SAMPLE CHAPTER

Aurelia IN ACTION

Sean Hunter

MANNING

www.itbook.store/books/9781617294785

Aurelia in Action

by Sean Hunter

Chapter 1

brief contents

1 Introducing Aurelia

This chapter covers

- Examining what Aurelia is and is not, and why you should care
- Identifying applications suited to development using the Aurelia framework
- Looking at what you'll learn in this book
- Touring the Aurelia framework

Aurelia is a frontend JavaScript framework focused on building rich web applications. Like other frameworks, such as Angular and Ember.js, Aurelia is a single-page application (SPA) development framework. This means that Aurelia applications deliver the entire user experience (UX) on one page without requiring the page to be reloaded during use. At its core, writing an Aurelia application means writing a JavaScript application. But Aurelia applications are written with the latest versions of JavaScript (ES2015 and beyond, which we'll dig into as we go along, or TypeScript). The Aurelia framework has all the tools you need to build the rich and responsive web applications that users expect today, using coding conventions closely aligned to web standards.

1.1 Why should you care?

Imagine you're having a Facebook chat session with your friend Bob, and every time you send a message, you need to wait because there's a three-second delay while the page reloads to show you whether you've successfully sent the message, whether Bob has received it, and whether you've received any other messages from Bob in the meantime. In this scenario, it would be difficult to have a fluid conversation because of the jarring pause between entering your message and receiving feedback from the application. You may ask Bob a question, only to find that by the time the page reloads, he's already answered it. Today, however, the experience is much different. As soon as you start typing a message, Bob can see that you're composing a message for him, and when you click Send you receive visual feedback in the form of a checkmark to indicate that the message was delivered successfully. It's easy to gloss over functionality like this today because it's a part of so many applications we use all the time, such as Slack, Skype, and Facebook Messenger.

Now imagine that you've been tasked with building a line-of-business application for your department. The HR department has implemented an employee-of-the-month system where staff nominate and vote on who most deserves a monthly prize. This app would be expected to provide things like the following:

- A responsive voting system
- Live updating charts showing an overview of who has the most votes
- Validation to prevent employees from voting more than once

DEFINITION *Responsive web applications* can be used across a variety of devices, from smartphones to desktop PCs. Typically, this is achieved using CSS media queries to resize various sections of the page or even hide them entirely so that the UX is optimal for the device at hand.

The features listed for your HR application are common examples of the kinds of things that users—like your fictional HR department—expect in rich web applications. Applications like Facebook and Slack have raised the bar in terms of what users expect from all web applications. I've noticed a trend over the past few years where clients have begun to expect the same kind of richness out of a line-of-business application that they're used to seeing in applications they use outside of the office. Using an SPA framework like Aurelia makes it vastly simpler to build these kinds of applications, compared with the traditional request/response style of architecture used with frameworks such as ASP.NET MVC, JSP, or Ruby on Rails, to name a few.

Given that you want to create rich, responsive web applications, the next logical question is, which technology should you use to do this? A useful technique for answering this kind of question is to analyze the kinds of attributes that are important to you and your team with a set of questions like this:

- *How complicated are the applications you're trying to build?* You don't want to use a jackhammer to crush a walnut when a nutcracker will do the job. Conversely,

you want to make sure that you have a sufficient foundation in place that will support the kind of application you're trying to build.

- *How important is web-standards compliance to the team?* A framework that adheres more closely to web standards is more likely to look familiar to anybody with web-development experience, regardless of whether they've used a given framework in the past. Such a framework is also more likely to play nicely with other web technologies such as third-party libraries and frameworks.
- *What past development experience does the team have?* Frameworks and libraries can have a steeper or shallower learning curve, depending on the experience of the team.
- *How is the team organized?* Do you need designers and developers to be able to work together on the same project? Do you have a team of 1 to 5 or 100 to 500?
- *What kind of commercial and community support do you need?* How important is it to be able to pick up the phone or send an email to the team or company responsible for the framework? What kind of community are you looking to join?

Let's look at where Aurelia sits in terms of each of these questions; in doing so, you'll get a feel for the kinds of problems Aurelia helps you solve, and some of the features available in Aurelia's toolbox.

1.1.1 How complicated are the applications you're building?

With most SPAs, it's helpful to have a minimal set of tools to build the kind of experience that users expect. If these tools aren't present in the framework, then you may need to either bring them in as a third-party project dependency or build a bespoke implementation. Aurelia provides a core set of functionalities that most SPAs need out of the box, as a set of base modules. Most of these modules are available as optional plugins, so if you don't need a part, you can leave it out. The following subsection presents a basic list of the features that Aurelia offers. I'll include only a brief definition at this point to give you a taste of what's available. We'll dive into each of these topics in more detail later.

THE BASICS: SPA BREAD AND BUTTER

The following functionality is bread and butter to almost every SPA, regardless of the complexity level:

- *Routing*—SPA users expect your application to behave like a standard website. This means that they should be able to bookmark a URL to get back to it later and navigate between the different states of your application using the browser's Forward and Back buttons. The Aurelia router solves this problem by allowing you to build URL-based routing into the core of your application. Routing also allows you to take advantage of a technique called *deep linking*, which allows users to bookmark a URL from deep inside the application (for example, a specific product on an e-commerce site) and return to it later.

- *Data binding and templating*—Virtually every SPA needs a way to take input from the page (either via DOM events or input fields) and push it through to the JavaScript application. Conversely, you'll also need to push state changes back to the DOM to provide feedback to the user. Take a contact form as an example. You need a way of knowing when the email field is modified so that you can validate it in your JavaScript application. You also need to know what the value of the input field is so you can determine the validation result. Once you validate the input field, you need to return the result to the user. Data binding and templating are Aurelia's way of achieving this.
- *HTTP services*—Most SPAs aren't standalone; they need to communicate with or get their data from external services. Aurelia provides several options out of the box to make this easy, without the need to pull in any third-party JavaScript libraries like jQuery AJAX.

GETTING MORE ADVANCED—BEYOND BREAD AND BUTTER

As an SPA increases in size and complexity, you'll often run into a new set of problems. When you run into these problems, it's useful to have the tools to solve them:

- *Components*—One set of tools Aurelia provides for dealing with complexity is components. Components are a way of taking a user-interface (UI) layout and breaking it into small chunks to be composed into an entire view of your SPA. In a way, you can think of the components of your page like objects in a back-end system built using object-oriented programming (OOP), such as Ruby, Java, or C#.
- *Intercomponent communication*—Following the OOP analogy, wherein OOP objects can notify each other of application-state changes, your components also need a way of talking to each other. Aurelia has several options for how you can implement this kind of behavior. The appropriate option again depends on the complexity of your application in terms of the number of components and how interrelated they are. We'll dive into intercomponent communication in depth in chapter 6.

Most SPAs start basic and become more complicated over time. Aurelia allows you to reach for tools to deal with a given level of complexity when you need to but avoids overloading you with that complexity unnecessarily. By the end of this book, you'll be equipped to handle SPAs with varying complexity levels, and you'll know the suitable tool to retrieve from your Aurelia toolbox to solve the problem at hand.

