Understanding single-page web applications

SAMPLE CHAPTER

SPA Design and Architecture

Emmit A. Scott, Jr.

FOREWORD BY Burke Holland

www.itbook.store/books/9781617292439

SPA Design and Architecture

by Emmit A. Scott, Jr

Sample Chapter 1

Copyright 2016 Manning Publications

www.itbook.store/books/9781617292439

brief contents

PART 1 THE BASICS 1

1 ▪ What is a single-page application? 3
2 ▪ The role of MV* frameworks 22
3 ▪ Modular JavaScript 52

PART 2 CORE CONCEPTS 83

4 ▪ Navigating the single page 85
5 ▪ View composition and layout 106
6 ▪ Inter-module interaction 129
7 ▪ Communicating with the server 156
8 ▪ Unit testing 186
9 ▪ Client-side task automation 209

appendix A Employee directory example walk-through 229
appendix B Review of the XMLHttpRequest API 259
appendix C Chapter 7 server-side setup and summary 266
appendix D Installing Node.js and Gulp.js 277

www.itbook.store/books/9781617292439

Part 1

The basics

This part of the book will get you acquainted with some basic concepts you'll need to know before developing your first single-page web application.

In chapter 1, we'll talk about what an SPA is in very clear terms. It's important to know what this type of architecture involves and why you might choose it over that of a traditional web application.

Keeping your application's code base clean and maintainable becomes critical when working within the context of a single page. Chapter 2 compares different styles of JavaScript framework that help you achieve that goal. The chapter frames the discussion with an introduction to the three architectural patterns that heavily influenced these frameworks: MVC, MVP, and MVVM. The chapter then progresses into how the same application must change based on the style of framework that's implemented.

In chapter 3, you'll get a crash course on the module pattern and how it will change the way you think about organizing your JavaScript code. Using this pattern, you'll be able to create functions and variables as you normally would but within the cozy confines of a structure that mimics classic encapsulation in other languages. As you'll find out in this chapter, modular programming is crucial for a successful SPA.

www.itbook.store/books/9781617292439

www.itbook.store/books/9781617292439

1 What is a single-page application?

This chapter covers

- The definition of a single-page application (SPA)
- An overview of the basic elements of an SPA
- The benefits of SPAs over traditional web applications

Developers have been chasing the dream of delivering web applications with the look and feel of native desktop applications for about as long as they've been writing them. Various solutions for a more native-like experience, such as IFrames, Java applets, Adobe Flash, and Microsoft Silverlight, have been tried with varying degrees of success. Though different technologies, they all have at least one goal in common: bringing the power of a desktop app to the thin, cross-platform environment of a web browser. The single-page (web) application, or SPA, shares in this objective, but without a browser plugin or a new language to learn. The idea that a native-like experience can be realized using only JavaScript, HTML, and Cascading Style Sheets (CSS) is a tantalizing thought, but what is an SPA under the covers, and where did this idea begin?

www.itbook.store/books/9781617292439

The stage was set in the early 2000s. A brand-new way of thinking about web-page design came about when the AJAX movement started to gain steam. It began with an interesting, yet obscure, ActiveX control in Microsoft's Internet Explorer browser, used to send and receive data asynchronously. These humble beginnings eventually led to a revolution, when the control's functionality was officially adopted by the major browser vendors as the *XMLHttpRequest (XHR)* API.

Developers who began to merge this API with JavaScript, HTML, and CSS obtained remarkable results. The blending of these techniques became known as *AJAX*, or Asynchronous JavaScript and XML. AJAX's unobtrusive data requests, combined with the power of JavaScript to dynamically update the Document Object Model (DOM), and the use of CSS to change the page's style on the fly, brought AJAX to the forefront of modern web development.

Piggybacking off this successful movement, the SPA concept takes web development to a whole new level by expanding the page-level manipulation techniques of AJAX to the entire application. Additionally, the patterns and practices commonly used in the creation of an SPA can lead to overall efficiencies in application design, code maintenance, and development time. Having a successful implementation of a single-page application, though, will be greatly impacted by your understanding of SPA architecture.

As with most emerging solutions, single-page application design comprises a variety of approaches. Varying opinions by today's experts, plus a multitude of competing libraries and frameworks, can make finding the right solution for your SPA project challenging. The more you know going into it, the more successful you'll be in finding the implementation that's right for you and your development goals. That's why I'll start by providing a clear understanding of an SPA and its benefits. Over the course of the book, you'll examine each facet of SPA development by using a style of JavaScript frameworks commonly called *MV* frameworks*.

> **Not everything is MV***
>
> Our discussion of SPAs in this book is limited to MV* frameworks (and you'll learn more about them in chapter 2). It's important to make this distinction up front, however, because other approaches can be used to create an SPA, including using React (https://facebook.github.io/react) or Web Components (a W3C specification for a set of standards for component-based web development), for example.

1.1 SPA in a nutshell

In an SPA, the entire application runs as a single web page. In this approach, the presentation layer for the entire application has been factored out of the server and is managed from within the browser. To get a better idea of what this looks like, you'll review a couple of illustrations.

