

EXPERT INSIGHT



Angular for Enterprise Applications

Build scalable Angular apps using the minimalist Router-first architecture

Third Edition



<packt>

Doguhan Uluca

Angular for Enterprise Applications

Third Edition

Build scalable Angular apps using the minimalist Router-first architecture

Doguhan Uluca



BIRMINGHAM—MUMBAI

“Angular” and the Angular Logo are trademarks of the Google LLC

Angular for Enterprise Applications

Third Edition

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the ...

Contents

[Preface](#)

[Who this book is for](#)

[What this book covers](#)

[To get the most out of this book](#)

[Get in touch](#)

1. [Angular's Architecture and Concepts](#)

[Two Angulars](#)

[A brief history of web frameworks](#)

[The jQuery era](#)

[The server-side MVC era](#)

[Rich client era](#)

[Angular and the philosophies behind it](#)

[Deterministic releases](#)

[First-class upgrades](#)

[Maintainability](#)

[Angular Evergreen](#)

[TypeScript](#)

[Component architecture](#)

[Reactive programming](#)

[RxJS](#)

[Reactive data streams](#)

[Modular architecture](#)

[Standalone architecture](#)

[Angular Router](#)

[Lazy loading](#)

[State management](#)

[The Flux pattern](#)

[NgRx](#)

[NgRx component store](#)

[React.js architecture](#)

[Future of Angular](#)

[Summary](#)

[Further reading](#)

[Questions](#)

2. [Forms, Observables, Signals, and Subjects](#)

[Technical requirements](#)

[Great UX ...](#)

Preface

Welcome to the wonderful world of Angular enterprise development! Standalone projects, signals, and the control flow syntax have injected fresh blood into the framework. At the time that this book is published, Angular 17.1 has been released with features to bring Signals-based components closer to reality, keeping the Angular ecosystem as vibrant as ever. If this trajectory holds, by Angular 20, the framework will be easier than ever to use and will make it possible to create reliable and high-performance applications of any size. This new edition of the book refocuses the content on enterprise architecture and continues the journey toward mastering sophisticated and scalable Angular solutions ready for complex business needs.

Much ...

1

Angular's Architecture and Concepts

Angular is a popular **Single-Page Application (SPA)** framework for building web applications. It is often preferred in enterprise application development because it is an opinionated, batteries-included framework that supports type-checking with TypeScript and concepts like **Dependency Injection (DI)** that allow for engineering scalable solutions by large teams. In contrast, React is a flexible and unopinionated library rather than a complete framework, requiring developers to pick their flavor from the community to build fully featured applications.

React is undoubtedly the more popular choice of the two. The numbers don't lie. React's easier learning curve and deceptively small and simple starting point have attracted the attention of many developers. The many "Angular vs React" articles you have undoubtedly encountered online add to the confusion. These articles are usually too shallow, often contain misleading information about Angular, and lack insights into the very bright future of Angular.

This chapter aims to give you a deeper understanding of why Angular exists, the variety of patterns and paradigms you can leverage to solve complex problems, and, later in the book, the

pitfalls to avoid as you scale your solution. It's important to take your time to read through this material because every journey begins with a choice. The real story of your choice today can only be written several years into a project when it's too late and expensive to switch technologies.

This chapter covers the following topics:

- Two Angulars
- A brief history of web frameworks
- Angular and the philosophies behind it
- Component architecture
- Reactive programming
- Modular architecture
- Standalone architecture
- Angular Router
- State management
- React.js architecture
- The future of Angular

Chapter 2, Forms, Observables, Signals, and Subjects, covers Angular fundamental concepts and building blocks. *Chapter 3, Architecting an Enterprise App*, covers technical, architectural, and tooling concerns for delivering large applications. With *Chapter 4, Creating a Router-First Line-of-Business App*, we dive into creating scalable Angular applications ready for the enterprise.

Each chapter introduces new concepts and progressively builds on best practices while covering optimal working methods with popular open-source tools. Along the way, tips and information

boxes provide additional background and history, numbered steps, and bullet points that describe actions you need to take.

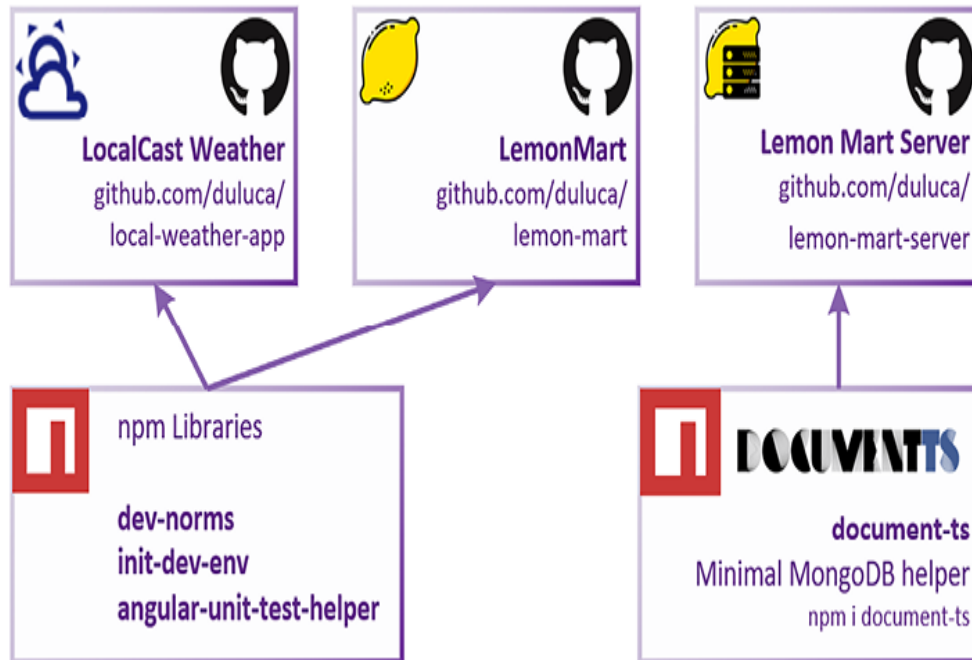
The code samples provided in this book have been developed using Angular 17. Since the second edition, significant changes occurred in the JavaScript and Angular ecosystems. The transition to Angular's Ivy engine meant some third-party tools stopped working. ESLint has superseded TSLint. Karma and Jasmine have become outdated and superseded by Jest or the more modern Vitest. Significant headway was made in replacing `commonjs` modules with **ES modules** (ESM). The totality of these changes meant that much of the second edition's supporting tools were beyond repair. As a lesson learned, the example projects now utilize minimal tooling to allow for the best possible DevEx with the least possible amount of npm packages installed. The core samples of the book, which intentionally avoided third-party libraries, were initially written for Angular 5 and have survived the test of time. This book adopts the Angular Evergreen motto and encourages incremental, proactive, and timely upgrades of your dependencies to maintain the health of your project and your team.

This book is supported by the companion site <https://AngularForEnterprise.com>. Visit the site

for the latest news and updates.