1.1.2 How important is web-standards compliance to the team?

Imagine you're tasked with building an international website that needs to be accessible by users across the globe, some of whom may be vision impaired or have poor-quality internet connections. Enter *web standards*. Web standards provide a common base for the web. Devices and browsers are built to web-standards specifications, and as such, sticking to these specifications when building websites gives you the best

chance of supporting a plethora of devices. These standards are also focused on accessibility; a working group called the Web Content Accessibility Guidelines (WCAG) is devoted to it. Following web standards also gives you a set of tools to enable support for users with a diverse set of accessibility requirements. A simple example of this is alt text (alternative text) on images, which allows screen readers to give vision-impaired users a description of the images on your site.

At the same time, you can reduce your future development costs by building on a stable and well-understood technology set. This makes it easier to bring new team members onto a project who don't necessarily know Aurelia, and allows you to make use of the vast array of third-party JavaScript and CSS libraries that weren't built with Aurelia in mind, which in turn improves maintainability and reduces development costs.

Wherever possible, Aurelia uses existing browser technology rather than reinventing the wheel. A simple example of this is HTML markup. Aurelia uses standards-compliant HTML, which allows both humans and screen readers to read the page source without needing to understand how Aurelia works. As we explore the framework, I'll highlight various points where the core team have leaned on an existing standard web technology to implement a given feature.

1.1.3 What past development experience does the team have?

In the world of web development, the number of new technologies to learn can often seem overwhelming. Given this reality, any boost your team can get in terms of building on past development experience can save you a lot of time. Some of the concepts in Aurelia will feel familiar to those with OOP experience, such as patterns like dependency injection, Model-View-ViewModel, or Event Aggregator. Don't worry if these concepts aren't familiar to you at this point, because we'll delve into each of these throughout the course of the book. On the other hand, Aurelia should also be easy to pick up for people with a good amount of experience with vanilla JavaScript, HTML, and CSS due to Aurelia's close adherence to web standards.

1.1.4 How is the team organized?

The concept of *separation of concerns* between your HTML file (which provides the structure of the page), your CSS file (which determines how the page looks), and your JavaScript (which determines how the page behaves) has existed in web development for some time. The idea is that by splitting these concerns and managing them separately, you should be able to work on any of them independently of the others. It also means that team members with the relevant skill set should be able to work on one piece of the picture without needing to know the in-depth details of the other pieces. For example, a developer should be able to put together the basic HTML structure and JavaScript behavior to then be styled by a team member with more of a focus on UX or design.

Aurelia's opinion on this is that that this concept of separation of concerns is no less important in the world of SPA development than it is in a more traditional server-centric web-development approach.

This approach gives you maximum flexibility. You still have the option of having one person manage an entire vertical slice of the application—JavaScript, HTML, and CSS—but you also have the option of splitting these out if that's the way you'd prefer to work within your team or company, as depicted in figure 1.1.

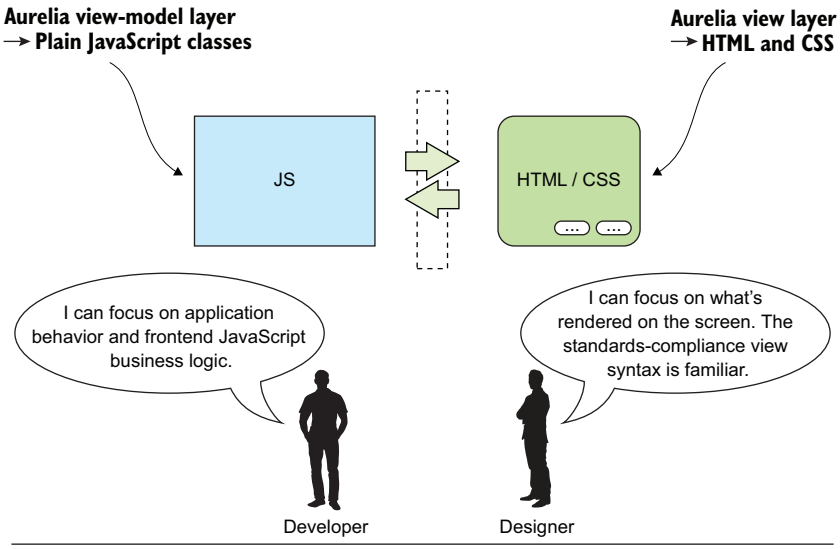

Figure 1.1 Aurelia maintains a separation of concerns between the structure, style, and behavior of pages. This allows for these pieces to be worked on independently, and by the person best suited for the job.

1.1.5 What kind of commercial and community support do you need?

Besides the technical details of any technology choice, an important aspect to consider is how well the product is supported. The level of support available for a given technology can have an enormous impact on how successful it is within your company. Following are some metrics to consider.

LEARNING/TRAINING

How easy is it to take somebody with no experience in the technology and up-skill them to the stage that they're productive with it? Several factors play into this:

- *Documentation*—Aurelia has a detailed set of documentation on the project website (http://aurelia.io/docs). This is actively maintained by the project core team. Often, issues that are raised on GitHub result in a documentation update that clarifies how a feature should be used.
- *Training*—Aurelia has a training program that makes it possible to receive in-person or online training from an Aurelia expert. This training is official and endorsed by the Aurelia core team. Often, it's even provided by core team

members, giving you direct access to people with the most experience working with Aurelia.

- *Community*—If you haven't come across it before, Gitter is a chat client like Slack but focused on allowing threaded conversations on open source projects. The Aurelia community has an active Gitter chat room, and often you'll get a detailed answer to any questions you might have. You can also ask questions and learn about best practices on the Aurelia Discourse forum (https://discourse.aurelia.io/).

SUPPORT

Like most popular JavaScript frameworks today, Aurelia is open source. But unlike most alternatives, Aurelia is one of two SPA frameworks that has commercial support available. Where frameworks such as Angular or React are developed and maintained by Google and Facebook respectively, it's not possible to pay for somebody from these companies to assist you if you get into trouble or want a little extra guidance on a project. Conversely, Blue Spire—the company behind Aurelia—offers commercial support contracts that can be tailored to the needs of your company.

Support is also available in the standard forms that you'd expect from an open source project. Aurelia core team members are quick to respond to GitHub issues or questions on Gitter.

For somebody from a technical background, it can be easy to overlook the fluffier aspects of choosing a technology like support and training. But these aspects make a difference when you look at how a technology will be picked up and used by the company long term.

1.2 What kind of projects make Aurelia shine?

To understand what kind of projects Aurelia works best with, let's look at the web-development models that are available, and how they might consider the context of a sample application. Imagine that you're tasked with building an ecommerce system. This system consists of a set of the following distinct groups of functionalities:

- *Blog*—News and updates about new products or events. This needs to be searchable and is mainly a read-only system.
- *Product list*—A listing of all the products that your company has on offer.
- *Administration*—Administrators need to be able to add new products and view statistics of what users are doing on the site.

You can structure an SPA several ways, but for our purposes, you can split these into the following four main categories:

- Server-side application with a sprinkling of JavaScript
- Client-side-rendered SPA
- Hybrid SPA
- Server-side rendered SPA with client-side continuation

1.2.1 Server-side application with a sprinkling of JavaScript

Figure 1.2 represents a traditional PHP/JSP/ASP.NET/Ruby on Rails–style website. In this model, the user requests the product list. The server is then responsible for rendering it to HTML and returning it to the user. Once the page has reached the client side, you may have a few simple JavaScript widgets such as a product-image lightbox jQuery plugin.