www.itbook.store/books/9781617292439

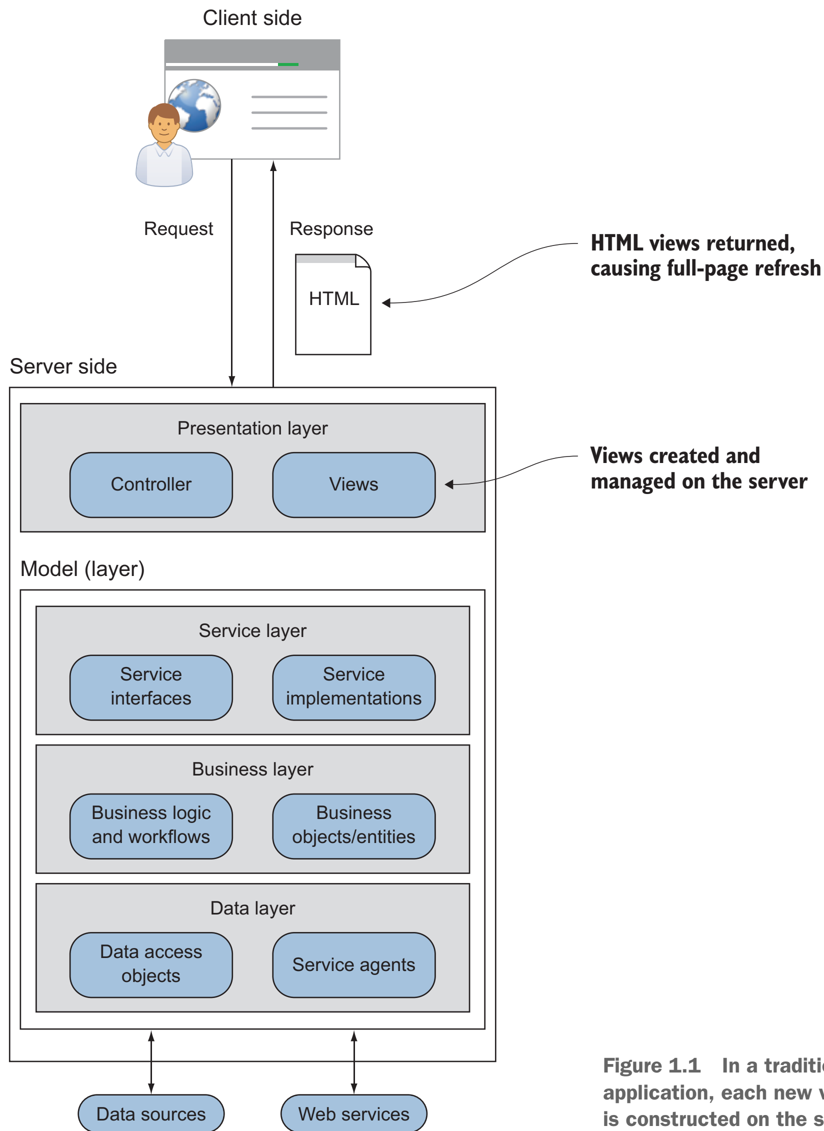


Figure 1.1 In a traditional web application, each new view (HTML page) is constructed on the server.

First, let's take a look at a web application that's not an SPA. Figure 1.1 shows a large web application that uses a traditional server-side design.

With this design, each request for a new view (HTML page) results in a round-trip to the server. When fresh data is needed on the client side, the request is sent to the server side. On the server side, the request is intercepted by a controller object inside the presentation layer. The controller then interacts with the model layer via the service layer, which determines the components required to complete the model layer's task. After the data is fetched, either by a data access object (DAO) or by a service

www.itbook.store/books/9781617292439

agent, any necessary changes to the data are then made by the business logic in the business layer.

Control is passed back to the presentation layer, where the appropriate view is chosen. Presentation logic dictates how the freshly obtained data is represented in the selected view. Often the resulting view starts off as a source file with placeholders, where data is to be inserted (and possibly other rendering instructions). This file acts as a kind of template for how the view gets stamped whenever the controller routes a request to it.

After the data and view are merged, the view is returned to the browser. The browser then receives the new HTML page and, via a UI refresh, the user sees the new view containing the requested data.

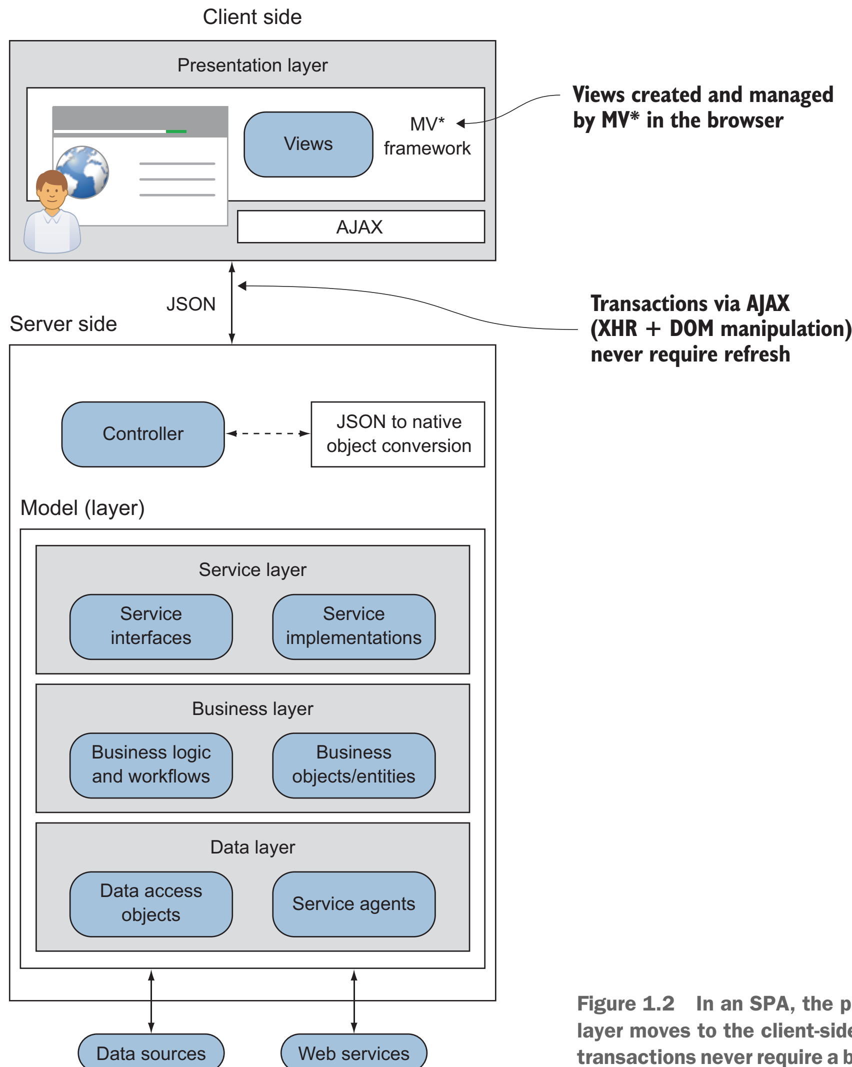


Figure 1.2 In an SPA, the presentation layer moves to the client-side code, and transactions never require a browser refresh.

www.itbook.store/books/9781617292439

Figure 1.2 demonstrates how this design could look as an SPA. Notice what has happened with the presentation layer and our transactions.

Moving the process for creating and managing views into the UI decouples it from the server. From an architectural standpoint, this gives the SPA an interesting advantage. Unless you're doing partial rendering on the server, the server is no longer required to be involved in how the data is presented.

The overall SPA design is nearly the same as the traditional design. The key changes are as follows: no full browser refreshes, the presentation logic resides in the client, and server transactions can be data-only, depending on your preference for data rendering.

