Angular IN ACTION

Jeremy Wilken

Sample Chapter

MANNING

www.itbook.store/books/9781617293313

Angular in Action

by Jeremy Wilken

Chapter 1

brief contents

1 Angular: a modern web platform

This chapter covers

- Angular as a platform for modern applications
- Key reasons for choosing Angular
- Angular's architecture and how components form the basis of it
- How AngularJS differs from Angular
- ES2015 and TypeScript and how Angular uses them

Angular is a modern web application platform that promises to provide developers with a comprehensive set of tools and capabilities to build large, robust applications. The core value proposition of Angular is to make it possible to build applications that work for nearly any platform—whether mobile, web, or desktop. The Angular team has focused on building much more than a robust application framework; they've also built an entire ecosystem.

All that's a bit of a mouthful, which is partly what makes Angular such an exciting technology to work with. Let's start by taking a closer look at why you would choose Angular for your next project.

1.1 Why choose Angular?

Building web applications that can meet the needs of users is not a trivial task. The quality and complexity of applications is ever increasing, and so are users' expectations for quality and capabilities. Angular exists to help developers deliver applications to meet these demands.

If you haven't settled on Angular as a tool of choice yet, let's quickly cover some of the top reasons that you should seriously consider Angular. Some items are covered more in section 1.3, but here are the top highlights in my experience:

- *Inspired by web standards, enhanced by modern capabilities*—Anyone building web applications today knows there are many different ways and ideas about how to design applications. Angular tries to design its framework and the development process around common standards (like leveraging the latest JavaScript language features), using modern capabilities (such as embracing TypeScript for type enforcement).
- *Development tooling included, customizations available*—Angular provides a common developer experience through its CLI tooling (for generating, building, testing, and deploying apps), while making those same tools available to be easily integrated into custom solutions (such as a custom build toolchain) and third-party tools (like different editors or IDEs).
- *Powerful ecosystem with a large community*—There is an ever-growing number of third-party libraries, UI libraries, blog posts, and events. Angular's large and active community provides a great foundation on which to learn and should instill confidence that it will remain a valuable technology.
- *Sponsored by Google, open source community driven*—Google has a team of engineers, managers, and evangelists solely dedicated to bringing Angular to the rest of Google and the entire web community. With thousands of "internal customers" who rely on Angular inside Google, the Angular team uses those experiences to inform future development and receives large volumes of external contributions that together shape Angular's future (you can join in too!).

Angular is much more than just a JavaScript library that powers some of the top websites in the world. I'm passionate about open source communities, and I'm an advocate for people to get engaged in a project as part of their regular routine. Projects in the Angular community are where I put a lot of my energy and contributions, and I invite you to join me. Although I do engage with the Angular project itself, I primarily contribute to projects in the Angular ecosystem, such as Clarity, a UI component library and design language.

You may be a developer trying to figure out whether Angular will meet your needs, or you may be a manager trying to understand the role of the technology, or trying to figure out how to improve your current applications. Regardless of where you're starting from, the Angular ecosystem has a lot to offer.

1.2 What you'll learn

This book is designed to be a comprehensive walk through Angular, but it's also meant to get you informed about various aspects of the ecosystem. The approach is always experiential, where you'll learn about a topic and build it yourself to see the concepts come to life. At the end of this book you should be able to make high-quality Angular applications and have the foundational knowledge and experience on which to build a career and applications.

The key takeaways in this book include the following:

- *How Angular works*—We'll look at some of the key internal concepts that make it such a compelling platform for building your applications. You'll learn the concepts and build examples to illustrate them as part of a functional application.
- *How to build applications*—In most chapters, we'll walk step-by-step through a number of real-life examples. The code examples are comprehensive and focus on a certain set of goals for each chapter.
- *Learn about the ecosystem*—Each example uses some third-party libraries and capabilities. This helps you see more of a realistic development experience and gain a foundation for building your own applications.
- *Get practical insights from my experiences*—In many of the examples and notes about them, I share practical advice from my experience, including suggestions on things to avoid (even if it's perfectly legitimate code) and how to choose between different approaches when they're provided.

You should be equipped to design and build web applications with Angular by the end of the book. If you're not as interested in the technical aspects (perhaps as a manager), you'll still glean a lot of the same lessons to get a solid frame of reference for how Angular works and what it provides for your project.

There are a few things I won't be able to cover in this book, but just because these items aren't specifically discussed, it doesn't mean you can't learn many things related to them. The following are *not* core topics covered in this book:

- *How to write libraries*—This book focuses on how to build applications with Angular, and in many ways building a library has different guidelines and recommendations. That would be another book. But building a library is also difficult if you don't know how to build an application first.
- *Every available API and features*—Many APIs and features aren't covered in this book, mostly because they're rarely used. I believe this book will empower you to build your skills to the level that you can quickly learn these additional features as your project needs require.
- *How to design your app and UX principles*—This is such a large topic that I can't cover it fully. I've tried to show several different ideas and patterns in the chapter examples to give you some ideas, but it's often opinion-based. I hope you'll take time to compare the design of each and know that there can be limitations as well due to these being examples and not actual projects.

Angular is an evolving project, with new features and sometimes deprecation of existing ones. I've taken great care to ensure that the concepts taught are the core ideas that are unlikely to change (though they may be enhanced). If there are any changes that break some of the example code or concepts, please check the GitHub project for each chapter or the book's forums, which should have a list of known changes and errata.

To better understand the impact of Angular in today's web, let's go back a few years to look at the history that brought us here.

1.3 The journey from AngularJS to Angular

Web applications came of age around 2009–2010, when the Web 2.0 fad finally gave way to better application approaches and frameworks. The term *web application* also became refined, due perhaps in large part to the standardization of HTML5 and EcmaScript 5 (the basis of JavaScript), and focused primarily on the ability to build robust applications that run almost entirely in the browser.

In 2009, Miško Hevery announced AngularJS, which became one of the most popular frameworks (if not *the* most) for building web applications. The AngularJS project was brought into Google, and version 1.0 was officially launched in October 2010. There were many other viable frameworks, but AngularJS struck a chord with a wide audience of developers.

Angular vs. AngularJS

There has been some confusion about Angular versions. The Angular team has decided to provide guidance and call the first version *AngularJS*. That's the name it was given initially, and it separates it architecturally from later versions. Any release from 1.0 through 2.0 is known as AngularJS.

For versions 2.0 and greater, it's known as just *Angular*. Version 2.0 was a complete rewrite, and all versions after it are planned as incremental changes upon it.

Angular version 2, officially announced in September 2014, was developed over the course of two years (plus some time prior to its announcement). It was released as Angular version 2 in September 2016, with Angular 4 being released in March 2017. The Angular team will continue to provide major releases on a six-month schedule, with a focus on easy upgrades. Depending on when you read this, Angular 6, or even 10, could be the most current release.

But you aren't looking at this book to learn about the past—you're interested in building modern web applications. Perhaps you've built Angular 1 applications, or even started with some of the Angular 2 guides. The focus of this book is on building modern web applications, and Angular provides the platform to elegantly accomplish that.

Throughout the book, I'll mention AngularJS occasionally to draw connections for readers who have experience with it, but when I use *Angular* without a number, I'm always referring to Angular version 2 or greater. Check out https://angular.io (figure 1.1) for more info.