The world of JavaScript, TypeScript, and Angular is constantly changing. To maintain consistency for my readers, I published a collection of open-source projects that support the content of the book:

Open-Source Projects



Dev Tools

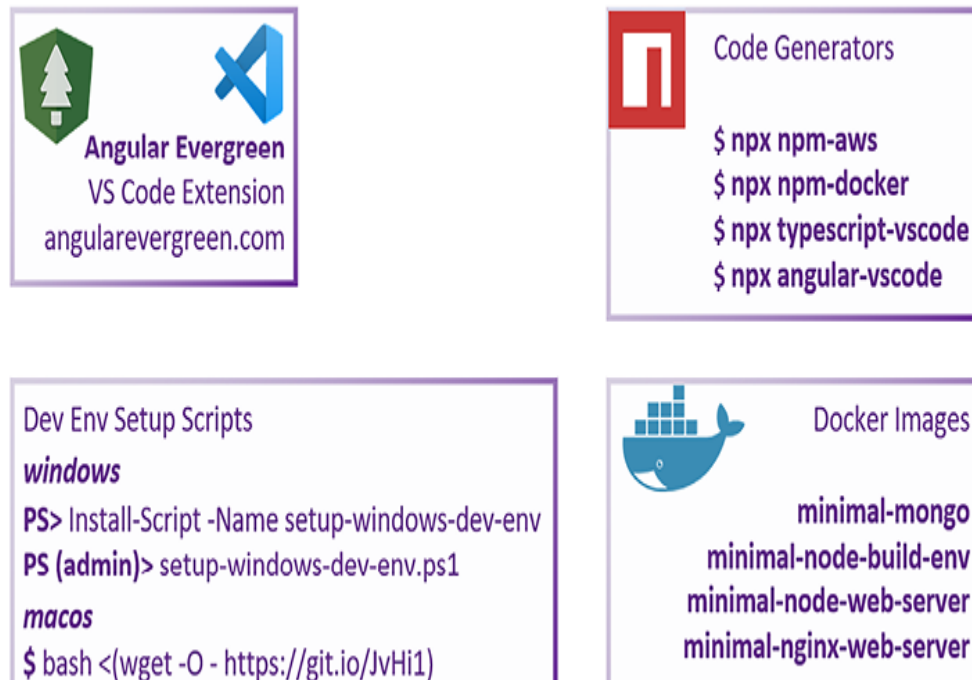


Figure 1.1: Code developed in support of this book

The diagram above shows you the moving parts that make up the technical content supporting this book. Each component is detailed in the coming chapters. The code samples contain chapter-by-chapter snapshots and the final state of the code. The most up-to-date versions of the sample code for the book are on GitHub at the repositories linked below:

- For *Chapters 2 and 9*, LocalCast Weather:
<https://github.com/duluca/local-weather-app>
- For *Chapters 4 to 10*, Lemon Mart:
<https://github.com/duluca/lemon-mart>
- For *Chapter 5*, Lemon Mart Server:
<https://github.com/duluca/lemon-mart-server>

You may read more about updating Angular in the supplemental reading, *Keeping Angular and Tools Evergreen*, available at
<https://angularforenterprise.com/evergreen>.

Now that you're oriented with the book's structure and supporting content, and before we dive into a prolonged history of the web, let's first disambiguate the two major architectures of Angular and the underlying themes that motivated a dramatic rewrite of the framework in 2016.

Two Angulars

In its original incarnation, Angular.js, aka 1.x, pioneered the SPA era, a technique that tricks the browser into thinking that a single `index.html` houses an interactive application containing many pages. Angular.js also popularized the concept of two-way binding in web development, which automatically updates the view to match the state of the ViewModel. To implement such a feature, Angular.js used **Change Detection** to keep track of **Document Object Model (DOM)** elements of the browser and the ViewModel state of the application.

Change Detection depends on a sophisticated rendering loop to detect user interactions and other events to determine if the application needs to react to changes. Whenever a rendering loop is involved, like in games, performance can be measured as a frame rate expressed in **Frames per Second (FPS)**. A slow change detection process results in a low FPS count, translating into a choppy **User Experience (UX)**. With the demand for more interactive and complicated web applications, it became clear that the internal architecture of Angular.js couldn't be improved to maintain a consistent FPS output. However, UX and performance are only one side of the experience story. As an application grows more complicated better tooling is needed to support a great **Developer Experience (DevEx)** – sometimes called **DevX** or **DX** – which is key to developer wellbeing.

The Angular 2 rewrite, now simply referred to as Angular, aimed to solve both sides of the problem. Before frameworks and libraries like React, Angular, and Vue, we suffered from unmanaged complexity and JavaScript-framework-of-the-week syndrome. These

frameworks succeeded with promises to fix all problems, bring about universally reusable web components, and make it easier to learn, develop, and scale web applications- at least for a while, some being better than others during different periods. The same problems that plagued early SPA are returning as the demand for ever more complicated web experiences increases, and the tooling to resolve these problems grows ever complex. To master Angular or any other modern framework, it is critical to learn about the past, present, and future of web development. The adolescent history of the web has taught us a couple of essential lessons. First, change is inevitable, and second, the developer's happiness is a precious commodity that can make or break entire companies.

As you can see, Angular's development has been deeply impacted by performance, UX, and DevEx concerns. But this wasn't a unique problem that only impacted Angular. Let's roll back the clock further and look at the last quarter century or so of web development history so that you can contextualize modern frameworks like Angular, React, and Vue.

A brief history of web frameworks

It is essential to understand why we use frameworks such as Angular, React, or Vue in the first place to get an appreciation of the value they bring. As the web evolves, you may find that, in some cases, the framework is no longer necessary and should be discarded, and in others, critical to your business and must be retained. Web frameworks rose as JavaScript became more popular and capable in the browser. In 2004, the **Asynchronous JavaScript**

and XML (AJAX) technique became very popular in creating websites that did not have to rely on full-page refreshes to create dynamic experiences, utilizing standardized web technologies like HTML, JavaScript/ECMAScript, and CSS. Browser vendors are supposed to implement these technologies as defined by the **World Wide Web Consortium (W3C)**.

Internet Explorer (IE) was the browser that most internet users relied on at the time. Microsoft used its market dominance to push proprietary technologies and APIs to secure IE's edge as the go-to browser. Things started to get interesting when Mozilla's Firefox challenged IE's dominance, followed by Google's Chrome browser. As both browsers successfully gained significant market share, the web development landscape became a mess. New browser versions appeared at breakneck speed. Competing corporate and technical interests led to the diverging implementation of web standards.

This fracturing created an unsustainable environment for developers to deliver consistent experiences on the web. Differing qualities, versions, and names of implementations of various standards created an enormous challenge: successfully writing code that could manipulate the DOM of a browser consistently. Even the slightest difference in the APIs and capabilities of a browser would be enough to break a website.

The jQuery era

In 2006, jQuery was developed to smooth out the differences between APIs and browser capabilities. So instead of repeatedly writing code to check browser versions, you could use jQuery, and

you were good to go. It hid away all the complexities of vendor-specific implementations and gracefully filled the gaps when there were missing features. For almost a decade, jQuery became the web development framework. It was unimaginable to write an interactive website without using jQuery.

However, to create vibrant user experiences, jQuery alone was not enough. Native web applications ran all their code in the browser, which required fast computers to run the dynamically interpreted JavaScript and render web pages using complicated object graphs. In the 2000s, many users ran outdated browsers on relatively slow computers, so the user experience wasn't great.