1.1.1 No browser refreshes

In an SPA, views aren't complete HTML pages. They're merely portions of the DOM that make up the viewable areas of the screen. After the initial page load, all the tools required for creating and displaying views are downloaded and ready to use. If a new view is needed, it's generated locally in the browser and dynamically attached to the DOM via JavaScript. No browser refreshes are ever needed.

1.1.2 Presentation logic in the client

Because our presentation logic is mostly client side in an SPA, the task of combining HTML and data is moved from the server to the browser. As on the server side, source HTML contains placeholders where data is to be inserted (and possibly other rendering instructions). This client-side template is used as a basis for stamping out new views in the client. It's not template HTML for a complete page, though. It's for only the portion of the page the view represents.

The heavy lifting of routing to the correct view, combining data with the HTML template, and managing a view's lifecycle is typically delegated to a third-party JavaScript file commonly referred to as an *MV* framework* (sometimes called an *SPA framework*). Chapter 2 covers templates and MV* frameworks in detail.

1.1.3 Server transactions

In an SPA, several approaches can be used to render data from the server. These include server-side partial rendering, in which snippets of HTML are combined with data in the server's response. This book focuses on an alternative approach, in which rendering is done on the client and only data is sent and received during business transactions. This is always done asynchronously via the XHR API. The data-exchange format is typically JavaScript Object Notation (JSON), though it doesn't have to be. Even using client-side rendering, though, the server still plays a vital role in the SPA. Chapter 7 reviews the role of the server in more detail.

Even if you're already using a server-side design pattern such as Model-View-Controller (MVC) to separate views, data, and logic, reconfiguring your MVC framework for use with SPAs is relatively easy. Therefore, frameworks such as ASP.NET MVC or Spring MVC can still be used with an SPA.

www.itbook.store/books/9781617292439

1.2 A closer look

Now that you have a bird's-eye view of the SPA, let's break it down a little further. Let's talk about what's going on in the presentation layer now that it's moved to the browser. Because upcoming chapters provide more detail, I'll keep this discussion at a high level.

1.2.1 An SPA starts with a shell

The *single-page* part of the SPA refers to the initial HTML file, or *shell*. This single HTML file is loaded once and only once, and it serves as the starting point for the rest of the application. This is the only full browser load that happens in an SPA. Subsequent portions of the application are loaded dynamically and independently of the shell, without a full-page reload, giving the user the perception that the page has changed.

Typically, the shell is minimal in structure and often contains a single, empty DIV tag that will house the rest of the application's content (see figure 1.3). You can think of this shell HTML file as the mother ship and the initial container DIV as the docking bay.

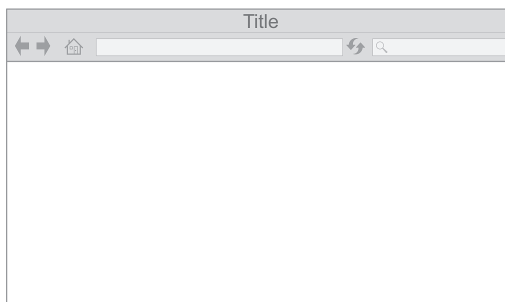
The code for the shell has some of the basic starting elements of a traditional web page, such as a `HEAD` and `BODY`. The following listing illustrates a basic shell file.

Listing 1.1 Example SPA shell

```
<!DOCTYPE html>
<html>
<head>
     <title>Shell Example</title>
     <link rel="stylesheet"
          type="text/css"
          href="app/css/default.css">
</head>
<body>
     <div id="container"></div>
</body>
</html>
```

Load the application's style sheets

Initial container DIV



Shell starts empty

Figure 1.3 The HTML shell is the beginning structure. It has no content yet, only an empty `DIV` tag.

www.itbook.store/books/9781617292439

The initial container DIV can have child containers beneath it if the application's viewable area is divided into subsections. The child containers are often referred to as *regions*, because they're used to visually divide the screen into logical zones (see figure 1.4).

Figure 1.4 Subsections of the shell are called regions. A region's content is provided by a view.

Regions help you divide the viewable area into manageable chunks of content. The region container DIV is where you tell the MV* framework to insert dynamic content. It's worth noting, though, that other paradigms are used by frameworks not covered in this book. React, for example, uses DOM patching rather than the replacement of particular regions.

1.2.2 From traditional pages to views

The "pages" of the application aren't pages at all, at least not in the traditional sense. As the user navigates, the parts of the screen that appear to be pages are actually independent sections of the application's content, called *views*. Chapter 2 covers views in detail. For now, it's enough to know that the view is a portion of the application that the end user sees and interacts with.

Imagining the difference between the average web page and the view of an SPA can be difficult. To help you visualize the difference, take a look at the following figures. Figure 1.5 shows a simple website composed of two web pages. As you can see, both web pages of the traditional site contain the complete HTML structure, including the HEAD and BODY tags.

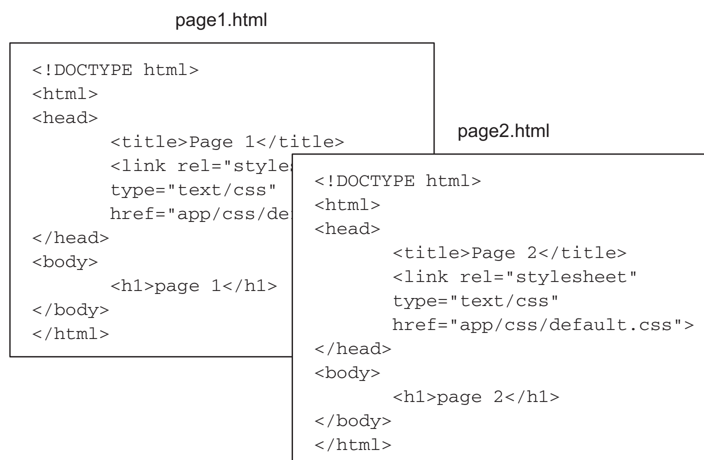


Figure 1.5 In traditional site design, each HTML file is a complete HTML page.