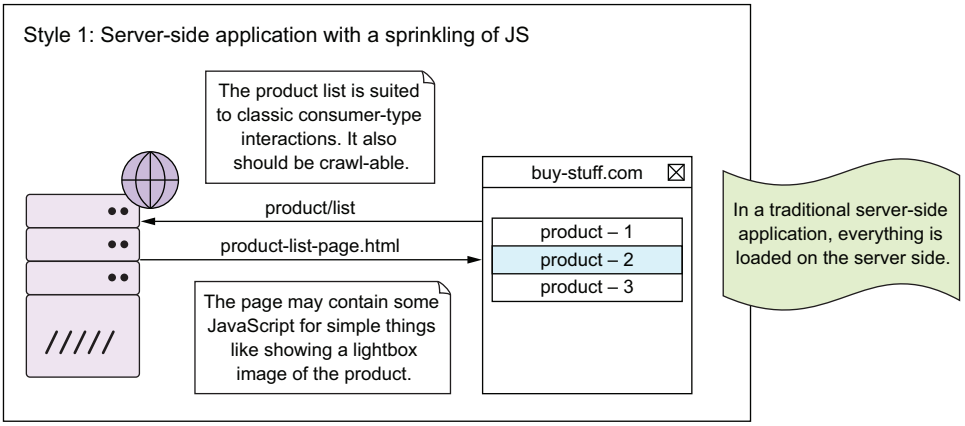

Figure 1.2 Server-side applications retrieve the entire page load as one rendered resource from the server (in addition to any CSS and JavaScript that is typically loaded separately).

The benefits of this approach are as follows:

- *The entire page is rendered on the server*—By scaling the server, you can improve page-load times without worrying about a device's native client-side render performance.
- *The crawler issue*—Page crawlers can effectively crawl a site for products and blog entries because JavaScript (which has patchy support by these technologies at best) isn't required to render the page.
- *Initial page load is generally fast*—The user isn't returned to the page by the server until the page is ready to go.

But there are drawbacks, such as slower subsequent page loads. Although the initial page-load time is relatively quick, subsequent interactions to the page also go through the same lifecycle of a full HTTP request to the server and rerender the entirely new page in the browser. This is OK for features such as the blog and product list but isn't ideal when the user needs to perform a set of interactions on the site and receive rapid feedback (such as commenting on a blog or creating a new product).

1.2.2 SPA rendered on the client side

In contrast to the server-side application, in this scenario the entire payload is loaded up front and returned to the browser in a single batch in a client-side SPA. This payload contains the application scripts and templates along with (optionally) some seed data required on the initial page load (see figure 1.3). In this case, the page is rendered on the client side rather than the server side. Traditionally, in this model, you have the entire e-commerce website returned when the user visits the initial page. Following this, as the user browses through various pages such as products and blog entries, you rerender the page on the client side based on data you received from the server via AJAX.

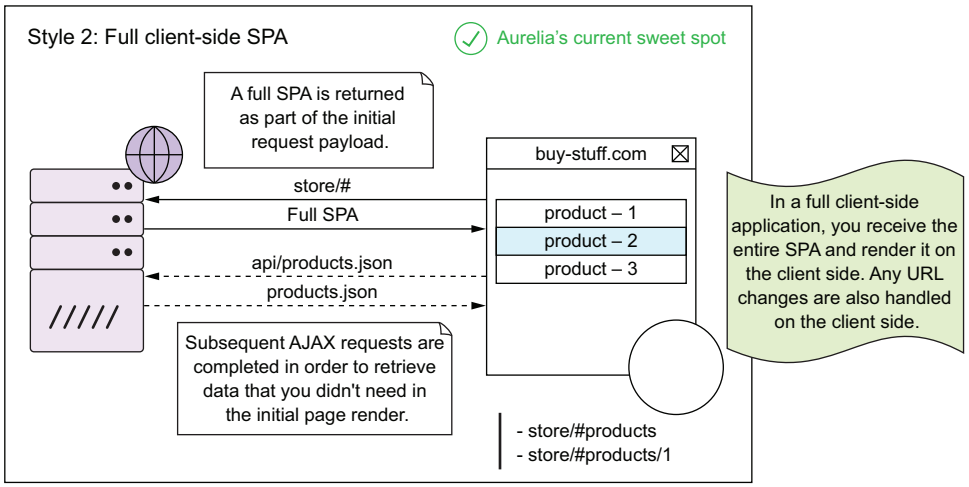

Figure 1.3 Full client-side SPA. An entire application is bundled and returned in one or two responses from the server and rendered in the browser. Subsequent AJAX requests completed to populate data that isn't required in the initial page load.

The benefits of this approach are as follows:

- *Your app feels lightning fast*—As the user clicks around, viewing products and comments on blog threads, the response times are fantastic, with minimal friction. The only time the client needs to talk to the server is when it needs data that the user hasn't seen before. This is often optional data that may not need to be loaded at all. In these cases, it's generally easy to show a spinner or a similar indicator to give the user instant feedback that the data they want is being fetched, and then proceed to show them the rerendered UI once the data has been received.
- *Selectively load and render UI fragments with AJAX*—After initial page load, you can also make the page feel more responsive by executing multiple concurrent asynchronous calls to the backend API. When these asynchronous calls com-

plete, you can provide the results to the user by rerendering only the impacted DOM fragment. An example is retrieving the list of products as JSON from an HTTP response and rerendering only the product list, leaving the rest of the page intact.

The drawbacks are as follows:

- *Large response payload*—Because the entire page loads in one request, users are forced to wait while this happens. Typically, you'd show a spinner or a similar visual device to smooth this interaction. But with large applications, this can be time consuming. Because 47% of users expect a website to load in 2 seconds or less, there's a risk that you may lose some of the visitors to your site if the initial application load exceeds this boundary.
- *Unpredictable page-load times*—Because you're delivering a substantial chunk of JavaScript and state to the application, page-render times can also be unpredictable. If you visit the page from a modern device with a fast processor and high internet bandwidth, there's a good chance you'll have the lightning-fast experience you've come to expect from SPAs. But what if you're in the outback in Australia attempting to load the application on a 5-year-old smartphone over a flaky 3G connection? The experience will be altogether different. Conversely, a traditional server-rendered web application is more predictable (at least in terms of the time required to render the application before it's returned to the user). HTTP servers can be more easily updated and scaled to improve these render times, bringing the issue back into something that you can control, rather than being subject to whatever device the user happens to be visiting the page from.
- *The crawler issue*—Some technologies, such as web crawlers, aren't built with the ability to process JavaScript. This can be a significant issue when building a public-facing SPA that needs to appear in search engine results.

BEST OF BOTH WORLDS

A good option would be to split the application into several separate sites. The blog might still be done in the traditional server-side rendered style (mainly for SEO purposes). But you could potentially develop the administrative site and shopping cart as separate microsites, providing the user with that rich interaction needed for this style of UI.

