the value to a string.

attributeChangedCallback

Listing 4-4 had the value of the color attribute in the `connectedCallback` method and applied that value to the DOM. But after the initial render, what happens if we change the attribute to a click event handler, as in seen Listing 4-5?

Listing 4-5. Changing the Color of the HelloWorld Component

```
const changeColorTo = color => {
  document
    .querySelectorAll('hello-world')
    .forEach(helloWorld => {
      helloWorld.color = color
    })
}

document
  .querySelector('button')
  .addEventListener('click', () => {
    changeColorTo('blue')
  })
```

When the button is clicked, the handler changes the `color` attribute of every `HelloWorld` component to blue. But on the screen, nothing happens. A quick-and-dirty solution to this problem would be adding some kind of DOM manipulation in the setter itself.

```
set color (value) {
  this.setAttribute('color', value)
  //Update DOM with the new color
}
```

But this approach is very fragile, because if we use the `setAttribute` method instead of the `color` setter, the DOM would not be updated either. The right way to manage attributes that can change during a component's lifecycle is with the `attributeChangedCallback` method. This method (like its name suggests) is invoked every time attributes change. We can modify the code of our `HelloWorld` component (see Listing 4-6) to update the DOM every time that a new `color` attribute is provided.

Listing 4-6. Updating the Color of the Label

```
const DEFAULT_COLOR = 'black'

export default class HelloWorld extends HTMLElement {
  static get observedAttributes () {
    return ['color']
  }

  get color () {
    return this.getAttribute('color') || DEFAULT_COLOR
  }

  set color (value) {
    this.setAttribute('color', value)
  }
  attributeChangedCallback (name, oldValue, newValue) {
    if (!this.div) {
      return
    }

    if (name === 'color') {
      this.div.style.color = newValue
    }
  }

  connectedCallback () {
    window.requestAnimationFrame(() => {
      this.div = document.createElement('div')
      this.div.textContent = 'Hello World!'
      this.div.style.color = this.color
      this.appendChild(this.div)
    })
  }
}
```

The `attributeChangedCallback` method accepts three parameters: the name of the attribute that has changed, the attribute's old value of the attribute, and the attribute's new value.

Note Not every attribute will trigger `attributeChangedCallback`, only the attributes listed in the `observedAttributes` array.

Virtual DOM Integration

Our virtual DOM algorithm from Chapter 2 is completely pluggable in any custom element. In Listing 4-7, there is a new version of the `HelloWorld` component. Every time that the color changes, it invokes the virtual DOM algorithm to modify the color of the label. The complete code for this example is at https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter04/00.3.

Listing 4-7. Using Virtual DOM in a Custom Element

```
import applyDiff from './applyDiff.js'

const DEFAULT_COLOR = 'black'

const createDomElement = color => {
  const div = document.createElement('div')
  div.textContent = 'Hello World!'
  div.style.color = color
  return div
}

export default class HelloWorld extends HTMLElement {
  static get observedAttributes () {
    return ['color']
  }

  get color () {
    return this.getAttribute('color') || DEFAULT_COLOR
  }

  set color (value) {
    this.setAttribute('color', value)
  }

  attributeChangedCallback (name, oldValue, newValue) {
    if (!this.hasChildNodes()) {
      return
    }
    applyDiff(
      this,
      this.firstElementChild,
      createDomElement(newValue)
    )
  }

  connectedCallback () {
    window.requestAnimationFrame(() => {
      this.appendChild(createDomElement(this.color))
    })
  }
}
```

Using a virtual DOM for this scenario is clearly over-engineering, but it can be useful if your component has a lot of attributes. In a case like that, the code would be much more readable.

Custom Events

For the next example, we will analyze a more complex component called GitHubAvatar. The purpose of this component is to show the avatar of a GitHub user given their username. To use this component, you need to set the user attribute.

```
<github-avatar user="francesco-strazzullo"></github-avatar>
```

When the component is connected to the DOM, it shows a “loading” placeholder. Then it uses GitHub REST APIs to fetch the avatar image URL. If the request succeeds, the avatar is shown; otherwise, an error placeholder is shown. A diagram that explains how this component works is shown in Figure 4-2.

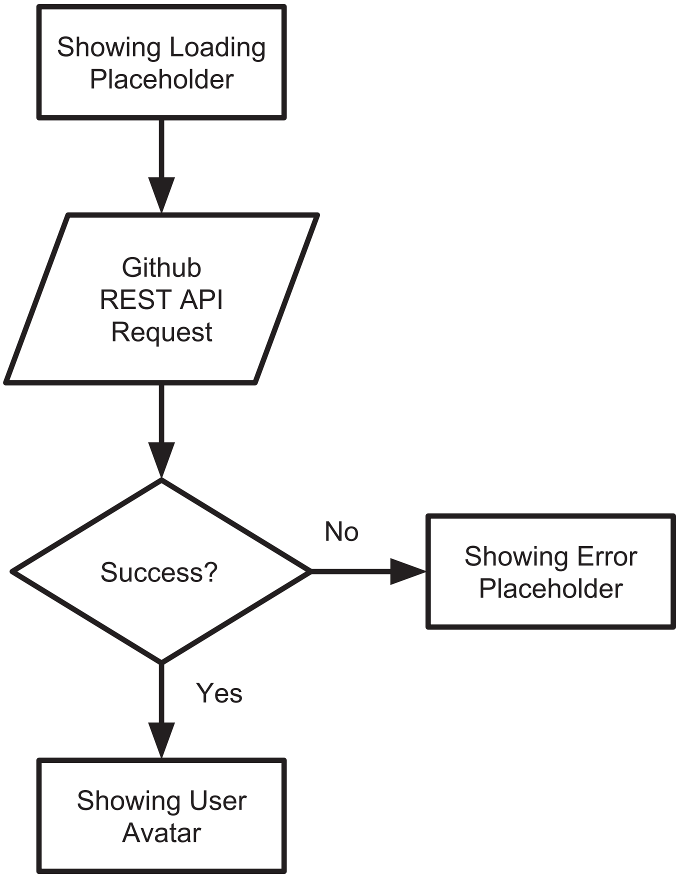


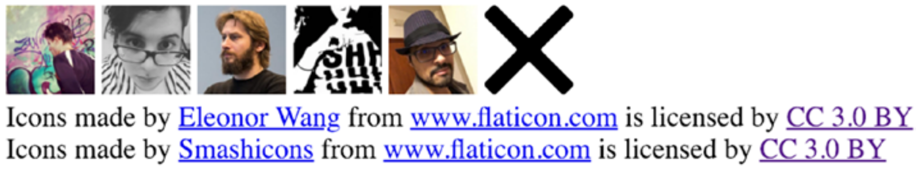
Figure 4-2. *Flowchart of GitHubAvatar component*

You can look at the code of the GitHubAvatar component in Listing 4-8. For the sake of simplicity, I didn't manage changes in the `user` attribute with the `attributeChangedCallback` method.

Listing 4-8. GitHubAvatar Component

```
const ERROR_IMAGE = 'https://files-82ee7vgzc.now.sh'
const LOADING_IMAGE = 'https://files-8bga2nnt0.now.sh'

const getGitHubAvatarUrl = async user => {
  if (!user) {
    return
  }

  const url = `https://api.github.com/users/${user}`

  const response = await fetch(url)
  if (!response.ok) {
    throw new Error(response.statusText)
  }
  const data = await response.json()
  return data.avatar_url
}

export default class GitHubAvatar extends HTMLElement {
  constructor () {
    super()
    this.url = LOADING_IMAGE
  }

  get user () {
    return this.getAttribute('user')
  }

  set user (value) {
    this.setAttribute('user', value)
  }
  render () {
    window.requestAnimationFrame(() => {
      this.innerHTML = ''
      const img = document.createElement('img')
      img.src = this.url
      this.appendChild(img)
    })
  }

  async loadNewAvatar () {
    const { user } = this
    if (!user) {
      return
    }
    try {
      this.url = await getGitHubAvatarUrl(user)
    } catch (e) {
      this.url = ERROR_IMAGE
    }

    this.render()
  }

  connectedCallback () {
    this.render()
    this.loadNewAvatar()
  }
}
```

If you follow the flowchart shown in Figure 4-2, the code should be easy to read. To fetch the data from the GitHub API, I used `fetch`, a native way in modern browsers to make asynchronous HTTP requests. I talk more about this topic in Chapter 5. In Figure 4-3, you can see the result of various instances of the component.



Icons made by Eleonor Wang from www.flaticon.com is licensed by CC 3.0 BY
Icons made by Smashicons from www.flaticon.com is licensed by CC 3.0 BY

Figure 4-3. *GitHubAvatar example*

What if we want to react to the result of the HTTP request from the *outside* of the component itself? Remember that when it's possible, a custom element should behave exactly like a standard DOM element. Earlier, we used attributes to pass information to a component, just like any other element. Following the same reasoning to get information from a component, we should use DOM events. In Chapter 3, I talked about the Custom Events API, which makes it possible to create DOM events that are bounded to the domain and not the user interaction with the browser.

In Listing 4-9, a new version of the GitHubAvatar component emits two events: one when the avatar is loaded and one when an error occurs.

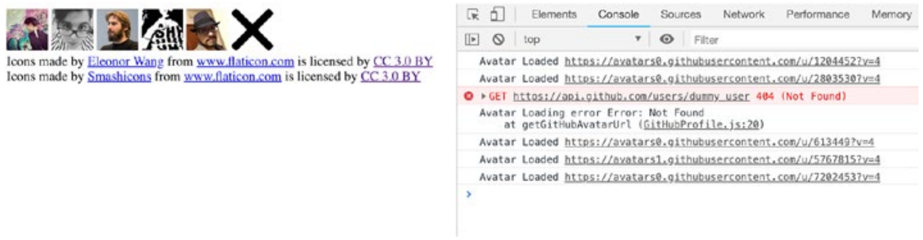
Listing 4-9. GitHubAvatar with Custom Events

```
const AVATAR_LOAD_COMPLETE = 'AVATAR_LOAD_COMPLETE'
const AVATAR_LOAD_ERROR = 'AVATAR_LOAD_ERROR'

export const EVENTS = {
  AVATAR_LOAD_COMPLETE,
  AVATAR_LOAD_ERROR
}

export default class GitHubAvatar extends HTMLElement {
  ...
  onLoadAvatarComplete () {
    const event = new CustomEvent(AVATAR_LOAD_COMPLETE, {
      detail: {
        avatar: this.url
      }
    })

    this.dispatchEvent(event)
  }

  onLoadAvatarError (error) {
   const event = new CustomEvent(AVATAR_LOAD_ERROR, {
     detail: {
       error
     }
   })

   this.dispatchEvent(event)
  }

  async loadNewAvatar () {
    const { user } = this
    if (!user) {
      return
    }
    try {
      this.url = await getGitHubAvatarUrl(user)
      this.onLoadAvatarComplete()
    } catch (e) {
      this.url = ERROR_IMAGE
      this.onLoadAvatarError(e)
    }

    this.render()
  }
  ...
}
```

In Listing 4-10, we attach event handlers to the two kinds of events. In Figure 4-4, you can see that the right handlers are invoked.

Listing 4-10. Attaching Event Handlers to GitHubAvatar Events

```
import { EVENTS } from './components/GitHubAvatar.js'

document
  .querySelectorAll('github-avatar')
  .forEach(avatar => {
    avatar
      .addEventListener(
        EVENTS.AVATAR_LOAD_COMPLETE,
        e => {
          console.log(
            'Avatar Loaded',
            e.detail.avatar
          )
        })

    avatar
      .addEventListener(
        EVENTS.AVATAR_LOAD_ERROR,
        e => {
          console.log(
            'Avatar Loading error',
            e.detail.error
          )
        })
  })
```


Icons made by Eleonor Wang from www.flaticon.com is licensed by CC 3.0 BY
Icons made by Smashicons from www.flaticon.com is licensed by CC 3.0 BY

```
Avatar Loaded https://avatars0.githubusercontent.com/u/1204452?v=4
Avatar Loaded https://avatars0.githubusercontent.com/u/2803530?v=4
▸GET https://api.github.com/users/dummy_user 404 (Not Found)
Avatar Loading error Error: Not Found
    at getGitHubAvatarUrl (GitHubProfile.js:20)
Avatar Loaded https://avatars0.githubusercontent.com/u/613449?v=4
Avatar Loaded https://avatars1.githubusercontent.com/u/5767815?v=4
Avatar Loaded https://avatars0.githubusercontent.com/u/7202453?v=4
>
```

Figure 4-4. *GitHubAvatar with events*

Using Web Components for TodoMVC

It's time to build the usual TodoMVC application. This time, we are going to use web components. Most of the code will be similar to the previous versions based on functions. I decided to split the application into three components: `todomvc-app`, `todomvc-list`, and `todomvc-footer`, as shown in Figure 4-5.

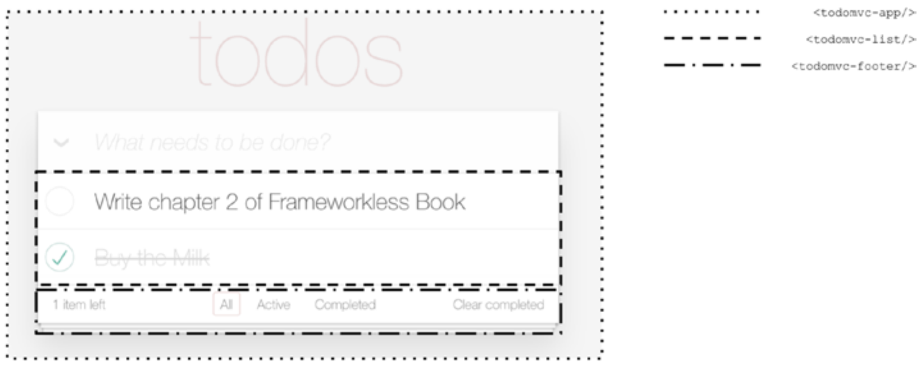


Figure 4-5. *TodoMVC components*

The first thing that I want to analyze is the HTML part of the application. As you see in Listing 4-11, I made extensive usage of the `<template>` element.

Listing 4-11. HTML for TodoMVC Application with Web Components

```
<body>
    <template id="footer">
        <footer class="footer">
            <span class="todo-count">
            </span>
            <ul class="filters">
                <li>
                    <a href="#/">All</a>
                </li>
                <li>
                    <a href="#/active">Active</a>
                </li>
                <li>
                    <a href="#/completed">
                      Completed
                    </a>
                </li>
            </ul>
            <button class="clear-completed">
                Clear completed
            </button>
        </footer>
    </template>
    <template id="todo-item">
        <li>
            <div class="view">
                <input
                  class="toggle" type="checkbox">
                <label></label>
                <button class="destroy"></button>
            </div>
            <input class="edit">
        </li>
    </template>
    <template id="todo-app">
        <section class="todoapp">
            <header class="header">
                <h1>todos</h1>
                <input class="new-todo"
                  autofocus>
            </header>
            <section class="main">
                <input
                  id="toggle-all"
                  class="toggle-all"
                  type="checkbox">
                <label for="toggle-all">
                    Mark all as complete
                </label>
                <todomvc-list></todomvc-list>
            </section>
            <todomvc-footer></todomvc-footer>
        </section>
    </template>
    <todomvc-app></todomvc-app>
</body>
```

To keep the code simple, I only implemented two of the many events that are present in the complete TodoMVC: adding an item and deleting it. This way, we can skip the `todomvc-footer` code and concentrate on `todomvc-app` and `todomvc-list`. If you're interested, you can check the complete code on GitHub at `https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter04/01`. Let's start with the list in Listing 4-12.

Listing 4-12. TodoMVC List Web Component

```
const TEMPLATE = '<ul class="todo-list"></ul>'

export const EVENTS = {
  DELETE_ITEM: 'DELETE_ITEM'
}

export default class List extends HTMLElement {
  static get observedAttributes () {
    return [
      'todos'
    ]
  }

  get todos () {
    if (!this.hasAttribute('todos')) {
      return []
    }

    return JSON.parse(this.getAttribute('todos'))
  }

  set todos (value) {
    this.setAttribute('todos', JSON.stringify(value))
  }

  onDeleteClick (index) {
    const event = new CustomEvent(
      EVENTS.DELETE_ITEM,
      {
        detail: {
          index
        }
      }
  )

    this.dispatchEvent(event)
}

createNewTodoNode () {
    return this.itemTemplate
      .content
      .firstElementChild
      .cloneNode(true)
}

getTodoElement (todo, index) {
  const {
    text,
    completed
  } = todo

  const element = this.createNewTodoNode()

  element.querySelector('input.edit').value = text
  element.querySelector('label').textContent = text

  if (completed) {
    element.classList.add('completed')
    element
      .querySelector('input.toggle')
      .checked = true
  }
  element
    .querySelector('button.destroy')
    .dataset
    .index = index

  return element
}

updateList () {
  this.list.innerHTML = "

  this.todos
    .map(this.getTodoElement)
    .forEach(element => {
      this.list.appendChild(element)
    })
  }

  connectedCallback () {
    this.innerHTML = TEMPLATE
    this.itemTemplate = document
      .getElementById('todo-item')

    this.list = this.querySelector('ul')

    this.list.addEventListener('click', e => {
      if (e.target.matches('button.destroy')) {
        this.onDeleteClick(e.target.dataset.index)
      }
    })

    this.updateList()
  }
  attributeChangedCallback () {
    this.updateList()
  }
}
```

Most of this code is very similar to the one in Chapter 3. One of the differences is that we use a custom event to tell the outer world what is happening when the user clicks the Destroy button. The only attribute that this component accepts as input is the list of todo items; every time the attribute changes, the list is rendered. As you saw earlier in this chapter, it's very easy to attach a virtual DOM mechanism in here.

Let's continue to the `todomvc-app` component code, shown in Listing 4-13.

Listing 4-13. TodoMVC Application Components

```
import { EVENTS } from './List.js'

export default class App extends HTMLElement {
  constructor () {
    super()
    this.state = {
      todos: [],
      filter: 'All'
    }

    this.template = document
      .getElementById('todo-app')
  }

  deleteItem (index) {
    this.state.todos.splice(index, 1)
    this.syncAttributes()
  }
```

```
addItem (text) {
  this.state.todos.push({
    text,
    completed: false
  })
  this.syncAttributes()
}

syncAttributes () {
  this.list.todos = this.state.todos
  this.footer.todos = this.state.todos
  this.footer.filter = this.state.filter
}

connectedCallback () {
  window.requestAnimationFrame(() => {
    const content = this.template
      .content
      .firstElementChild
      .cloneNode(true)

    this.appendChild(content)

    this
      .querySelector('.new-todo')
      .addEventListener('keypress', e => {
        if (e.key === 'Enter') {
          this.addItem(e.target.value)
          e.target.value = "
        }
      })

    this.footer = this
      .querySelector('todomvc-footer')
      this.list = this.querySelector('todomvc-list')
      this.list.addEventListener(
        EVENTS.DELETE_ITEM,
        e => {
          this.deleteItem(e.detail.index)
        }
      )

      this.syncAttributes()
    })
  }
}
```

This component has no attributes; it has an internal state instead. Events from the DOM (standard or custom) change this state, and then the component syncs its state with the attributes of its children in the `syncAttributes` method. I talk more about which components should have an internal state in Chapter 7.

Web Components vs. Rendering Functions

Now that you have seen web components in action, let's compare them to the rendering functions approach that we analyzed in Chapter 2 and Chapter 3. Next, I discuss some of the pros and cons of these two ways to render DOM elements.

Code Style

To create a web component means to extend an HTML element, so it requires you to work with classes. If you're a functional programming enthusiast, you may feel uncomfortable working in this way. On the other hand, if you're familiar with languages based on classes like Java or C#, you may feel more confident with web components than functions.

There is no real winner here; it's really up to what you like most. As you saw in the last TodoMVC implementation, you can take your rendering functions and wrap them with web components over time so that you can adapt your design to your scenario. For example, you can start with simple rendering functions and then wrap them in a web component if you need to release them in a library.