www.itbook.store/books/9781617292439

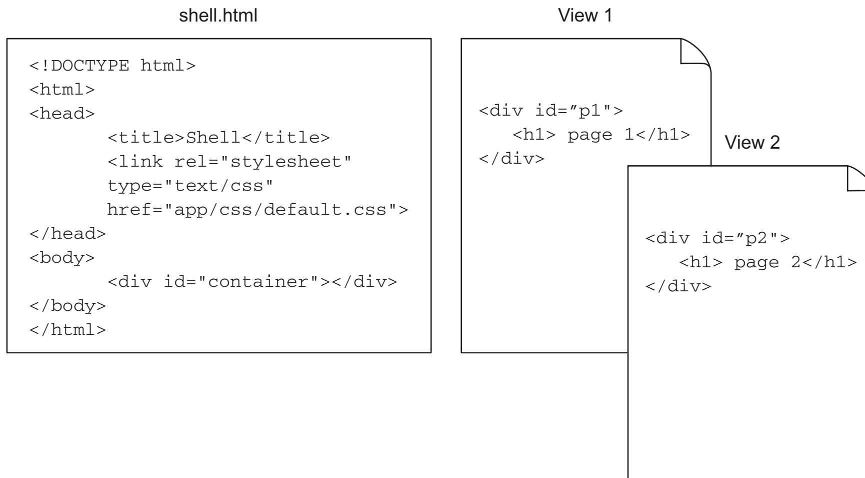


Figure 1.6 In an SPA design, one complete HTML file contains placeholders for the HTML fragments stored in view files.

Figure 1.6 shows the same website as an SPA. The SPA "pages" are only HTML fragments. If the content of the viewable area of the screen changes, that's the equivalent of changing pages in a traditional website.

When the application starts, the MV* framework inserts view 1. When the user navigates to what appears to be a new page, the framework is swapping view 1 for view 2. Chapter 4 covers SPA navigation in detail.

1.2.3 The birth of a view

If sections (or views) of the application aren't part of the initial shell, how do they become part of the application? As mentioned previously, the various sections of the SPA are presented on demand, usually as a result of user navigation. The skeletal HTML structure of each section, called a *template*, contains placeholders for data. JavaScript-based libraries and frameworks, commonly referred to as *MV**, are used to marry data and at least one template. This marriage ultimately results in the final view (see figure 1.7). All the screen's content beyond the shell gets placed into separate views.

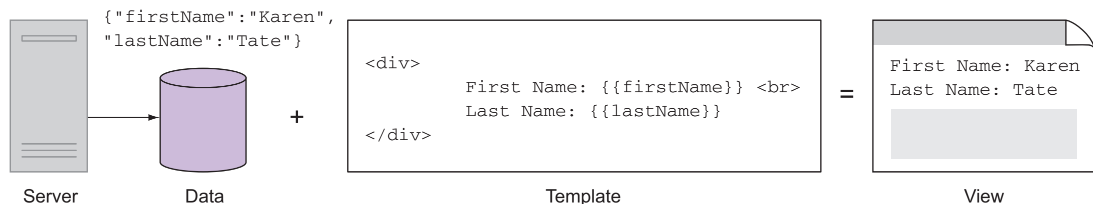


Figure 1.7 A view is the marriage of data and one or more templates.

www.itbook.store/books/9781617292439

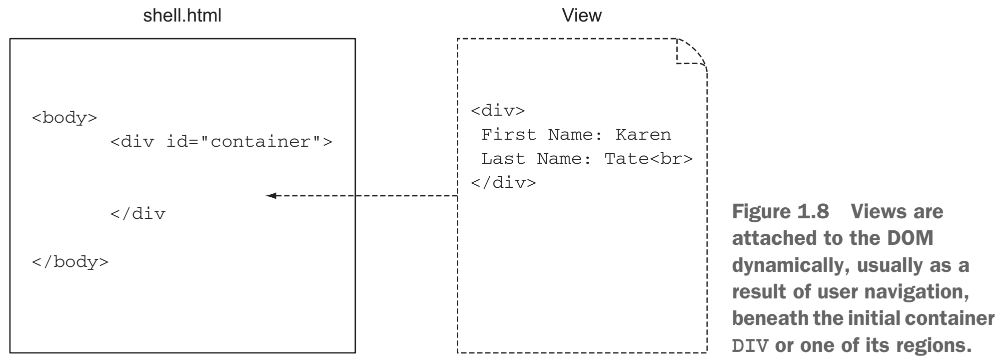


Figure 1.8 Views are attached to the DOM dynamically, usually as a result of user navigation, beneath the initial container `DIV` or one of its regions.

The completed view is attached to the DOM, as needed, either directly under the initial container `DIV`, as illustrated in figure 1.8, or in one of the regions if there are any.

1.2.4 View swapping for zero reload navigation

All of this happens without having to refresh the shell. So instead of getting served a new static page for every navigation request, the SPA can display new content without a disruption for the user. For a particular part of the screen, content of one view is merely replaced by the content of another view. This gives the illusion that the page itself is changing as the user navigates (see figure 1.9). Navigation without a reload is a key feature of the single-page application that gives it the feel of a native application.

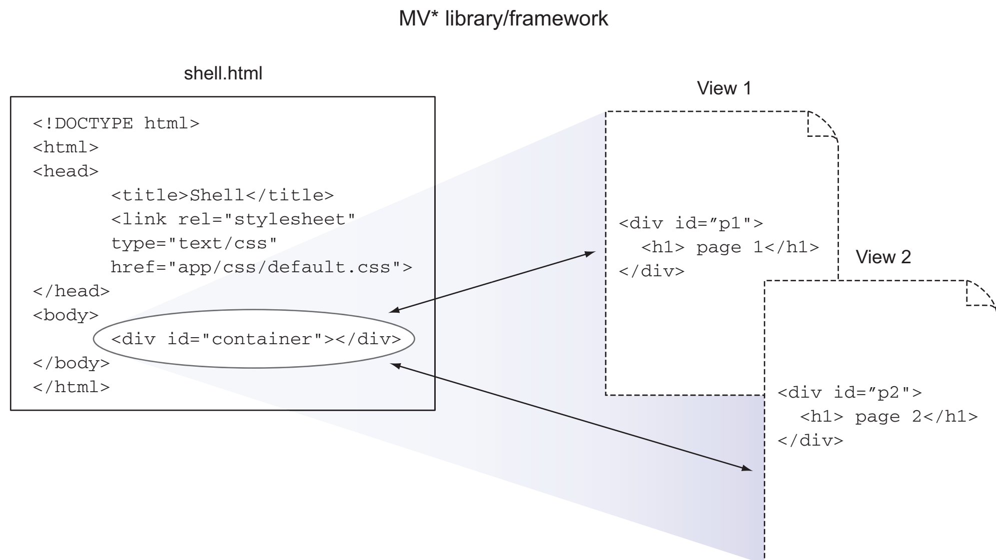


Figure 1.9 Views in an SPA are seamlessly swapped (through DOM manipulation) for a given area of the screen, giving the user a more desktop-like feel.

www.itbook.store/books/9781617292439

The interesting thing about navigation in an SPA is that, to the user, it looks like the page is changing. The URL will look different, and even the Back button can be used to take the user to the previous "page."