1.2.3 The hybrid approach: server side with SPA islands

Imagine a new set of requirements have come down from management. After great initial success, the company is looking to expand the use of the site and sell directly from the site. As such, the site needs some interactive pieces like a shopping cart and order screen. Taking the hybrid approach, you'd create the following three new routes in the website (see figure 1.4):

- site/store/cart
- site/store/order
- site/admin

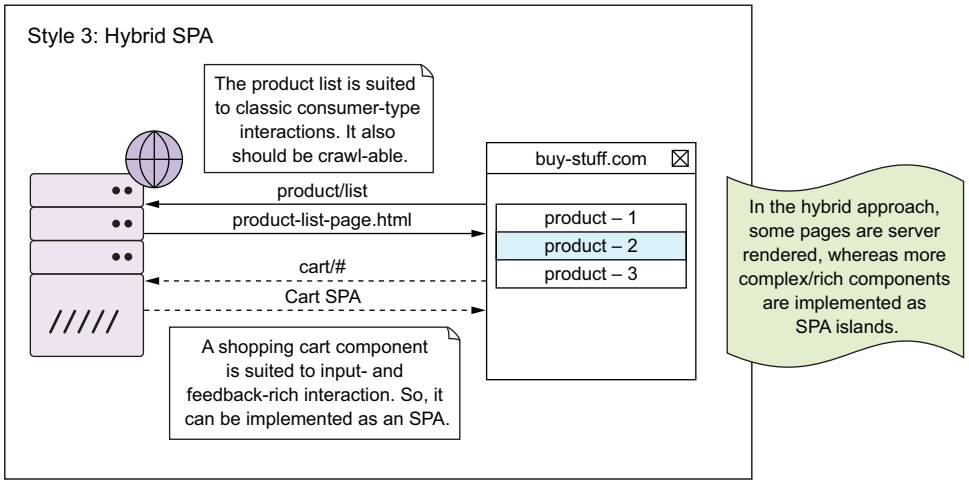

Figure 1.4 Hybrid SPA approach. In this model, the main *content* parts of the site are done with the traditional server-side approach. The *product list* and *blog* endpoints return the relevant HTML rendered from the server. But you then create multiple smaller SPAs to add rich interaction to the parts of the site that need it, such as the store.

Each of these new routes hosts an individual SPA. This approach avoids the need to immediately rewrite the entire site as an SPA, but still gives you the benefit of building the rich interactive parts of the site in a technology designed for that purpose. It also avoids the large payload size to some extent by splitting the SPA into smaller chunks that can be more quickly retrieved from the server and rendered.

The benefits of this approach are as follows:

- *The crawler issue has been solved*—This resolves the SEO and unpredictable page-load issues while still providing an interactive experience.
- *Faster on-ramp*—If you've already got a server-side application, you don't need to rewrite it as an SPA from the ground up, but instead can break out components of the application that are good candidates for SPA-style interaction over time.

The drawbacks are as follows:

- *Added complexity*—Managing application state and page navigation can get tricky when these concerns are dealt with both at the client-side and server-side level. With each new feature, there's a cognitive burden of deciding where everything fits best.
- *Tightly coupled frontend and backend*—In this style of architecture, your backend API is vulnerable to changes on the frontend, and vice versa. Without a clean boundary between the UI and the backend, API changes on either side are likely to trigger changes on the other. This increases the maintenance burden and makes it less likely you'll be able to use your backend API for other client types (such as mobile) in the future.

1.2.4 Server-side-rendered SPA

The Aurelia team has released a technology called *Server Render*. As the name implies, using this approach, the application is rendered on the server side before it's returned to the browser. This technology resolves both the *unpredictable page-load times* and *crawler* issues. This goes a long way toward expanding the set of applications that fall into Aurelia's sweet spot. You can find out more about server-side rendering on the Aurelia documentation site (https://aurelia.io/docs/ssr/introduction/). Take a look at figure 1.5.

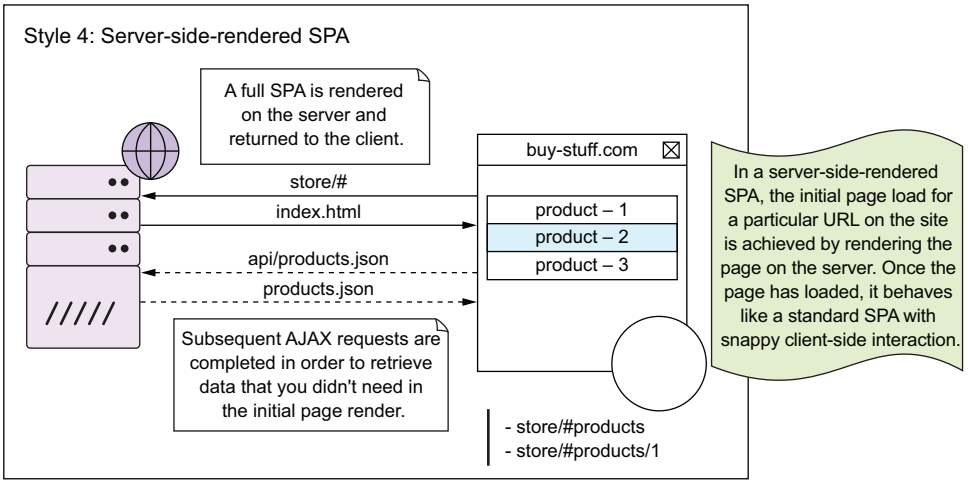

Figure 1.5 In this server-side-rendered SPA model, the initial page render is done on the server side, but once the application has been loaded into the browser, the SPA framework takes over, and subsequent interactions are done on the client side.

The benefit of this approach is that, in a way, this model gives you the best of both worlds. The initial render can be done on the server side, which resolves the issues of unpredictable page-load times and crawler accessibility, but still provides the rich interactive experience that users expect when the page is loaded. This approach may add more complexity to the system architecture by requiring the setup of additional components on the server side.

1.2.5 Where does Aurelia sit?

At its heart, Aurelia is designed to manage your entire web application in the style of a client-side-rendered SPA. Any web application where you need a significant level of interaction from the user that goes beyond simple content consumption is a good fit for Aurelia. Example applications that would be a great fit for Aurelia include the following:

- A messaging client
- A reporting/analytics portal for a website

- A CRUD (create, read, update, delete) application (like a typical forms-over-data example)
- An office-style application (such as Google Docs)
- An admin portal for a website (like the WordPress admin panel)

What these have in common is a user-interaction model more like a traditional desktop application than what you'd historically think of as a website. In the e-commerce example in figures 1.2–1.5 (buy-stuff.com), a great candidate would be a separate sidekick site to the main website that would be used to handle the administrative operations. You might also use Aurelia to build the shopping-cart or blog-comment website components as modules of the larger site.

1.2.6 What makes Aurelia different?

With the substantial number of SPA frameworks available today from the heavy hitters like Angular, React, and Vue, to up-and-comers like Mithril, it's important to consider what makes Aurelia different. What unique value does Aurelia provide that makes it a standout choice for building your web applications? I present you with the Aurelia cheat sheet, four reasons that you can give to your teammates when they ask you this question:

- *It gets out of your way.* Aurelia applications are developed by combining components built with plain JavaScript and HTML. In contrast, many other MV* frameworks today require a comparatively large amount of framework-specific code in both the view layer and the model/controller layer. This increases the concept count, making them more difficult to master and maintain.
- *Aurelia is developed following the convention-over-configuration pattern.* Convention over configuration means having reasonable defaults rather than requiring developers to manually specify every option. But what if the convention doesn't suit you? Aurelia makes it easy to override the default conventions when necessary, and we'll go into this in more detail as we proceed through the chapter.
- *When it comes to web standards, Aurelia's a pro.* Although other frameworks may pay lip service to web standards, Aurelia has them at its core, in its bones. Wherever possible, Aurelia adopts the standard browser implementation of a feature, rather than creating a framework-specific abstraction. A simple example of this is Aurelia's HTML templates, which come directly out of the Web Components Specifications (covered in depth in chapter 12).
- *When it comes to open source, community is king.* Aurelia has a thriving open source community. With core team members, and other Aurelia aficionados available on Slack (https://aurelia-js.slack.com/), Gitter, Stack Overflow, and GitHub help, you can always get the help you need to keep up the momentum while building your Aurelia applications.