Testability

To easily test rendering functions, you only need a test runner integrated with a JSDOM like Jest (`https://jestjs.io`). JSDOM is a mock DOM implementation used for Node.js that is extremely useful for testing rendering. The problem is that JSDOM doesn't support (for now) custom elements. To test a custom element, you may need to use a real browser with a tool like Puppeteer (`https://developers.google.com/web/tools/puppeteer`), but your tests will be slower and likely more complicated.

Portability

Web components exist to be portable. The fact that they act exactly like any other DOM element is a killer feature if you need to use the same component between other applications.

Community

Component classes are a standard way to create DOM UI elements in most frameworks. This is a very useful thing to keep in mind if you have a large team or a team that needs to grow quickly. Your code is more readable if it is similar to other code that people are familiar with.

Disappearing Frameworks

A very interesting side effect of the emergence of web components is the birth of a bunch of tools that are called *disappearing frameworks* (or *invisible frameworks*). The basic idea is to write code like with any other UI framework, like React. When you create the production bundle, the output will be standard web components. In other words, during compile time, the framework will simply dissolve.

The two most popular disappearing frameworks are Svelte (`https://svelte.technology`) and Stencil.js (`https://stenciljs.com`). Stencil.js is based on TypeScript and JSX. At first, it seems a strange mix between Angular and React. I consider Stencil.js particularly interesting because it's the tool that the team behind Ionic (`https://ionicframework.com`) built to create a new version of the famous mobile UIKit entirely based on web components. Listing 4-14 shows how to build a simple Stencil.js component.

Listing 4-14. A Simple Stencil.js Component

```
import { Component, Prop } from '@stencil/core'

@Component({
  tag: 'hello-world'
})
export class HelloWorld {

  @Prop() name: string

  render() {
    return (
      <p>
        Hello {this.name}!
      </p>
    )
  }
}
```

Once the code is compiled, you can use this component like any other custom element.

```
<hello-world name="Francesco"></hello-world>
```

Summary

In this chapter, you learned about the main APIs behind the web component standard and explored the Custom Elements API.

We built a new version of our TodoMVC application based on web components, and we evaluated the differences between this approach and rendering functions.

Finally, you learned about disappearing frameworks and saw how to create a very simple component with Stencil.js.

The next chapter focuses on building a frameworkless HTTP client to make asynchronous requests.

CHAPTER 5

HTTP Requests

In the previous chapters, you learned to render DOM elements and to react to events from the system or users, but a front-end application feeds on asynchronous data from a server. The purpose of this chapter is to show you how to build an HTTP client in a frameworkless way.

A Bit of History: The Birth of AJAX

Before 1999, a complete page reload was required for every user action needing any kind of data from the server. For people that are approaching web development (or the web in general) today, it's very hard to imagine web applications built in this way. In 1999, applications like Outlook, Gmail, and Google Maps started using a new technique: loading data from a server after the initial page load, without completely reloading the page. Jesse James Garrett, in his 2005 blog post (`https://adaptivepath.org/ideas/ajax-new-approach-web-applications/`), named this technique AJAX, an acronym for Asynchronous JavaScript and XML.

The main part of any AJAX applications is the `XMLHttpRequest` object. As you will see later in this chapter, with this object, you can fetch data from a server with an HTTP request. The W3C made a first draft of the specification for this object in 2006.

As I mentioned, the "X" in AJAX stands for XML. When AJAX came out, the web applications received from the server data in XML form. Today, however, the friendlier JSON (for JavaScript applications) format is used. In Figure 5-1 you can see the differences between AJAX and Non-AJAX architectures.

© Francesco Strazzullo 2019
F. Strazzullo, *Frameworkless Front-End Development*,
https://doi.org/10.1007/978-1-4842-4967-3_5

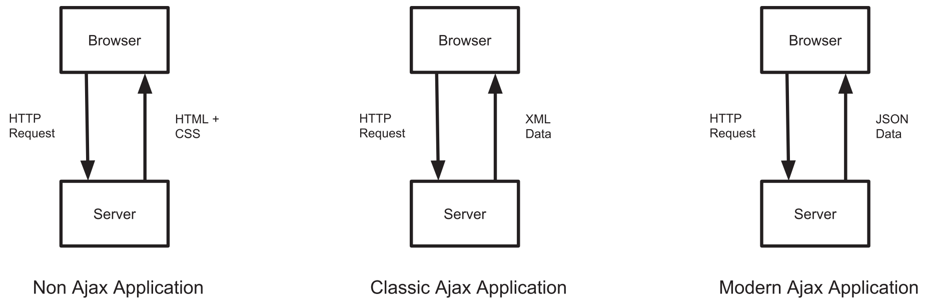


Figure 5-1. *AJAX vs. Non-AJAX architecture*

A todo-list REST Server

To test the clients that we are going to develop, we need a server to fetch data from. Listing 5-1 shows a very simple REST server for Node.js with Express (`https://expressjs.com`), a simple library to quickly create REST servers . This dummy server will use a temporary array to store the data related to our todo-list, instead of a real database. To generate fake IDs, I used a small npm package called UUID (`www.npmjs.com/package/uuid`) that lets developers generate universally unique identifiers (UUIDs).

Listing 5-1. A Dummy REST Server for Node.js

```
const express = require('express')
const bodyParser = require('body-parser')
const uuidv4 = require('uuid/v4')
const findIndex = require('lodash.findindex')

const PORT = 8080

const app = express()
let todos = []

app.use(bodyParser.json())
app.get('/api/todos', (req, res) => {
  res.send(todos)
})

app.post('/api/todos', (req, res) => {
  const newTodo = {
    completed: false,
    ...req.body,
    id: uuidv4()
  }

  todos.push(newTodo)

  res.status(201)
  res.send(newTodo)
})

app.patch('/api/todos/:id', (req, res) => {
  const updateIndex = findIndex(
    todos,
    t => t.id === req.params.id
  )
  const oldTodo = todos[updateIndex]

  const newTodo = {
    ...oldTodo,
    ...req.body
  }

  todos[updateIndex] = newTodo

  res.send(newTodo)
})
```

```
app.delete('/api/todos/:id', (req, res) => {
  todos = todos.filter(
    t => t.id !== req.params.id
  )

  res.status(204)
  res.send()
})

app.listen(PORT)
```

Representational State Transfer

In this section, I explain the meaning of REST, which is the architecture behind our dummy server. If you already know about REST, you can simply skip this section.

REST is an acronym for REpresentational State Transfer, which is a way to design and develop web services. The main abstraction of any REST API is its resources. You need to split your domain into resources. Each resource should be read or manipulated and accessed at a specific URI (Uniform Resource Identifier). For example, to see a list of the users in your domain, you should use this URI: `https://api.example.com/users/`. To read the data for a specific user, the URI should have this form: `https://api.example.com/users/id1` (where `id1` is the ID of the user that you want to read).

To manipulate the users (add, remove, or update), the same URIs are used, but with different HTTP verbs. Table 5-1 contains some examples of REST APIs for manipulating a list of users.

Table 5-1. *REST API Cheat Sheet*

Action	URI	HTTP Verb
Read all users' data	`https://api.example.com/users/`	GET
Read data of user with ID "1"	`https://api.example.com/users/1`	GET
Create a new user	`https://api.example.com/users/`	POST
Replace user data with ID "1"	`https://api.example.com/users/1`	PUT
Update user data with ID "1"	`https://api.example.com/users/1`	PATCH
Delete the user with ID "1"	`https://api.example.com/users/1`	DELETE

The actions listed in this table are straightforward. The only topic that may need an explanation the difference between updating the data (with PATCH) and replacing the data (with PUT). When you use the verb PUT, you need to pass in the body of the HTTP requests the new user, complete in all its parts. When PATCH is used, the body should contain only the differences with the previous state. In this scenario, the `new todo` object is the result of the merging of the old todo with the request body.

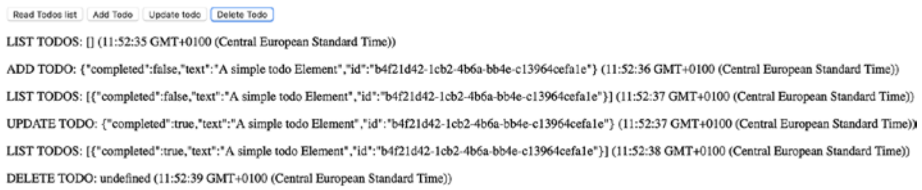
We have only scratched the surface of REST APIs. If you want to go deeper, I suggest reading *RESTful Web APIs: Services for a Changing World* by Leonard Richardson and Mike Amundsen (O'Reilly Media, 2013).

Code Examples

We are going to create three different HTTP clients with three different technologies: XMLHttpRequest, Fetch, and axios. We will analyze each client's strengths and weaknesses.

The Basic Structure

To show how our HTTP clients work, we will always use the same simple application that is shown in Figure 5-2. To keep the focus on the HTTP client, we are not going to use a TodoMVC application, but a simpler application with buttons that execute the HTTP requests and print the result on the screen. The code for this application (and the other implementations) is at `https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter05/`.



```
LIST TODOS: [] (11:52:35 GMT+0100 (Central European Standard Time))
ADD TODO: {"completed":false,"text":"A simple todo Element","id":"b4f21d42-1cb2-4b6a-bb4e-c13964cefa1e"} (11:52:36 GMT+0100 (Central European Standard Time))
LIST TODOS: [{"completed":false,"text":"A simple todo Element","id":"b4f21d42-1cb2-4b6a-bb4e-c13964cefa1e"}] (11:52:37 GMT+0100 (Central European Standard Time))
UPDATE TODO: {"completed":true,"text":"A simple todo Element","id":"b4f21d42-1cb2-4b6a-bb4e-c13964cefa1e"} (11:52:37 GMT+0100 (Central European Standard Time))
LIST TODOS: [{"completed":true,"text":"A simple todo Element","id":"b4f21d42-1cb2-4b6a-bb4e-c13964cefa1e"}] (11:52:38 GMT+0100 (Central European Standard Time))
DELETE TODO: undefined (11:52:39 GMT+0100 (Central European Standard Time))
```

Figure 5-2. *The application used to test the HTTP clients*

In Listing 5-2, you can see our application's `index.html`. Listing 5-3 shows the main controller.

Listing 5-2. HTML for HTTP Client Application

```
<html>

<body>
    <button data-list>Read Todos list</button>
    <button data-add>Add Todo</button>
    <button data-update>Update todo</button>
    <button data-delete>Delete Todo</button>
    <div></div>
</body>

</html>
```

Listing 5-3. Main Controller for HTTP Client Application

```
import todos from './todos.js'

const NEW_TODO_TEXT = 'A simple todo Element'

const printResult = (action, result) => {
  const time = (new Date()).toTimeString()
  const node = document.createElement('p')
  node.textContent = `${action.toUpperCase()}: ${JSON.
  stringify(result)} (${time})`

  document
    .querySelector('div')
    .appendChild(node)
}

const onListClick = async () => {
  const result = await todos.list()
  printResult('list todos', result)
}

const onAddClick = async () => {
  const result = await todos.create(NEW_TODO_TEXT)
  printResult('add todo', result)
}

const onUpdateClick = async () => {
  const list = await todos.list()

  const { id } = list[0]
  const newTodo = {
    id,
    completed: true
  }
  const result = await todos.update(newTodo)
  printResult('update todo', result)
}

const onDeleteClick = async () => {
  const list = await todos.list()
  const { id } = list[0]

  const result = await todos.delete(id)
  printResult('delete todo', result)
}

document
  .querySelector('button[data-list]')
  .addEventListener('click', onListClick)

document
  .querySelector('button[data-add]')
  .addEventListener('click', onAddClick)

document
  .querySelector('button[data-update]')
  .addEventListener('click', onUpdateClick)

document
  .querySelector('button[data-delete]')
  .addEventListener('click', onDeleteClick)
```

In this controller, I didn't use the HTTP client directly; instead, I wrapped the HTTP request in a `todos` model object. This kind of encapsulation is useful for a lot of reasons.

One of the reasons is testability. It's possible to replace the todos object with a mock that returns a static set of data (also called a *fixture*). In this way, I can test my controller in isolation.

Another reason is readability; model objects make your code more explicit.

Tip Never directly use HTTP clients in controllers. Try to encapsulate these functions in model objects.

Listing 5-4 shows the `todos` model object.

Listing 5-4. Todos Model Object

```
import http from './http.js'

const HEADERS = {
  'Content-Type': 'application/json'
}

const BASE_URL = '/api/todos'

const list = () => http.get(BASE_URL)

const create = text => {
  const todo = {
    text,
    completed: false
  }

  return http.post(
    BASE_URL,
    todo,
    HEADERS
  )
}
```

```
const update = newTodo => {
  const url = `${BASE_URL}/${newTodo.id}`
  return http.patch(
    url,
    newTodo,
    HEADERS
  )
}

const deleteTodo = id => {
  const url = `${BASE_URL}/${id}`
  return http.delete(
    url,
    HEADERS
  )
}

export default {
  list,
  create,
  update,
  delete: deleteTodo
}
```

The signature of our HTTP client is `http[verb](url, config)` for verbs that don't need a body, like GET or DELETE. For the other verbs, we can add the request body as a parameter, with this signature: `http[verb](url, body, config)`.

There is no rule that forces your team to use this kind of public API for an HTTP client. Another option is to use `http` as a function and not as an object, adding the verb as a parameter: `http(url, verb, body, config)`. Whatever you decide, try to keep it consistent.

Now that we have defined our HTTP client's public contract, it's time to look at the implementations.

XMLHttpRequest

The implementation in Listing 5-5 is based on XMLHttpRequest (`https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest`), which was W3C's first attempt to create a standard way to make asynchronous HTTP requests.

Listing 5-5. HTTP Client with XMLHttpRequest

```
const setHeaders = (xhr, headers) => {
  Object.entries(headers).forEach(entry => {
    const [
      name,
      value
    ] = entry

    xhr.setRequestHeader(
      name,
      value
    )
  })
}

const parseResponse = xhr => {
  const {
    status,
    responseText
  } = xhr
  let data
  if (status !== 204) {
    data = JSON.parse(responseText)
  }

  return {
    status,
    data
  }
}

const request = params => {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest()

    const {
      method = 'GET',
      url,
      headers = {},
      body
    } = params

    xhr.open(method, url)

    setHeaders(xhr, headers)

    xhr.send(JSON.stringify(body))

    xhr.onerror = () => {
      reject(new Error('HTTP Error'))
    }

    xhr.ontimeout = () => {
      reject(new Error('Timeout Error'))
    }
    xhr.onload = () => resolve(parseResponse(xhr))
  })
}

const get = async (url, headers) => {
  const response = await request({
    url,
    headers,
    method: 'GET'
  })

  return response.data
}

const post = async (url, body, headers) => {
  const response = await request({
    url,
    headers,
    method: 'POST',
    body
  })
  return response.data
}

const put = async (url, body, headers) => {
  const response = await request({
    url,
    headers,
    method: 'PUT',
    body
  })
  return response.data
}
```

```
const patch = async (url, body, headers) => {
  const response = await request({
    url,
    headers,
    method: 'PATCH',
    body
  })
  return response.data
}

const deleteRequest = async (url, headers) => {
  const response = await request({
    url,
    headers,
    method: 'DELETE'
  })
  return response.data
}

export default {
  get,
  post,
  put,
  patch,
  delete: deleteRequest
}
```

The core part of our HTTP client is the `request` method. XMLHttpRequest is an API defined in 2006, so it's based on callbacks. We have the `onload` callback for a completed request, the `onerror` callback for any HTTP that ends with an error, and the `ontimeout` callback for a request that times out. There is no timeout by default, but you can change it by modifying the `timeout` property of the `xhr` object.

The HTTP client's public API is based on promises (`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`). So the `request` method encloses the standard XMLHttpRequest request with a new `Promise` object. The public methods `get`, `post`, `put`, `patch`, and `delete` are wrappers around the `request` method (passing the appropriate parameters) to make the code more readable.

The following is the flow of an HTTP request with XMLHttpRequest (also see Figure 5-3).

1. Create a new XMLHttpRequest object (**`new XMLHttpRequest()`**).
2. Initialize the request to a specific URL (**`xhr.open(method, url)`**).
3. Configure the request (setting headers, timeout, etc.).
4. Send the request (**`xhr.send(JSON.stringify(body))`**).
5. Wait for the end of the request.
    a. If the request ends successfully, the **`onload`** callback is invoked.
    b. If the request ends with an error, the **`onerror`** callback is invoked.
    c. If the request times out, the **`ontimeout`** callback is invoked.

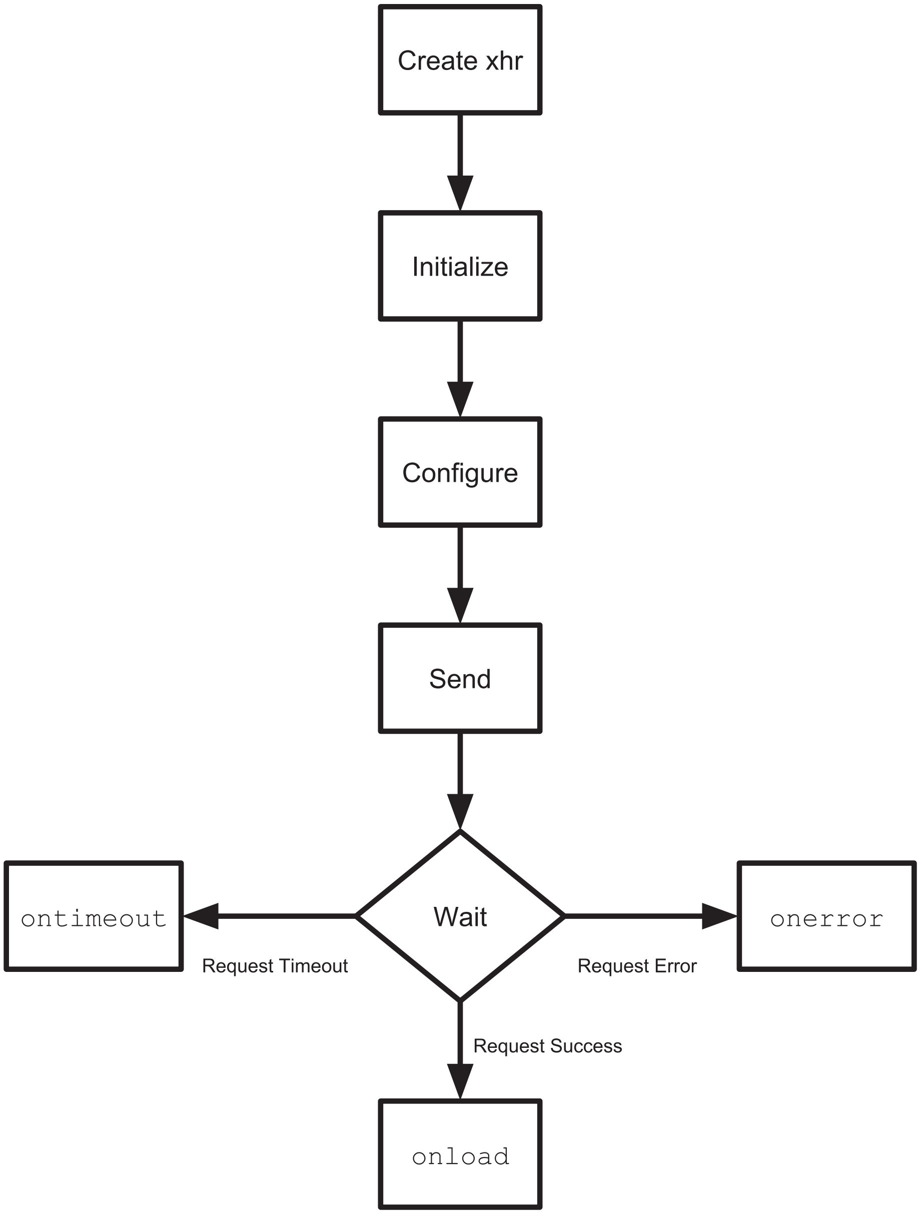


Figure 5-3. *Flow of an HTTP request with XMLHttpRequest*

Fetch

Fetch is a new API created for access to remote resources. Its purpose is to provide a standard definition for many network objects, such as Request or Response. This way, these objects are interoperable with other APIs, such as ServiceWorker (https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker) and Cache (https://developer.mozilla.org/en-US/docs/Web/API/Cache).