Keep in mind that the heavy lifting of creating and managing the views in the client is handled by MV* frameworks. In chapter 2, you'll dissect their various parts to get an even clearer picture.

1.2.5 Fluidity through dynamic updates

Another defining aspect of the SPA is how data from the server can be retrieved asynchronously and inserted dynamically into the application. So not only does the page not reload during navigation, it also doesn't reload while requesting and receiving server data. This, too, gives the appearance and feel of a native application. The techniques of AJAX make this all possible. I began this chapter by talking about the natural evolution of web development and how AJAX played a pivotal role in the development of the SPA concept. So I'd be remiss if I didn't include AJAX as part of the SPA definition.

Previously, I explained in great detail how the page, or view, is swapped dynamically during navigation. Domain data from the server, or from cache, can also be added and removed in the same fashion. The retrieval of the data, which happens silently in the background, can happen in parallel with other data requests. After the data is fetched, it's combined with the HTML template, and the view is updated in real time. The ability to update the page right in front of the user's eyes without even as much as a flicker gives the application a certain fluidity and sleekness that can't be attained with a traditional web application. Chapter 7 covers accessing data in greater detail.

1.3 Benefits of SPAs over traditional web applications

The web browser is still a great way to distribute software because of its "thinness," ubiquity, and standardized environment. End users will already have a web browser. It's also great for software updates, because the updates happen on the server instead of users having to worry about the installation process. Unfortunately, jarring, full-page reloads, content being duplicated with every request, and heavy transaction payloads have all diminished the benefits of browser-delivered content.

Web-based customer interactions are far from over, though. Just the opposite is true, and SPAs are at the forefront of this user-experience revolution. The idea of the single-page application was born out of our desire to give end users the best experience possible. Here are some reasons you should consider single-page application architecture:

- *Renders like a desktop application, but runs in a browser*—The SPA has the ability to redraw portions of the screen dynamically, and the user sees the update instantly. Because the SPA downloads the web-page structure in advance, there's no need for the disruptive request to get a new page from the server. This is similar to the experience a user would get from a native desktop application;

www.itbook.store/books/9781617292439

therefore, it "feels" more natural. An advantage over even the desktop application, the SPA runs in the browser, making its native-like, browser-based environment the best of both worlds.

- *Decoupled presentation layer*—As mentioned previously, the code that governs how the UI appears and how it behaves is kept on the client side instead of the server. This leaves both server and client as decoupled as possible. The benefit here is that each can be maintained and updated separately.
- *Faster, lightweight transaction payloads*—Transactions with the server are lighter and faster, because after initial delivery, only data is sent and received from the server. Traditional applications have the overhead of having to respond with the next page's content. Because the entire page is re-rendered, the content returned in traditional applications also includes HTML markup. Asynchronous, data-only transactions make the operational aspect of this architecture extremely fast.
- *Less user wait time*—In today's web-centric world, the less time a user has to wait for the page to load, the more likely the person is to stay on the site and return in the future. Because the SPA loads with a shell and a small number of supporting files and then builds as the user navigates, application startup is perceived as being quick. As the previous points state, screens render quickly and smoothly, and transactions are lightweight and fast. These characteristics all lead to less user wait time. Performance isn't just a nice-to-have. It equates to real dollars when online commerce is involved. A study by Walmart that was published in *Web Performance Today*[1] indicated that for every 100 ms of performance improvement, incremental revenue grew by up to 1%. In Walmart terms, that's huge.
- *Easier code maintenance*—Software developers are always looking for better ways to develop and maintain their code base. Traditionally, web applications are a bit of a Wild West kind of environment, where HTML, JavaScript, and CSS can be intertwined into a maintenance nightmare. Add in the ability to combine server-side code with the HTML source (think Active Server Pages or JavaServer Pages scriptlets) and you've got a giant, steaming pile of goo. As you'll see in upcoming chapters, MV* frameworks like the ones covered in this book help us separate our code into different areas of concern. JavaScript code is kept where it needs to be—out of the HTML and in distinct units. With the help of third-party libraries and frameworks (for example, Knockout, Backbone.js, and AngularJS), the HTML structure for an area of the screen and its data can be maintained separately. The amount of coupling between the client and the server is dramatically reduced as well.

---

[1] www.webperformancetoday.com/2012/02/28/4-awesome-slides-showing-how-page-speed-correlates-to-business-metrics-at-walmart-com

www.itbook.store/books/9781617292439

1.4 Rethinking what you already know

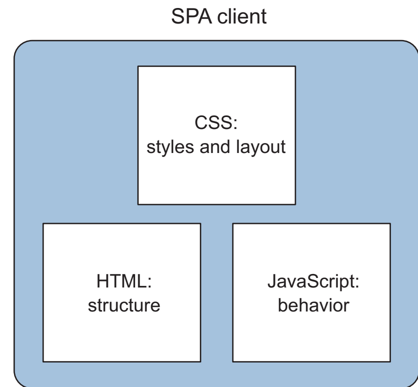


Figure 1.10 CSS, HTML, and JavaScript are the building blocks for the single-page application. There's no special language to learn and no browser plugins required.

In a single-page web application, you use the same languages that you normally use when creating a web application: HTML, CSS, and JavaScript. There's no browser plugin required and no magic SPA language to learn. HTML and CSS continue to be the primary building blocks for the UI's structure and layout, whereas JavaScript is still the cornerstone for interactivity and UI logic (see figure 1.10).

The difference to the user is in how the application will feel using SPA architecture. The navigation feels more like a native desktop application, delivering a smoother, more enjoyable experience. This difference for you, the developer, is that to create an application that functions within a single HTML page, you'll need to rethink your normal approach to web development.

As mentioned in the previous section, in an SPA, the application is broken into independent sections, or views. So you'll no longer create entire pages in which common elements, such as a header or a main menu, are repeated. Even the common sections are views in an SPA. You'll also have to stop thinking about the layout of individual pages and start thinking in terms of view placement in the available real estate of the screen. As it turns out, this is easy after you get the hang of it. Global layout areas, such as a main menu, remain fixed throughout the user experience. Shared areas of the screen, such as the center content well, are reused by the application to swap the various views (as well as entire regions) during user navigation.