Combined with AJAX, jQuery enabled any web developer to create interactive and dynamic websites that could run on any browser without running expensive server hardware and software. To have a solid understanding of the architectural nuances of code that runs on the client and server side, consider a traditional three-tier software architecture. Each tier is described in three primary layers, as shown in the following diagram:



Figure 1.2: Three-tiered software architecture

The presentation layer contains **User Interface (UI)** related code. This is primarily code that runs on the client, referred to as a **thick client**. However, the presentation logic can instead reside on the server. In these cases, the client becomes a **thin client**. The business layer contains business logic and normally resides on the server side. An undisciplined implementation can result in business logic spreading across all three layers. This means a bug or a change in the logic needs to be implemented in many locations. In reality, no individual can locate every occurrence of this logic and can only partially repair code. This, of course, results in the creation of more exotic bugs. The persistence layer contains code related to data storage.

To write easy-to-maintain and bug-free code, our overall design goal is to aim for low coupling and high cohesion between the components of our architecture. Low coupling means that pieces of

code across these layers shouldn't depend on each other and should be independently replaceable. High cohesion means that pieces of code related to each other, like code regarding a particular domain of business logic, should remain together. For example, when building an app to manage a restaurant, the code for the reservation system should be together and not spread across other systems like inventory tracking or user management.

With jQuery and AJAX, writing thick clients for the web became possible, making it easier than ever to write unmaintainable code. Modern web apps have way more moving parts than a basic three-tiered application. The diagram that follows shows additional layers that fit around the presentation, business, and persistence layers:

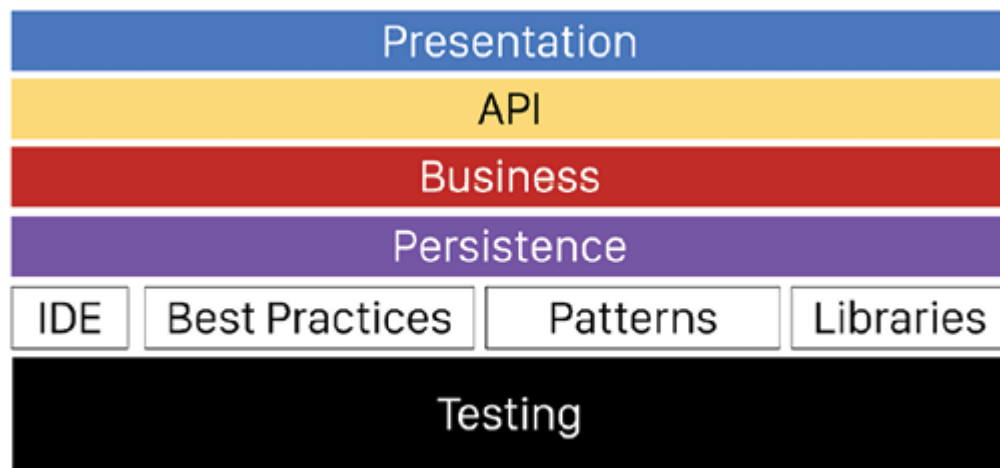


Figure 1.3: Modern Web Architecture

You can observe the essential components of modern web development in the expanded architecture diagram, which includes an API layer that usually transforms and transfers data between the presentation and business layers. Beyond code within the operating environment, the tools and best practices layer defines and enforces

patterns used to develop the software. Finally, the testing layer defines a barrage of automated tests to ensure the correctness of code, which is crucial in today's iterative and fast-moving development cycles.

While there was a big appetite to democratize web development with thick clients primarily consuming client-side computing resources, the tooling wasn't ready to enforce proper architectural practices and deliver maintainable software. This meant businesses kept investing in server-side rendering technologies.

The server-side MVC era

In the late 2000s, many businesses still relied on server-side rendered web pages. The server dynamically created all the HTML, CSS, and data needed to render a page. The browser acted as a glorified viewer that would display the result. The following is a diagram that shows an example architectural overview of a server-side rendered web application in the ASP.NET MVC stack:

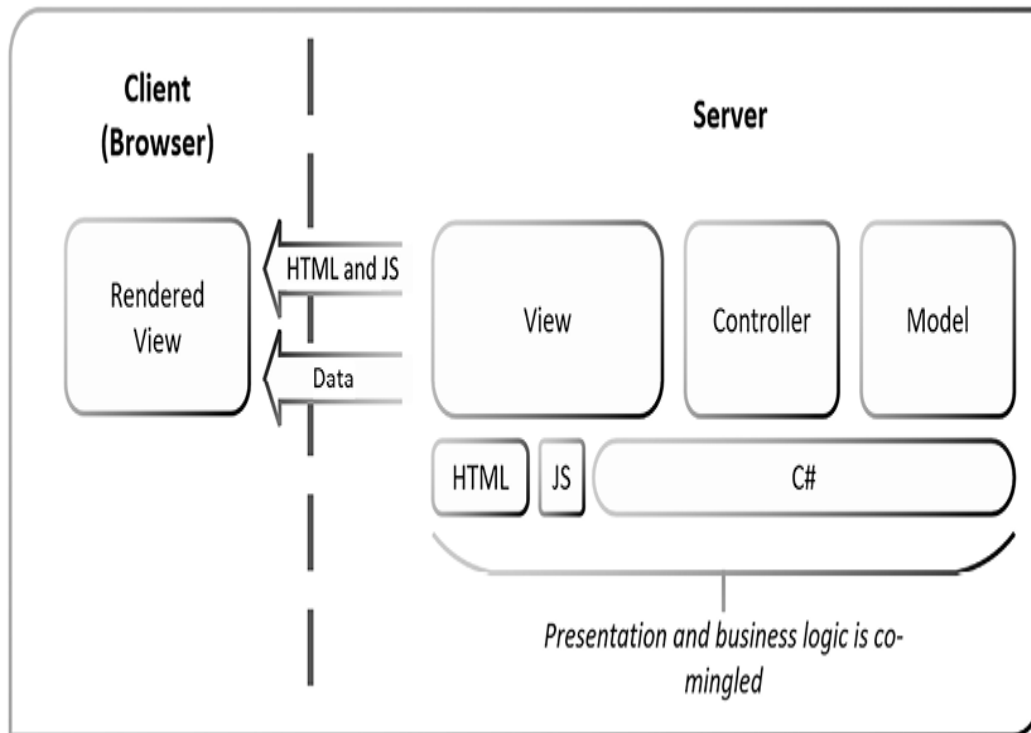


Figure 1.4: Server-side rendered MVC architecture

Model-View-Controller (MVC) is a typical pattern of code that has data manipulation logic in models, business logic in controllers, and presentation logic in views. In the case of ASP.NET MVC, the controller and model are coded using C#, and views are created using a templated version of HTML, JavaScript, and C#. The result is that the browser receives HTML, JavaScript, and needed data, and through jQuery and AJAX magic, web pages look to be interactive. Server-side rendering and MVC patterns are still popular and in use today. There are justified niche uses, such as [Facebook.com](https://www.facebook.com). Facebook serves billions of devices that range from the very slow to the very fast. Without server-side rendering, it would be impossible for Facebook to guarantee consistent UX across its user base.

The combination of server-side rendering and MVC is an intricate pattern to execute; there are a lot of opportunities for presentation

and business logic to become co-mingled. To ensure the low coupling of components, every member of the engineering team must be very experienced. Teams with a high concentration of senior developers are hard to come by, which would be an understatement.

Further complicating matters is that C# (or any other server-side language) cannot run natively in the browser. So developers who work on server-side rendered applications must be equally skilled at using frontend and backend technologies. It is easy for inexperienced developers to unintentionally co-mingle presentation and business logic in such implementations. When this happens, the inevitable UI modernization of an otherwise well-functioning system becomes impossible. In other words, to replace the sink in your kitchen with a new one, you must renovate your entire kitchen. Due to insufficient architecture, organizations spend millions of dollars writing and rewriting the same applications every decade.