To create a request, you need to use the window.fetch method, as you can see in Listing 5-6, which is the implementation of the HTTP client made with the Fetch API.

Listing 5-6. HTTP Client Based on the Fetch API

```
const parseResponse = async response => {
  const { status } = response
  let data
  if (status !== 204) {
    data = await response.json()
  }

  return {
    status,
    data
  }
}

const request = async params => {
  const {
    method = 'GET',
    url,
    headers = {},
    body
  } = params
  const config = {
    method,
    headers: new window.Headers(headers)
  }

  if (body) {
    config.body = JSON.stringify(body)
  }

  const response = await window.fetch(url, config)

  return parseResponse(response)
}

const get = async (url, headers) => {
  const response = await request({
    url,
    headers,
    method: 'GET'
  })

  return response.data
}

const post = async (url, body, headers) => {
  const response = await request({
    url,
    headers,
    method: 'POST',
    body
  })
  return response.data
}

const put = async (url, body, headers) => {
  const response = await request({
    url,
    headers,
    method: 'PUT',
    body
  })
  return response.data
}

const patch = async (url, body, headers) => {
  const response = await request({
    url,
    headers,
    method: 'PATCH',
    body
  })
  return response.data
}

const deleteRequest = async (url, headers) => {
  const response = await request({
    url,
    headers,
    method: 'DELETE'
  })
  return response.data
}

export default {
  get,
  post,
  put,
  patch,
  delete: deleteRequest
}
```

This HTTP client has the same public API as the one built with XMLHttpRequest: a request function wrapped with a method for each HTTP verb that we want to use. The code of this second client is much more readable because of `window.fetch` returning a `Promise` object, so we don't need a lot of boilerplate code to transform the classic callback-based approach of XMLHttpRequest in a more modern promise-based one.

The promise returned by `window.fetch` resolves a `Response` object. We can use this object to extract the body of the response sent by the server. Depending on the format of the data received, there are different methods available; for example, `text()`, `blob()`, or `json()`. In our scenario, we always have JSON data, so it's safe to use `json()`.

Nevertheless, in a real-world application, you should use the right method accordingly with the `Content-Type` header. You can read the complete reference of all the objects of the Fetch API on the Mozilla Developer Network at `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch`.

Axios

The last HTTP client that we are going to analyze is built with axios, a small open source library. The documentation and the source code is on GitHub (`https://github.com/axios/axios`).

The main characteristic that differentiates axios from the other approaches is that it works out-of-the-box for browsers and Node.js. Its API is based on promises, and thus it's very similar to the Fetch API. In Listing 5-7, you can see the implementation of the HTTP client based on axios.

Listing 5-7. HTTP Client Based on Axios

```
const request = async params => {
  const {
    method = 'GET',
    url,
    headers = {},
    body
  } = params

  const config = {
    url,
    method,
    headers,
    data: body
  }

  return axios(config)
}

const get = async (url, headers) => {
  const response = await request({
    url,
    headers,
    method: 'GET'
  })

  return response.data
}

const post = async (url, body, headers) => {
  const response = await request({
    url,
    headers,
    method: 'POST',
    body
  })
  return response.data
}
```

```
const put = async (url, body, headers) => {
  const response = await request({
    url,
    headers,
    method: 'PUT',
    body
  })
  return response.data
}

const patch = async (url, body, headers) => {
  const response = await request({
    url,
    headers,
    method: 'PATCH',
    body
  })
  return response.data
}

const deleteRequest = async (url, headers) => {
  const response = await request({
    url,
    headers,
    method: 'DELETE'
  })
  return response.data
}

export default {
  get,
  post,
  put,
  patch,
  delete: deleteRequest
}
```

Among the three versions of the HTTP client, this is the easiest to read. The `request` method in this case rearranges the parameter to match the axios signature with the public contract.

Let's Review Our Architecture

Let's review our architecture. The three clients have the same public API. This characteristic of our architecture lets us change the library used for HTTP requests (XMLHttpRequest, Fetch, or axios) with minimal effort. JavaScript is a dynamically typed language, but all clients implement the HTTP client interface. Figure 5-4 shows a UML diagram that represents the relationship between the three implementations.

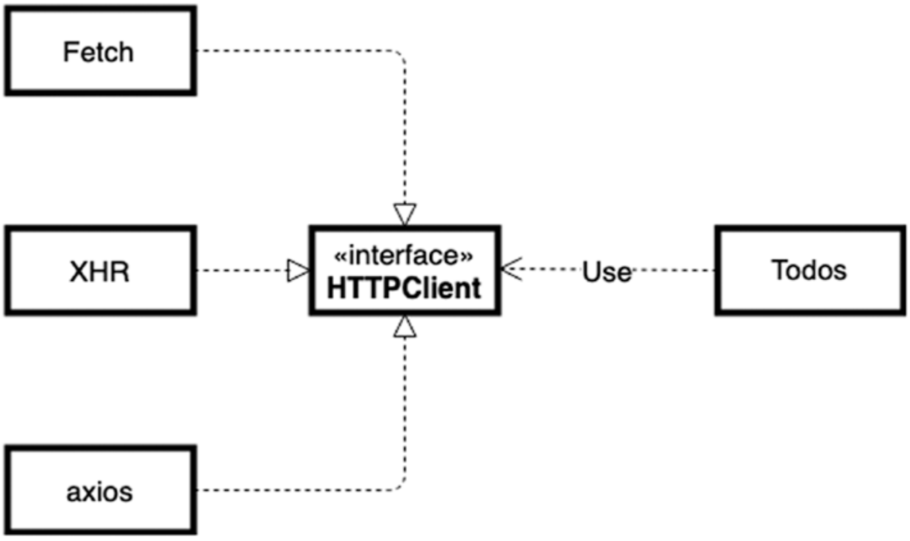


Figure 5-4. *UML diagram of HTTP client*

We applied one of the most important principles of software design.

Program to an interface, not an implementation.

—Gang of Four

This principle, found in the Gang of Four's book *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional, 1994), is very important when working with libraries.

Imagine having a very large application with dozens of model objects that need to access network resources. If we use axios directly, without using our HTTP client, changing the implementation to the Fetch API would be a very expensive (and tedious) task. Using axios in our model objects means programming to an implementation (the library) and not to an interface (the HTTP client).

Caution When using a library, always create an interface around it. It will be easier to change the library to a new one if you need to.

How to Choose the Right HTTP API

I talk about how to choose the "right" framework in the last chapter of this book. As I will explain more, there is no "right" framework. You can pick the "right" framework, but only for a "right" context.

So, for now, I will just point out the characteristics of XMLHttpRequest, the Fetch API, and axios from different perspectives.

Compatibility

If supporting Internet Explorer is important for your business, you have to rely on axios or XMLHttpRequest because the Fetch API works only on modern browsers.

Axios supports Internet Explorer 11, but if you need to work with older versions of Internet Explorer, you probably need to use XMLHttpRequest. Another option is to use a polyfill for the Fetch API (`https://github.github.io/fetch/`), but I suggest this approach only if your team plans to remove support for Internet Explorer very soon.

Portability

Both the Fetch API and XMLHttpRequest work only on browsers. If you need to run your code in other JavaScript environments, such as Node.js or React Native, axios is a very good solution.

Evolvability

One of the most important features of the Fetch API is to provide a standard definition of network-related objects like `Request` or `Response`. This characteristic makes the Fetch API a very useful library if you want to evolve your codebase quickly, because it fits seamlessly with new APIs like the ServiceWorker API or the Cache API.

Security

Axios has a built-in protection system against cross-site request forgery, or XSRF (`https://en.wikipedia.org/wiki/Cross-site_request_forgery`). If you stick with XMLHttpRequest or the Fetch API, and you need to implement this kind of security system, I suggest that you examine axios unit tests about this topic (`https://github.com/axios/axios/blob/master/test/specs/xsrf.spec.js`).

Learning Curve

If your code is based on axios or the Fetch API, it will be easier for newcomers to grasp the bulk of its meaning. XMLHttpRequest may look a bit strange to junior developers because they may not be used to working with callbacks. In that scenario, be sure to wrap the internal API with a promise, as I did in my example.

Summary

In this chapter, you learned about the rise of AJAX and how it changed web development. Then you looked at three distinct ways to implement an HTTP client. The first two were completely frameworkless and based on standard libraries (XMLHttpRequest and the Fetch API). The third one was based on an open source project called axios. Finally, you saw the differences between them from different points of view.

In the next chapter, you learn how to create a frameworkless routing system, an essential element in a single-page application.

CHAPTER 6

Routing

In the last chapter, I talked about AJAX and how it changed web development forever. The single-page application (SPA) is another very important technique that drastically changed the way users interact with web applications.

In this chapter, you learn what a SPA is and how to build one of its core features: the client-side routing system.

Single-Page Applications

A single-page application is a web application that runs inside a single HTML page. When the user navigates from one view to another, the application dynamically repaints the view, giving the illusion of standard web navigation. This approach removes the delay that users can experience when navigating between pages in a standard, multipage application. It usually provides a better user experience.

This kind of application relies on AJAX for any interactions with the server. Not every AJAX application has to be a SPA, however. Figure 6-1 shows the differences between a standard web application, a simple AJAX application, and a single-page application.

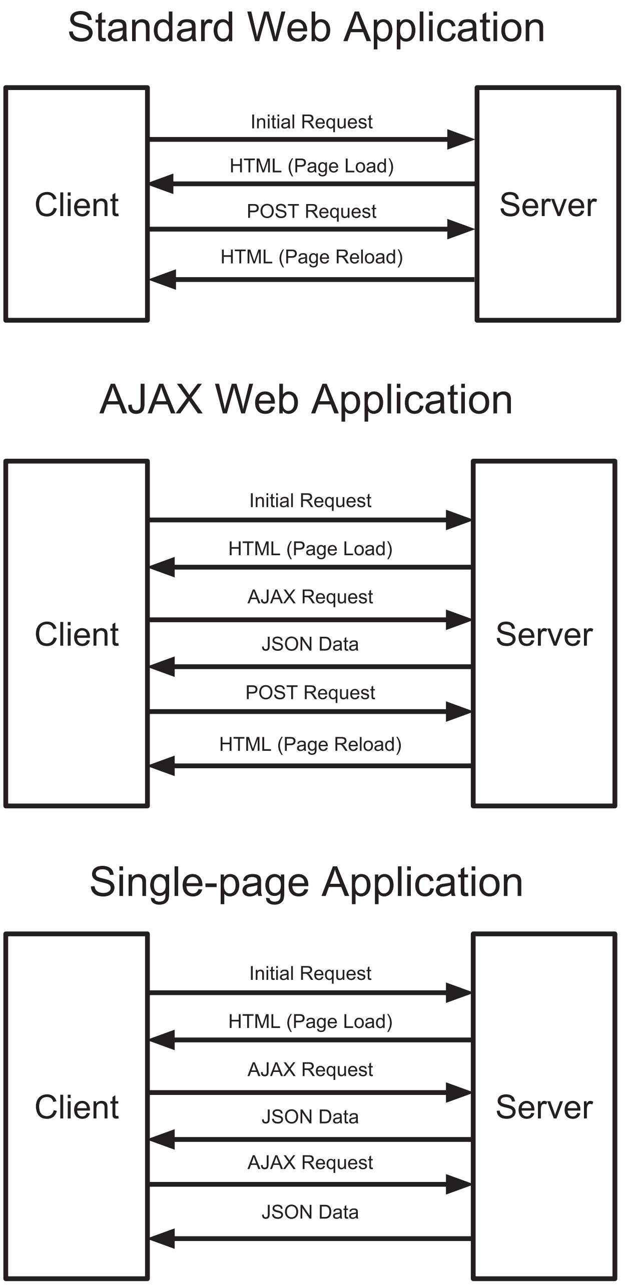


Figure 6-1. *Web application architectures comparison*

Frameworks like AngularJS and Ember contributed in making SPAs a mainstream approach for building web applications. These frameworks offer an out-of-the-box system to define routes via a routing system.

From an architectural point of view (see Figure 6-2), every routing system has at least two core elements. The first element is a *registry* that collects a list of our application's routes. A *route*, in its simplest form, is an object that maps a URL to a DOM component. The second important element is a listener on the current URL. When the URL changes, the router swaps the content of the body (or the main container) with the component bound to the route that matches the current URL.

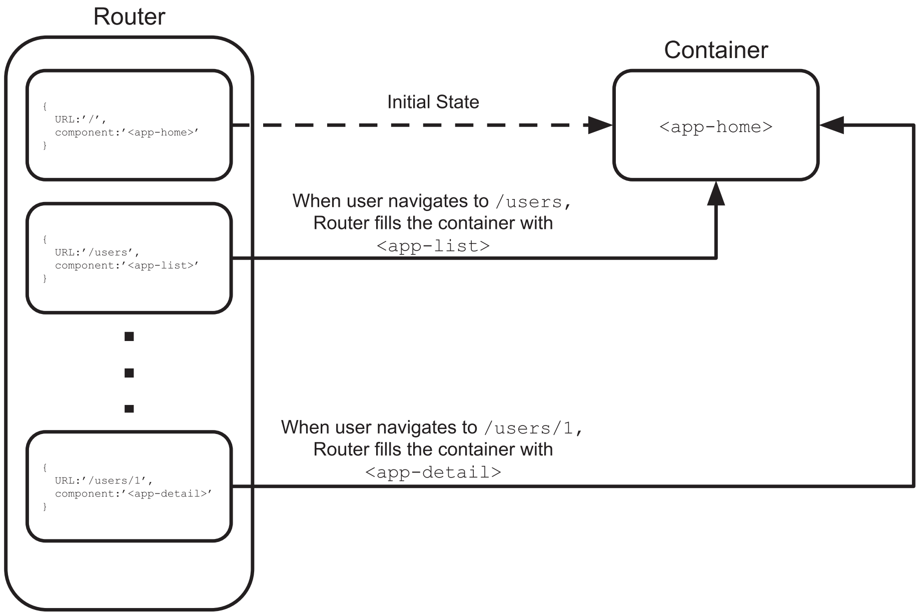

Router
{ URL:'/', component:'<app-home>' }
{ URL:'/users', component:'<app-list>' }
{ URL:'/users/1', component:'<app-detail>' }
Container
<app-home>
Initial State
When user navigates to /users, Router fills the container with <app-list>
When user navigates to /users/1, Router fills the container with <app-detail>

Figure 6-2. *Routing system high-level architecture*

Code Examples

Similar to the last chapter, we are going to create three versions of a routing system. We will begin two frameworkless approaches: one based on fragment identifiers and one based on the History API. Finally, we will use an open source library called Navigo.

All the code examples are available at https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter06.

Fragment Identifier

Every URL can contain an optional part introduced by a hash, which is called a *fragment identifier*. Its purpose is to identify a specific section of a web page. Let's use www.domain.org/foo.html#bar as an example. In this URL, bar is the fragment identifier. It is identifying the HTML element with id="bar".

When navigating a URL that contains a fragment identifier, browsers scroll the page until the element identified by the fragment is at the top of the viewport. We will use fragment identifiers to implement our first Router object. We will start with a simple example and make it more complete in an iterative way.

The First Example

In this example, we will build a very simple SPA with links and a main container. Listing 6-1 is the HTML template of this application.

Listing 6-1. Basic SPA Template

```
<body>
    <header>
        <a href="#/">Go To Index</a>
        <a href="#/list">Go To List</a>
        <a href="#/dummy">Dummy Page</a>
    </header>
    <main>
    </main>
</body>
```

By using anchors in the header, our URL will change `http://localhost:8080#/` to `http://localhost:8080#/list`, and so on. Our code injects the current component inside the main container when the URL changes. In this simple use case, the components are plain functions that update the content of a DOM element, as you can see in Listing 6-2.

Listing 6-2. Basic SPA Components

```
export default container => {
  const home = () => {
    container
      .textContent = 'This is Home page'
  }

  const list = () => {
    container
      .textContent = 'This is List Page'
  }

  return {
    home,
    list
  }
}
```

When defining routes, it's a good practice to define a "not found" component to show users when they navigate to a URL that is not linked to any component. This component (see Listing 6-3) should have the same structure as every other component.

Listing 6-3. A Not Found Component

```
const notFound = () => {
  container
    .textContent = 'Page Not Found!'
}
```

To make the router work, we need to configure it and link the component to the right fragment. Listing 6-4 shows how to configure our router.

Listing 6-4. Configuring a Basic Router

```
import createRouter from './router.js'
import createPages from './pages.js'

const container = document.querySelector('main')

const pages = createPages(container)

const router = createRouter()

router
  .addRoute('#/', pages.home)
  .addRoute('#/list', pages.list)
  .setNotFound(pages.notFound)
  .start()
```

The router has three public methods. The `addRoute` method defines a new route, a configuration object formed by the fragment, and the component. The `setNotFound` method sets a generic component for any fragment that is not present in the registry. The `start` method initializes the router and starts to listen for URL changes.

Now that we have analyzed the public interface of the router, it's time to look at the implementation in Listing 6-5.

Listing 6-5. Basic Router Implementation

```
export default () => {
  const routes = []
  let notFound = () => {}

  const router = {}

  const checkRoutes = () => {
    const currentRoute = routes.find(route => {
      return route.fragment === window.location.hash
    })

    if (!currentRoute) {
      notFound()
      return
    }

    currentRoute.component()
  }

  router.addRoute = (fragment, component) => {
    routes.push({
      fragment,
      component
    })

    return router
  }

  router.setNotFound = cb => {
    notFound = cb
    return router
  }
  router.start = () => {
    window
      .addEventListener('hashchange', checkRoutes)

    if (!window.location.hash) {
      window.location.hash = '#/'
    }

    checkRoutes()
  }

  return router
}
```

As you can see, the current fragment identifier is stored in the `hash` property of the `location` object. There's also a very handy `hashchange` event that we can use to be notified every time that the current fragment changes.

The `checkRoutes` method is the router's core method. It looks for the route that matches with the current fragment. If a route is found, its corresponding component function replaces the content that is present in the main container; otherwise, the generic `notFound` function is invoked. This method is called when the router starts and every time the `hashchange` event is fired. Figure 6-3 is a diagram of the router's flow.

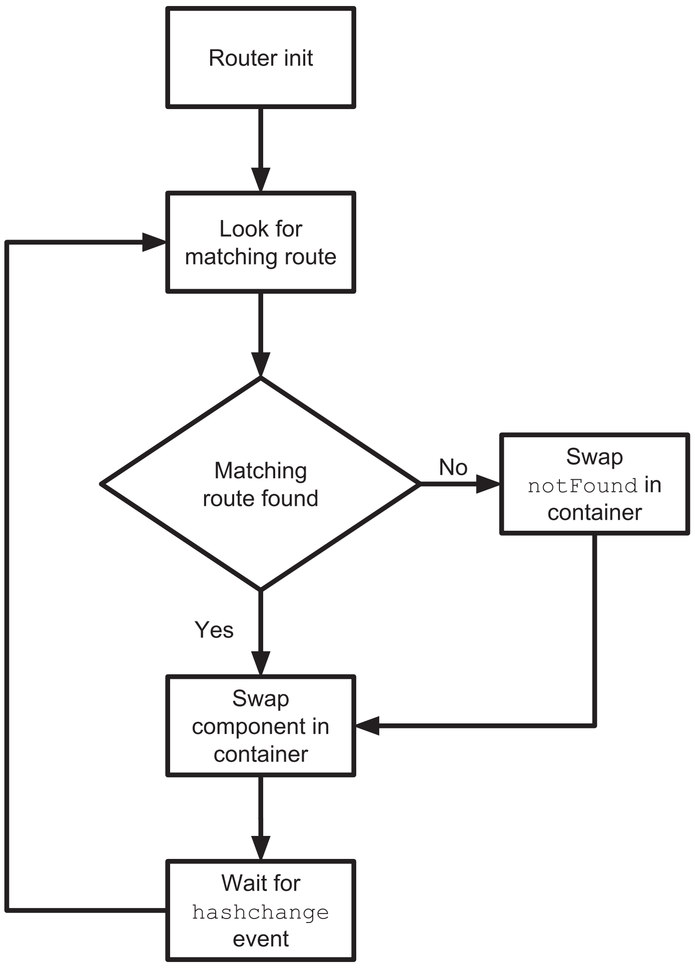

Router init
Look for matching route
Matching route found
No
Swap notFound in container
Yes
Swap component in container
Wait for hashchange event

Figure 6-3. *Router flow*

Navigating Programmatically

In the previous example, the navigation was activated in the classic way of clicking an anchor. Sometimes, however, you need to change view *programmatically*. Think, for example, at redirecting the user to their personal page after a successful login. To do that, let's change a bit our application, like in Listing 6-6, by swapping the links in the header with buttons.