1.3 A tour of Aurelia

Often, when you arrive at your hotel in a new city, one of the first things the concierge provides is a tourist map. Perhaps it's not a street-level map with the detail of the individual winding roads you need to take, but it's enough to give you an idea of where the major attractions and suburbs are. With a high-level map in hand, if you get lost, you can generally find your way by aiming for the attraction you want to visit and heading in the right direction. If you need a bit more detail, you might pull out a smartphone to navigate your way through some tricky areas. Similarly, you can think of figure 1.6 as your high-level map of the Aurelia framework.

Aurelia applications are built by combining view/view-model pairs called *components*. A view is a standards-compliant HTML template, and a view-model is a simple JavaScript class. Binding is used to connect fragments of the view (such as an `<input value>`) with properties on a view-model. Events raised on the view (such as an input-value change) trigger a corresponding method call in the view-model. Aurelia uses dependency injection to construct instances of view-models, providing them with their dependencies at runtime. These dependencies can be service classes (as seen in figure 1.6) or framework dependencies, such as the `@inject` decorator.

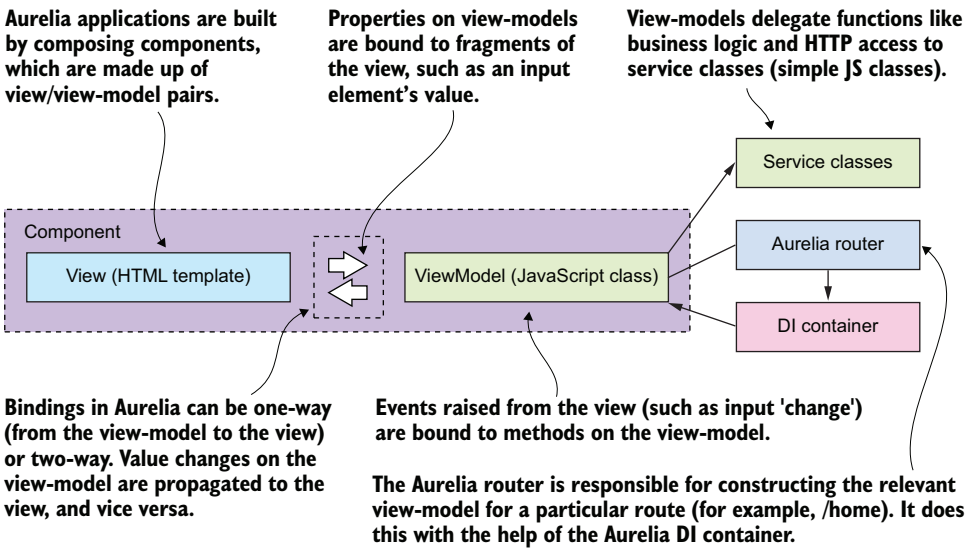

Figure 1.6 A high-level map of the Aurelia framework. At its core, Aurelia is an MV* framework.

DEFINITION Decorators are a new ECMAScript feature that at the time of writing are in stage 2 of the TC-39's proposal process for ECMAScript. Fortunately, the Babel transpiler allows us to use them today, even though they need to make it to stage 4 of this proposal process before they're officially adopted as a part of the language. Decorators allow you to easily and transparently

augment the behavior of an object, wrapping it with additional functionality (for example, logging). Decorators are used throughout the Aurelia framework to change the way that objects behave and can be recognized by their `@` prefix. One example of this is the `@bindable` decorator from the Aurelia framework package. Properties can then be declared as `@bindable` using the imported decorator.

The Aurelia router is used to pick the correct component to load for a given application URL.

The map is highly detailed, but don't worry if you see blocks that you're not familiar with. We'll delve into each area of this diagram to give you a well-rounded understanding of how Aurelia does its thing. Like the concierge, I'm going to draw a line through some of the paths that you'll follow through the framework. These are the code paths that users interacting with the system will trigger every time they load a page or click a button. Like pulling out a smartphone to see a given location in greater detail, we'll zoom in on specific parts of the map that represent important aspects of the framework that we'll expand on throughout this book. Like any good tour, there will be a few interesting detours along the way, but by the time I'm finished guiding you through, you'll have a much clearer idea of where you're going. Let's get started.

1.3.1 Binding

Aurelia's core building blocks, components, are view/view-model pairs, where the view is an HTML template and the view-model is a simple JavaScript class.

> **NOTE** Some components, such as custom attributes and value converters, don't have a corresponding HTML file, but we'll come back to these later.

A technique called *binding* is used to connect DOM fragments on the view (such as `<input value>` or `<h1>` content) to corresponding properties on the view-model. Changes to properties on the view-model are automatically propagated to the view, causing the relevant fragment of the DOM to be rerendered with the updated value. Several options are available to control the behavior of how and when the view or view-model is notified of changes in the pair. This binding workflow is illustrated in figure 1.7.

1.3.2 Handling DOM events

Aurelia's binding system also handles DOM events. Events raised on the view, such as `input value change`, `checkbox checked`, and `button clicked`, are connected to corresponding view-model methods via *binding commands*. Binding commands in Aurelia are declared in the view template and are used to connect a view event with a view-model method, such as `delegate` and `trigger`—we'll delve into these in detail in chapter 4. As in the case of binding, Aurelia provides tools (such as binding behaviors, which you'll encounter in chapter 3) to control when and why the view-model is notified of events raised from the view. Aurelia's event-handling workflow is illustrated in figure 1.8.

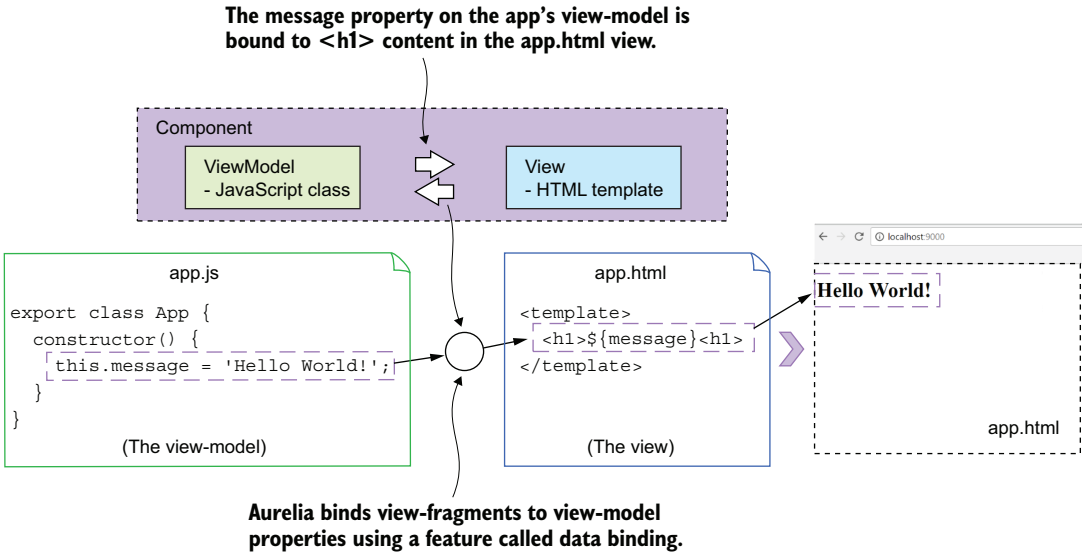

Figure 1.7 The `app` view-model (from the app.js file) is bound to its pair (the `app` view from the app.html file). When the `app` view-model is constructed, the `'message'` property is set to the `'Hello World!'` value. This value is bound to the view using data binding. When this view is rendered, the bound value of the message property is rendered to the view, and you can see the "Hello World!" text rendered in the Chrome browser.