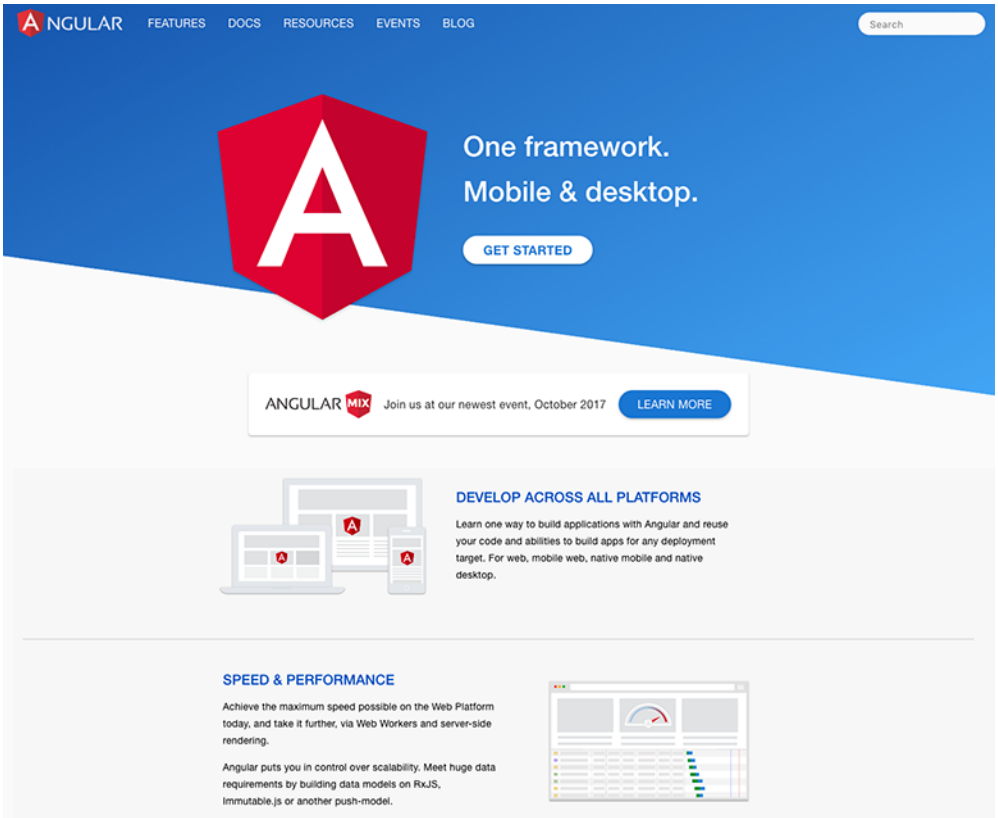


Figure 1.1 The Angular website is a great resource for documentation, events, and everything about Angular.

1.4 Angular: a platform, not a framework

There are a few important distinctions between a framework and a platform. A *framework* is usually just the code library used to build an application, whereas a *platform* is more holistic and includes tooling and support beyond a framework. AngularJS was focused solely on building web applications in the browser and was clearly a framework. It had a large ecosystem of third-party modules that could be easily used to add features to your application, but at the heart of it all, it simply built web applications in the browser.

Angular comes with a leaner core library and makes additional features available as separate packages that can be used as needed. It also has many tools that push it beyond a simple framework, including the following:

- Dedicated CLI for application development, testing, and deployment
- Offline rendering capabilities on many back-end server platforms
- Desktop-, mobile-, and browser-based application execution environments

- Comprehensive UI component libraries, such as Material Design

Some of these things existed in some shape with AngularJS, but most were community solutions and were bolted into AngularJS after the fact. In contrast, Angular was developed with these platform features in mind.

These parts are still in refinement and will continue to evolve into more robust options.

1.4.1 Angular CLI

Modern development typically requires setting up many tools in order to start a project, which has given rise to *more* tools to help manage *those* tools. A typical project needs to manage handling a build process (asset optimization), testing (unit and end-to-end testing), and local development support (local server).

> **The CLI is always improving**
>
> The Angular CLI is a wonderful tool that has an ever-growing list of capabilities. Over time, it will likely do many more things than I mention here, and perhaps the capabilities may change as well.

The Angular CLI (often just referred to as the CLI) is the official toolchain for building Angular applications that provide these features and more. This book uses the CLI for all examples, and you're encouraged to use it for your projects as well. You could roll your own build tooling, but that's suggested only if the CLI doesn't meet your needs.

You can install the CLI using npm. It does require that you have a recent version of NodeJS installed to run properly:

```
npm install -g @angular/cli
```

The CLI has a number of features that aid in the development of Angular apps. Here are the primary features:

- *Generates new project scaffolding*—Instead of having to create a new project from an existing project or creating all the files yourself, the CLI will generate a full project with a basic app already started for you.
- *Generates new application pieces*—Need a new component? Easy; it can generate the files for you. It can generate components, services, routes, and pipes, and it also will automatically ensure they are fully wired up in the build process.
- *Manages the entire build toolchain*—Because files need to be processed before being served to the client (such as TypeScript compilation), the CLI will process your source files and build them into an optimized version for development or production.
- *Serves a localhost development server*—The CLI handles the build flow and then starts a server listening on localhost so you can see the results, with a live reload feature.

- *Incorporates code linting and formatting code*—Helps enforce quality code by using the CLI to lint your code for style and semantic errors, and it can also help format your code automatically to the style rules.
- *Supports running unit and e2e tests*—Tests are vital, so the CLI sets up Karma for running your unit tests and works with Protractor to execute your e2e tests. It will automatically pick up and execute new tests as they're generated.

You can add other features and capabilities to the CLI. To see the full list of features, you can run `ng help` to output the current help documentation. You can also read more about the CLI at https://cli.angular.io.

1.4.2 Server rendering and the compiler

Compiling output in Angular is decoupled from the browser in a way that allows Angular applications to be rendered in different environments, such as a server or desktop app. There are many great side effects of this design pattern, because Angular is much more versatile by being able to render on the client and server, and it opens many different opportunities.

There are two things in play here—first, the decoupled compiler of Angular, and then optional support for universal rendering. It takes a decoupled compiler to enable the universal rendering, because you can implement different rendering patterns depending on the environment.

The compiler in Angular is a very important piece of the puzzle. It's responsible for resolving data bindings, registering event handlers, and rendering out the resulting HTML for components.

The term *server rendering* is about the notion that it shouldn't matter where you run the JavaScript engine that executes Angular code. It should be possible to run Angular universally, such as with browser JavaScript engines, NodeJS, or even less common engines like Java's Nashorn engine. This greatly increases the ways in which Angular can be used.

Why does this matter? Let's explore a few primary use cases:

- *Server rendering for faster loading*—Mobile devices are the primary way to access the internet these days, and mobile connections are frequently slow and unreliable. A server-side rendering option allows you to resolve data bindings and render components on the server so the initial payload sent to the user is pre-initialized. It can also optimize and send the necessary bytes for a quick initial load time and lazy load the other assets as needed.
- *Performance in the browser*—One of the major pain points of JavaScript is that it's single threaded, which means that JavaScript can only handle one instruction at a time. In modern browsers, a newer technology known as web workers allows Angular to push some of the execution of the compiler into another process. This means that a lot more processing can occur, and it allows things like animations and user interactions to be smoother.

- *SEO*—There's a major concern about how heavy JavaScript applications are crawled by search engines. Universal rendering means we can detect crawlers and render the site for them so that content is ready without having to worry if the crawler executes JavaScript (some do, some don't). This will certainly enhance SEO efforts for Angular applications.
- *Multiple platforms*—Many developers want to use other platforms for their back ends, such as .NET or PHP. Angular can be compiled in the platform of choice, assuming there's a supported renderer. Angular will provide support for NodeJS, but the community is actively building and maintaining rendering support for other platforms such as Java and Go.