Rich client era

In the 2000s, it was possible to build rich web applications decoupled from their server APIs using Java Applets, Flash, or Silverlight. However, these technologies relied on browser plugins that needed a separate installation. Most often, these plugins were outdated, created critical security vulnerabilities, and consumed too much power on mobile computers. Following the iPhone revolution in 2008, it was clear such plugins wouldn't run on mobile phones, despite the best attempts by the Android OS. Besides, Apple CEO Steve Jobs' disdain for such inelegant solutions marked the

beginning of the end for the support of such technologies in the browser.

In the early 2010s, frameworks like Backbone and AngularJS started showing up, demonstrating how to build rich web applications with a native feel and speed and do so in a seemingly cost-effective way. The following diagram shows a **Model-View-ViewModel (MVVM)** client with a **Representational State Transfer (REST)** API. When we decouple the client from the server via an API, we can architecturally enforce the implementation of presentation and business logic separately. Theoretically, this RESTful web services pattern would allow us to replace the kitchen sink as often as possible without remodeling the entire kitchen.

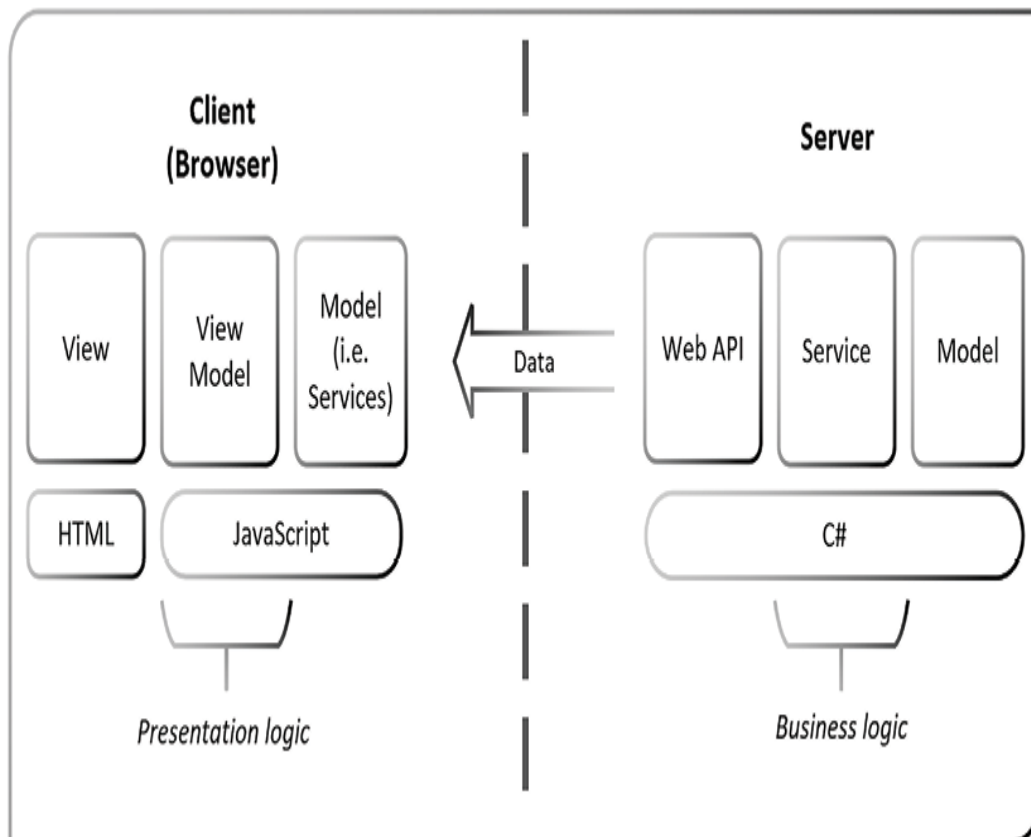


Figure 1.5: Rich-client decoupled MVVM architecture

The MVVM architecture above shows a near doubling of boxes compared to the server-side MVC architecture. Does this mean we need to write twice as much code? Yes and no. Yes, we need to write more code to maintain a disciplined architecture; however, over time, we'll write a lot less code because of the overall maintainability of the solution. The architecture surrounding the presentation logic indeed becomes a lot more complicated. The client and server must implement their presentation/API, business, and persistence layers.

Unfortunately, many early development efforts leveraging frameworks like Backbone and AngularJS collapsed under their weight because they failed to implement the client-side architecture properly.

These early development efforts also suffered from ill-designed RESTful Web APIs. Most APIs didn't version their URIs, making it very difficult to introduce new functionality while supporting existing clients. Further, APIs often returned complicated data models exposing their internal relational data models to web apps. This design flaw creates a tight coupling between seemingly unrelated components/views written in HTML and models created in SQL. If you don't implement additional layers of code to translate or map the structure of data, then you create an unintentional and uncontrolled coupling between layers. Over time, dealing with such coupling becomes very expensive very quickly, in most cases necessitating significant rewrites.

Today, we use the API layer to flatten the data model before sending it to the client to avoid such problems. Newer technologies like GraphQL go further by

exposing a well-defined data model and letting the consumer query for the exact data it needs. Using GraphQL, the number of HTTP requests and the amount of data transferred over the wire is optimal without the developers having to create many specialized APIs.

Backbone and AngularJS proved that creating web applications that run natively in the browser was viable. All SPA frameworks at the time relied on jQuery for DOM manipulation. Meanwhile, web standards continued to evolve, and evergreen browsers supporting new standards became commonplace. However, change is constant, and the evolution of web technologies made it unsustainable to gracefully evolve this first generation of SPA frameworks, as I hinted in the *Two Angulars* section.

The next generation of web frameworks needed to solve many problems; they needed to enforce good architecture, be designed to evolve with web standards and be stable and scalable to enterprise needs without collapsing. Also, these new frameworks needed to gain acceptance from developers, who were burned out with too many rapid changes in the ecosystem. Remember, unhappy developers do not create successful businesses. Achieving these goals required a clean break from the past, so Angular and React emerged as platforms to address the problems of the past in different ways. As you'll discover in the following sections, Angular offers the best tools and architecture for building scalable enterprise-grade applications.

Angular and the philosophies behind it

Angular is an open-source project maintained by Google and a community of developers. The new Angular platform vastly differs from the legacy framework you may have used. In collaboration with Microsoft, Google made TypeScript the default language for Angular. TypeScript is a superset of JavaScript that enables developers to target legacy browsers, such as Internet Explorer 11, while allowing them to write modern JavaScript code that works in evergreen browsers such as Chrome, Firefox, and Edge. The legacy version of Angular in the 1.x range, called AngularJS, was a monolithic JavaScript SPA framework. The modern version, Angular 2+, is a platform capable of targeting browsers, hybrid-mobile frameworks, desktop applications, and server-side rendered views.

In the prior generation, upgrading to new versions of AngularJS was risky and costly because even minor updates introduced new coding patterns and experimental features. Each update introduced deprecations or refactored API surfaces, requiring rewriting of large portions of code. Also, updates were delivered in uncertain intervals, making it impossible for a team to plan resources to upgrade to a new version. The release methodology eventually led to an unpredictable, ever-evolving framework with seemingly no guiding hand to carry code bases forward. If you used AngularJS, you were likely stuck on a particular version because the specific architecture of your code base made it very difficult to move to a new version. In 2018, the Angular team released the last major update to AngularJS

with version 1.7. This release marked the beginning of the end for the legacy framework, with end-of-life coming in January 2022.