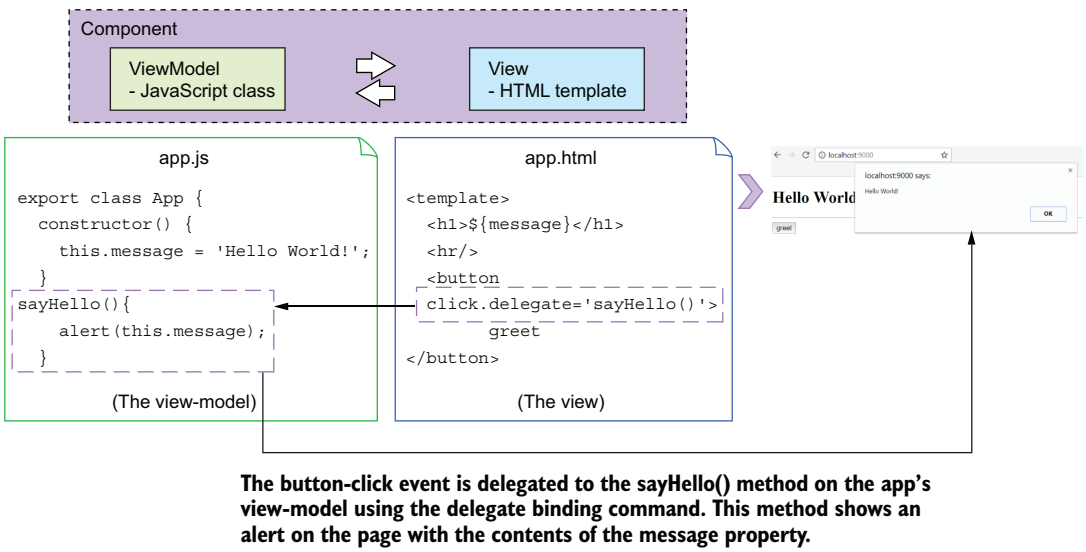

Figure 1.8 The Aurelia `delegate` binding command is used to delegate the click event to the `sayHello` method on the `app.js` view-model. When the button is clicked, the event is raised, and the `sayHello` method is called. This results in a greeting alert being presented to the user.

1.3.3 Routing

One of the tools required in most SPAs today is routing. Routing provides a way of mapping URLs to application routes. The benefit is that you can build an SPA in a way that makes it feel like a real website, so conventions such as the Back button returning to the previous page work like the user expects. As mentioned, taking advantage of a routing tool in your SPA allows you to implement deep linking, which allows users to visit a path deep within your application (for example, /products?id=1) and have the page rendered with the state that they expect (in that case, the product with the given ID). Aurelia lets you configure an array of URLs called *routes*. When you navigate to a route, it looks up the URL entry in the dictionary and finds the view-model that corresponds to this route. It then constructs that view-model on your behalf. But how do you get from that constructed JavaScript class to something that's rendered on the page? Figure 1.9 gives you a vital clue, outlining how a typical routing setup in Aurelia works.

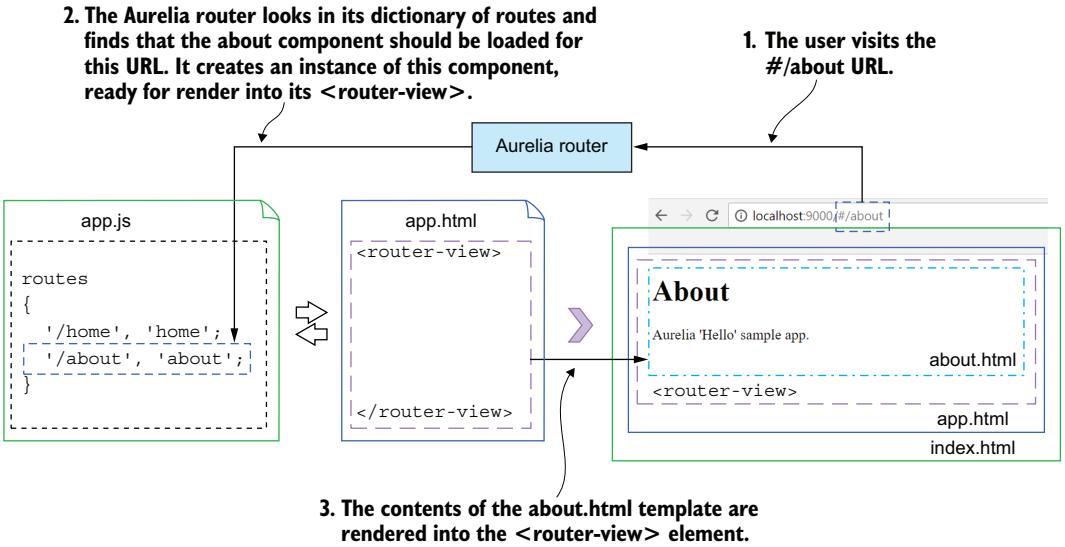

Figure 1.9 Zooming in on the routing component of your Aurelia map, the router works by looking up a URL (in the case of the example, the /about URL) and matching it to a route found in the route dictionary. The route dictionary determines which view-model to load. In this case, you've specified that you want the `about` view-model to load whenever a user visits the #/about URL.

In this example, you've added routing to your `app.js` view-model by configuring a set of URLs (home and about) that corresponds to components in your application. Figure 1.9 consists of two components: `about` and `index`. The router takes care of constructing the view-model for a component. The component is then rendered inside the `<router-view>` element in the `app.html` view.

To get a better understanding of how this process works, bear with me as I take you on a brief diversion that will give you some insights into Aurelia's personality.

Understanding Aurelia's personality provides some clues as to why the framework behaves the way it does. Aurelia is a framework with many strong opinions, held weakly. This means that for any scenario you might face in your application—in this case, picking the view to show in relation to a view-model—Aurelia thinks it should be handled a certain way. But like an open-minded person, Aurelia's opinions are flexible. If you have a different opinion about the framework, you can tell Aurelia, "No, in this case I want you to pick the view to render based on the Fibonacci sequence," and it will do that instead.

These opinions are often called *conventions*. Typically, the default conventions will get you where you need to be about 80% of the time; for the other 20%, you'll need to override them with your own. Your mileage may vary based on how opinionated you are, and how much your opinions diverge from the default.