To the end user, though, the application can *look* exactly like a traditional web application. As figure 1.11 illustrates, it can have a header, a sidebar, or any other typical web-page element.

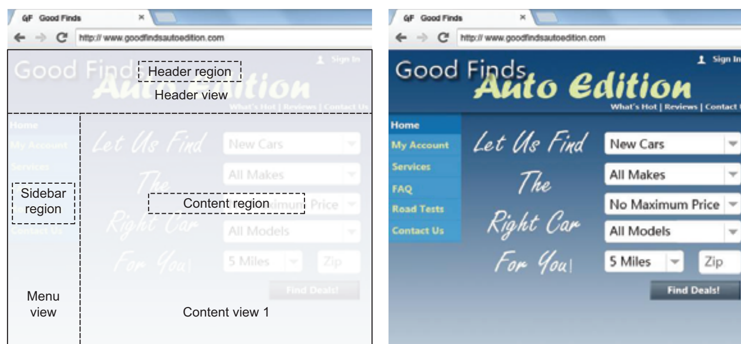


Figure 1.11 Using regions, an SPA's views can be placed so that it looks exactly like a traditional web page.

www.itbook.store/books/9781617292439

On the JavaScript side, you'll continue to code as you normally would, with one major exception. Because you're dealing with a single page that doesn't refresh, simple global scope for variables and functions won't suffice. You'll divide your code into workable units and house it in special functions called *modules* that have their own scope. This frees you from having to create all your variables and functions in the global namespace.

Communication with the server in an SPA is via AJAX. Though the name implies XML, most modern SPAs use AJAX techniques but use JSON as the preferred data-exchange format. It's an ideal format for the SPA because it's lightweight and compact, and its syntax is well-suited for describing object structure. But AJAX should be nothing new to most developers. Even traditional web applications typically use at least some AJAX.

Your overall design will revolve around keeping all the SPA code easily manageable and decoupled from other areas of concern. But don't worry about any extra complexity. Once you get the hang of the unusual syntax of the module pattern, your life as a developer will get easier. I present modular programming in detail later in the book and use variants of the module design pattern in all the examples. So no worries—you'll see it so much that by the end of the book it'll be second nature to you!

1.5 Ingredients of a well-designed SPA

If you researched the topic of single-page applications before picking up this book, you may have felt a little overwhelmed at your choices. As you've seen so far, the SPA isn't a single technology. It's a federation of technologies that work together to create the finished product. There are almost as many libraries and frameworks as there are opinions about the correct approach to take. So admittedly, trying to find the pieces of the puzzle that not only fit together but also fit the needs of your project and the preferences of your team can be rather daunting.

The good news is that there's a method to the madness. If you look at the single-page application concept as a whole, it can be broken into a list of categories that can fit any style of solution you adopt as your own.

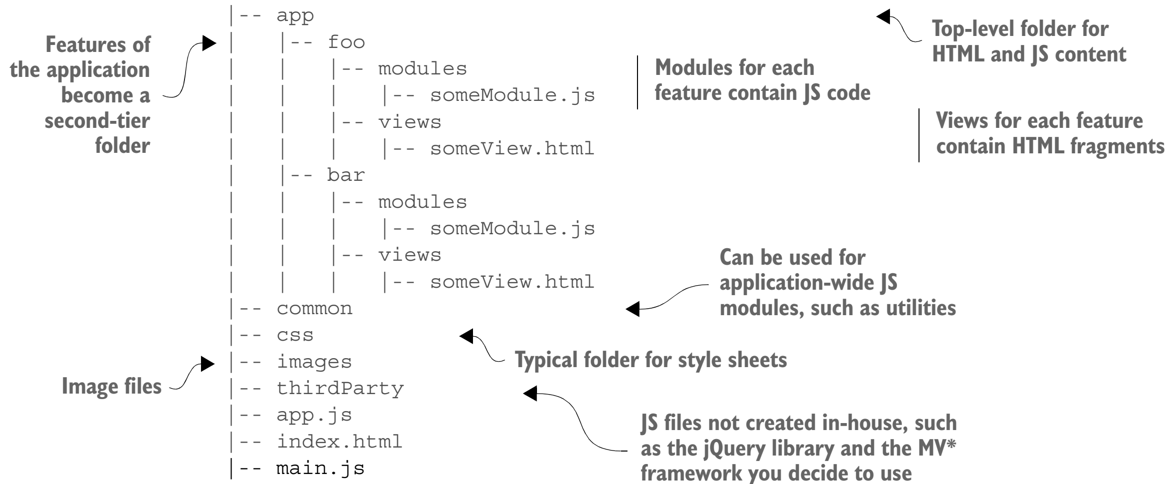
1.5.1 Organizing your project

Having a well-organized project isn't complicated, but it does require some thought and shouldn't be taken for granted. Fortunately, no hard-and-fast rules apply to directory structures. The general rule of thumb is that you should use whatever style works for the development team. A couple of common ways to organize your files are by feature and by functionality.

Grouping similar files by feature is somewhat akin to organizing code in a compiled language, such as Java, into packages. It's clean, discourages the cross-referencing of features, and visually segments files related to a particular feature within the project's file structure. The following listing illustrates how the client code for an application might be arranged using this style.

www.itbook.store/books/9781617292439

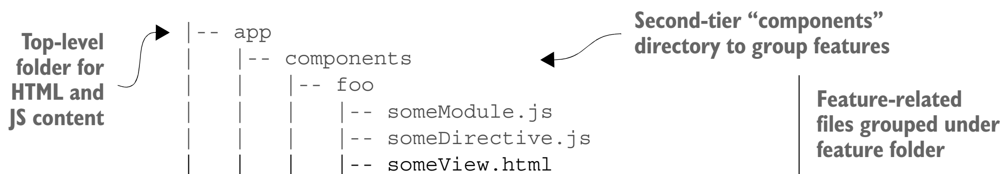
Listing 1.2 Sample directory structure (by feature)




```
|-- app
|   |-- foo
|   |   |-- modules
|   |   |   |-- someModule.js
|   |   |-- views
|   |   |   |-- someView.html
|   |-- bar
|   |   |-- modules
|   |   |   |-- someModule.js
|   |   |-- views
|   |   |   |-- someView.html
|-- common
|-- css
|-- images
|-- thirdParty
|-- app.js
|-- index.html
|-- main.js
```

A modified version of the *by feature* directory structure was proposed in the AngularJS style guide.[2] It favors a simplified version of listing 1.2, which eliminates the named functionality folders under each feature. The blog entry is a good read and has several variations based on the size and complexity of the application; the gist of the structure is specified in the following listing. In this version, boundaries are removed from the various file types within a feature. The style guide argues that this simpler version still groups things by feature but is more readable and creates a more standardized structure for AngularJS tools.