Listing 6-6. Adding Data Attributes to Buttons

```
<body>
    <header>
        <button data-navigate="/">
          Go To Index
        </button>
        <button data-navigate="/list">
          Go To List
        </button>
        <button data-navigate="/dummy">
        Dummy Page
      </button>
    </header>
    <main>
    </main>
</body>
```

Now we have to add an event handler for the buttons in our controller, as shown in Listing 6-7.

Listing 6-7. Adding Navigation to Buttons

```
const NAV_BTN_SELECTOR = 'button[data-navigate]'

document
  .body
  .addEventListener('click', e => {
    const { target } = e
    if (target.matches(NAV_BTN_SELECTOR)) {
      const { navigate } = target.dataset
      router.navigate(navigate)
    }
  })
```

To navigate to another view programmatically, I created a new public method for the router. This method gets the new fragment and replaces it in the `location` object. You can see the code for the `navigate` method in Listing 6-8.

Listing 6-8. Navigating Programmatically

```
router.navigate = fragment => {
  window.location.hash = fragment
}
```

It's important to wrap this line in a function to keep a standard interface when changing the internals of the router.

Route Parameters

The last feature that we will add to our router is reading route parameters. A route parameter is a part of the URL that is relative to a domain variable. For example, we can get the ID of an "order" domain model from `http://localhost:8080#/order/1`. In this case, the 1 is a route parameter called `id`.

When creating a route with a parameter, this form usually indicates that the URL contains a parameter: `http://localhost:8080#/user/:id`. I will stick to this de facto standard in my implementation.

The first thing to do is slightly modify the component, as shown in Listing 6-9, in order to let them accept an argument. This argument will be filled with route parameters.

Listing 6-9. Components with Parameters

```
const detail = (params) => {
  const { id } = params
  container
    .textContent = `
       This is Detail Page with Id ${id}
     `
}

const anotherDetail = (params) => {
  const { id, anotherId } = params
  container
    .textContent = `
      This is another Detail Page with Id ${id}
      and AnotherId ${anotherId}
    `
}
```

Listing 6-10 shows how to bind these two new components with the relative URLs.

Listing 6-10. Defining Routes with Parameters

```
router
  .addRoute('#/', pages.home)
  .addRoute('#/list', pages.list)
  .addRoute('#/list/:id', pages.detail)
  .addRoute('#/list/:id/:anotherId', pages.anotherDetail)
  .setNotFound(pages.notFound)
  .start()
```

Now it's time to modify the router implementation in order to manage the routes parameters. This implementation will be strongly based on regular expressions (also known as regex). If you feel uncomfortable with regular expressions (like I do), I suggest using Regex101 (https://regex101.com), which is a very helpful tool for understanding what a specific regular expression does.

The first thing to do is to extract the parameter names from the URL that is used as the first argument of the addRoute method. For example, from #/list/:id/:anotherId, we have to extract an array with id and anotherId. You can see how to do that in Listing 6-11.

Listing 6-11. Extracting Parameters Name from Fragments

```
const ROUTE_PARAMETER_REGEXP = /:(\w+)/g
const URL_FRAGMENT_REGEXP = '([^\\/]+)'

router.addRoute = (fragment, component) => {
  const params = []

  const parsedFragment = fragment
    .replace(
      ROUTE_PARAMETER_REGEXP,
      (match, paramName) => {
        params.push(paramName)
        return URL_FRAGMENT_REGEXP
      })
    .replace(/\//g, '\\/')

  routes.push({
    testRegExp: new RegExp(`^${parsedFragment}$`),
    component,
    params
  })

  return router
}
```

To extract parameter names from the fragment, use this regex: `/:(\w+)/g`. It matches with `:id` and `:anotherId`. You can use the following schema to better understand the purpose of this regex.

- `:(\w+)`
    - `:` matches the exact character :
    - `()` indicates the start of a capturing group
    - `\w` is a shortcut for `[a-zA-Z0-9_]` and matches any standard character
    - `+` indicates that we accept at least one occurrence of a standard character

This regex is used with the `replace` function of the `String` object. The callback is called for every match of the regex with the target `String` (in this case, the fragment). The second argument of this callback is the name of the parameter, which we add in the `params` array. Then the match is replaced with another regex snippet: `([^\\/]+)`. Finally, we wrap our new fragment between a ^ and a `$`.

So, passing the fragment `#/list/:id/:anotherId` as an argument to the `addRoute` method will result in `testRegExp` with the value `^#\/list\/([^\\/]+)\/([^\\/]+)$`, which we will use when checking if this route matches with the current fragment in the `location` object.

Here's a schema that explains the regex.

- `^#\/list\/([^\\/]+)\/([^\\/]+)$`
    - ^ indicates the beginning of the string
    - `#\/list\/` matches with the exact string #/list/
    - `()` indicates the start of the first capturing group
    - `[^\\/]` matches any character apart from / or \
    - + indicates that we accept at least one occurrence of the previous match

- `()` indicates the start of the second capturing group
- `[^\\/]` matches any character apart from / or \
- `+` indicates that we accept at least one occurrence of the previous match
- `$` indicates the end of the string

Now that we have parsed our fragments, the generated regular expressions are used to select the right route for the current fragment and extract the actual parameters, as shown in Listing 6-12.

Listing 6-12. Extracting the URL Params from the Current Fragment

```
const extractUrlParams = (route, windowHash) => {
  if (route.params.length === 0) {
    return {}
  }

  const params = {}

  const matches = windowHash
    .match(route.testRegExp)

  matches.shift()

  matches.forEach((paramValue, index) => {
    const paramName = route.params[index]
    params[paramName] = paramValue
  })

  return params
}

const checkRoutes = () => {
  const { hash } = window.location
  const currentRoute = routes.find(route => {
    const { testRegExp } = route
    return testRegExp.test(hash)
  })

  if (!currentRoute) {
    notFound()
    return
  }

  const urlParams = extractUrlParams(
    currentRoute,
    window.location.hash
  )

  currentRoute.component(urlParams)
}
```

As you can see, we use `testRegExp` to check if the current fragment matches with one of the routes in the registry, and then we use the same regex to extract the parameters that will be used as arguments for our component functions.

Notice the usage of `shift` in `extractUrlParams`. The `String.matches` method returns an array where the first element is the match itself, while the other elements are the result of the capturing groups. With `shift`, we remove the first element from that array.

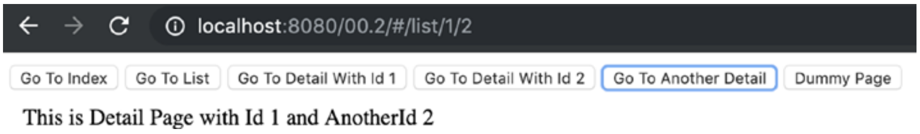
This is a recap of what happens when managing a route with parameters.

1. The fragment `#/list/:id/:anotherId` is passed to the `addRoute` method.

2. The `addRoute` method extracts the two parameter names (`id` and `anotherId`) and transforms the fragment in the regex `^#\/list\/([^\\/]+)\/([^\\/]+)$`.

3. When the user navigates a matching fragment, such as `#/list/1/2`, the `checkRoutes` method selects the right route thanks to the regex.
4. The `extractUrlParams` method extracts the actual parameters from the current fragment in this object: `{id:1, anotherId:2}`.
5. The object is passed to the component function that updates the DOM.

Figure 6-4 shows what the user sees when navigating to `#/list/1/2`.



This is Detail Page with Id 1 and AnotherId 2

Figure 6-4. *Example Project with Route Parameters*

Listing 6-13 is the complete code for the router based on fragment identifiers.

Listing 6-13. Router Based on Fragment Identifiers

```
const ROUTE_PARAMETER_REGEXP = /:(\w+)/g
const URL_FRAGMENT_REGEXP = '([^\\/]+)'

const extractUrlParams = (route, windowHash) => {
  const params = {}

  if (route.params.length === 0) {
    return params
  }

  const matches = windowHash
    .match(route.testRegExp)

  matches.shift()
  matches.forEach((paramValue, index) => {
    const paramName = route.params[index]
    params[paramName] = paramValue
  })

  return params
}

export default () => {
  const routes = []
  let notFound = () => {}

  const router = {}

  const checkRoutes = () => {
    const { hash } = window.location

    const currentRoute = routes.find(route => {
      const { testRegExp } = route
      return testRegExp.test(hash)
    })

    if (!currentRoute) {
      notFound()
      return
    }

    const urlParams = extractUrlParams(
      currentRoute,
      window.location.hash
    )

    currentRoute.component(urlParams)
  }
```

```
router.addRoute = (fragment, component) => {
  const params = []

  const parsedFragment = fragment
    .replace(
      ROUTE_PARAMETER_REGEXP,
      (match, paramName) => {
        params.push(paramName)
        return URL_FRAGMENT_REGEXP
      })
    .replace(/\//g, '\\/')

  console.log(`^${parsedFragment}$`)

  routes.push({
    testRegExp: new RegExp(`^${parsedFragment}$`),
    component,
    params
  })

  return router
}

router.setNotFound = cb => {
  notFound = cb
  return router
}

router.navigate = fragment => {
  window.location.hash = fragment
}
```

```
  router.start = () => {
    window
      .addEventListener(
        'hashchange',
        checkRoutes
      )

    if (!window.location.hash) {
      window.location.hash = '#/'
    }

    checkRoutes()
  }

  return router
}
```

Note The public API of this first implementation will be the base of the other implementations that we cover in this chapter.

History API

The History API lets developers manipulate the users browsing history. For the second implementation of the router, we are going to use this API, or to be exact, one of its methods. Table 6-1 is a small cheat sheet for the History API; for the complete reference, take a look at the dedicated page on the Mozilla Development Network (`https://developer.mozilla.org/en-US/docs/Web/API/History`).

Table 6-1. *History API Cheat Sheet*

Signature	Description
`back()`	Goes to the previous page in the history.
`forward()`	Goes to the next page in the history.
`go(index)`	Goes to a specific page in the history.
`pushState(state, title, URL)`	Pushes the data in the history stack and navigate to the provided URL.
`replaceState(state, title, URL)`	Replaces the most recent data in the history stack and navigates to the provided URL.

When using the History API for routing, we don't need to base our routes on fragment identifiers. Instead, we can utilize a real URL, like `http://localhost:8080/list/1/2`.

Listing 6-14 is a version based on the History API.

Listing 6-14. Router Built with the History API

```
const ROUTE_PARAMETER_REGEXP = /:(\w+)/g
const URL_FRAGMENT_REGEXP = '([^\\/]+)'
const TICKTIME = 250

const extractUrlParams = (route, pathname) => {
  const params = {}

  if (route.params.length === 0) {
    return params
  }

  const matches = pathname
    .match(route.testRegExp)

  matches.shift()
  matches.forEach((paramValue, index) => {
    const paramName = route.params[index]
    params[paramName] = paramValue
  })

  return params
}

export default () => {
  const routes = []
  let notFound = () => {}
  let lastPathname

  const router = {}

  const checkRoutes = () => {
    const { pathname } = window.location
    if (lastPathname === pathname) {
      return
    }

    lastPathname = pathname

    const currentRoute = routes.find(route => {
      const { testRegExp } = route
      return testRegExp.test(pathname)
    })

    if (!currentRoute) {
      notFound()
      return
    }

    const urlParams = extractUrlParams(currentRoute, pathname)

    currentRoute.callback(urlParams)
  }
```

```
router.addRoute = (path, callback) => {
  const params = []

  const parsedPath = path
    .replace(
      ROUTE_PARAMETER_REGEXP,
      (match, paramName) => {
        params.push(paramName)
        return URL_FRAGMENT_REGEXP
      })
    .replace(/\//g, '\\/')

  routes.push({
    testRegExp: new RegExp(`^${parsedPath}$`),
    callback,
    params
  })

  return router
}

router.setNotFound = cb => {
  notFound = cb
  return router
}

router.navigate = path => {
  window
    .history
    .pushState(null, null, path)
}
```

```
  router.start = () => {
    checkRoutes()
    window.setInterval(checkRoutes, TICKTIME)
  }

  return router
}
```

I will point out the differences between this listing and Listing 6-13 based on fragment identifiers. The `pushState` method is only method that we need from the History API to navigate to a new URL.

The greatest difference between Listing 6-14 and Listing 6-13 is the absence of a DOM event that we can employ to be notified when the URL changes. To achieve a similar result, I used `setInterval` to regularly check if the path name has changed.

The public API is unchanged. The only thing that we need to adapt is the routes by removing the hash at the beginning, as shown in Listing 6-15.

Listing 6-15. Defining Routes Without Fragment Identifiers

```
router
  .addRoute('/', pages.home)
  .addRoute('/list', pages.list)
  .addRoute('/list/:id', pages.detail)
  .addRoute('/list/:id/:anotherId', pages.anotherDetail)
  .setNotFound(pages.notFound)
  .start()
```

Using Links

To switch completely to the History API, we need to update the links that we have in our template. Listing 6-16 is an updated version of the initial template of our sample application. In this case, the links point to a real URL rather than fragment identifiers of the same page.

Listing 6-16. History API Link Navigation

```
<header>
  <a href="/">Go To Index</a>
  <a href="/list">Go To List</a>
  <a href="/list/1">Go To Detail With Id 1</a>
  <a href="/list/2">Go To Detail With Id 2</a>
  <a href="/list/1/2">Go To Another Detail</a>
  <a href="/dummy">Dummy Page</a>
</header>
```

Simply changing the `href` attribute is not enough. These links will not work as expected; for example, clicking the Go To List link will navigates to `http://localhost:8080/list/index.html`, which causes a 404 HTTP error.

To make these links work, we need to change their *default behavior*. The first thing to do is mark all the links that are used for internal navigation, as shown in Listing 6-17.

Listing 6-17. History API Navigation Marked Links

```
<header>
  <a data-navigation href="/">Go To Index</a>
  <a data-navigation href="/list">Go To List</a>
  <a data-navigation href="/list/1">Go To Detail With Id 1</a>
  <a data-navigation href="/list/2">Go To Detail With Id 2</a>
  <a data-navigation href="/list/1/2">Go To Another Detail</a>
  <a data-navigation href="/dummy">Dummy Page</a>
</header>
```

We easily recognize these links in Listing 6-18, which disables standard navigation and uses the routers `navigate` method.

Listing 6-18. Changing the Behavior of Internal Navigation Links

```
const NAV_A_SELECTOR = 'a[data-navigation]'

router.start = () => {
  checkRoutes()
  window.setInterval(checkRoutes, TICKTIME)

  document
    .body
    .addEventListener('click', e => {
      const { target } = e
      if (target.matches(NAV_A_SELECTOR)) {
        e.preventDefault()
        router.navigate(target.href)
      }
    })

  return router
}
```

The router intercept clicks every internal navigation anchor by using the event delegation technique discussed in Chapter 3. It's possible to disable the standard handler of any DOM element with the `preventDefault` method of the `Event` object.

Navigo

The last implementation that we will analyze in this chapter is based on Navigo, a very simple and small open source library that you can find on GitHub at `https://github.com/krasimir/navigo`.

It's very important to wrap any library with your own public interface. In the implementation shown in Listing 6-19, we keep the same API, but change the internals of the router itself.

Listing 6-19. Router Implementation with Navigo

```
export default () => {
  const navigoRouter = new window.Navigo()
  const router = {}

  router.addRoute = (path, callback) => {
    navigoRouter.on(path, callback)
    return router
  }

  router.setNotFound = cb => {
    navigoRouter.notFound(cb)
    return router
  }

  router.navigate = path => {
    navigoRouter.navigate(path)
  }

  router.start = () => {
    navigoRouter.resolve()
    return router
  }

  return router
}
```

Managing the internal navigation links is very similar to the previous implementation. We just need to change `data-navigation` in `data-navigo`, as you can see in Listing 6-20.

Listing 6-20. Internal Navigation Links with Navigo

```
<header>
  <a data-navigo href="/">Go To Index</a>
  <a data-navigo href="/list">Go To List</a>
  <a data-navigo href="/list/1">Go To Detail With Id 1</a>
  <a data-navigo href="/list/2">Go To Detail With Id 2</a>
  <a data-navigo href="/list/1/2">Go To Another Detail</a>
  <a data-navigo href="/dummy">Dummy Page</a>
</header>
```

How to Choose the Right Router

There is no meaningful difference between the three implementations. Just remember that the History API is not supported on Internet Explorer 9 or lower, but it's safe to say that it's not a problem anymore. My suggestion is to start with a frameworkless implementation, and switch to a third-party library only if you need something very complex.

Routing is the nervous system of any SPA. It decides how to match URLs with what the users see on their screens. Keep this in mind when working with a framework too. If you use a React router for a project, you probably will not be able to remove React from your project because it's very hard to change the routing system of a single-page application. Nevertheless, if your routing system is independent, you can start changing the framework one view at a time.

Tip When using a framework, try to keep a separate layer for routing.

Summary

In this chapter, you learned about single-page applications. You also learned how to create a client-side routing system. You built two different frameworkless versions of a router; one was based on fragment identifiers and one was based on the History API. Finally, you created a router based on Navigo, a very small JavaScript library.

In the next chapter, I cover how to manage the state of our applications with different state management techniques.

CHAPTER 7

State Management

In the previous chapters, you learned how to display data and how to manage user inputs, and made HTTP requests and managed routes. You can consider these skills the basic building blocks. But before you can actually start writing frameworkless code, you need to know how to manage the data (or the *state*) that links all of these elements together. In front-end applications or more generally, in all kinds of client applications (web, desktop, and mobile) effectively managing data is called *state management*.

State management doesn't solve a new problem. Model-view-controller (the most famous state management pattern) was introduced in the 1970s. When React became a mainstream library, the term started to show up in blogs, conferences, and so on. Today, there are dedicated libraries for front-end state management. Some of them are tied to existing frameworks, such as Vuex (for Vue.js) and NgRx (for Angular), while other libraries are agnostic, like MobX and Redux.

Choosing the right architecture for your state management code is crucial in keeping the application healthy and maintainable. In this chapter, we will build three state management strategies, compare them, and analyze their pros and cons.

Let's Review the TodoMVC Application

As a base for the examples in this chapter, we will use TodoMVC, which we developed in Chapter 3, with a functional rendering engine. In Listing 7-1, you can see the code of the controller with all the events to manipulate the todos and the filter.

The complete code for this application is available at `https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter07/00`.