All of these have been issues for years in building web applications, and Angular provides a comprehensive solution. The great thing is you don't have to do a lot of work to enable these features in your application.

This is an area of evolution at the time of writing, and setting it up correctly is an advanced topic that I can't cover in depth. But the Angular documentation and CLI are being constantly improved to show you how to incorporate these types of benefits easily.

1.4.3 Mobile and desktop capabilities

The rendering capabilities enable Angular to work with native mobile and desktop applications. Tools like Cordova have been around for a while; they let you create *hybrid* applications—web applications wrapped up inside some type of native shell. But Angular's rendering design makes it possible to support rendering out to different native platforms entirely.

The major value is that you can share a lot of code between your Angular applications, even if some are designed to build mobile apps and others are web applications. This is particularly valuable in large teams.

The mobile and desktop capabilities of Angular are extensions of the design of the compiler. The following tools are all outside of Angular's core but use the design of Angular to power some powerful design patterns:

- *Ionic (mobile)*—This fantastic and popular hybrid app framework (figure 1.2) for Angular JS has been updated to work with Angular. Millions of mobile apps have been created with Ionic, and it's primarily focused on building hybrid apps. The UI components are all created to run in the browser, but look and feel like native UI components.
- *NativeScript (mobile)*—This is another popular mobile framework that creates native mobile apps. NativeScript implements the native UI components but allows you to write Angular components to describe your application.
- *React Native (mobile, desktop)*—By the name, you'd be correct to assume that React Native is really part of the React framework ecosystem. But with a custom render, it's possible to use the React Native tool to generate native mobile apps.

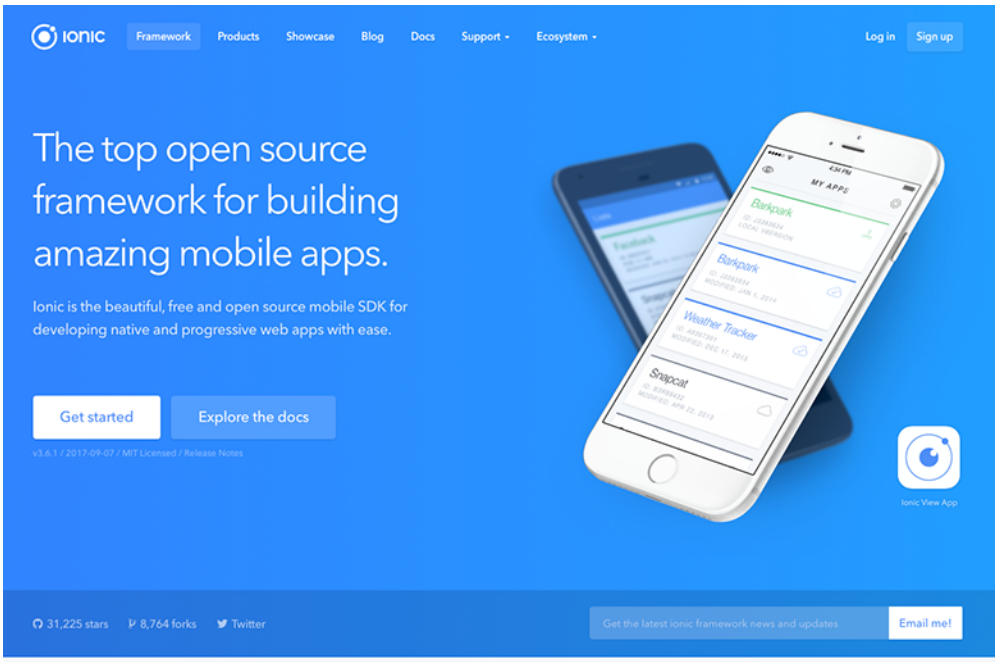


Figure 1.2 Ionic is a popular and powerful mobile framework for Angular.

- *Windows Universal (desktop)*—Windows has support for building native Windows applications using JavaScript. It's possible to use Angular as your application layer but still have to build out a native Windows application.
- *Electron (desktop)*—Based on NodeJS, Electron is a very popular cross-platform application framework. It implements a set of APIs to hook into the native OS, and you can leverage Angular to power the internal logic of your app.
- *Progressive Web Apps (mobile, desktop)*—The capabilities of Progressive Web Apps (PWAs) aren't limited to Angular. They're fundamentally about blurring the line between the web and native. As of this writing, they're in experimental support. This is an exciting potential avenue for building applications of tomorrow.

These different options support the power of the decoupled compiler in Angular. It also means that there will likely be many, many more examples and use cases that allow you to build Angular applications that can run nearly anywhere.

1.4.4 UI libraries

There's an ever-growing catalog of UI libraries built for Angular. They bring different sets of UI components to developers for easy consumption. Rather than having to build your own charts or tabs components, you can use one of the many prebuilt options.

Depending on your team's size and skill sets, implementing your own UI components may be challenging. Making truly reusable and hardened UI components is difficult. These components are rarely what make your application really unique, so it's hard to spend the time (and money) to build them.

These libraries are plentiful. There are so many that I can't cover all the options. You'll notice there's a lot of overlap in the functionalities that each of them provides, so comparing them can be difficult. We'll take a look at some of the most popular options, but I recommend doing additional research before selecting an option:

- *Angular Material (https://github.com/angular/material2)*—Material Design is the official design specification created by Google. It has strong roots in concepts of real-world objects, hence Material in the name. Angular Material is the official UI component library provided by the Angular team and it implements a number of UI components according to the design specification. It has an open source license.
- *Covalent (https://teradata.github.io/covalent)*—This library extends the Angular Material project with a number of additional components and capabilities, but still retains the principles of Material Design. It's a result of work done at Teradata. It has an open source license.
- *Clarity (https://vmware.github.io/clarity)*—This library, shown in figure 1.3, comes from VMware. It's designed as both a library and a design specification for web applications. It contains many components that are specific to Angular but also has some icons and a general CSS framework. It has an open source license.
- *ng-bootstrap (https://ng-bootstrap.github.io)*—Based on the very popular Bootstrap CSS framework, ng-bootstrap implements the components based on the design of Bootstrap. It's built by the same team that created the very popular AngularJS UI Bootstrap project. It has an open source license.
- *Kendo UI (https://www.telerik.com/kendo-angular-ui/)*—From the same company as NativeScript, Kendo UI is a UI library that's been integrated into many different frameworks, but the company is building a set of native Angular UI components that are custom for Angular. It has a commercial license.
- *PrimeNG (www.primefaces.org/primeng/)*—A rich collection of UI components, PrimeNG is developed by PrimeTek and has more than 60 components. It comes with many themes and is designed for mobile and desktops. It has an open source license.
- *Wijmo (http://wijmo.com/angular2/)*—Containing some very complex data grid components, Wijmo implements this set of Angular components without support from other libraries like jQuery. The UI library has a commercial license.