Deterministic releases

Angular improves upon AngularJS in every way imaginable. The platform follows **semver**, as defined at <https://semver.org/>, where minor version increments denote new feature additions and potential deprecation notices for the following major version, but no breaking changes. Furthermore, the Angular team at Google has committed to a deterministic release schedule with major versions released every 6 months. After this 6-month development window, starting with Angular 4, all major releases receive LTS with bug fixes and security patches for an additional 12 months. From release to end-of-life, each major version receives updates for 18 months. Refer to the following chart for the tentative release and support schedule for Angular:

Version	Status	Released	Active Ends	LTS Ends
^14.0.0	LTS	2022-06-02	2022-11-18	2023-11-18
^15.0.0	LTS	2022-11-18	2023-05-03	2024-05-18
^16.0.0	LTS	2023-05-03	2023-11-03	2024-11-18
^17.0.0	Active	2023-11-06	2024-05-08	2025-05-15

Figure 1.6: Actively supported versions

What does this mean for you? You can be confident that your Angular code is supported and backward compatible for approximately 24 months, even if you make no changes to it. For

example, if you wrote an Angular app in version 17 in November 2023, and you didn't use any deprecated functionality, your code will be runtime compatible with Angular 18 and supported through May 2025. To upgrade your Angular 17 code to Angular 19, you must ensure that you're not using any deprecated APIs that receive a deprecation notice in Angular 18.

In practice, most deprecations are minor and are straightforward to refactor. Unless you work with low-level APIs for highly specialized user experiences, the time and effort it takes to update your code base should be minimal. However, this is a promise made by Google and not a contract. The Angular team has a significant incentive to ensure backward compatibility because Google runs around 1,000+ Angular apps with a single version of Angular active at any one time throughout the organization. So, by the time you read this, all of Google's 1,000+ apps will be running on the latest version of Angular.

First-class upgrades

You may think Google has infinite resources to update thousands of apps regularly. Like any organization, Google, too, has limited resources. It would be too expensive to assign a dedicated team to maintain every app. So the Angular team must ensure compatibility through automated tests and make it as painless as possible to move through major releases in the future. In Angular 6 `ng update` was introduced, making the update process a first-class experience.

The Angular team continually improves its release process with automated CLI tools to make upgrading deprecated functionality a

mostly automated, reasonable endeavor. Air France and KLM demonstrated this strategy's benefits, reducing their upgrade times from 30 days in Angular 2 to 1 day in Angular 7.

A predictable and well-supported upgrade process is excellent news for developers and organizations. Instead of being perpetually stuck on a legacy version of Angular, you can plan and allocate the necessary resources to keep moving your application to the future without costly rewrites. As I wrote in a 2017 blog post, *The Best New Feature of Angular 4*, at <https://bit.ly/NgBestFeature>, the message is clear:

For developers and managers: Angular is here to stay, so you should be investing your time, attention, and money in learning it – even if you're currently in love with some other framework.

For decision makers (CIOs, CTOs, and so on): Plan to begin your transition to Angular in the next 6 months. It'll be an investment you'll be able to explain to business-minded people, and your investment will pay dividends for many years to come, long after the initial LTS window expires, with graceful upgrade paths to Angular vNext and beyond.

So why do Google (Angular) and Microsoft (TypeScript and Visual Studio Code) give away such technologies for free? There are

multiple reasons:

- A sophisticated framework that makes it easy to develop web apps demonstrates technical prowess, which retains and attracts developer talent.
- An open-source framework enables the proving and debugging of new ideas and tools with millions of developers at scale.
- Allowing developers to create great web experiences drives more business for Google and Microsoft.

I don't see any nefarious intent here and welcome open, mature, and high-quality tools that, if necessary, I can tinker with and bend to my own will. Not having to pay for a support contract for a proprietary piece of tech is a welcome bonus.

Beware - looking for Angular help on the web may be tricky. You'll need to disambiguate between AngularJS or Angular, which may be referred to as Angular2, but also be aware that some advice given about versions 13 or below may not apply to 14+ because of the rendering engine change to Ivy. I always recommend reading the official documentation when learning. Documentation for Angular is at <https://angular.dev>. This should not be confused with angularjs.org, which is about the legacy AngularJS framework or the retired angular.io site.

For the latest updates on the upcoming Angular releases, view the official release schedule at

<https://angular.dev/reference/releases>.

Maintainability

Your time is valuable, and your happiness is paramount, so you must carefully choose the technologies to invest your time in. With this in mind, we must answer why Angular is the tool you should learn over React, Vue, or others. Angular is a great framework to start learning. The framework and the tooling help you get off the ground quickly and continue being successful, with a vibrant community and high-quality UI libraries you can use to deliver exceptional web applications. React and Vue are great libraries with their strengths and weaknesses. Every tool has its place and purpose.

In some cases, React is the right choice for a project, while Vue is the right one in others. Becoming somewhat proficient in other web frameworks can only help further your understanding of Angular and make you a better developer overall. SPAs such as Backbone and AngularJS grabbed my full attention in 2012 when I realized the importance of decoupling frontend and backend concerns. Server-side rendered templates are nearly impossible to maintain and are the root cause of many expensive rewrites of software systems. If you care about creating maintainable software, you must abide by the prime directive: keep the business logic behind the API decoupled from the presentation logic implemented in the UI.

Angular neatly fits the Pareto principle or the 80-20 rule. It has become a mature and evolving platform, allowing you to achieve 80% of tasks with 20% of the effort. As mentioned in the previous

section, every major release is supported for 18 months, creating a continuum of learning, staying up to date, and deprecating old features. From the perspective of a full-stack developer, this continuum is invaluable since your skills and training will remain relevant and fresh for many years to come.

The philosophy behind Angular is to err on the side of configuration over convention. Although convention-based frameworks may seem elegant from the outside, they make it difficult for newcomers to pick up the framework. Configuration-based frameworks aim to expose their inner workings through explicit configuration and hooks, where you can attach your custom behavior to the framework. In essence, where AngularJS had tons of magic, which can be confusing, unpredictable, and challenging to debug, Angular tries to be non-magical.

Configuration over convention results in verbose coding. Verbosity is a good thing. Terse code is the enemy of maintainability, only benefiting the original author. As Andy Hunt and David Thomas put it in *The Pragmatic Programmer*:

Remember that you (and others after you) will be reading the code many hundreds of times, but only writing it a few times.

Further, Andy Hunt's *Law of Design* dictates:

If you can't rip every piece out easily, then the design sucks.

Verbose, decoupled, cohesive, and encapsulated code is the key to future-proofing your code. Through its various mechanisms, Angular enables the proper execution of these concepts. It eliminates many custom conventions invented in AngularJS, such as `ng-click`, and introduces a more natural language that builds on the existing HTML elements and properties. As a result, `ng-click` becomes `(click)`, extending HTML rather than replacing it.

Next, we'll review Angular's evergreen mindset and the reactive programming paradigm, the latest extensions of Angular's initial philosophy.

Angular Evergreen

When you're learning Angular, you're not learning one specific version of Angular but a platform that is continually evolving. Since the first drafts, I designed this book to deemphasize the specific version of Angular you're using. The Angular team champions this idea. Over the years, I have had many conversations with the Angular team and thought leaders within the community and listened to many presentations. As a result, you can depend on Angular as a mature web development platform. Angular frequently receives updates with great attention to backward compatibility. Furthermore, any code made incompatible by a new version is brought forward with help from automated tools or explicit guidance on updating your code by locating the **Angular Update**

Guide on <https://angular.dev/update>, so you're never left guessing or scouring the internet for answers. The Angular team is committed to ensuring you – the developer – have the best web development experience possible.