Listing 7-1. TodoMVC Controller

```
import todosView from './view/todos.js'
import counterView from './view/counter.js'
import filtersView from './view/filters.js'
import appView from './view/app.js'
import applyDiff from './applyDiff.js'

import registry from './registry.js'

registry.add('app', appView)
registry.add('todos', todosView)
registry.add('counter', counterView)
registry.add('filters', filtersView)

const state = {
  todos: [],
  currentFilter: 'All'
}
```

```
const events = {
  addItem: text => {
    state.todos.push({
      text,
      completed: false
    })
    render()
  },
  updateItem: (index, text) => {
    state.todos[index].text = text
    render()
  },
  deleteItem: (index) => {
    state.todos.splice(index, 1)
    render()
  },
  toggleItemCompleted: (index) => {
    const {
      completed
    } = state.todos[index]
    state.todos[index].completed = !completed
    render()
  },
  completeAll: () => {
    state.todos.forEach(t => {
      t.completed = true
    })
    render()
  },
  clearCompleted: () => {
    state.todos = state.todos.filter(
      t => !t.completed
    )
    render()
  },
  changeFilter: filter => {
    state.currentFilter = filter
    render()
  }
}

const render = () => {
  window.requestAnimationFrame(() => {
    const main = document.querySelector('#root')

    const newMain = registry.renderRoot(
      main,
      state,
      events)

    applyDiff(document.body, main, newMain)
  })
}

render()
```

The state management code is defined in the events object, which we pass to our View function to attach its methods to DOM handlers.

Model-View-Controller

To keep your state in the controllers is not a good way to manage it. The first step in enhancing our design is simply moving all that code to a separate file. Listing 7-2 is an updated version of the controller with an external model that manages the state of the application.

Listing 7-2. Controller with Separated Model

```
import modelFactory from './model/model.js'

const model = modelFactory()

const events = {
  addItem: text => {
    model.addItem(text)
    render(model.getState())
  },
  updateItem: (index, text) => {
    model.updateItem(index, text)
    render(model.getState())
  },
  deleteItem: (index) => {
    model.deleteItem(index)
    render(model.getState())
  },
  toggleItemCompleted: (index) => {
    model.toggleItemCompleted(index)
    render(model.getState())
  },
  completeAll: () => {
    model.completeAll()
    render(model.getState())
  },
  clearCompleted: () => {
    model.clearCompleted()
    render(model.getState())
  },
  changeFilter: filter => {
    model.changeFilter(filter)
    render(model.getState())
  }
}

const render = (state) => {
  window.requestAnimationFrame(() => {
    const main = document.querySelector('#root')

    const newMain = registry.renderRoot(
      main,
      state,
      events)

    applyDiff(document.body, main, newMain)
  })
}

render(model.getState())
```

Notice that the actual data used to render is returned from the getState method of the model object. You can see its code in Listing 7-3. For simplicity, I reported only the addItem and the updateItem methods (this is done in all the other listings regarding the model in this chapter). To check the complete code, you can visit the GitHub repository at https://github.com/Apress/frameworkless-front-end-development/blob/master/Chapter07/00/model/model.js.

Listing 7-3. Simple Model Object for a TodoMVC Application

```
const cloneDeep = x => {
  return JSON.parse(JSON.stringify(x))
}

const INITIAL_STATE = {
  todos: [],
  currentFilter: 'All'
}

export default (initalState = INITIAL_STATE) => {
  const state = cloneDeep(initalState)

  const getState = () => {
    return Object.freeze(cloneDeep(state))
  }

  const addItem = text => {
    if (!text) {
      return
    }

    state.todos.push({
      text,
      completed: false
    })
  }

  const updateItem = (index, text) => {
    if (!text) {
      return
    }
    if (index < 0) {
      return
    }

    if (!state.todos[index]) {
      return
    }

    state.todos[index].text = text
  }

  //Other methods...

  return {
    addItem,
    updateItem,
    deleteItem,
    toggleItemCompleted,
    completeAll,
    clearCompleted,
    changeFilter,
    getState
  }
}
```

The values extracted from a `model` object should be *immutable*. You do that by generating a clone every time `getState` is invoked, and then by freezing it with `Object.freeze` (`https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze`). To clone the object, I used the `parse` and `serialize` methods of the JSON object. I first serialize the state to a string, and then I parse back the object from the JSON string, obtaining a deep clone of the original object. This approach can be slow, and I used it here for simplicity. In a real-world application, I usually rely on Lodash's `cloneDeep` function (www.npmjs.com/package/lodash.clonedeep). I will use these two functions in most of the other listings in this chapter; but to keep the listings short, I will not show the definition.

Using an immutable state to transfer data forces the consumers of this API to use public methods to manipulate the state. This way, the business logic is completely contained in the `Model` object and not scattered in various parts of the application. This approach helps keep our state management code with a high level of testability during the lifespan of our codebase. Listing 7-4 shows part of the test suite for the `Model` object.

Listing 7-4. Test Suite for TodoMVC State Object

```
import stateFactory from './state.js'

describe('external state', () => {
  test('data should be immutable', () => {
    const state = stateFactory()

    expect(() => {
      state.get().currentFilter = 'WRONG'
    }).toThrow()
  })

  test('should add an item', () => {
    const state = stateFactory()

    state.addItem('dummy')

    const { todos } = state.get()

    expect(todos.length).toBe(1)
    expect(todos[0]).toEqual({
      text: 'dummy',
      completed: false
    })
  })
```

```
test('should not add an item when a falsy text is provided',
() => {
  const state = stateFactory()

  state.addItem('')
  state.addItem(undefined)
  state.addItem(0)
  state.addItem()
  state.addItem(false)

  const { todos } = state.get()

  expect(todos.length).toBe(0)
})

test('should update an item', () => {
  const state = stateFactory({
    todos: [{
      text: 'dummy',
      completed: false
    }]
  })

  state.updateItem(0, 'new-dummy')

  const { todos } = state.get()

  expect(todos[0].text).toBe('new-dummy')
})
```

```
  test('should not update an item when an invalid index is
  provided', () => {
    const state = stateFactory({
      todos: [{
        text: 'dummy',
        completed: false
      }]
    })

    state.updateItem(1, 'new-dummy')

    const { todos } = state.get()

    expect(todos[0].text).toBe('dummy')
  })
})
```

This first version of a state management library for a TodoMVC application is a classic model-view-controller (MVC) implementation. Historically, MVC is one of the first patterns dedicated to managing the state of a client application. You can see a schema of this pattern in Figure 7-1.

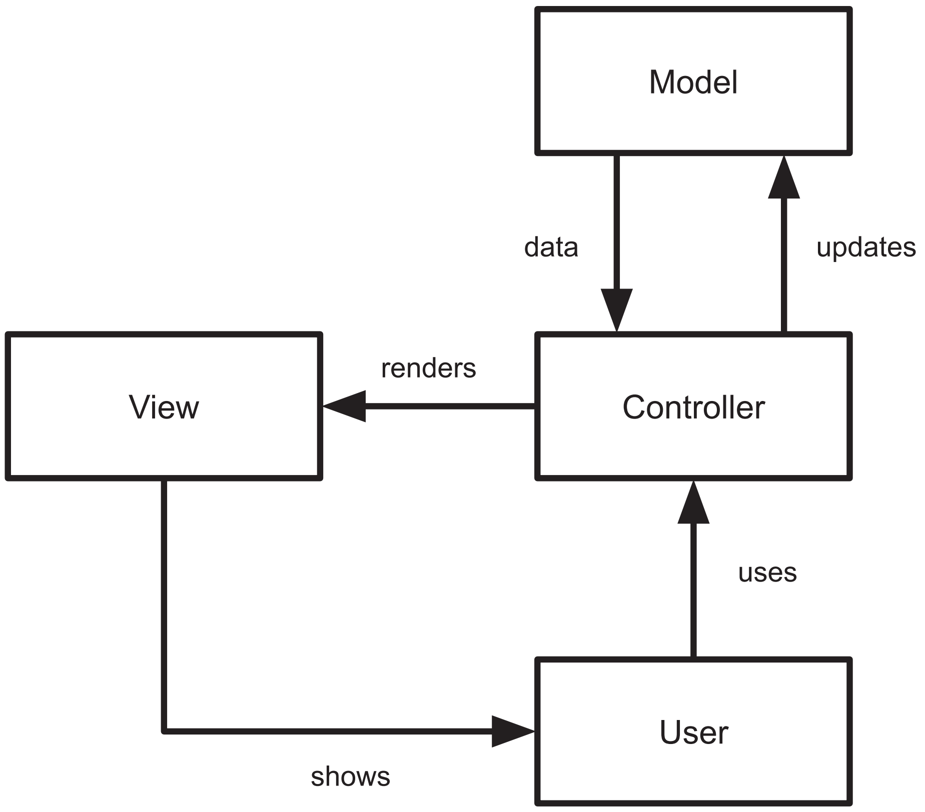


Figure 7-1. *MVC pattern schema*

This model object will be the base for all the other implementations. So, before continuing, let's review the workflow of our application and the relationship between its parts.

1. The controller gets the initial state from the model.
2. The controller invokes the view to render the initial state.
3. The system is ready to receive user inputs.
4. The user does something (for example, they add an item).

5. The controller maps the user action with the correct `Model` method (`model.addItem`).
6. The model updates the state.
7. The controller gets the new state form the model.
8. The controller invokes the view to render the new state.
9. The system is ready to receive user inputs.

This workflow is quite generic for any front-end application. It is summarized in Figure 7-2. The loop between the render and the user action is called the *render cycle*.

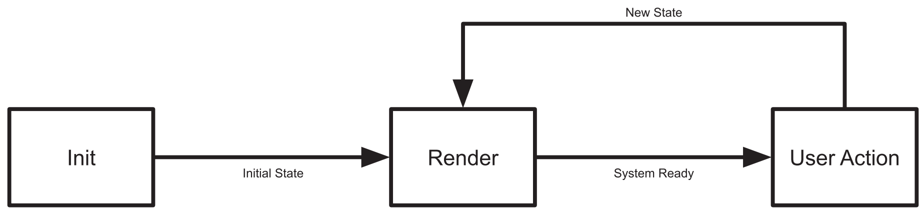


Figure 7-2. *Render cycle*

Observable Model

This first version of the state management code based on MVC works well for our use case. Nevertheless, the integration between the model and the controller is quite clumsy because we need to *manually* invoke the render method every time that the user performs some kind of action. This is not an optimal solution for two significant reasons. First, manually invoking the render after every state change is a very *error-prone* approach. Second, the render method is also invoked when the action *does not modify the* *state*; for example, adding an empty item to the list. Both of these issues are resolved in the next version of our model, which is based on the observer pattern (https://en.wikipedia.org/wiki/Observer_pattern).

Listing 7-5 is the code for the new version of the model. The differences with the previous version are highlighted for better readability. The complete code is available at https://github.com/Apress/frameworkless-front-end-development/blob/master/Chapter07/01/model/model.js.

Listing 7-5. Observable TodoMVC Model

```
const INITIAL_STATE = {
  todos: [],
  currentFilter: 'All'
}

export default (initalState = INITIAL_STATE) => {
  const state = cloneDeep(initalState)
  let listeners = []

  const addChangeListener = listener => {
    listeners.push(listener)

    listener(freeze(state))

    return () => {
      listeners = listeners.filter(
        l => l !== listener
      )
    }
  }
```

```
const invokeListeners = () => {
  const data = freeze(state)
  listeners.forEach(l => l(data))
}

const addItem = text => {
  if (!text) {
    return
  }

  state.todos.push({
    text,
    completed: false
  })

  invokeListeners()
}

const updateItem = (index, text) => {
  if (!text) {
    return
  }

  if (index < 0) {
    return
  }

  if (!state.todos[index]) {
    return
  }

  state.todos[index].text = text

  invokeListeners()
}

//Other methods...
```

```
  return {
    addItem,
    updateItem,
    deleteItem,
    toggleItemCompleted,
    completeAll,
    clearCompleted,
    changeFilter,
    addChangeListener
  }
}
```

To better understand the public API of the observable model, take a look at Listing 7-6, which shows a simple test suite for the new model.

Listing 7-6. Unit Tests for Observable Model

```
import modelFactory from './model.js'
let model

describe('observable model', () => {
  beforeEach(() => {
    model = modelFactory()
  })

  test('listeners should be invoked immediatly', () => {
    let counter = 0
    model.addChangeListener(data => {
      counter++
    })
    expect(counter).toBe(1)
  })
  test('listeners should be invoked when changing data', () => {
    let counter = 0
    model.addChangeListener(data => {
      counter++
    })
    model.addItem('dummy')
    expect(counter).toBe(2)
  })

  test('listeners should be removed when unsubscribing', () => {
    let counter = 0
    const unsubscribe = model
        .addChangeListener(data => {
        counter++
      })
    unsubscribe()
    model.addItem('dummy')
    expect(counter).toBe(1)
  })

  test('state should be immutable', () => {
    model.addChangeListener(data => {
      expect(() => {
        data.currentFilter = 'WRONG'
      }).toThrow()
    })
  })
})
```

After reading the tests, it's clear that the only way to get the state from the `Model` object is to add a listener callback. This callback will be invoked at the moment of subscription and every time that the internal state changes. This approach simplifies the controller, as you can see in Listing 7-7.

Listing 7-7. Using the Observable Model in Controller

```
import modelFactory from './model/model.js'

const model = modelFactory()

const {
  addChangeListener,
  ...events
} = model

const render = (state) => {
  window.requestAnimationFrame(() => {
    const main = document.querySelector('#root')

    const newMain = registry.renderRoot(
      main,
      state,
      events)

    applyDiff(document.body, main, newMain)
  })
}

addChangeListener(render)
```

The code for the controller is simpler now. To bind the `render` method to the model is enough reason to use that method as a listener. Notice that we extract all methods (apart from `addEventListener`) from the model to use them as events that we pass to the view.

The observable model is useful for adding new features to the controller without modifying the public interface of the model. In Listing 7-8, you see a new version of the controller that creates two new change listeners. The first one is a simple logger on the console. The second one saves the state to `window.localStorage`. This way, the controller can load the initial data from the storage when the application starts.

Listing 7-8. More Listeners Used with the Observable Model

```
import stateFactory from './model/state.js'

const loadState = () => {
  const serializedState = window
    .localStorage
    .getItem('state')

  if (!serializedState) {
    return
  }

  return JSON.parse(serializedState)
}

const state = stateFactory(loadState())

const {
  addChangeListener,
  ...events
} = state

const render = (state) => {
  // Render Code
}

addChangeListener(render)

addChangeListener(state => {
  Promise.resolve().then(() => {
    window
      .localStorage
      .setItem('state', JSON.stringify(state))
  })
})
```

```
addChangeListener(state => {
  console.log(
    `Current State (${(new Date()).getTime()})`,
    state
  )
})
```

Implementing the same features without the observable model would have been difficult and not maintainable. Keep this pattern in mind when your controller becomes *too coupled* with the model.

In this section, I said that "the model" was a single object. This is true for a simple application like TodoMVC, but in a real scenario, "the model" is a collection of `Model` objects that manage all the different domains in your application.

Reactive Programming

Reactive programming has been a buzz phrase in the front-end community for quite a while. It became popular when the Angular team announced that their framework was heavily based on RxJS (React Extensions for JavaScript), a library built to create applications based on reactive programming. In my opinion, the best source for understanding reactive programming is a GitHub Gist titled "The Introduction to Reactive Programming You've Been Missing," by André Staltz, one of the maintainers of RxJS. You can read it at `https://gist.github.com/staltz/868e7e9bc2a7b8c1f754`.

In a nutshell, to implement the reactive paradigm means to work in an application where everything is an observable that can *emit* events: model changes, HTTP requests, user actions, navigation, and so forth.

Tip If you're using a lot of observables in your code, you're working in a reactive paradigm.

Reactive programming is a very interesting topic, and in this chapter, we will just scratch the surface by creating a reactive state management library in a couple of different ways. If you'd like to study this topic in depth, I suggest reading *Front-End Reactive Architectures* by Luca Mezzalira (Apress, 2018).

A Reactive Model

The model created in Listing 7-5 is an example of reactive state management because it's an observable. But, in a non-trivial application, there should be a lot of different model objects, so we need an easy way to create observables. This way, we can focus on the domain logic and leave the architectural part in a separate library. In Listing 7-9, you see a new version of our model object based on an observable factory. Listing 7-10 shows the observable factory.

Listing 7-9. An Observable TodoMVC Model Built with a Factory

```
import observableFactory from './observable.js'

const INITIAL_STATE = {
  todos: [],
  currentFilter: 'All'
}

export default (initalState = INITIAL_STATE) => {
  const state = cloneDeep(initalState)

  const addItem = text => {
    if (!text) {
      return
    }
    state.todos.push({
      text,
      completed: false
    })
  }

  const updateItem = (index, text) => {
    if (!text) {
      return
    }

    if (index < 0) {
      return
    }

    if (!state.todos[index]) {
      return
    }

    state.todos[index].text = text
  }

  ...

  const model = {
    addItem,
    updateItem,
    deleteItem,
    toggleItemCompleted,
    completeAll,
    clearCompleted,
    changeFilter
  }

  return observableFactory(model, () => state)
}
```

Listing 7-10. Observable Factory

```
export default (model, stateGetter) => {
  let listeners = []

  const addChangeListener = cb => {
    listeners.push(cb)
    cb(freeze(stateGetter()))
    return () => {
      listeners = listeners
        .filter(element => element !== cb)
    }
  }

  const invokeListeners = () => {
    const data = freeze(stateGetter())
    listeners.forEach(l => l(data))
  }

  const wrapAction = originalAction => {
    return (...args) => {
      const value = originalAction(...args)
      invokeListeners()
      return value
    }
  }

  const baseProxy = {
    addChangeListener
  }
  return Object
    .keys(model)
    .filter(key => {
      return typeof model[key] === 'function'
    })
    .reduce((proxy, key) => {
      const action = model[key]
      return {
        ...proxy,
        [key]: wrapAction(action)
      }
    }, baseProxy)
}
```

The code of the observable factory may seem a little obscure, but its functioning is quite simple. It creates a *proxy* of the `Model` object, in which every method in the original model creates a new method with the same name that wraps the original one and invokes all the listeners. To pass the state to the proxy, a simple getter function is used to get the current state after every modification made by the model.

From an external point of view, the observable models in Listing 7-5 and Listing 7-9 have the same public interface. So a good way to design a reactive state management architecture is to create a simple observable model, and when you need more than one `Model` object, you can create the observable factory abstraction. This is another example of the YAGNI principle that I introduced in Chapter 3.

In Figure 7-3, you can see the relationship between the controller, the model, and the proxy.

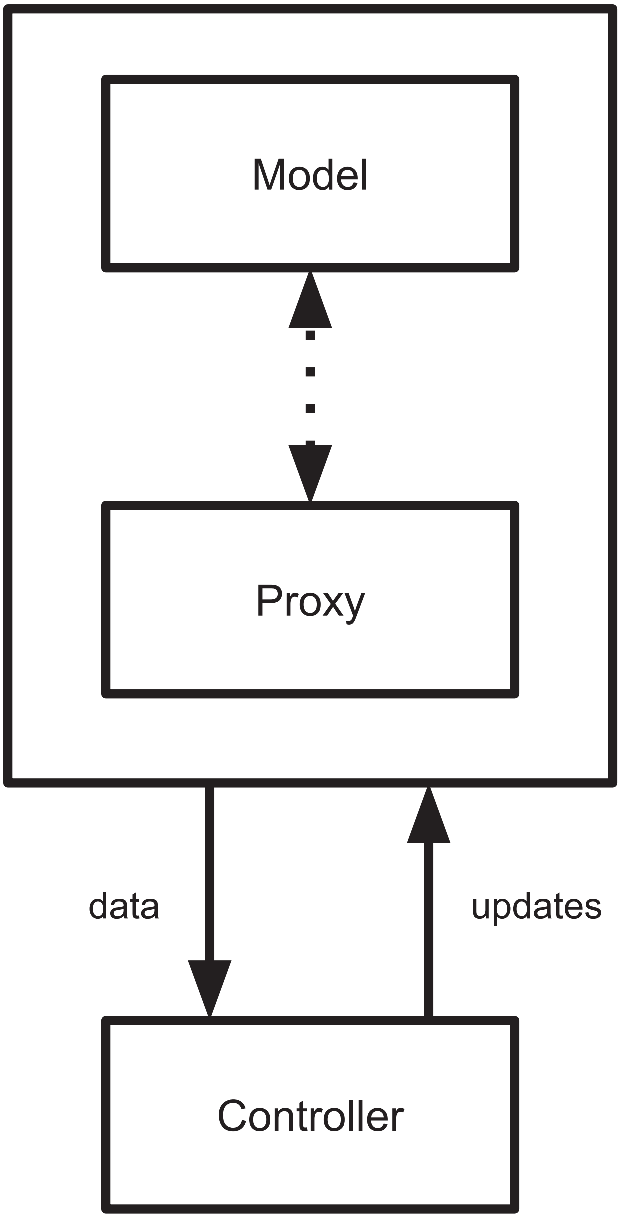


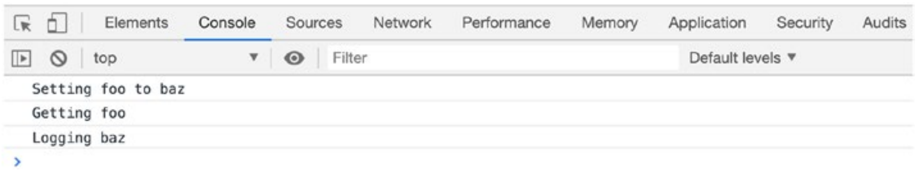
Figure 7-3. *Observable model with a proxy*

Native Proxies

JavaScript has a native way to create proxies via the Proxy object (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy). This new API makes it easy to wrap the default behavior of any object with custom code. Listing 7-11 creates a simple proxy that logs a message every time that we get or set a property of the base object. In Figure 7-4, you can see the result in the browser's console.