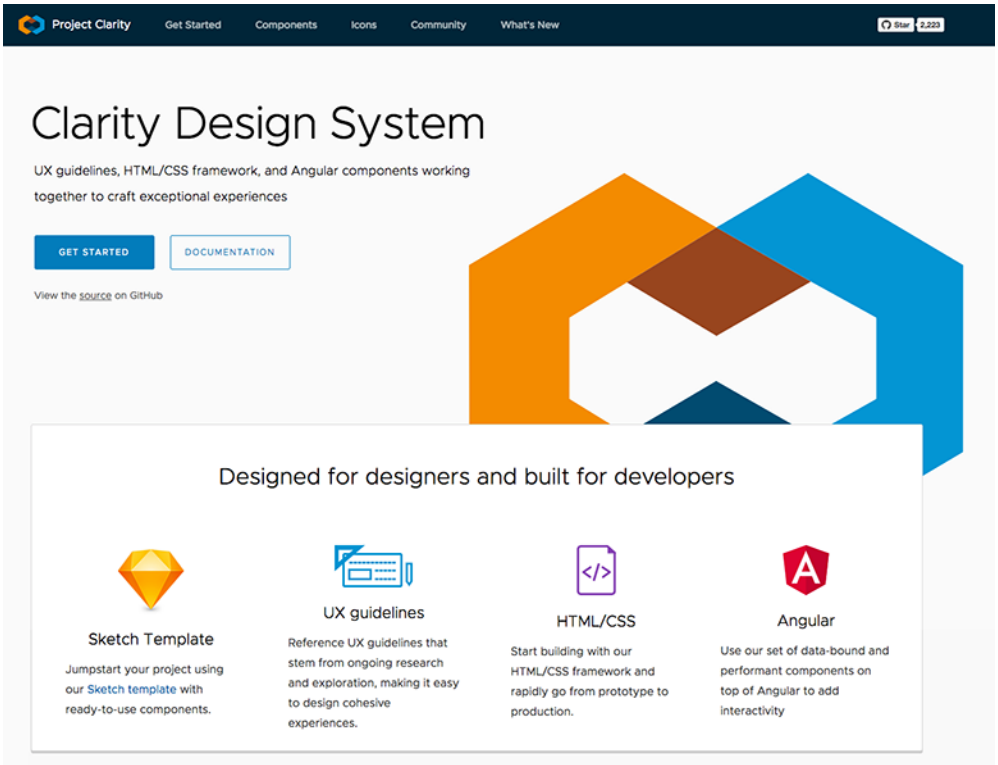


Figure 1.3 Clarity Design System is one of the most popular Angular UI libraries.

- *Ionic (http://ionic.io)*—Primarily for mobile, Ionic is a comprehensive library of components with easy theming, native device integrations, practical services, and its own CLI for app development workflows. The company also provides commercial services for mobile app development. It has an open source license.
- *Fuel-UI (http://fuelinteractive.github.io/fuel-ui/)*—Another Bootstrap CSS framework-based set of components, directives, and pipes by Fuel Travel. It has an open source license.

You're certainly not required to use a UI library, but most developers will find them to be useful. Any reasonable UI library should be fairly well tested, allowing you to focus more on what makes your application unique.

1.5 Component architecture

Many modern applications have adopted a component-based approach to developing applications. The intention is to design each piece of your application in a standalone manner that limits the amount of coupling and duplication across various parts of the program. In many ways, a *component* is a way to create custom HTML elements in your application.

The easiest way to think about a component architecture is to look at an example of a page with a large number of discrete parts and inspect how the various parts relate to one another. Figure 1.4 shows an example from a future chapter and visually breaks down the various component parts.

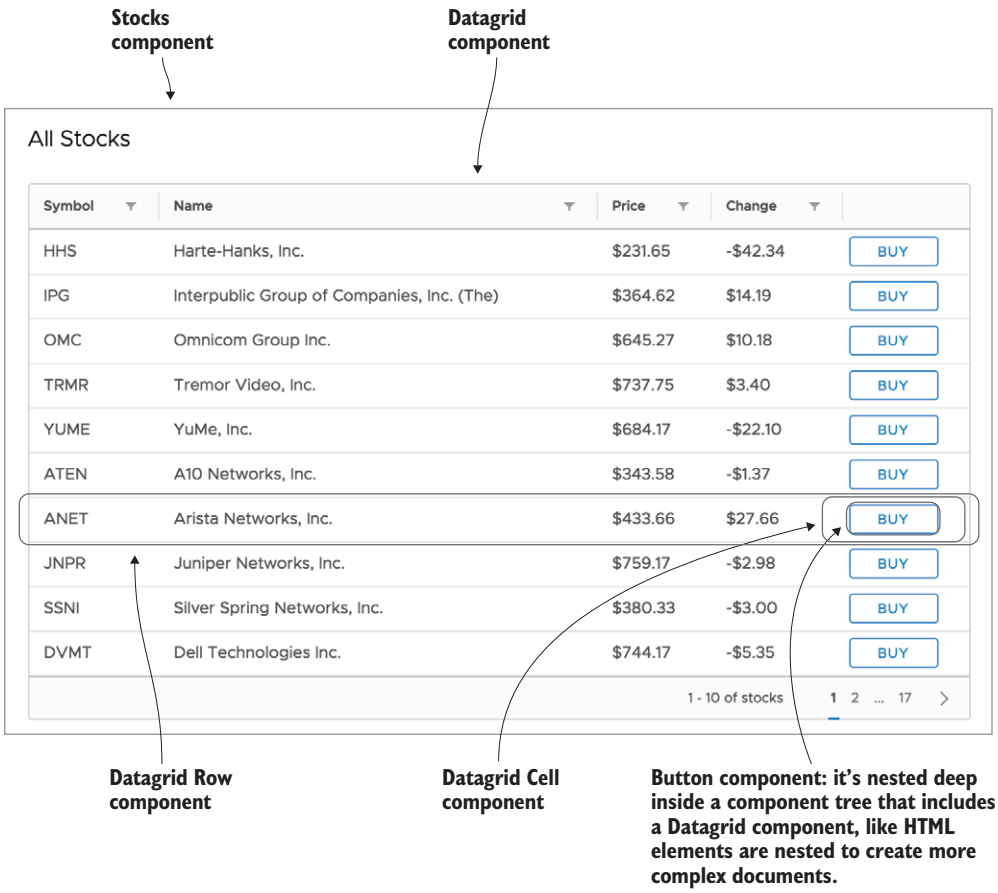


Figure 1.4 Component architecture illustrated by showing how components are nested and combined to create more complex layouts

The figure shows an isolated section from one of the book chapter examples, illustrating that several components combine to create this display. You can see that various parts are independent from the others, but they also work together to create the list of items. There's clearly a hierarchy between them. The list of components on the right shows the parent-to-child relationship each of the components has with the others, and this is essentially how HTML elements work together on the page.

HTML itself is a language of components. Each element has a certain role and functionality, and they're all easily nested to create more complex functionality. They're isolated but still easily manipulated to do whatever is needed at the moment. Some

elements work in tandem. For example, INPUTs are used inside of a FORM to describe a set of input controls. Many elements can also emit events when things happen; a FORM can emit an event when the form is submitted, for example. This allows you to wire up additional logic to manipulate HTML elements based on the events that fire—the fundamentals of front-end application development.

Hopefully a component architecture seems fairly approachable and consistent with your current understanding of the web. The intention is to focus on breaking down individual parts of the application (particularly the visual UI elements) into discrete, modular components.

There are many ways to implement a component architecture, as evidenced by the many web application libraries such as React and Ember. Angular has a very obvious component-based architecture (all Angular applications are components). React and Ember also have first-class support for components in their applications. Those with jQuery experience can also imagine that jQuery plugins can be conceptually similar to components, though they're not as consistent or regulated. Even the basic concepts of the Web 2.0 days (think widgets!) are based around building components.