To bring this idea front and center with developers, several colleagues and I have developed and published a Visual Studio Code extension called Angular Evergreen, as shown in the following image:



File Edit Selection View Go Debug Terminal




ANGULAR EVERGREEN



▼ VERSIONS




▼  Current CLI: 9.0.0

Latest: 9.0.1



Next: 9.0.0

▼  Current Core: 9.0.0

Latest: 9.0.0



Next: 9.0.0



▼ QUICK COMMANDS



Configure VS Code for Angular

▼  Update with Angular CLI



Update Angular CLI & Core



Update Angular --all



Update Angular --all --force




Update Angular CLI & Core --next



Update Angular --next --all



Update Angular --next --all --force

▼  Update npm Packages



Check npm packages



Apply npm updates



Run Post-Update Checkup



▼ NEED HELP?



How to Update



Visit blog.angular.io



Request Consulting

Figure 1.7: Angular Evergreen VS Code extension

This extension detects your current version of Angular and compares it to the latest and next releases of Angular. Releases labeled next are meant for early adopters and testing your code's compatibility with an upcoming version of Angular. Do not use next-labeled releases for production deployments.

Find more information, feature requests, and bug reports on the Angular Evergreen extension at <https://AngularEvergreen.com>.

One of the critical components of Angular that allows the platform to remain evergreen is TypeScript. TypeScript allows new features to be implemented efficiently while supporting older browsers, so your code can reach the widest audience possible.

TypeScript

Angular is coded using TypeScript. Anders Hejlsberg of Microsoft created TypeScript to address several major issues with applying JavaScript at a large enterprise scale.

Anders Hejlsberg is the creator of Turbo Pascal and C# and Delphi's chief architect. Anders designed C# to be a developer-friendly language built upon the familiar syntax of C and C++. As a result, C# became the language behind Microsoft's popular .NET Framework. TypeScript shares a similar pedigree with Turbo Pascal and C# and their ideals, which made them a great success.

JavaScript is a dynamically interpreted language where the browser parses and understands the code you write at runtime. Statically typed languages like Java or C# have an additional compilation step where the compiler can catch programming and logic errors during compile time. Detecting and fixing bugs at compile time versus runtime is much cheaper. TypeScript brings the benefits of statically typed languages to JavaScript by introducing types and generics. However, TypeScript is not a compiler in the traditional sense. It is a transpiler. A compiler builds code into machine language with C/C++ or **Intermediary Language (IL)** with Java or C#. A transpiler, however, transforms the code from one dialect to another. So, when TypeScript code is built, compiled, or transpiled, the result is pure JavaScript.

JavaScript's official name is ECMAScript. The language's feature set and syntax are maintained by the ECMA Technical Committee 39, or TC39 for short.

Transpilation has another significant benefit. The same tooling that converts TypeScript to JavaScript can be used to rewrite JavaScript with a new syntax to an older version that older browsers can parse and execute. Between 1999 and 2009, the JavaScript language didn't see any new features. ECMAScript abandoned version 4 due to various technical and political reasons. Browser vendors have struggled to implement new JavaScript features within their browsers, starting with the introduction of ES5 and then ES2015 (also known as ES6).

As a result, user adoption of these new features has remained low. However, these new features meant developers could write code more productively. This created a gap known as the JavaScript Feature Gap, as demonstrated by the graphic that follows:

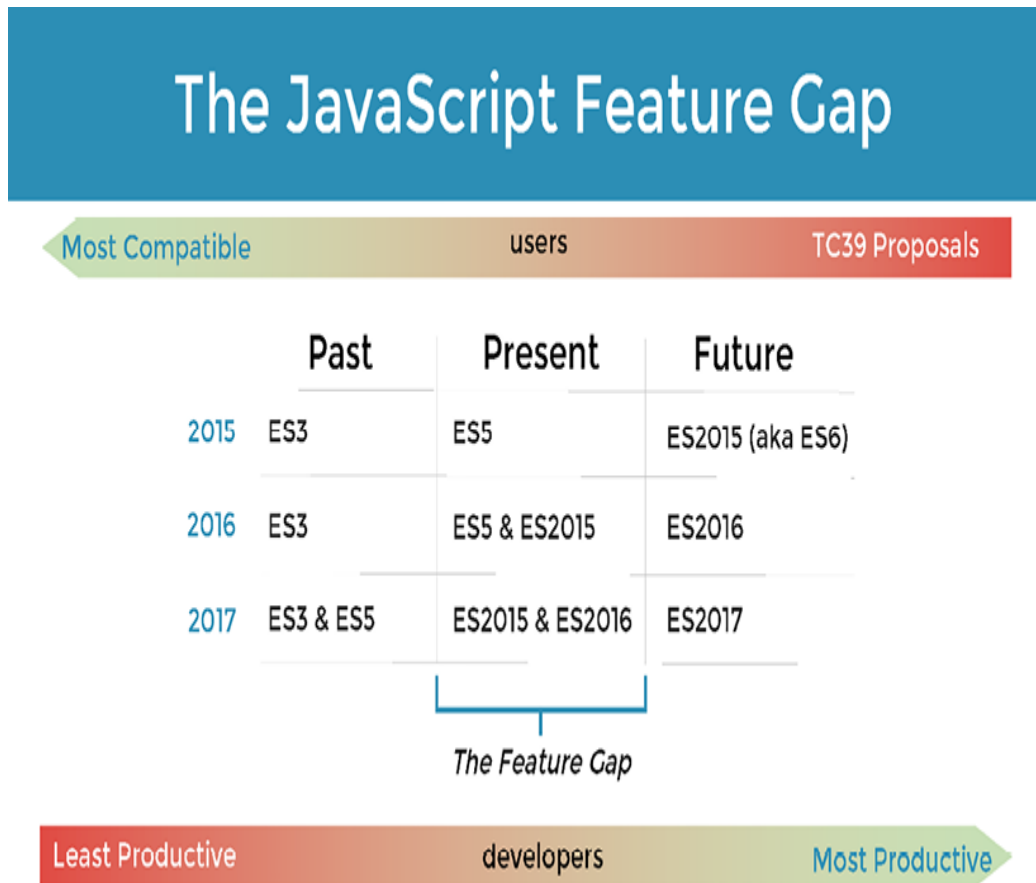


Figure 1.8: The JavaScript Feature Gap

The JavaScript Feature Gap is sliding, as TC39 has committed to updating JavaScript every year. As a result, TypeScript represents JavaScript's past, present, and future. You can use future features of JavaScript today and still be able to target browsers of the past to maximize the audience you can reach. In 2023, this gap is smaller

than ever, with ES2022 being a mature language with wide support from every major browser.

Now, let's go over Angular's underlying architecture.