Listing 7-11. Basic Proxy Object Usage

```
const base = {
  foo: 'bar'
}

const handler = {
  get: (target, name) => {
    console.log(`Getting ${name}`)
    return target[name]
  },
  set: (target, name, value) => {
    console.log(`Setting ${name} to ${value}`)
    target[name] = value
    return true
  }
}

const proxy = new Proxy(base, handler)

proxy.foo = 'baz'
console.log(`Logging ${proxy.foo}`)
```



```
Setting foo to baz
Getting foo
Logging baz
```

Figure 7-4. *Basic proxy result*

To create a proxy that wraps a base object, you need to provide a handler that consists of a set of traps. A trap is a method that wraps a basic operation on the base object. In our simple case, we overwrote the setters and the getters of all the properties. Notice that the set handler should return a boolean value that represents the success of the operation. In Listing 7-12, the `Proxy` object creates an observable factory.

Listing 7-12. Observable Factory with Proxy Object.freeze

```
export default (initialState) => {
  let listeners = []

  const proxy = new Proxy(cloneDeep(initialState), {
    set: (target, name, value) => {
      target[name] = value
      listeners.forEach(l => l(freeze(proxy)))
      return true
    }
  })

  proxy.addChangeListener = cb => {
    listeners.push(cb)
    cb(freeze(proxy))
    return () => {
      listeners = listeners.filter(l => l !== cb)
    }
  }
  return proxy
}
```

Even if the signature is similar, the usage is slightly different, as in Listing 7-13, which shows the new version of the model created with this new observable factory.

Listing 7-13. An Observable TodoMVC Model Built with a Proxy Factory

```
export default (initialState = INITIAL_STATE) => {
  const state = observableFactory(initialState)

  const addItem = text => {
    if (!text) {
      return
    }

    state.todos = [...state.todos, {
      text,
      completed: false
    }]
  }

  const updateItem = (index, text) => {
    if (!text) {
      return
    }

    if (index < 0) {
      return
    }

    if (!state.todos[index]) {
      return
    }
    state.todos = state.todos.map((todo, i) => {
      if (i === index) {
        todo.text = text
      }
      return todo
    })
  }

  ...

  return {
    addChangeListener: state.addChangeListener,
    addItem,
    updateItem,
    deleteItem,
    toggleItemCompleted,
    completeAll,
    clearCompleted,
    changeFilter
  }
}
```

There's a very important difference between the two versions. In the one based on proxy, the `todos` array is *overwritten* every time. In the first one, the `todos` array is modified in place, which invokes the array's `push` method or substitutes an element. When using a `Proxy` object, it's mandatory to overwrite the properties to invoke the `set` trap.

Caution When working with a `Proxy` object, always replace properties instead of modifying them in place.

Event Bus

In this section, I cover how to manage the state of an application using the event bus pattern. An event bus is one way to implement an event-driven architecture (EDA). When working with EDAs, every state change is represented by an event that is dispatched in the system. To learn more about the various kinds of EDA and their differences, I suggest reading *Building Evolutionary Architectures: Support Constant Change* by Neal Ford, Rebecca Parsons, and Patrick Kua (O'Reilly Media, 2017).

An event is defined by a name that identifies what happened and a payload that contains meaningful information to process the event. Listing 7-14 is an example event that should be dispatched when creating a new item in our TodoMVC domain.

Listing 7-14. Add Item Event

```
const event = {
  type: 'ITEM_ADDED',
  payload: 'Buy Milk'
}
```

The main idea behind the event bus pattern is that every event is processed by a single object that connects all the "nodes" that make up the application. The event is then processed, and the result is sent to all the connected nodes. When using an event bus for state management, the result of any event processing is an updated version of the state of the application. Figure 7-5 is a diagram of the event bus pattern.

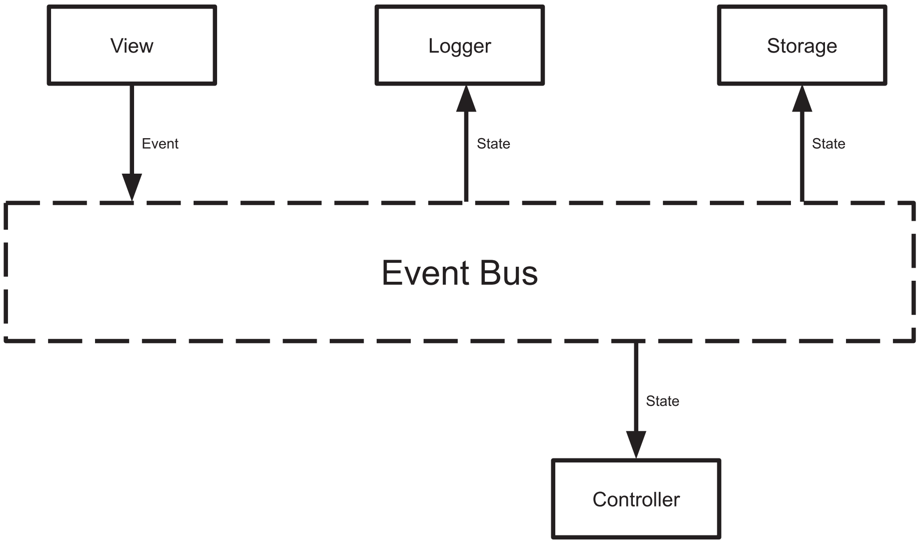


Figure 7-5. *Event bus pattern*

To better understand how an event bus works, let's analyze the flow of an `ITEM_ADDED` event.

1. The view renders the initial state.
2. The user fills out the form and presses Enter.
3. The DOM event is captured by the view.
4. The view creates the `ITEM_ADDED` event and dispatches it to the bus.
5. The bus processes the event generating a new state.
6. The new state is sent to the controller.
7. The controller invokes the view to render the new state.
8. The system is ready to receive user inputs.

Step 5 states that the bus processes the event generating a new state. This is not correct because the event bus is an architectural element and **should not contain any kind of domain-related code**. We need to add the model to the mix to implement the event bus pattern. In this scenario, the model is a function that accepts the old state and an event and returns a new version of the state, as shown in Figure 7-6.

It's important to notice that in this pattern, the state that travels from the model to the subscribers is a single object. This object contains all the data useful to our application. This does not mean that the model should be one big JavaScript function. Later, you see how it is possible to split this model into submodels that together build the `State` object.

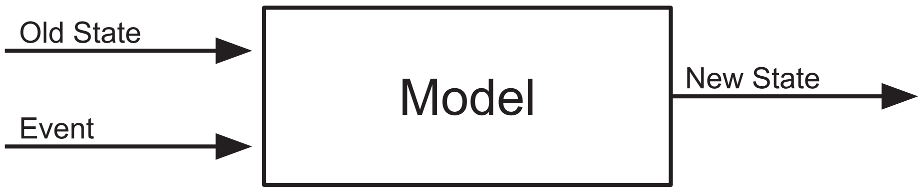


Figure 7-6. *Model structure in an event bus application*

Figure 7-7 is an updated diagram of the event bus pattern with the addition of the model.

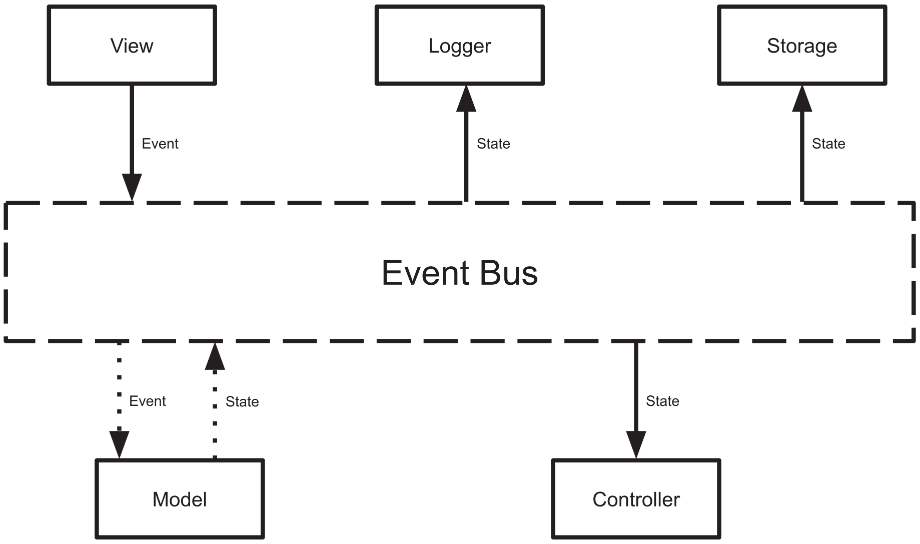


Figure 7-7. *Event bus pattern with model*

To complete this section, we will analyze two event bus implementations: the first is frameworkless and the second is based on Redux. Redux is a state management library born in the React ecosystem but usable in any kind of environment.

A Frameworkless Implementation

The first element that we are going to analyze is the event bus. Like the previous examples, not all the code is shown in this book. The complete code for this implementation is available at `https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter07/03`. The code for the event bus is shown in Listing 7-15.

Listing 7-15. Frameworkless Event Bus

```
export default (model) => {
  let listeners = []
  let state = model()
  const subscribe = listener => {
    listeners.push(listener)

    return () => {
      listeners = listeners
        .filter(l => l !== listener)
    }
  }

  const invokeSubscribers = () => {
    const data = freeze(state)
    listeners.forEach(l => l(data))
  }

  const dispatch = event => {
    const newState = model(state, event)

    if (!newState) {
      throw new Error('model should always return a value')
    }

    if (newState === state) {
      return
    }

    state = newState


    invokeSubscribers()
  }
  return {
    subscribe,
    dispatch,
    getState: () => freeze(state)
  }
}
```

In this scenario, the model is a function that gets the previous state and the event as inputs, and then returns a new state. There is another important characteristic of the model: a pure function. A pure function is where the return value is only determined by its input values—just like any standard mathematical function, such as `Math.cos(x)`.

A model designed as a pure function offers a big boost to testability because the new state cannot depend on an internal status of the model itself. We can also use this aspect to optimize performances, because every time that the state is updated, it has to be a new object. So if the old state and new state are equal, it means that we can skip the subscribers. In this implementation, invoking the model without parameters will result in obtaining the *initial* state of the application.

To better understand the inner workings of the event bus, Listing 7-16 shows the related test suite.

Listing 7-16. Test Suite for Event Bus

```
import eventBusFactory from './eventBus'
let eventBus

const counterModel = (state, event) => {
  if (!event) {
    return {
      counter: 0
    }
  }

  if (event.type !== 'COUNTER') {
    return state
  }

  return {
    counter: state.counter++
  }
}
```

```
describe('eventBus', () => {
  beforeEach(() => {
    eventBus = eventBusFactory(counterModel)
  })

  test('subscribers should be invoked when the model catch the
  event', () => {
    let counter = 0

    eventBus.subscribe(() => counter++)

    eventBus.dispatch({ type: 'COUNTER' })

    expect(counter).toBe(1)
  })

  test('subscribers should not be invoked when the model does
  not catch the event', () => {
    let counter = 0

    eventBus.subscribe(() => counter++)

    eventBus.dispatch({ type: 'NOT_COUNTER' })

    expect(counter).toBe(0)
  })

  test('subscribers should receive an immutable state', () => {
    eventBus.dispatch({ type: 'COUNTER' })
    eventBus.subscribe((state) => {
      expect(() => {
        state.counter = 0
      }).toThrow()
    })
  })
  test('should throw error if the model does not return a
  state', () => {
    const eventBus = eventBusFactory(() => {
      return undefined
    })

    expect(() => {
      eventBus.dispatch({ type: 'EVENT' })
    }).toThrow()
  })
})
```

The `counterModel` object gives us a glimpse of how a model should work in an event bus architecture. When an event of the `COUNTER` type is dispatched a new state is created with an incremented counter property. For all the other events, nothing is changed and the old state is returned. Listing 7-17 shows part of the model of the TodoMVC application.

Listing 7-17. TodoMVC Model for Event Bus Architecture

```
const INITIAL_STATE = {
  todos: [],
  currentFilter: 'All'
}

const addItem = (state, event) => {
  const text = event.payload
  if (!text) {
    return state
  }

  return {
    ...state,
    todos: [...state.todos, {
      text,
      completed: false
    }]
  }
}

const updateItem = (state, event) => {
  const { text, index } = event.payload
  if (!text) {
    return state
  }

  if (index < 0) {
    return state
  }

  if (!state.todos[index]) {
    return state
  }

  return {
    ...state,
    todos: state.todos.map((todo, i) => {
      if (i === index) {
        todo.text = text
      }
      return todo
    })
  }
}

const methods = {
  ITEM_ADDED: addItem,
  ITEM_UPDATED: updateItem
}
```

```
export default (initalState = INITIAL_STATE) => {
  return (prevState, event) => {
    if (!prevState) {
      return cloneDeep(initalState)
    }

    const currentMethod = methods[event.type]

    if (!currentMethod) {
      return prevState
    }

    return currentMethod(prevState, event)
  }
}
```

To avoid a very long switch statement to choose the right method based on the event type, I used a simple object that maps the event type with a method. If no method is found, it means that the model does not manage that event, and so the previous state is returned.

In the previous section, I stated that in a real application, the model function should be separated into smaller submodules. Another version of the model in Listing 7-17 is on GitHub at `https://github.com/Apress/frameworkless-front-end-development/blob/master/Chapter07/03.1/model/model.js`. In that version of the model, there are two submodels. The first one manages the todos and the second one manages the filter. The main model function merges the results of the submodels into a single `State` object.

Tip When working with an event bus, split the model into submodels to achieve good readability for your code.

Listing 7-18 shows the controller for the TodoMVC application based on the event bus.

Listing 7-18. Controller for an Event Bus–based TodoMVC Application

```
import eventBusFactory from './model/eventBus.js'
import modelFactory from './model/model.js'

const model = modelFactory()
const eventBus = eventBusFactory(model)

const render = (state) => {
  window.requestAnimationFrame(() => {
    const main = document.querySelector('#root')

    const newMain = registry.renderRoot(
      main,
      state,
      eventBus.dispatch)

    applyDiff(document.body, main, newMain)
  })
}

eventBus.subscribe(render)

render(eventBus.getState())
```

As you can see, the major difference between this version and previous versions is that we don't provide the events to the render function; instead, we use the `dispatch` method of the event bus. This way, the view is able to dispatch events in the system, as you can see in Listing 7-19, which shows part of the view's code.

Listing 7-19. View Function Using Event Bus

```
import eventCreators from '../model/eventCreators.js'

let template

const getTemplate = () => {
  if (!template) {
    template = document.getElementById('todo-app')
  }

  return template
    .content
    .firstElementChild
    .cloneNode(true)
}

const addEvents = (targetElement, dispatch) => {
  targetElement
    .querySelector('.new-todo')
    .addEventListener('keypress', e => {
      if (e.key === 'Enter') {
        const event = eventCreators
          .addItem(e.target.value)
        dispatch(event)
        e.target.value = ''
      }
    })
}

export default (targetElement, state, dispatch) => {
  const newApp = targetElement.cloneNode(true)
  newApp.innerHTML = ''
  newApp.appendChild(getTemplate())

  addEvents(newApp, dispatch)

  return newApp
}
```

Note the usage of `eventCreators.addItem` to create the `Event` object to dispatch. The `eventCreators` object is a simple collection of factories used to easily build consistent events. The code is shown in Listing 7-20.

Listing 7-20. Event Creators

```
const EVENT_TYPES = Object.freeze({
  ITEM_ADDED: 'ITEM_ADDED',
  ITEM_UPDATED: 'ITEM_UPDATED'
})

export default {
  addItem: text => ({
    type: EVENT_TYPES.ITEM_ADDED,
    payload: text
  }),
  updateItem: (index, text) => ({
    type: EVENT_TYPES.ITEM_UPDATED,
    payload: {
      text,
      index
    }
  })
}
```

These functions help ensure that every event is in the canonical form seen in Listing 7-14.

Redux

Redux is a state management library that was first announced at the React-Europe conference in 2015 during a talk by Dan Abramov (www.youtube.com/watch?v=xsSnOQynTHs). After that, it rapidly became a mainstream approach to working with React applications. Redux is one (and surely the most successful) of the so-called Flux-like libraries, a group of tools that implemented Facebook's Flux architecture. To learn more about Flux, consult its website at https://facebook.github.io/flux/.

Working with Redux is very similar to working with a frameworkless event bus. But, since it was created after the Flux pattern, the words used to define the components of the architecture are different, as you can see in Table 7-1.

Table 7-1. *Comparison of Event Bus and Redux Elements*

Event Bus	Redux
Event Bus	Store
Event	Action
Model	Reducer

To better understand the principles behind Redux, I strongly suggest reading the "Three Principles" chapter in the Redux documentation at https://redux.js.org/introduction/three-principles.

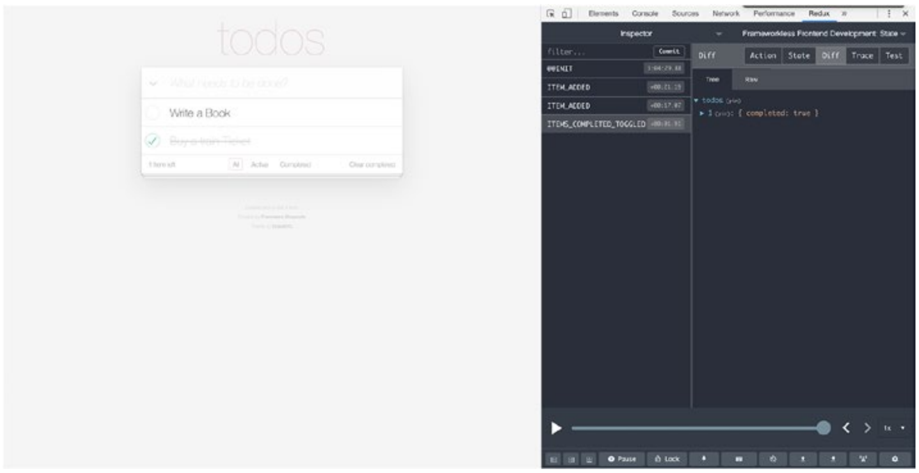
Apart from the naming, the elements are very similar. In fact, Listing 7-21 is the code for the controller of a TodoMVC application built with Redux.