One of Aurelia's opinions is that naming is important. Programmers typically put a lot of thought into how we name things. First, it appeals to our sense of organization (sometimes to the annoyance of those around us). But beyond that, it allows us to remember where everything is in a project and what it does. Naming conventions also allow our team members, when they first start a project, to make educated guesses about a file's content and have a fighting chance of being correct. Aurelia has the convention that each view-model file should be named the same as the view file but with a different file extension. For example, a view-model class named `App` should live in a file called app.js. The view corresponding to this view-model should then be named `app.html` (see figure 1.10).

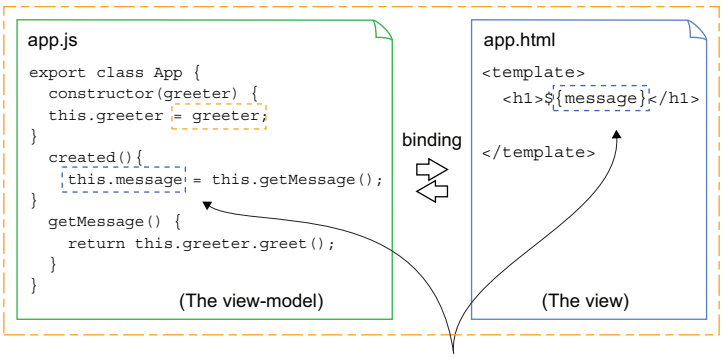

Figure 1.10 Naming the view-model and view files in this way allows Aurelia to bind the view-model and view together by convention.

Aurelia has many other opinions, and we'll look at those later, but for now let's return to the dilemma of how you get from the view-model that the router loaded to the view rendered on the page. One option is to define a property in your view-model to hold

the path of the view that you want to load. Then, when the view-model is loaded, the framework will look up this property, fetch the page, and render it into the DOM. This is a fine approach that's taken by many frameworks, but Aurelia instead uses conventions to create smart defaults regarding which view should be loaded given a specific view-model.

1.3.4 Inside the view-model, and what's this about components?

Now you know that users interact with the Aurelia framework either by navigating to a URL or by interacting with the view, which raises DOM events. In the first case, the appropriate view-model will be loaded and instantiated. In the second case, you're already on the page, so the view-model has been instantiated as a part of the application startup.

In the example application shown in figure 1.8, you've only got one view-model view pair (the app.js and app.html files). This pair is called a *component*. A component in Aurelia can represent a section of the UI (for example, it could be a nav bar or a product list). Components can also be used to encapsulate functionality within your application (for example, formatting a date field for display in the view with a value converter). Examples of these kinds of components include value converters, binding behaviors, and view-engine hooks. We'll cover these kinds of components in chapter 5. Using custom-element components is a way of reducing complexity as your application grows by splitting the application into a set of small, well-defined pieces that do one thing well. Those of you from an OOP background can think of this as another use case of the single-responsibility principle.

The first thing that Aurelia does after the view-model has been initialized is call the view-model constructor. This is the first step in something called the *component lifecycle*.

1.3.5 The Aurelia component lifecycle

As you interact with the components in the Aurelia application, these components go through a lifecycle. The lifecycle includes from when a component is constructed (for example, when you first visit a route that causes a component to be created) and rendered into the DOM to when you navigate away from this route, causing the component to be cleaned up and removed from the DOM.

Aurelia provides hooks into this lifecycle, allowing you to execute behavior relevant at that point in the life of the component. An example of this is the *activate* lifecycle hook. To hook into when a component is attached to the DOM, you can create an attached method on your component's view-model. You can think of this lifecycle like a tour bus, where you let the driver know in advance which attractions you're interested in seeing. The driver will then let you know at certain points in the tour that "We've reached the picnic spot," or "We've reached the scenic lookout destination." Then, when you arrive at a destination, you can decide on the action you want to take. We'll cover the Aurelia component lifecycle and look at each of the lifecycle hooks available in chapter 6 when we cover intercomponent communication with Aurelia.

1.3.6 Loading data from an API

Suppose that when your page loads, you want to retrieve a greeting from a REST API and render it in the app view. To load this data, you can hook into the `created` callback method from the component lifecycle (as shown in figure 1.11), or the `activate` callback method from the router lifecycle in the case of a routed component (covered in chapter 9). Zooming in on the data-retrieval area of your map shows the `app.js` view-model using an external service called app-service to retrieve data from a back-end API using `aurelia-fetch-client`. Figure 1.11 illustrates a typical workflow used to retrieve data from an HTTP API and render it in the DOM.

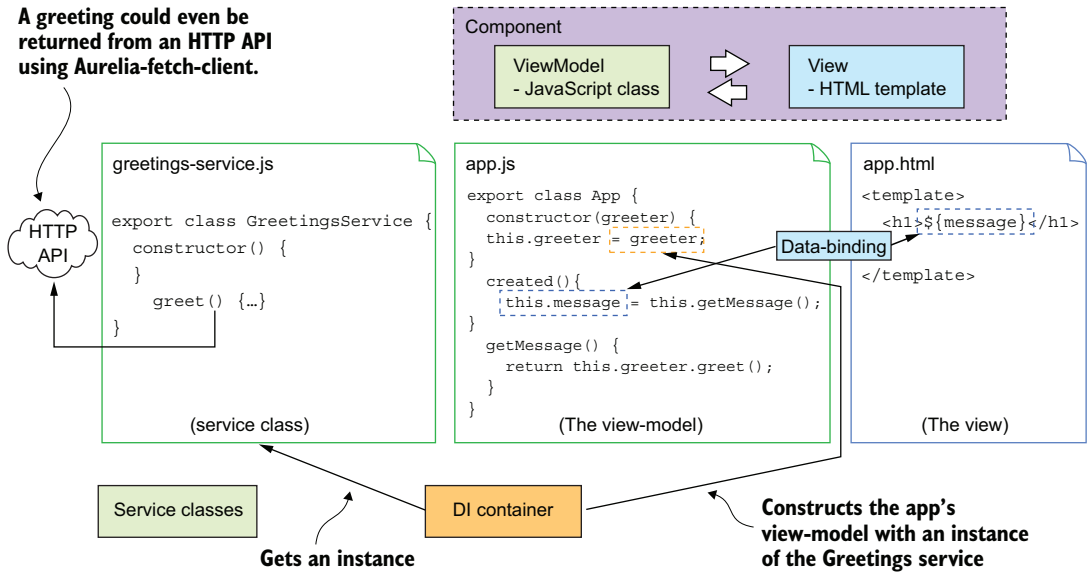

Figure 1.11 The architecture of a sample Hello World application modified to show how a service class could be used to encapsulate functionality such as HTTP API access

The Aurelia DI container is used to get an instance of the `GreetingsService` class and inject this into the app view-model when it constructs it. This service is then used to retrieve a greeting from an HTTP API. The greeting message is data-bound from the `app` view-model to the `app` view, so when your response returns from the HTTP API, it's immediately rendered to the DOM.