1.5.1 Components' key characteristics

Components have some concepts that drive their design and architecture. This section will explore these concepts in more detail, but also keep an eye open for how Angular applies these concepts to practice throughout the book:

- *Encapsulation*—Keeping component logic in a single place
- *Isolation*—Keeping component internals hidden from external actors
- *Reusability*—Allowing component reuse with minimal effort
- *Evented*—Emitting events during the lifecycle of the component
- *Customizable*—Making it possible to style and extend the component
- *Declarative*—Using a component with simple declarative markup

When we build components, the preceding are the tenets we should consider when designing the best components possible. These concepts have existed in various forms before, but rarely have they all been clearly implemented and standardized into the web platform.

The World Wide Web Consortium (W3C), the primary standards body for the web, is developing an official Web Component specification. Several standards are required in order to implement the full vision of web components:

- Custom elements (encapsulation, declarative, reusability, evented)
- Shadow DOM (isolation, encapsulation, customizable)
- Templates (encapsulation, isolation)
- JavaScript modules (encapsulation, isolation, reusability)

As of this writing, the specification isn't fully adopted in all browsers and possibly never will be. Standards are also subject to change, but it's not crucial that we dive into the

specifics of the specification here. The important thing is that these four concepts are central to the idea of components. Let's explore them a little more in detail and see how they enable a component architecture.

CUSTOM ELEMENTS

HTML is the language of the web because it describes the content of a page in a fairly concise set of elements. As a markup language, it's a declarative way to describe your content. *Custom elements* mean being able to extend HTML with our own additional elements, adding to the vocabulary of what is possible. You can read about the official specification at www.w3.org/TR/custom-elements/.

The official specification for custom elements is intended to allow developers to create new HTML elements that essentially blend naturally and natively into the DOM. In other words, using a custom element should be no different from using any other HTML element. For example, imagine you want to create a custom element that implements a tabbing interface. You would likely want to create custom elements like the following code and in figure 1.5:

```
<tabs>
  <tab title="About">
    <h1>This is the about tab</h1>
  </tab>
  <tab title="Profile">
    <h2>This is the profile tab</h2>
  </tab>
  <tab title="Contact Us">
    <form>
      <textarea name="message"></textarea>
      <input type="submit" value="Send">
  </tab>
</tabs>
```

This looks and feels like natural HTML because these would be two custom elements: `tabs` and `tab` elements. The real value here is how easy it is to implement tabs. Using jQuery, you would end up creating a lot of `div` elements, applying a number of custom IDs or classes, and sprinkling some JavaScript on top.

These tabs could also emit events. For example, anytime the active tab changes there could be a `tabChange` event. Anything in your application could then listen for this event and act accordingly. Each custom element could implement any number of events that seem practical to the lifecycle of the component.

A custom element can also implement its own styling, so the tabs can come by default with a particular look and feel. Anyone using the tabs could write their own CSS to modify it to their particular use case, but custom elements can have a default appearance much like many HTML elements.

Custom elements have a lot of the stuff necessary for building components. In fact, we could stop with custom elements and be fairly happy. It gives us a *declarative* way to create a *reusable* component, which *encapsulates* the internal mechanics of the component away from the rest of the application, but can emit *events* to enable other components to hook into the lifecycle. Angular uses these concepts in its implementation of components.

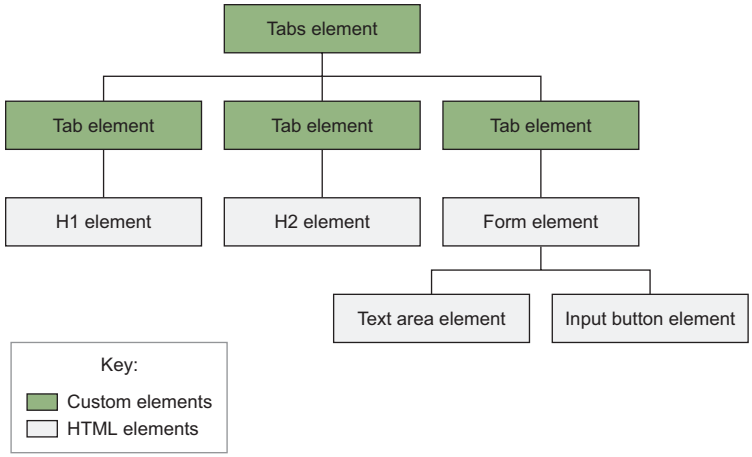


Figure 1.5 Custom elements fit into a normal HTML hierarchy but can implement new behaviors.

Angular provides its own mechanics to create a custom element, which is just an Angular component. Every Angular component is a custom element and fulfills the four tenets (and more) that we expect to get from a custom element.

1.5.2 Shadow DOM

Despite the rather ominous-sounding name, the *Shadow DOM* is really your best friend when it comes to trying to isolate styling behaviors inside of a component. The Shadow DOM is an isolated Document Object Model (DOM) tree that's detached from the typical CSS inheritance, allowing you to create a barrier between markup inside and outside of the Shadow DOM. For example, if you have a button inside of a Shadow DOM and a button outside, any CSS for the button written outside the Shadow DOM won't affect the button inside it. This is important for Angular because it allows us to have better control over how CSS affects the way the components display.

CSS is a powerful language, but most web developers have run into issues where CSS styles have accidentally modified elements other than the intended ones, particularly when adding CSS from external sources. Shadow DOM provides a way to truly encapsulate your component HTML and CSS from other parts of the page, which is known as Light DOM. You can read about the official specification at www.w3.org/TR/shadow-dom.

Developers should be familiar with the standard Light DOM, defining the standard DOM behaviors with regard to element styling and visibility. When you write a CSS rule, the CSS selector is the only way to limit which elements receive that particular styling. Outside of some fairly small, hand-crafted web pages, most CSS is written with some kind of systematic approach to set clear rules about how CSS styles get applied. This gave rise to many of the great CSS grid and component frameworks, such as Bootstrap and Foundation. It also gave us a selection of CSS selector nomenclatures, such as Scalable Modular Architecture for CSS (SMACSS) and Block Element Modifier (BEM). Although we've found ways to manage the Light DOM with these systems, it doesn't change the

underlying behavior that someone could still manage to break your whole application by adding a single rule that doesn't adhere to the guidelines.

There's always been pain associated with scaling page styling with CSS due to the greedy nature of CSS selectors always trying to match as many things as possible. In contrast with the Light DOM, the Shadow DOM gives us the ability to denote that a fragment of the DOM be shifted into a new realm that doesn't play with the Light DOM styles.

In many science fiction stories, characters may get caught somehow in a new dimension of reality that is separated from normal reality, and they're usually unable to interact between these realities except through some "bridge" between the realities. Similarly, I like to think of using Shadow DOM as like shifting the current context to a new dimension that has very limited connection to the Light DOM and therefore allows us to write CSS and HTML that gets rendered without having the ability to modify other styles.

Developers can create a new Shadow DOM (known as a *shadow root*) that will carve out an isolated DOM tree that has limited interaction with the Light DOM. You still attach this root inside the DOM tree as a node. The *shadow boundary* is the line between the Light and Shadow DOMs. There are many nuances and features that enable certain forms of styles to target inside or outside of the boundary, but I'll leave those details for you to dive into if they become needed.

In figure 1.6, you can see a simple example where the first line of text output in the middle of the image has the black background and white text, whereas the second line of text (which is inside the shadow root) doesn't.