Component architecture

Angular follows the MV* pattern, a hybrid of the MVC and MVVM patterns. Previously, we went over the MVC pattern. At a high level, the architecture of both patterns is relatively similar, as shown in the diagram that follows:

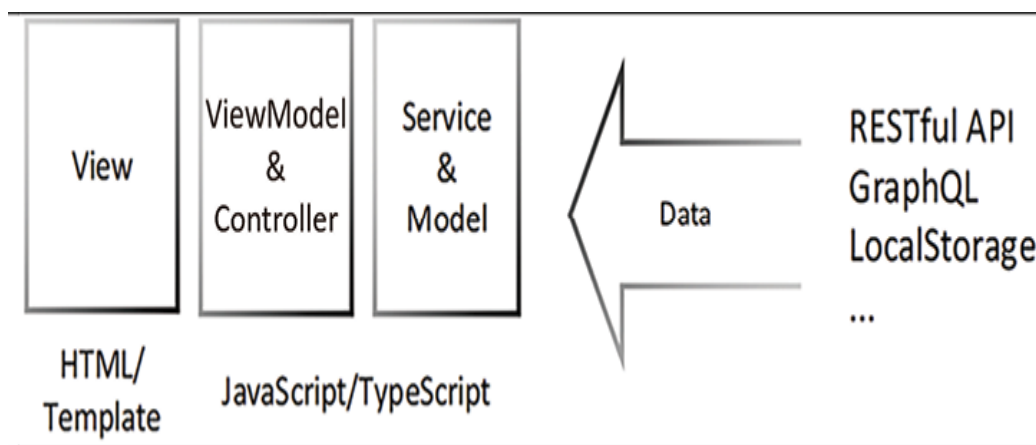


Figure 1.9: MV architecture*

The new concept here is the ViewModel, which represents the glue code that connects your view to your model or service. In Angular, this glue is known as binding. Whereas MVC frameworks like Backbone or React must call a **render** method to process their HTML templates, in Angular, this process is seamless and transparent for the developer. Binding is what differentiates an MVC application from an MVVM one.

The most basic unit of an Angular app is a component. A component combines a JavaScript class, written in TypeScript, and an Angular template, written in HTML, CSS, and TypeScript, as one element. The class and the template fit together like a jigsaw puzzle through bindings so that they can communicate with each other, as shown in the diagram that follows:

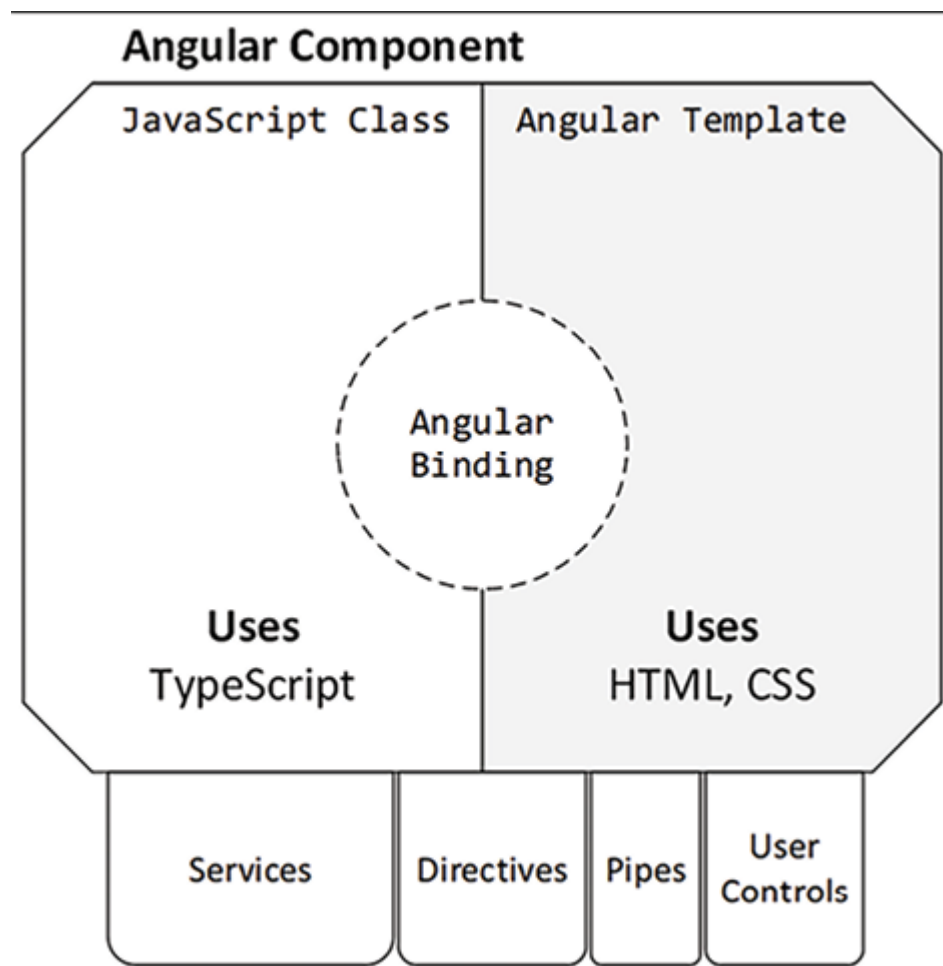


Figure 1.10: Anatomy of a component

Classes are an **Object-Oriented Programming (OOP)** construct. If you invest the time to dig deeper into the OOP paradigm, you will vastly improve your understanding of how Angular works. The

OOP paradigm allows for the **Dependency Injection (DI)** of dependent services in your components, so you can make HTTP calls or trigger a toast message to be displayed to the user without pulling that logic into your component or duplicating your code. DI makes it very easy for developers to use many interdependent services without worrying about the order of the instantiation, initialization, or destruction of such objects from memory.

Angular templates allow similar code reuse via directives, pipes, user controls, and other components. These are pieces of code that encapsulate highly interactive end-user code. This kind of interactivity code is often complicated and convoluted and must be kept isolated from business logic or presentation logic to keep your code maintainable.

Angular 17 introduces a new **control flow syntax** (in preview), which replaces directives like `*ngIf` with `@if`, `*ngFor` with `@for`, and `*ngSwitch` with `@switch` and introduces `@empty`, `@defer`, contextual variables, and conditional statements. The new syntax makes templates easier to read and avoids importing legacy directives to every component in a standalone project. This book will exclusively use the control flow syntax.

You can run `npx ng generate @angular/core:control-flow` to convert your existing template to the new syntax.

Angular apps can be created in two different ways:

- An NgModule project
- A standalone project

As of Angular 17, the default way is to bootstrap your app as a standalone project. This approach has many benefits, as further explained in *The Angular Router* section below. There is a lot of new terminology to learn, but modules as a concept aren't going away. It's just that they're no longer required.

Whether your app starts with `bootstrapApplication` or `bootstrapModule`, at the root level of your application, Angular components, services, directives, pipes, and user controls are provided to the `bootstrapApplication` function or organized under modules. The root level configuration renders your first component, injects any services, and prepares any dependencies it may require. In a standalone app, you can lazily load individual components.

You may also introduce **feature modules** to lazy load groups of services and components. All these features help the initial app load up very quickly, improving First Contentful Paint times because the framework doesn't have to download and load all web application components in the browser simultaneously. For instance, sending code for the admin dashboard to a user without admin privileges is useless.

Being able to create standalone components allows us to ditch contrived modules. Previously, you were forced to place shared components in a shared module, leading to inefficiencies in reducing

app size because developers wouldn't necessarily want to create a module per shared component. For example, the LocalCast Weather app is a simple app that doesn't benefit from the concept of a module, but the LemonMart app naturally reflects a modular architecture by implementing separate business functions in different modules. More on this later in the chapter in the *Modular architecture* section.