Listing 7-21. Controller of a Redux-based TodoMVC Application

```
import reducer from './model/reducer.js'

const INITIAL_STATE = {
  todos: [],
  currentFilter: 'All'
}

const {
  createStore
} = Redux

const store = createStore(
  reducer,
  INITIAL_STATE
)

const render = () => {
  window.requestAnimationFrame(() => {
    const main = document.querySelector('#root')

    const newMain = registry.renderRoot(
      main,
      store.getState(),
      store.dispatch)

    applyDiff(document.body, main, newMain)
  })
}

store.subscribe(render)

render()
```

Using Redux's store instead of the event bus build makes almost no difference for the controller. As you can see in the complete application code at `https://github.com/Apress/frameworkless-front-end-development/tree/master/Chapter07/04`, the reducer has exactly the same code of the model from the frameworkless event bus.

One of the main advantages of using Redux instead of a frameworkless event bus is the large number of tools and plugins that are readily available. One of the most popular tools is Redux DevTools. With this tool, developers can easily log all the actions that are dispatched in the system to see how they affect the state. Moreover, it is possible to import or export the state in JSON format. Figure 7-8 shows the Redux DevTools in action.



```
@@INIT
ITEM_ADDED
ITEM_ADDED
ITEMS_COMPLETED_TOGGLED
```

```
▼ todos
  ▶ 1 : { completed: true }
```

Figure 7-8. *Redux DevTools*

Comparing State Management Strategies

In this last section, I point out the characteristics of the three kinds of state management strategies from three different points of views: simplicity, consistency, and scalability.

Model-View-Controller

Model-view-controller is fairly simple to implement and offers developers a lot of advantages. For example, there is a good deal of separation of concerns and testability of your domain's business logic.

The real problem with MVC is that it is not a strict pattern. The definition of the elements and the relations between them could be unclear. If you ask, *What is the difference between the view and the controller?,* you may get a lot of different answers. This happens because every MVC framework fills in the "gray areas" of the MVC pattern with their own ideas, so every framework implements a slightly different version of the MVC. To effectively work with a frameworkless MVC, the first task is to define your team's MVC rules.

The same characteristic is also a scalability problem. As your application grows bigger, the number of "gray areas" may increase, and if consistency is not addressed, your code may become unreadable.

Reactive Programming

The main idea behind reactive programming is that everything in your application is an observable. We covered how to easily build observable models, but there are libraries (like RxJS) that transform every aspect of a front-end application in an observable, from user inputs to timers and HTTP requests. This approach guarantees good consistency because you work with objects of the "same type."

Nevertheless, wrapping everything in an observable is not simple. It may become easy if you use a third-party library like RxJS, but that does not mean that it would be *simple*.

Caution Implementing an easy architecture is not the same as building a simple one. Your goal should be to create the simplest architecture that matches your requirements, rather than the easiest one to build.

It may not be that simple because you're working with a very *big abstraction*, but **everything is an observable**. Working with abstractions may become a problem as your application becomes bigger because it starts to "leak" (`https://en.wikipedia.org/wiki/Leaky_abstraction`). *Leakiness* is not a specific problem of reactive programming, but it is related to any pattern (or framework) based on a central abstraction. This is greatly explained by the Law of Leaky Abstractions, coined by Joel Spolsky, which states, "All non-trivial abstractions, to some degree, are leaky."

As your application grows, there will be parts that are not suited for that abstraction, which may become a big problem for scalability.

Event Bus

An event bus (and event-driven architectures in general) is based on a single strict rule: Every state change is generated by an event. This rule helps keep the complexity of your application proportional to its size, while in other architectures, the complexity is *exponential* to the size of the application. That is one reason why a big application's code is usually less readable than a small application's code.

With the increase in the number of elements that make up your application, there more possibilities related to how to let them communicate, as shown in Figure 7-9.

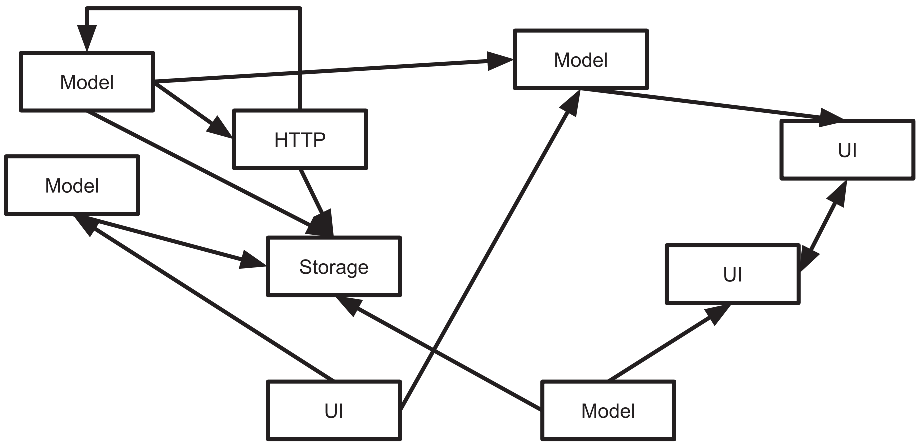


Figure 7-9. *Complexity in a big application*

Strictly following the event bus pattern will remove this complexity, because the only way to communicate is through the bus itself (compare Figure 7-9 with Figure 7-7). This feature makes the event bus a very good approach if your first concern is the scalability of your codebase.

As you saw with frameworkless implementations of the event bus, it is easy to use and build. It's also simple because the abstraction behind the pattern is not as strong as the one in reactive programming. The main problem with event bus is its verbosity. For every state update, the team needs to create the event, dispatch it through the bus, write the model that updated the state, and send the new state to the listeners. Because of the verbosity of this pattern, not all states of the application are managed with it. In the long run, developers tend to pair it with another state management strategy (MVC or reactive) to manage smaller or simpler domains, which results in a loss of consistency.

Table 7-2 is a summary of the comparisons made in this section.

Table 7-2. *State Management Strategies Comparison*

	Simplicity	Consistency	Scalability
MVC	✓	✗	✗
Reactive	✗	✓	—
Event bus	—	✗	✓

As you may have noticed, none of these characteristics is really measurable; they are just my personal thoughts based on my studies and experiences. Using the different patterns covered in this chapter may lead you to completely different conclusions.

Summary

In this chapter, you learned about state management and why it is important in creating any kind of client application. You also analyzed and implemented three state management strategies: model-view-controller, reactive programming, and event bus.

In the last chapter, I introduce some decision-making techniques to help you choose the right tool for the right job.

CHAPTER 8

The Right Tool for the Right Job

Programming is a social activity.

—Robert C. Martin

In Chapter 7, you learned about the last piece of the "frameworkless toolkit." You now know how to render DOM elements, manage user input, make HTTP requests, implement a client-side routing system, and manage the state of your application. You're now ready to create a complete frameworkless front-end application from scratch.

This last chapter will help you answer the question "Now that I'm able to work effectively without frameworks, *when* should I do that?" or more generally, "Which framework should I use for this product, or should I use no framework at all?" In a nutshell, I will talk about how to choose "the right tool for the right job." I will do that by defining principles that should be kept in mind when making a technical decision and by listing a collection of practical tools based on these principles.

F. Strazzullo, *Frameworkless Front-End Development*,
https://doi.org/10.1007/978-1-4842-4967-3_8

JavaScript Fatigue

If you are a front-end developer, you have probably heard the expression "JavaScript fatigue," which was coined around 2016 to express the sense of frustration that generates from the inability to keep up with the latest libraries or frameworks. JavaScript fatigue can be very hard for newcomers to manage; they may feel overwhelmed by all the possibilities that the community has to offer.

There are several reasons behind the constant change in the JavaScript ecosystem. The most important is that JavaScript now runs almost everywhere. Apart from the browser, its natural environment, JavaScript runs on servers (thanks to Node) and in a lot of other environments, such as mobile applications, blockchain, Internet of Things (IoT), and so forth. Jeff Atwood (Atwood's law) stated the following:

> *Any application that can be written in JavaScript, will eventually be written in JavaScript.*

For now, it seems that the rule is still valid. Table 8-1 is a noncomprehensive list of areas (excluded the front-end) where JavaScript is used. For each area, I included an example tool with a relative link to the project's homepage.

***Table 8-1.** JavaScript Ecosystem Cheat Sheet*

	Tool	Link
Back-end	Node.JS	https://nodejs.org
Ethereum Blockchain	Truffle Suite	https://truffleframework.com
Mobile Applications	React Native	https://facebook.github.io/react-native/
IoT	Johnny-Five	http://johnny-five.io
NES Programming	Nesly	https://github.com/emkay/nesly
Machine Learning	TensorFlow	www.tensorflow.org
Alexa Skill	ASK (Alexa Skill Kit)	https://github.com/alexa/alexa-skills-kit-sdk-for-nodejs

If we limit our reasoning about "fatigue" to only the front end, there are many options out there. In addition to the three mainstream frameworks (Angular, React, and Vue), there are many small libraries that solve specific problems. I talked about some of them in previous chapters, including axios for HTTP requests, Redux for state management, and Navigo for routing, but they are just the tip of the iceberg.

Personally, I don't like the expression "JavaScript fatigue." I am really happy to have a lot of choices in my ecosystem. This book would never have been published if I didn't have the opportunity to study React, Angular, and so on. Frameworks and libraries are great for learning. So, the more frameworks that we have, the faster we can learn new paradigms. I love to call this period the "JavaScript renaissance," because it's a great time to be a JavaScript developer.

The “Right” Framework

Why did I start this chapter with a section about the JavaScript renaissance? Because the great opportunities that this ecosystem offers to developers also brings a challenge: choosing the right framework. I hope that this chapter can help you and your team with this task. Remember that every time I talk about choosing a framework, I always consider the frameworkless approach.

Tip When choosing a framework, always consider a frameworkless option. You may notice that frameworks are not giving you any advantage in a particular scenario.

When I said, “choose,” I didn’t mean selecting a framework from a list, but analyzing and applying decision-making techniques in a structured way. Because of the magnitude of decision-making, in this chapter, I will introduce some basic principles that should drive you to choose a framework.

If you want to study decision-making in depth, these are some of the books that I suggest that you read:

- *Decision Making For Dummies* by Dawna Jones (For Dummies, 2014)
- *The Thinker’s Toolkit: 14 Powerful Techniques for Problem Solving* by Morgan D. Jones (Crown Business, 1998)
- *Thinking, Fast and Slow* by Daniel Kahneman (Farrar, Straus and Giroux, 2011)

What does the “right” framework mean? One definition that you may find in the dictionary is “true or correct as a fact.”

But can a framework be correct as a "fact"? I don't think so. There are probably more than one "right" framework for your project. So I will change the challenge from "choose the right framework" to "choose a good enough framework." By "good enough," I mean that it is helping your team achieve its goals.

Tip If a framework seems "good enough," you should stop searching. Trying to find the perfect match may cost you a lot of time.

So, throughout the rest of this chapter, I will talk about how to choose a "good enough" framework, rather than talk about the differences between React, Angular, and so on. I do that because I firmly agree with one of the main points of the Agile Manifesto (`https://agilemanifesto.org`).

Individuals and interactions over processes and tools.

In other words, I want to focus on the team that makes that decision and how they interact with each other. Therefore, our challenge becomes choosing a good enough framework in the right way.

Anti-patterns

In my work as a consultant, I often help teams during the bootstrap stage of a new project. One of my tasks is to help them choose the technology stack. The following sections are a collection of anti-patterns that I observed when teams chose tools and frameworks.

Fear of Obsolescence

One of the direct consequences of the JavaScript renaissance is that many teams are terrified that their stacks will become old right after they start working with it. Driven by this fear, the only parameter that they use to pick the next framework is its popularity: the more people that use it, the better. The problem here is that popularity is unrelated to the lifespan of a framework. Remember that AngularJS was very popular and almost a de facto standard; nonetheless, it "died" very quickly when Angular came out.

The truth is that if your project is successful, it will probably "last" longer than the popularity of any framework. When should you start worrying about the "age" of your framework? In my opinion, software (and frameworks) become "legacy" when they become a roadblock for a new business need that emerges during the lifespan of the project. For example, you need to hire new people but you can't find developers that want to work with your very old stack.

In these scenarios, a good refactoring approach is called the StrangleApplication (or StranglerFigApplication) pattern (`https://martinfowler.com/bliki/StranglerFigApplication.html`). In this pattern created by Martin Fowler, you attach a new application to an existing one. You let the new one grow slowly over the years until the old one is strangled. I talked about how to use this pattern in a front-end application at jsDay 2018 in Verona (`https://youtu.be/cTSoFvAUUF8`).

In the use case that I described in the talk, my team and I strangled an AngularJS application and replaced it with a frameworkless one. Figure 8-1 is a diagram of this refactoring process.

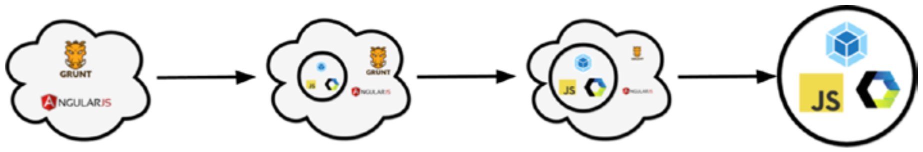


Figure 8-1. *From AngularJS to frameworkless with StranglerApplication*

Following the Hype

While some people are scared of the constant changes in the JavaScript ecosystem, other people are thrilled by it. Usually, these people tend to always choose the new "big thing" that becomes available. The problem with early adopters is that they are pioneers; they will be the first ones to explore the boundaries of new technologies and they cannot rely on the community to solve the problems that they face.

Hype can also be exploited. If you need to make some noise around your product, using terms like "VR," "blockchain," and "machine learning" can help your marketing team. But, you need to be aware of Gartner's Hype Cycle (`www.gartner.com/en/research/methodologies/gartner-hype-cycle`). The Hype Cycle (see Figure 8-2) visually represents the market's adoption of a technology.

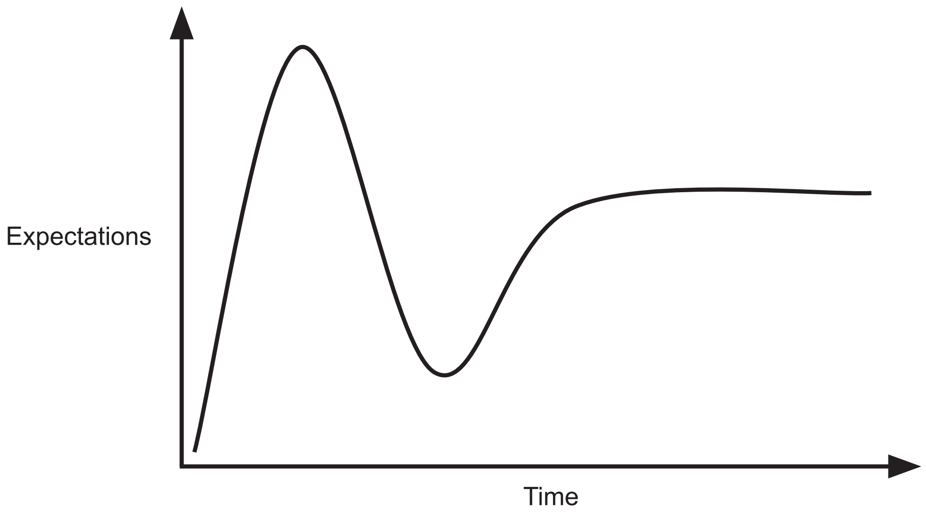


Figure 8-2. *Gartner's Hype Cycle*

To know the position of your stack on the Hype Cycle chart is crucial if you want to exploit the hype for marketing purposes.

Tip Use Gartner's Hype Cycle to make mindful decisions about hype.

The Usual Path

In some teams, there is no discussion about frameworks or tools; they just use the same technologies for every project. If they had a successful project with jQuery ten years ago, they will use jQuery for every new project. This usually happens in companies with a bad governance model, where deadlines come from above without any kind of negotiation with the development team. In these companies, developers are scared of failure, so they simply use the tools that they are more comfortable with. This kind of decision-making approach is one of the visible results of Conway's law, (https://en.wikipedia.org/wiki/Conway%27s_law), which states the following:

> *Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.*

So, if you have a bad governance model in your company, your technical decision will be affected by that model.

Caution When you have a suboptimal architecture in your application, try to discover if you have any kind of communication issues. You may need to solve them to fix your architecture.

The Expert

Another anti-pattern derived from the governance model of a company is to rely on “the expert.” In some companies, all the technical decisions are made by some kind of expert, who is may have an elevated title like “software architect” or “specialist.”

This is a very dangerous approach because this person might not have all the information needed to make a mindful decision. To choose a framework is a very important decision, and it should be as collaborative as possible.

Tip If you’re an architect, collaborate with your engineers when making framework-related decisions. If you’re an engineer, ask for collaboration from your architect.

Notice that the dangerousness of this approach is completely unrelated to the skills of the expert. It’s not a problem of skill, but a lack of information problem.

Rage-Driven Decisions

Some teams don’t know which framework they should use, but they know which framework they want to avoid: the last one that they used. Sometimes this happens when a team fails a project; they blame their problems on the unsuitable framework. When I decided to investigate the reasons behind the team’s frustration, failure was often related to their skills with using the framework or from a bad governance model.

Tip When a project fails, have a post-mortem meeting to understand the reasons. Your framework is usually innocent.

The Frameworkless Manifesto

As I explained in the front matter, this book is related to the Frameworkless Movement: a group of developers interested in developing without frameworks and in making mindful technical decisions. In the movement's manifesto (`https://github.com/frameworkless-movement/manifesto`), you find the principles that drive the people who believe in the Frameworkless Movement when they have to make a technical decision.

In this section, you analyze these principles and learn how they can be useful in your day-to-day job.

The First Principle

The first principle states the following.

> *The value of a software is not the code itself but in the reasons behind the existence of that code.*

In other words, to make mindful decisions about software (like choosing a framework), you should clarify the reasons behind why the software is being developed. Consult the Business Model Canvas (BMC) of your project. A BMC visually represents how the company wants to make money from your software. You can download an empty canvas at `www.strategyzer.com/canvas/business-model-canvas`. If you want more information, read *Business Model Generation* by Alexander Osterwalder and Yves Pigneur (John Wiley and Sons, 2010).

This canvas is composed of nine informative "blocks."

- **Customer segments**. The customers that the company is trying to serve.
- **Value proposition**. The products (or services) that your company offers to meet the needs of its customers.

- **Key activities**. The most important activities needed to develop the value proposition.
- **Key resources**. The resources necessary for developing the value proposition.

As you may imagine, your technical decisions should be influenced by the information that you get from a BMC.

Tip If your company does not have a BMC for your project, call your managers and try to create one. It contains a lot of useful information.

The Second Principle

The second principle states the following.

> *Every decision should be made considering the context. A good choice in a given context could be a bad choice in another one.*

This principle may seem quite obvious at first, but the main problem is to find a way to define the "context" of a software application. A way that I find really effective is to use a list of non-functional requirements (NFR). We all know what a functional requirement is—a way to define what software should do. Usually, they come in the form of user stories.

> *As an* ***anonymous user,*** *I want to* ***log in****, so that* ***I can access the premium area****.*

NFRs are a way to define how a software application should be, instead of what it should do. Take a look at the following second version of the user story.

> *As an anonymous user, I want to log in so that I can access the premium area* ***in less than one second****.*

As you can see, this new version of the user story is helpful in understanding if we are doing a good job developing the login feature of our software. In this case, our software should be performant enough to let the users log in in less than one second. Table 8-2 shows a non-comprehensive list of NFRs. For a more complete list, consult Wikipedia's entry about non-functional requirements at `https://en.wikipedia.org/wiki/Non-functional_requirement`.

***Table 8-2.** Partial List of NFRs*

Accessibility	Maintainability	Extensibility
Performances	Wow-Effect	Portability
Evolvability	Customizability	Testability
Deployability	Credibility	Reusability

The NFRs are a crucial aspect to consider when making any kind of technical decision. Two software applications that have the same functional requirements but different NFRs probably need different technologies. Alas, NFRs are usually completely ignored when describing software.

Caution You can't rely only on functional requirements to make mindful technical decisions. Keep NFRs in mind.

The Third Principle

The third principle states the following.