Listing 1.3 Simplified "by feature" directory structure




```
|-- app
|   |-- components
|   |   |-- foo
|   |   |   |-- someModule.js
|   |   |   |-- someDirective.js
|   |   |   |-- someView.html
```

Alternatively, you and your development team might elect to organize the project by functionality (see listing 1.4). This is perfectly acceptable as well. Most SPA libraries and frameworks aren't that opinionated when it comes to directory structure. The choices come down to preference. If you do choose to organize your directory by functionality, it's still a good idea to include the name of the feature as a subfolder under the functionality. Otherwise, under each functionality folder, you'll end up

[2] http://blog.angularjs.org/2014/02/an-angularjs-style-guide-and-best.html or http://angularjs.blogspot.co.uk/2014/02/an-angularjs-style-guide-and-best.html

www.itbook.store/books/9781617292439

having many unrelated files together. That might be all right for smaller applications, but for large applications, this leads to a sort of "junk drawer" effect.

Listing 1.4 Sample directory structure (by functionality)

```
|-- app
|    |-- modules
|    |    | -- foo
|    |    |     |-- someModule.js
|    |    | -- bar
|    |    |     |-- someModule.js
|    |-- views
|    |    | -- foo
|    |    |     |-- someView.html
|    |    | -- bar
|    |    |     |-- someView.html
```


The preceding two listings are pretty basic, to give you the idea. The size of the application, architecture choices, and personal preferences also influence the types of folders used and their names. The term *modules* might be labeled *js* or *scripts*. Instead of *views*, you might choose *templates*. Even the type of framework you incorporate might influence the way you choose to create your directory structure. If you're creating an AngularJS project, for example, you might also have other folders such as *controllers*, *directives*, and *services*.

However you choose to stack it, having an agreed-upon file structure and sticking to that organizational model will greatly enhance your chances for a successful project.

1.5.2 Creating a maintainable, loosely coupled UI

Having clean, organized JavaScript code is a step in the right direction for building scalable, maintainable single-page applications. Layering the code so that the JavaScript and HTML can be as loosely coupled as possible is another tremendous step. This approach still allows HTML and JavaScript to interact but removes the need for direct references in the code.

How are these separate layers achieved? Enter MV* patterns. Patterns to separate data, logic, and the UI's view have been around for years. Some of the most notable ones are Model-View-Controller (MVC), Model-View-Presenter (MVP), and Model-View-ViewModel (MVVM). In recent years, these patterns have begun appearing in the form of JavaScript libraries and frameworks to help apply these same concepts to the front end of web applications. The basic idea is that a framework or library, outside your own logic, manages the relationship between the JavaScript and the HTML. The MV* libraries and frameworks allow you to design the UI such that domain data (the model) and the resulting HTML "page" the user interacts with (the view) can communicate but are maintained separately in code. The last component of the MV* pattern, the controller or ViewModel or presenter, acts as the orchestrator of all this.

www.itbook.store/books/9781617292439

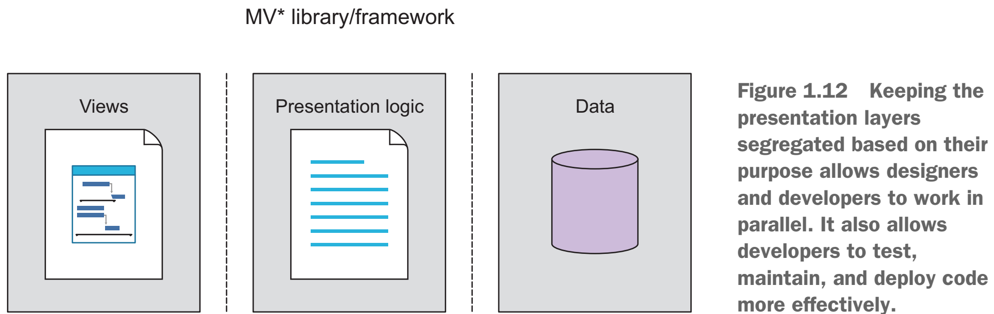


Figure 1.12 Keeping the presentation layers segregated based on their purpose allows designers and developers to work in parallel. It also allows developers to test, maintain, and deploy code more effectively.

Keeping the view, logic, and data separated, as in figure 1.12, is an effective tool in the design of a single-page application.

Achieving this level of separation in your SPA has the following advantages:

- Designers and developers can more effectively collaborate. When the view is void of logic, each resource can work in parallel toward the same goal without stepping on each other's toes.
- Separate view and logic layers can also help developers create cleaner unit tests, because they have to worry about only the nonvisual aspect of a feature.
- Separate layers help with maintenance and deployments. Isolated code can more easily be changed without affecting other parts of the application.

It's OK if this facet of SPA development still seems a little murky at this point. This is one of the harder concepts to grasp. Don't worry, though. Chapter 2 covers the MV* patterns thoroughly.

1.5.3 Using JavaScript modules

Having an elegant way of allowing all your JavaScript code to coexist harmoniously in the same browser page is a necessity in an SPA. You can achieve this by placing the functionality of your application into modules. Modules are a way to group together distinct pieces of functionality, hiding some parts while exposing others. In the ECMAScript 6 version of JavaScript, modules will be supported natively. Meanwhile, various patterns, such as the module pattern, have emerged that you can use as a fallback.

In a traditional web application, whenever the page is reloaded, it's like getting a clean slate. All the previous JavaScript objects that were created get wiped away, and objects for the new page are created. This not only frees memory for the new page but also ensures that the names of a page's functions and variables don't have any chance of conflicting with those of another page. This isn't the case with a single-page application. Having a single page means that you don't wipe the slate clean every time the user requests a new view. Modules help you remedy this dilemma.