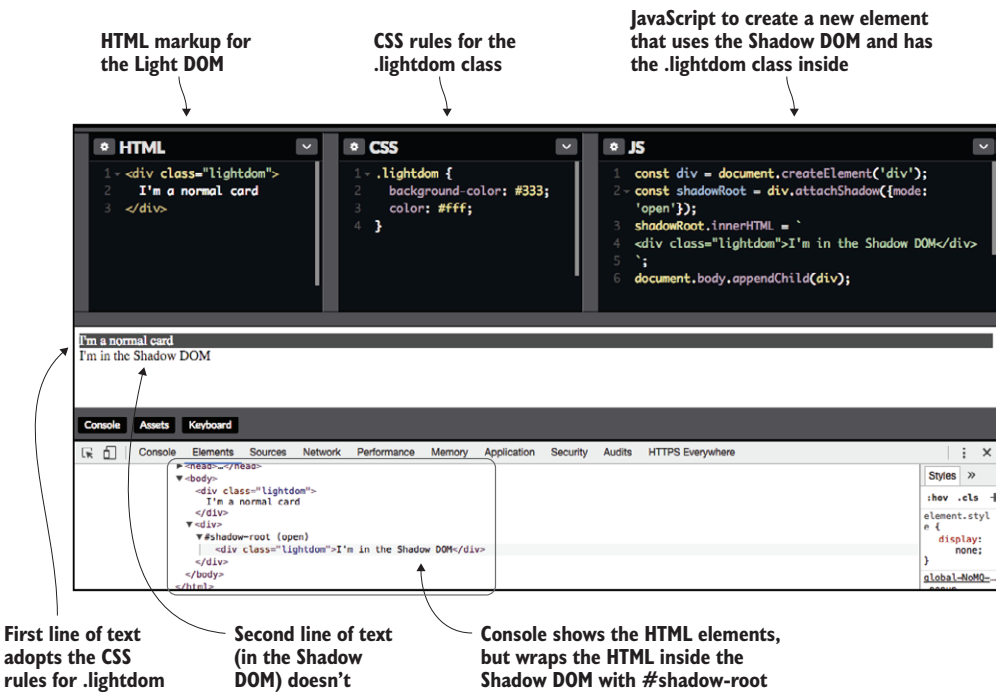


Figure 1.6 Shadow DOM example where the styles from outside the shadow root don't cross the boundary and apply to inner elements

Shadow DOM enables the best form of encapsulation available in the browser for styles and templates. It's able to isolate the internals of a component in such a way that outside styles and scripts won't accidentally attach and modify it. It does provide some customization features that allow you to communicate across the shadow boundary. These are particularly important features when we want to build out complex and reusable components that can be entirely self-contained with styling.

Unfortunately, Shadow DOM support may not be available in all browsers and may require a polyfill. Chapter 4 explores this in more detail, but Angular lets us write components that use either the Shadow DOM, an emulated version of the Shadow DOM, or just the Light DOM.

1.5.3 Templates

Templates are a powerful feature that allow us to create isolated fragments of the DOM to use in our components. Our custom elements need to have some kind of internal structure, and often we'll need to be able to reuse this markup. Ideally this shouldn't clutter the main document, and HTML5 introduces a new `template` tag to help us out. You can read the spec at https://www.w3.org/TR/html5/semantics-scripting.html#the-template-element.

Any markup written inside a template is just a fragment that's not part of the current page unless it's explicitly initialized. In other words, if you were to look at the DOM tree, the content in templates doesn't appear. If your markup has CSS, inline scripts, image elements, or other elements that typically trigger a browser action, those actions won't run until the template is used.

Templates are often used with the Shadow DOM because it allows you to define the template and then inject it into the shadow root. Without templates, the Shadow DOM APIs would require us to inject content node by node. They're also used by Angular as part of the lifecycle of components and the compilation process, allowing Angular to keep isolated, inert copies of the template as data changes and needs to be recompiled.

The role of templates folds in nicely with the overall component architecture and works in tandem with the Shadow DOM and custom elements. They provide a layer of encapsulation that lets you define a template that remains inactive until it's needed and therefore isolates the template from the rest of the application.

1.5.4 JavaScript modules

Neither HTML nor JavaScript has traditionally had a native means to load additional files or assets during the lifecycle of the application. You had to ensure that all the needed files were loaded on page load, or use some workaround that usually relied on making an XHR request or adding a new script tag to the page. Though these approaches worked, they weren't particularly elegant or always easy to use.

Today we have modules and module loaders in JavaScript, which give a native way to load and execute code throughout the entire lifecycle of the app, not just on page load. Previously, developers had to build a bundle of all the assets for the web application ahead of time and deliver the whole package to the user. Modules (figure 1.7) give us a lot of interesting capabilities, many of which are familiar to developers who have worked with other languages with package or module capabilities, like Java, Python, or Go.

Figure 1.7 Angular provides modules (like animation) that contain all the services and objects you'll need to build your applications, but first you'll have to import them.

Inherently, modules aren't strictly a component technology. Modules are an isolated piece of JavaScript that can be used to generate a component, create a reusable service, or do anything else JavaScript can do. They're fundamentally a way to encapsulate application code and choose what's available for the other parts of the application to use.

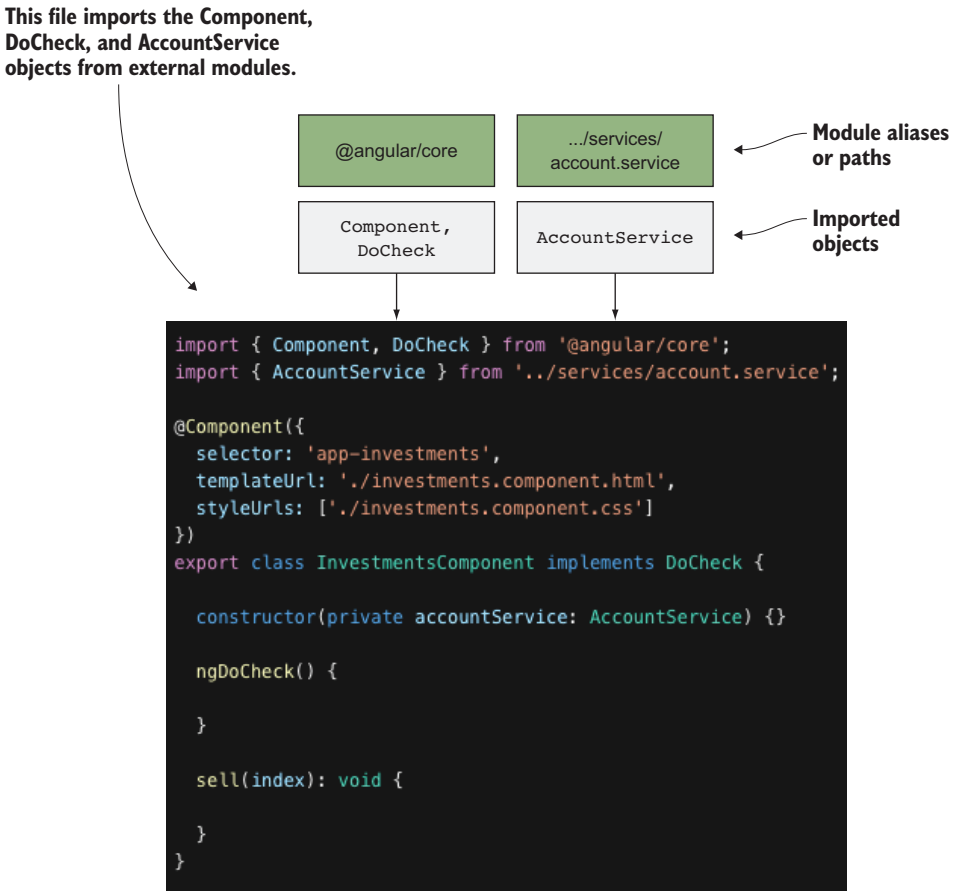


Figure 1.8 Loading objects into a file from different modules using imports

In JavaScript, a module is any file of JavaScript code that contains the `export` keyword. Modules export values that they want to expose to the application and can keep other parts of the internal logic private. Then, in order to use an exported value, you have to first import it from another module (figure 1.8).