Having determined where you're going to implement this logic, how do you go about doing it? One of the most popular choices in Aurelia is via the use of a service class (figure 1.11). There's nothing special about a service class. In fact, it's a plain old JavaScript object (or POJO). Service classes allow you to separate the concern of retrieving data from an API from the view-model concerns of getting data rendered to the screen. In this case, you'd fill out the logic for retrieving the data via AJAX (most

likely via `aurelia-fetch-client`, an HTTP client that makes your life much simpler than the jQuery AJAX `$.get` method, but we'll get into that in more detail later). An instance of this service class can be automatically injected into the view-model using dependency injection.

> **DEFINITION** Traditionally, objects in a system are responsible for managing their own dependencies. This can become challenging as the application grows in scale, increasing the complexity of the relationships between these objects. *Dependency injection* (DI) simplifies this problem by moving the responsibility of creating objects away from the objects themselves and placing it in the hands of the DI framework. This transition of responsibility is known as *inversion of control (IoC)*. Typically, in an application using DI, an object declares which dependencies it requires (often as constructor parameters), and the DI framework provides the relevant implementation of these dependencies at runtime.

1.3.7 Dependency injection with Aurelia

In your simple example view-model, there's only one basic screen (the `app` view), so it will be easy to create a new instance of the class directly in the constructor and do what you need to do. In the real world, however, applications are never that simple. Take Facebook as an example. It has a chat box, a notification widget that tells you how many unread messages you've got, an area that shows you all the ongoing conversations with your friends—and this is only on the chat side of things. If you were to build this in Aurelia, each area would be made up of a set of components. Dependency injection becomes useful when you have multiple components and need to manage dependences (such as service classes) between these components. We'll look at how DI simplifies this process in more detail when we explore intercomponent communication in chapter 6.

1.3.8 Rendering the view

After you've retrieved the data via the service class, you need some way of rendering it back to the screen. One approach that you may be familiar with is to use jQuery. In the world of jQuery, you start by pulling your JSON blob into an array of JavaScript objects. You then have to query the DOM to retrieve the `Table` element object. After that, you cycle through each of the values in the array and add these as rows to the table, but there are two downsides to this approach:

- *Performance*—It's possible to update the DOM efficiently by carefully replacing only the affected DOM branches that correspond to a change in your JavaScript model even with plain JavaScript or jQuery. But because this optimization step is tedious and time-consuming, many developers skip it and instead replace a larger fragment of the DOM than is necessary. Data binding performs this optimization for you, which makes skipping this step a non-issue.

- *Difference of abstraction level*—In an ideal world, when writing application code, you only need to concern yourself with the business problem you're solving. If you're in the flow of solving this problem, and then need to change gears and think instead about the mechanics of getting the relevant state changes reflected in the DOM, it can break you out of this flow. Any break in flow causes a slowdown in development pace and introduces an opportunity for errors to creep in. Imagine if every time you wanted to accelerate your car you needed to think about the process of how the internal combustion engine worked to produce forward motion. Focusing on these details might increase the likelihood of accidents. The accelerator pedal is an abstraction that obviates you needing to think about all of this. All you need to do is push the pedal, and off you go. Aurelia's binding system brings you to a higher level of abstraction, much like the accelerator pedal does when you're driving.

To render the results from your service-class call, save those results into a property bound to an element in the view. Aurelia then takes care of it, first notifying itself that a change has taken place, and then applying the relevant changes to the DOM. This keeps you at the same level of abstraction throughout the process and allows you to focus on the task at hand. It also allows Aurelia to optimize the changes to the DOM. Aurelia has a high-level picture of the changes that need to be made to the DOM. Given this perspective, it's able to find an optimal way to perform these changes to minimize browser rerendering, and so on. (We'll look at Aurelia's DOM optimizations in more depth later in this book.)

We started our virtual tour with user interaction on the page (either a URL navigation event or a DOM event). In the case of the navigation event, you've loaded the appropriate view-model and initiated it via the constructor, which is the first step in its component lifecycle. In the case of the DOM event, you've responded to an event triggered in the UI that was bound to a method in the view-model via data binding. In both cases, you responded to the event by retrieving data from a web API via a service class that was injected into your view-model via dependency injection. Once you received the data back from the service class, you saved it into a property value on the view-model that was data-bound to a property on the view. This caused that part of the view to rerender and display the list of values to the user.

You now have an idea of the kinds of problems that Aurelia solves and, at a high level, how it solves them. But if you're anything like me, you're eager to learn how to put this into practice. In the next chapter, you'll get your hands dirty creating an Aurelia application from the ground up. We'll build a virtual bookshelf SPA and begin by creating the ability to add and list books. By the end of this book, we'll have built a full-fledged SPA with multiple interrelated components, third-party libraries (such as Bootstrap and Font Awesome), and the ability to send and receive data from an HTTP REST API built with Node.js, MongoDB, and Express.js. This will give you an understanding of the variety of tools that Aurelia offers, and by the time you're finished, you'll have everything you need to build your own component-oriented SPA with Aurelia.

Summary

- Certain styles of web applications are difficult to develop using the traditional request/response style of web-application architecture.
- Many of these applications (such as admin portals or messaging applications) would've been built as desktop applications.
- The SPA architecture makes it easier to build this style of application, by providing a set of tools such as data binding and routing.
- Aurelia is an MV* SPA application framework that provides a similar set of tools to other frameworks, such as AngularJS or Ember.js, and is more similar to these frameworks than the SPA libraries, such as React, which are more of a rendering layer.
- Aurelia is the standout choice in today's web-development world due to its focus on clean code, simplicity, and convention over configuration.
- Aurelia applications are built by composing components of view/view-model pairs, where the view is an HTML template and the view-model is a JavaScript class.
- Data binding is used to handle events from the view in the view-model, propagate changes in view-model properties to the view, and propagate changes from the view back to view-models.
- Dependency injection is used to simplify the management of dependencies in Aurelia applications by moving the dependency-management responsibility out of the components themselves and into the DI container.
- Aurelia provides routing to allow developers to build SPAs that feel like real websites, supporting the standard web-interaction patterns that users are familiar with.

WEB DEVELOPMENT

Aurelia IN ACTION

Sean Hunter

Try Aurelia, and you may not go back to your old web framework. Flexible and efficient, Aurelia enforces modern design practices and a modular architecture based on web components. It's perfect for hybrid web + mobile apps, with hot features like dynamic routes, pluggable pipelines, and APIs for nearly every flavor of JavaScript.

Aurelia in Action teaches you how to build extraordinary web applications using the Aurelia framework. You'll immediately take advantage of key elements like web components and decorators when you start to explore the book's running example: a virtual bookshelf. As the app unfolds, you'll dig into templating and data binding the Aurelia way. To complete the project, you'll take on routing and HTTP, along with tuning, securing, and deploying your finished product.

What's Inside

- Templating and data binding
- Communication between components
- Server-side and SPA design techniques
- View composition

Written for developers comfortable with JavaScript and MVC-style web development.

Sean Hunter is a software developer in Melbourne, Australia, with nine years of web-development experience.

"Makes building single-page applications easy and fun."
—Alessandro Campeis, Vimar

"The perfect way to transform your applications into successes."
—Philippe Charrière, Clever Cloud

"All navigators need a map, and this book provides a path into the jungle of production-ready web-development frameworks."
—Joseph Tingsanchali, Netspend

"Learn how to build a single-page app that aims for simplicity, modularity, and convention over configuration."
—Peter Perlepes, Growth

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

ISBN-13: 978-1-61729-478-5
ISBN-10: 1-61729-478-0
54999
9 781617 294785

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]