The module limits the scope of your code. Variables and functions defined within each module have a scope that's local to its containing structure (see figure 1.13).

www.itbook.store/books/9781617292439

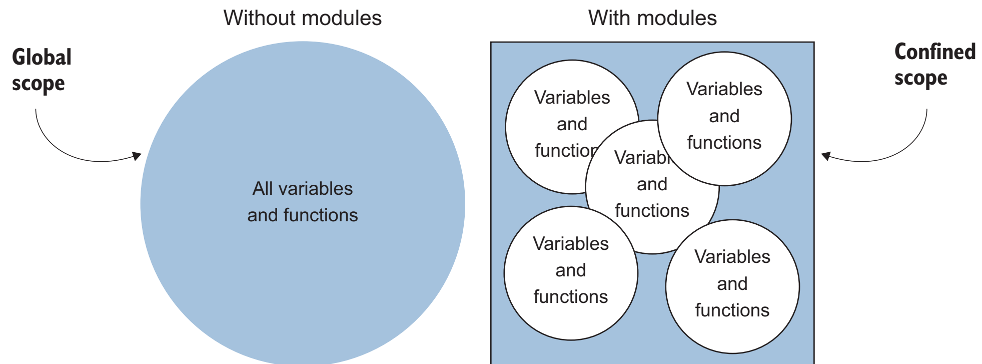


Figure 1.13 Using the module pattern limits the scope of variables and functions to the module itself. This helps avoid many of the pitfalls associated with global scope in a single-page application.

The module pattern, combined with other techniques to manage modules and their dependencies, gives programmers a practical way to design large, robust web applications with single-page architecture.

This book covers the topic of modular programming with JavaScript quite extensively. Chapter 3 provides an introduction. You'll also explore the topic of script loaders, which help manage the modules and their dependencies. Throughout the entire book, you'll rely on the module pattern to help build your examples.

1.5.4 Performing SPA navigation

Chapter 4 provides an in-depth look at client-side routing. To give users the feeling that they're navigating somewhere, single-page applications normally incorporate the idea of *routing* in their design: JavaScript code, either in the MV* framework or via a third-party library, associates a URL-style path with functionality. The paths usually look like relative URLs and serve as catalysts for arriving at a particular view as the user navigates through the application. Routers can dynamically update the browser's URL, as well as allow users to use the Forward and Back buttons. This further promotes the idea that a new destination is reached when part of the screen changes.

1.5.5 Creating view composition and layout

In a single-page application, the UI is constructed with views instead of new pages. The creation of content regions and the placement of views within those regions determine your application's layout. Client-side routing is used to connect the dots. All of these elements come together to impact both the application's usability and its aesthetic appeal.

In chapter 5, you'll look at how to approach view composition and layout in an SPA, tackling both simple and complex designs.

www.itbook.store/books/9781617292439

1.5.6 Enabling module communication

Modules encapsulate our logic and provide individual units of work. Although this helps decouple and privatize our code, we still need a way for modules to communicate with each other. In chapter 6, you'll learn the basic ways in which modules communicate. In doing so, you'll also learn about a design pattern called *pub/sub*, which allows one module to broadcast messages to other modules.

1.5.7 Communicating with the server

I began our definition of a single-page application by discussing the metamorphosis that web pages have undergone since the introduction of the XMLHttpRequest API. The collection of techniques, called AJAX, that revolve around this API is at the heart of the SPA. The ability to asynchronously fetch data and repaint portions of the screen is a staple of single-page architecture. After all, in an SPA we create the illusion for users that, as they navigate, the screen is somehow changing smoothly and effortlessly. So what would this feat of showmanship by the application be without the ability to acquire data for our users?

Chapter 7 focuses on using our MV* frameworks to make calls to our server. You'll see how these frameworks abstract away a lot of the boilerplate code used in making requests and processing results. In doing so, you'll learn about something called a *promise* and a style of web service called a *RESTful service*.

1.5.8 Performing unit testing

An important but overlooked part of designing a successful single-page application is testing your JavaScript code. We test our back-end code to smithereens. Unfortunately, JavaScript unit tests aren't always performed so religiously. Today, many good unit-testing libraries are available. In chapter 8, you'll get an introduction to basic JavaScript unit testing with a framework called QUnit.

1.5.9 Using client-side automation

In chapter 9, you'll learn about using client-side automation not only to create a build process for your SPA but also to automate common development tasks.

1.6 Summary

Here's a quick recap of what you've learned about SPAs so far:

- SPAs are an approach to web development in which the entire application is housed in a single page.
- In an SPA, no full-page refreshes occur after the application loads. Instead, presentation logic is loaded up front and presented in terms of view swapping within content regions.
- SPAs communicate with the server asynchronously. Often the data format used in this communication is JSON-formatted text.

www.itbook.store/books/9781617292439

- MV* frameworks provide the mechanism used by SPAs to marry data from our server requests with the views the user sees and interacts with. There are alternatives to MV* not covered in the book, particularly when using technologies such as React or Web Components.
- Instead of relying on global variables and functions, the JavaScript code in an SPA is organized using modules. Modules provide state and/or data encapsulation. They also help code stay decoupled and more easily maintained.
- Some of the benefits of an SPA include a desktop-like feel, a decoupled presentation layer, faster and lighter payloads, less user wait time, and easier code maintenance.

www.itbook.store/books/9781617292439

WEB DEVELOPMENT

SPA Design and Architecture

Emmit A. Scott, Jr.

The next step in the development of web-based software, single-page web applications deliver the sleekness and fluidity of a native desktop application in a browser. If you're ready to make the leap from traditional web applications to SPAs, but don't know where to begin, this book will get you going.

SPA Design and Architecture teaches you the design and development skills you need to create SPAs. You'll start with an introduction to the SPA model and see how it builds on the standard approach using linked pages. The author guides you through the practical issues of building an SPA, including an overview of MV* frameworks, unit testing, routing, layout management, data access, pub/sub, and client-side task automation. This book is full of easy-to-follow examples you can apply to the library or framework of your choice.

What's Inside

- Working with modular JavaScript
- Understanding MV* frameworks
- Layout management
- Client-side task automation
- Testing SPAs

This book assumes you are a web developer and know JavaScript basics.

Emmit Scott is a senior software engineer and architect with experience building large-scale, web-based applications.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/spa-design-and-architecture

"Takes a very complex topic and breaks it down into easily understandable and digestible pieces."
—From the Foreword by Burke Holland, Telerik

"A great resource for this hot development topic."
—Bruno Sonnino
Revolution Software

"Gives a crystal-clear, multi-faceted, and well-structured presentation of what state-of-the-art SPAs are. I highly recommend it!"
—Alain Couniot, STIB-MIVB

"The code examples are detailed, informative, and practical. They provide a real-world context to the topic."
—John Shea, Endicott College

ISBN 13: 978-1-61729-243-9
ISBN 10: 1-61729-243-5

$49.99 / Can $57.99 [INCLUDING eBOOK]

www.itbook.store/books/9781617292439