In figure 1.8 (a snippet from a later chapter), we're first importing some things from external modules that the rest of the code in this file depends on. The `Component` and `DoCheck` objects are being imported from the @angular/core package (which is part of our node modules directory), and `AccountService` is being imported based on the file path provided.

These modules are powerful because they *encapsulate* the contents of a single JavaScript file into a single coherent whole. They *isolate* the code and allow the developer to conditionally export values to share. They also support *reusability* by defining common mechanics for sharing values in a JavaScript application that previously could only be done by putting values directly on the global scope or by crafting some non-standard service to manage dependency injection, as Angular 1 did.

HTML imports are a similar concept that has been proposed as part of the HTML spec, which would provide similar capabilities. But it's likely that HTML imports won't be adopted, and instead JavaScript modules are used. There are libraries that use HTML imports, such as Polymer, by using a polyfill library.

Angular itself is built entirely around the notion of modules. The source code uses them extensively. When you write your own applications, it's recommended that you also use them. Executing an Angular application is fundamentally loading a module that contains the application bootstrapping logic, which in turn starts to load and trigger additional modules. It's possible to write your Angular applications without modules using ES5 syntax, which is not recommended but discussed next.

1.6 Modern JavaScript and Angular

Angular is designed to take advantage of many features that are fairly recent to the web platform. Most of these became part of the JavaScript specification in 2015 with the release of ES2015 (also known as ES6, but I'll refer to its official name ES2015); other features are still in development as of this writing but are likely to be adopted in a future version.

These features are well covered in many places, so I won't go into detail. Though they could be used with AngularJS, Angular was designed to work using these capabilities. I'll cover some of the most important aspects quickly, namely the following:

- Classes
- Decorators
- Modules
- Template literals

Let's look at an example with all these features working together and then review how they're put together. The following listing is a functional but simple Angular component, and you'll get to see many more examples that use the same concepts in more complex ways in this book.

Listing 1.1 Modern JavaScript Syntax

```
import {Component} from '@angular/core';

@Component({
  selector: 'my-component',
  template: `
<div>
  <h4>{{title}}</h4>
</div>
`
})
export class MyComponent {
  constructor() {
    this.title = 'My Component';
  }
}
```

Imports the Component object from another module

Uses a decorator to add metadata to the MyComponent object

Uses a template literal string to write inline HTML

Exports the MyComponent object, which was defined as a class

Let's start from the bottom and go from there. In ES2015, classes were introduced as a new way to define an object, which is in fact a function. Classes are used to create components, directives, pipes, and services, though they can be used in other ways as well. Using the `class` keyword, the class `MyComponent` is created and is an object that has a property called `title`.

Classes are syntactic sugar for creating objects in JavaScript. They don't introduce a new type of inheritance to JavaScript, which is important to remember. Developers familiar with class objects in other languages may accidentally carry over concepts into JavaScript, but in this case the concept of a class doesn't change the way prototypical inheritance works with JavaScript.

Inside of the class there's a special method called `constructor()`. It's executed immediately when a new copy of the object is created. As long as you name a method `constructor()`, it will be used during creation.

Classes are also useful because they help ensure that the keyword `this` references the object itself. The keyword `this` is a common barrier in JavaScript, and classes help ensure that it behaves more consistently.

The `export` keyword denotes the file as a module. Any module is isolated into a private space, and unless a value is exported, it won't be available for another file or module to use. This breaks away from the global scope that JavaScript has for values and provides a proper separation between modules. Because the `MyComponent` class is exported, it can be imported into another module (not shown here).

At the top of the file, the `import` statement imports the `Component` value from the `angular/core` module, which allows it to be used in this module.

Then in the middle we use the `@Component` decorator, which is a way to add metadata to the class. Decorators always start with the @ symbol, and Angular uses these decorators to understand what type of class has been declared. In this case, it's a component, and Angular will know how to render a component based on this decorator. There are several other ones, such as `Injectable` and `Pipe`, and we'll see those in action later.

Finally, the decorator accepts an object that contains the metadata associated with the component itself. In this example, it has two properties for the selector and an inline HTML template. The decorators define what properties can be passed here, but they allow you to customize the way the class is handled by Angular.

1.6.1 Observables

In addition to new syntax, *observables* are a newer pattern for JavaScript applications to manage asynchronous activities. They're also a draft for a feature to be natively implemented in the JavaScript language so it has weight behind the pattern. RxJS is the library we'll use to help us implement observables in our applications.

Promises are another construct to help deal with asynchronous calls, which are useful for making API requests, for example. Promises have a major limitation in that they're only useful for one call cycle. For example, if you wanted to have a promise return a value on an event like a user click, that promise would resolve on the first click. But you

might be interested in handling *every* user click action. Normally, you'd use an event listener for this, and that allows you to handle events over time. This is an important distinction: Observables are like event handlers in that they continue to process data over time and allow you to continuously handle that stream of data.

Reactive programming is the higher-level name for what observables provide, which is a pattern for dealing with asynchronous data streams. Many things in a web application are asynchronous data streams, if you think about it. A user typing keystrokes into a form input is really a stream of individual characters. Timers and intervals generate a stream of activity over time. Websockets pass data as a stream over time. It's that simple, but the challenge can be wrapping your mind around it all.

Angular uses observable patterns often, and having a grasp of the fundamentals is useful. During the course of this book, you'll see observables in a number of places, and they all work in the same basic way. We're not going to worry about constructing observables here. Instead we'll just focus on how to use them when they're given to you.

To use observables, you subscribe to the stream of data and pass a function that will run every time there's a new piece of data. We'll see this in action in chapter 2 when we make an HTTP request, but let's look at a quick sample just to see some syntax:

```
this.http.get('/api/user').subscribe(user => {
    // Do something with the user record
}, (error) => {
    // Handle the error
})
```

This snippet is using the HTTP library to make a `get` request, which returns an observable. Then we subscribe to that observable, and our callback function fires when the data is returned or the error is handled. It's not very different from a promise, except that an observable could continue to send data. Let's take a different example:

```
this.keyboardService.keypress().subscribe(key => {
    // Do something with the key record
}, (error) => {
    // Handle the error
})
```

In this example, imagine `keyboardService.keypress()` returns an observable, and it emits details about what key was pressed. This is like an event listener, except that it comes in a stream.

Another interesting capability of observables is that they are composable into many combinations. Observables can be combined, flattened into one, filtered, and more. We'll see one example in chapter 9, where we'll combine two observable streams and handle the data they emit in one place. We'll not use many of the more complex features in this book, but you'll likely be interested in how they work, so I recommend the book *RxJS in Action* (www.manning.com/books/rxjs-in-action).

1.7 TypeScript and Angular

Angular itself it written with TypeScript, which is a superset of JavaScript that introduces the ability to enforce typing information. It can be used with any version of JavaScript, so you can use it with anything ES3 (that's not a typo) or newer.