> *The mindful choice of a framework is a technical one and should be made by technical people, taking business needs into account.*

This is a very important point. Choosing a framework is a technical decision and a responsibility of the technical team. But the only way to make a mindful decision is to also take business needs into account. For example, if you work for a start-up, it is crucial to get feedback from the customers to lessen the time to market (TTM). So you need to reach a compromise between quality and the velocity needed to lessen TTM.

The Fourth Principle

The fourth principle states the following.

> *The decision-making criteria that led to the choice of a framework should be known to all the members in the team.*

This last principle is not directly related to "how" to make technical decisions. Nevertheless, it's a very important one. All the members (not just the developers) of your team should know the criteria that led to a particular decision. This is very important because it's really hard to judge the result of a decision without knowing the original context. When somebody enters a brownfield project, they usually have a lot of questions about the architecture and tools chosen to work with. Without knowing the criteria that brought the team to that decision, they are blind. They can blindly accept the decisions without questioning them, or they can blindly change them. Both of these scenarios are far from ideal. Developers should not make any kind of decisions blindly.

A very helpful tool that tries to address these problems is the Lightweight Architecture Decision Records (LADR), which is a way to keep track of all the meaningful decisions that are made during the lifespan of a project. For every architectural decision that the team make, an Architectural Decision Record (ADR) is created. This ADR is a numbered markdown file that should be kept in the project repository. An example of ADR is on GitHub at `https://github.com/Apress/frameworkless-front-end-development/blob/master/Chapter08/ADR-001.MD`. Every ADR should contain the following:

- Title
- Context (discussing, accepted, deprecated, superseded)
- Decision
- Status
- Consequences

An ADR should not be deleted if it is no longer valid. In that case, a new ADR is created to reflect the new decision, and the status of the old one is changed to superseded. When new members of the team enter a project, they should read all the ADRs in the repository.

Tools

In this section, I cover a very small collection of technical decision-making tools that you can use every time that you need to choose whether to work with a framework.

Matteo Vaccari's Tool

A tool (`http://matteo.vaccari.name/blog/archives/1022`) created by my friend Matteo Vaccari is very useful for classifying the list of libraries/frameworks that you're evaluating for your project. Take every library and place it on a two-axis graph, like the one shown in Figure 8-3.

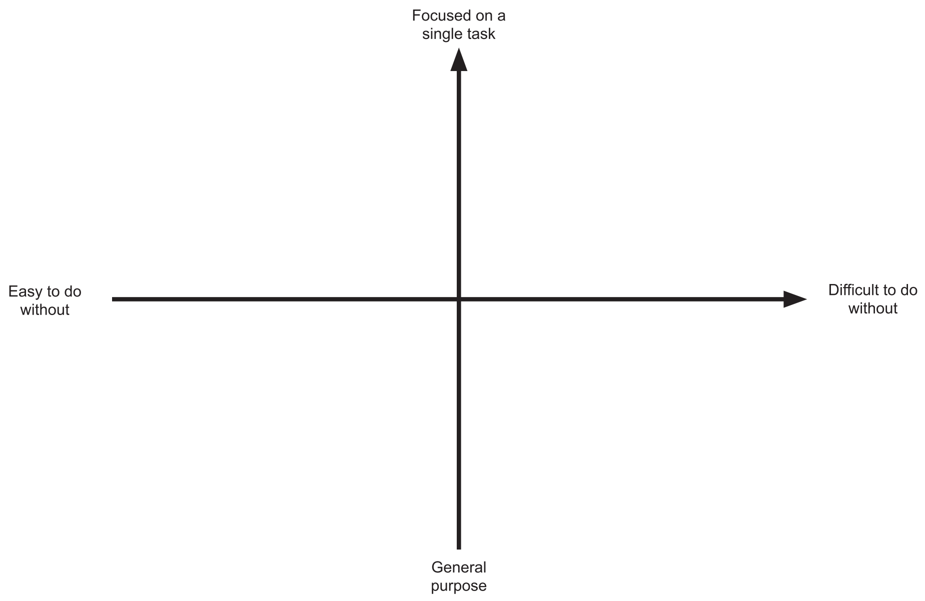


Figure 8-3. *Matteo Vaccari's tool*

After you place all the elements on the graph, you can use this tool to develop a strategy.

- **Upper-left quadrant**. These elements are good candidates for a frameworkless approach if you have the time to build the same features from scratch.
- **Upper-right quadrant**. These elements should be included in your codebase. Nevertheless, remember to write an interface around them.
- **Lower-right quadrant**. You may decide to add these elements to the codebase or to move them into the lower-left quadrant.
- **Lower-left quadrant**. You should avoid putting these elements in the codebase. If something is general purpose, it is usually hard to remove later.

Of course, this is not a strict rule. There might be exceptions that you need to discuss with your team.

Trade-off Sliders

This tool helps your team visualize the context of your software, which is a very important element in making mindful decisions. When working with this tool, the first task is to choose four or five metrics that you want to compare. Most of the time, I use quality, scope, budget, and deadline, but you can choose other metrics if you think that they are useful in your project. Next, order the metrics by decreasing "negotiability." **To protect the metrics that you put at the top of the list, you may need to sacrifice the other ones**.

Start with a silent voting phase, where each person writes a personal list, and then start a discussion to reach a consensus on the final list. You should obtain something similar to Figure 8-4.

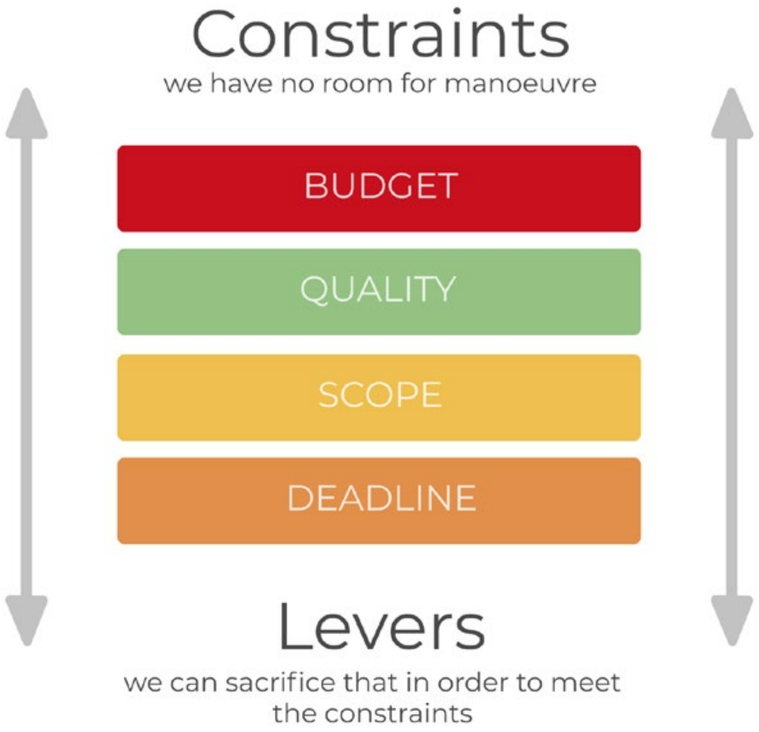

Constraints
we have no room for manoeuvre
BUDGET
QUALITY
SCOPE
DEADLINE
Levers
we can sacrifice that in order to meet the constraints

Figure 8-4. *Trade-off slider*

When using this tool, you should also involve your managers. You need their points of view on the metrics and they need to understand that in order to achieve something, you need to sacrifice something else. The name "trade-off sliders" is not accidental; every decision is usually the result of a trade-off of different aspects.

This simple "game" gives the team a lot of useful information about frameworks. If your first concern is the deadline, you probably have to choose the framework that your team knows best. This version of trade-off sliders is slightly different from the standard one. Read about the original one on Atlassian's website at `www.atlassian.com/team-playbook/plays/trade-off-sliders`.

Tip Every project has its trade-offs. Use this tool to visualize them and to help all the members of your team act accordingly.

Framework Compass Chart

I created this tool to specifically help teams to choose frameworks. It helps you visualize the most important NFRs for your project and the relationships between them. This tool is meant to bring developers and managers together in a meeting, just like trade-off sliders. The first step is to choose the five most important NFRs and place them on a radar chart, as shown in Figure 8-5.

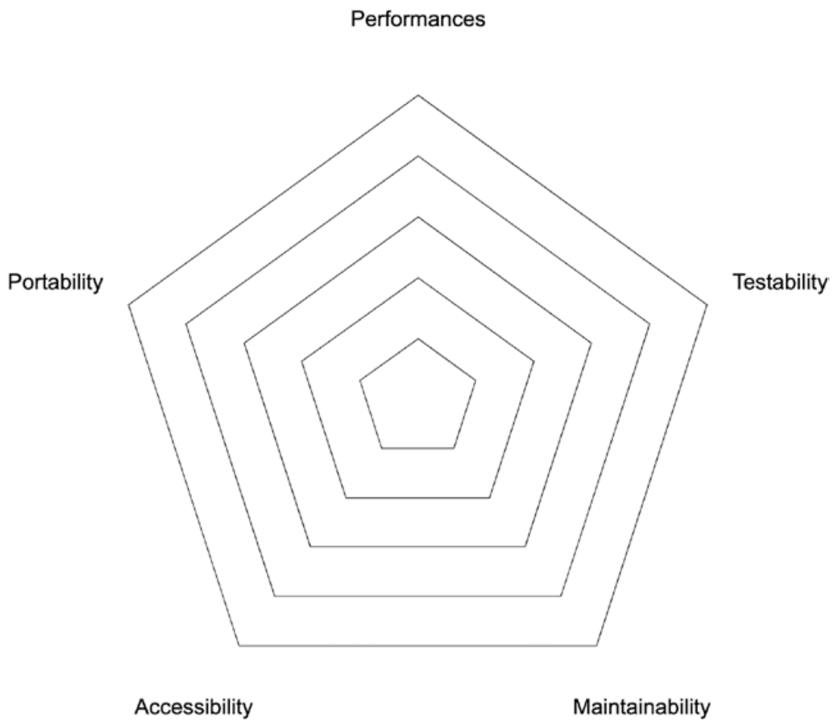


Figure 8-5. *Empty framework compass chart*

There are different ways to choose the NFRs to put on the chart. Table 8-3 lists some of the tools and includes links to instructions on how to use them.

Table 8-3. *Tools to Choose NFRs*

Tool	Link
Agile Retrospective	www.atlassian.com/team-playbook/plays/retrospective
SWOT Analysis	www.mindtools.com/pages/article/newTMC_05.htm
Impact Mapping	www.impactmapping.org
Lego Serious Play	www.lego.com/en-us/seriousplay

Next, you vote on the importance of each NFR on the chart (voting from 1 to 5), and reach a consensus among team members. You may use a technique similar to planning poker (https://en.wikipedia.org/wiki/Planning_poker). Each person calls the votes simultaneously, and then people with high and low votes justify their votes. Then repeat this procedure until you reach a consensus. The result of these votes should be placed on the chart, as shown in Figure 8-6.

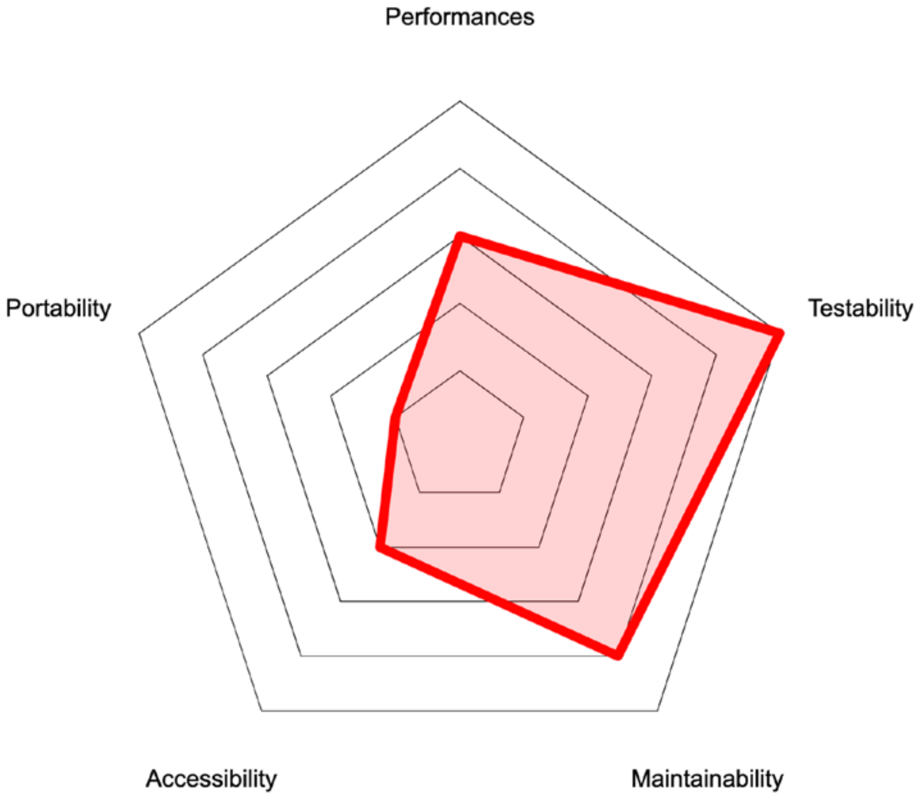


Figure 8-6. *Filled framework compass chart*

The technical team can now use this chart as a "compass" to choose a framework. For each framework that they want to evaluate, they can create a new chart and see how it fits on the compass, as shown in Figure 8-7.

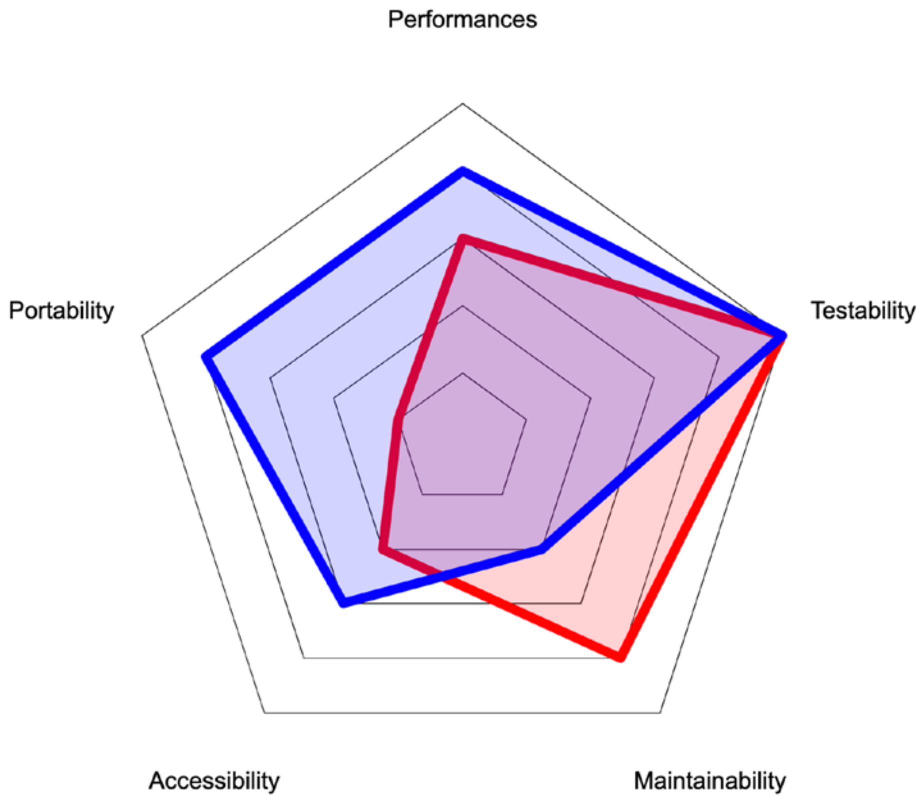


Figure 8-7. *Framework compass chart with a fitness check*

The most important advantage of this tool is to drive the discussion of the tech team toward topics that are useful for the project. In many of the teams that I helped, everyone talked about performances. When I spoke with the managers, they said that performances are not that important to their customer segments. This tool helps you avoid anti-patterns like that one.

Other Tools

There are many other tools that you can use when choosing a framework or making any other technical decision. These tools should gather information from the following four areas.

- Identity (Who are we?)
- Market (Who are the users?)
- Value (What should the software do?)
- Context (How should the software be?)

Figure 8-8 shows the relationship between these areas and decisions.

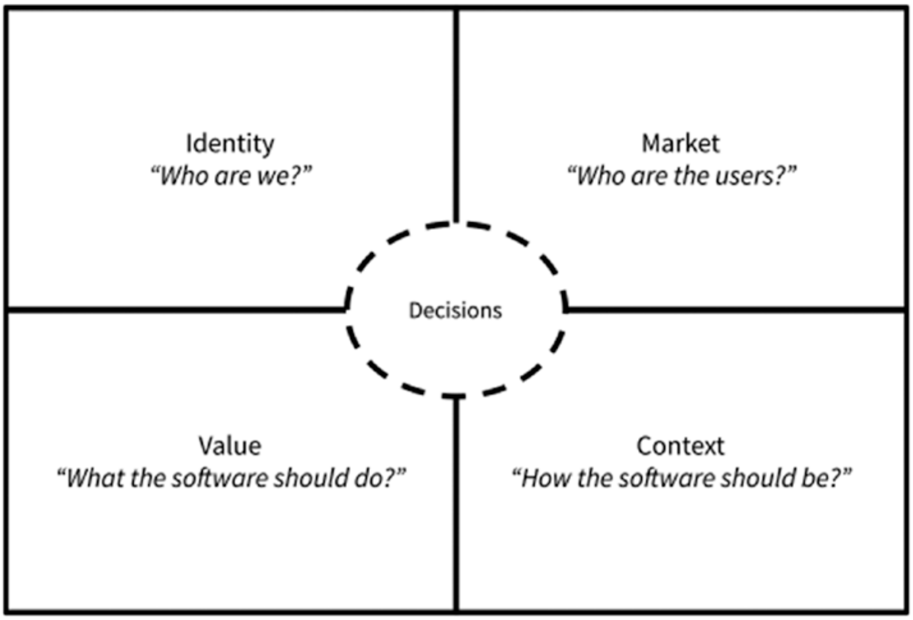


Figure 8-8. *Technical decision-making landscape*

Table 8-4 is a list of tools with the relative information area. Some of them were covered in the previous sections of this chapter.

***Table 8-4.** Decision-Making Tools*

Area	Tool	Link
Identity	Elevator Pitch	https://www.atlassian.com/team-playbook/plays/elevator-pitch
	5 Whys Analysis	https://www.atlassian.com/team-playbook/plays/5-whys
	Delegation Board	https://management30.com/practice/delegation-poker/
	Stakeholder Map	www.lucidchart.com/blog/how-to-do-a-stakeholder-analysis
Market	Business Model Canvas	www.strategyzer.com/canvas/business-model-canvas
	Customer Interview	www.atlassian.com/team-playbook/plays/customer-interview
	Customer Journey Mapping	www.atlassian.com/team-playbook/plays/customer-journey-mapping
	Value Proposition Canvas	www.strategyzer.com/canvas/value-proposition-canvas
Value	EventStorming	www.eventstorming.com
	Impact Mapping	www.impactmapping.org
	Lean Value Tree	https://blog.avanscoperta.it/it/2018/08/17/product-discovery-orchestrating-experiments-at-scale/
	User Story Mapping	www.jpattonassociates.com/user-story-mapping/

(*continued*)

Table 8-4. *(continued)*

Area	Tool	Link
Context	Trade-off Sliders	www.atlassian.com/team-playbook/plays/trade-off-sliders
	Framework Compass Chart	https://medium.com/flowingis/framework-compass-chart-d3851c25b45d
	SWOT Analysis	www.mindtools.com/pages/article/newTMC_05.htm

Summary

This chapter talked about the importance of decision-making principles when choosing a framework or making any other technical decision. We explored some technical decision-making anti-patterns and the problems that they can bring to your organization. We analyzed the principle behind the Frameworkless Movement and talked about some tools that can help you and your team in making mindful technical decisions.

Index