The basic value proposition of TypeScript is it can force restrictions on what types of values variables hold. For example, a variable may only hold a number or it may hold an array of strings. JavaScript has types (don't let anyone tell you otherwise!), but variables aren't typed, so you can store any type of value in any variable. This also gave birth to the many types of comparison operators, such as == for loose equality or === for strict equality.

TypeScript can help catch many simple syntax errors before they affect your application. Sometimes you can write valid JavaScript, but the real world shows that valid syntax doesn't always mean valid behavior. Take this example:

```
var bill = 20;
var tip = document.getElementById('tip').value; // Contains '5'
console.log(bill + tip); // 205
```

This snippet shows a simple tip calculator example where you take the value from an input element and add it to the bill to get the total payment amount. The problem here is that the `tip` variable is actually a string (because it's text input). Adding a number and a string together is perhaps one of the most common pitfalls for new JavaScript developers, though it can happen to anyone. If you used TypeScript to enforce types, this code could be written to alert about this common error:

```
var bill: number = 20;
var tip: number = document.getElementById('tip').value; // 5, error!
var total: number = bill + tip; // error!
```

Here we're using TypeScript to declare that all these variables must each hold a number value by using `:number`. This is a simple syntax that sits inside of JavaScript to tell TypeScript what type of value the variable should hold. The `tip` value will error because it's being assigned a string, and then the total value will error because it attempts to add a number and a string type, which results in a string.

This may seem like an obvious error to a seasoned JavaScript developer, but how often do you have new developers work on your code base? How often do you refactor your code? Can you still ensure that your application is passing around the same value types as you continue to maintain the application? Without TypeScript, you're responsible for doing a strict comparator check of every value before it's used.

Many developers wonder why they should bother learning and using TypeScript. Here are the primary reasons to use TypeScript, in my humble opinion:

- *Adds clarity to your code*—Variables that have types are easier to understand, because other developers (or yourself in six months) don't have to think very hard about what the variable should be.

- *Enables a smarter editor*—When you use TypeScript with a supported editor, you'll get automatic IntelliSense support for your code. As you write, the editor can suggest known variables or functions and tell you what type of value it expects.
- *Catches errors before you run code*—TypeScript will catch syntax errors before you run the code in the browser, helping to reduce the feedback loop when you write invalid code.
- *Entirely optional*—You can use types when you want, and optionally leave them out where it doesn't matter.

I hope you're sold on the value of TypeScript. If not, I hope you'll give it a closer look during the course of the book. This book uses it in examples to help provide more clarity and to help further demonstrate the power of TypeScript. I'll try to provide additional insight into TypeScript features and functionality as we use features in the examples, but you can always learn all there is to know at www.typescriptlang.org/docs/tutorial.html.

Even if you choose not to use TypeScript for type enforcement in your application, you can use TypeScript to compile your application. Because the Angular CLI already uses TypeScript internally, you may be using it without even knowing. If you decide to build your own build tooling, TypeScript is still a worthwhile compiler option.

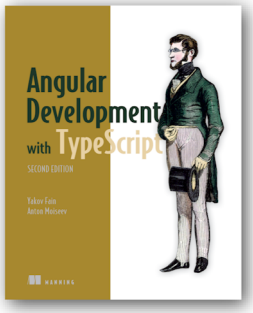
If you're wondering whether using TypeScript in your Angular application is required, the answer is technically no. There are ways to write your application in vanilla JavaScript and avoid TypeScript, to a certain degree. But it's intentionally not documented because there are simply too many features of Angular that don't work unless you use TypeScript. If you're afraid it will be hard to learn, don't be. It's straightforward, and in several places throughout the book I'll explain some nuances of TypeScript that you may not have seen before.

Summary

This chapter introduced you to Angular as a development platform, not just an application framework. There are so many features and capabilities with Angular. Here's a quick summary:

- Angular is a platform, with many key elements such as tooling, UI libraries, and testing built in or easily incorporated into your application projects.
- Applications are essentially combinations of components. These components build upon the core principles of encapsulation, isolation, and reusability, which should have events, be customizable, and be declarative.
- ES6 and TypeScript provide a lot of the underpinnings for Angular's architecture and syntax, making it a powerful framework without having to build a lot of custom language capabilities.

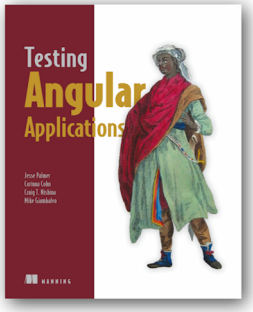
RELATED MANNING TITLES



Angular Development with Typescript, Second Edition

by Yakov Fain and Anton Moiseev

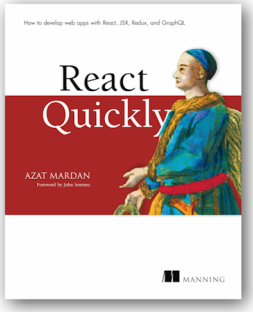
ISBN: 9781617295348
475 pages, $49.99
June 2018



Testing Angular Applications

by Jesse Palmer, Corinna Cohn, Michael Giambalvo, Craig Nishina

ISBN: 9781617293641
235 pages, $44.99
April 2018

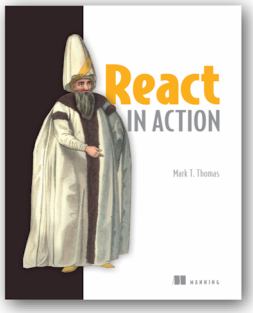


React Quickly

Painless web apps with React, JSX, Redux, and GraphQL

by Azat Mardan

ISBN: 9781617293344
528 pages, $49.99
August 2017



React in Action

by Mark Tielens Thomas

ISBN: 9781617293856
300 pages, $44.99
March 2018

For ordering information go to www.manning.com

JAVASCRIPT

Angular IN ACTION

Jeremy Wilken

Angular makes it easy to deliver amazing web apps. This powerful JavaScript platform provides the tooling to manage your project, libraries to help handle most common tasks, and a rich ecosystem full of third-party capabilities to add as needed. Built with developer productivity in mind, Angular boosts your efficiency with a modern component architecture, well-constructed APIs, and a rich community.

Angular in Action teaches you everything you need to build production-ready Angular applications. You'll start coding immediately, as you move from the basics to advanced techniques like testing, dependency injection, and performance tuning. Along the way, you'll take advantage of TypeScript and ES2015 features to write clear, well-architected code. Thoroughly practical and packed with tricks and tips, this hands-on tutorial is perfect for web devs ready to build web applications that can handle whatever you throw at them.

What's Inside

- Spinning up your first Angular application
- A complete tour of Angular's features
- Comprehensive example projects
- Testing and debugging
- Managing large applications

Written for web developers comfortable with JavaScript, HTML, and CSS.

Jeremy Wilken is a Google Developer Expert in Angular, Web Technologies, and Google Assistant. He has many years of experience building web applications and libraries for eBay, Teradata, and VMware.

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

"A comprehensive introduction to the world of Angular with great code samples to help readers get started."
—Tanya Wilke, Sanlam

"Detailed and clear explanations; lots of useful real-world examples."
—Harsh Raval, Zymr Systems

"You can never have enough Angular references ... this one will float to the top."
—Michael A. Angelo
US Department of Agriculture
Forestry Services

"The bible for Angular!"
—Phily Austria
Faraday Future

ISBN-13: 978-1-61729-331-3
ISBN-10: 1-61729-331-8
54499
9 781617 293313

MANNING $44.99 / Can $59.99 [INCLUDING eBOOK]