Standalone components shouldn't be confused with Angular elements, an implementation of the web standard, custom elements, also known as Web Components. Implementing components in this manner would require the Angular framework to be reduced to only a few KB in size, as opposed to the current framework of around 150 KB. If this is successful, you will be able to use an Angular component you develop in any web application. Exciting stuff but also a tall order. You can read more about Angular elements at <https://angular.dev/guide/elements>.

Angular heavily uses the RxJS library, which introduces reactive development patterns to Angular instead of more traditional imperative development patterns.

Reactive programming

Angular supports multiple styles of programming. The plurality of coding styles within Angular is one of the reasons it is approachable to programmers with varying backgrounds. Whether you come from an object-oriented programming background or are a staunch believer in functional programming, you can build viable apps using Angular. In *Chapter 2, Forms, Observables, Signals, and Subjects*, you'll begin leveraging reactive programming concepts in building the LocalCast Weather app.

As a programmer, you are most likely used to imperative programming. Imperative programming is when you, as the programmer, write sequential code describing everything that must be done in the order that you've defined them and the state of your application, depending on just the right variables to be set to function correctly. You write loops, conditionals, and call functions; you fire off events and expect them to be handled. Imperative and sequential logic is how you're used to coding.

Reactive programming is a subset of functional programming. In functional programming, you can't rely on variables you've set previously. Every function you write must stand on its own, receive its own set of inputs, and return a result without being influenced by the state of an outer function or class. Functional programming supports **Test Driven Development (TDD)** very well because every function is a unit that can be tested in isolation. As such, every function you write becomes composable. So you can mix, match, and combine any function you write with any other and construct a series of calls that yield the result you expect.

Reactive programming adds a twist to functional programming. You no longer deal with pure logic but an asynchronous data stream that you transform and mold into any shape you need with a composable set of functions. So when you subscribe to an event in a reactive stream, you're shifting your coding paradigm from reactive programming to imperative programming.

Later in the book, when implementing the LocalCast Weather app, you'll leverage `subscribe` in action in the `CurrentWeatherComponent` and `CitySearchComponent`.

Consider the following example, aptly put by Mike Pearson in his presentation *Thinking Reactively: Most Difficult*, of providing instructions to get hot water from the faucet to help understand the differences between imperative and reactive programming:

Instructions to get hot water from the faucet

Imperative



Reactive

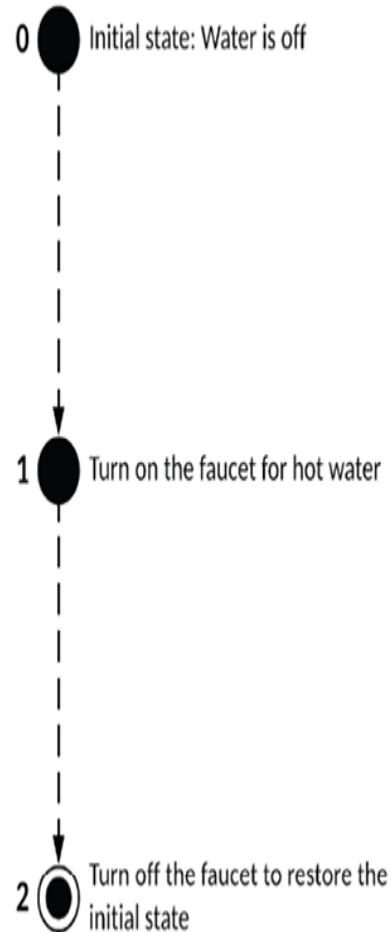


Figure 1.11: Imperative vs Reactive methodology

As you can see, with imperative programming, you must define every step of the code execution. There are six steps in total. Every step depends on the previous step, which means you must consider the state of the environment to ensure a successful operation. In such an environment, it is easy to forget a step and very difficult to test the correctness of every individual step. In functional reactive programming, you work with asynchronous data streams resulting

in a stateless workflow that is easy to compose with other actions. There are two steps in total, but *step 2* doesn't require any new logic. It simply disconnects the code in *step 1*.

RxJS is the library that allows you to implement your code in the reactive paradigm.

Angular 16 introduced signals, in a developer preview, as a new paradigm to enable fine-grained reactivity within Angular. In *Chapter 2, Forms, Observables, Signals, and Subjects*, you will implement signals in your Angular application. Refer to the *Future of Angular* section later in the chapter for more information.

RxJS

RxJS stands for **Reactive Extensions**, a modular library that enables reactive programming. It is an asynchronous programming paradigm that allows data stream manipulation through transformation, filtering, and control functions. You can think of reactive programming as an evolution of event-based programming.

Reactive data streams

In event-driven programming, you would define an event handler and attach it to an event source. In more concrete terms, if you had a **Save** button, which exposes an `onClick` event, you would implement a `confirmSave` function that, when triggered, would

show a popup to ask the user ‘**Are you sure?**’. Look at the following diagram for a visualization of this process:

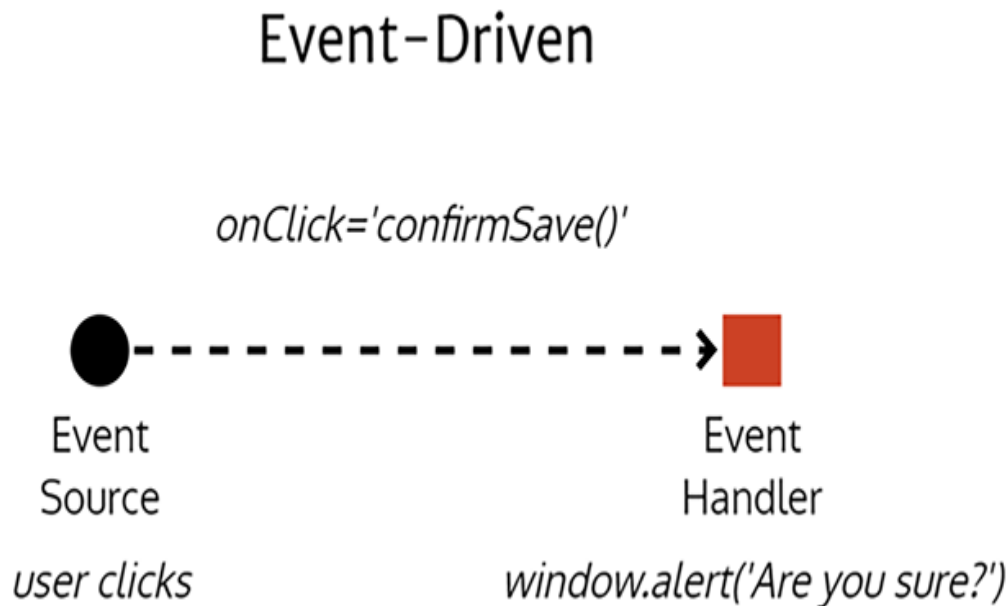


Figure 1.12: Event-driven implementation

In short, you would have an event firing once per user action. If the user clicks on the **Save** button many times, this pattern will gladly render as many popups as there are clicks, which doesn’t make much sense.

The **publish-subscribe (pub/sub)** pattern is a different type of event-driven programming. In this case, we can write multiple handlers to all simultaneously act on a given event’s result. Let’s say that your app just received some updated data. The publisher goes through its list of subscribers and passes the updated data to each.

Refer to the following diagram on how the updated data event triggers multiple functions:

- An `updateCache` function updates your local cache with new data
- A `fetchDetails` function retrieves further details about the data from the server
- A `showToastMessage` function informs the user that the app just received new data

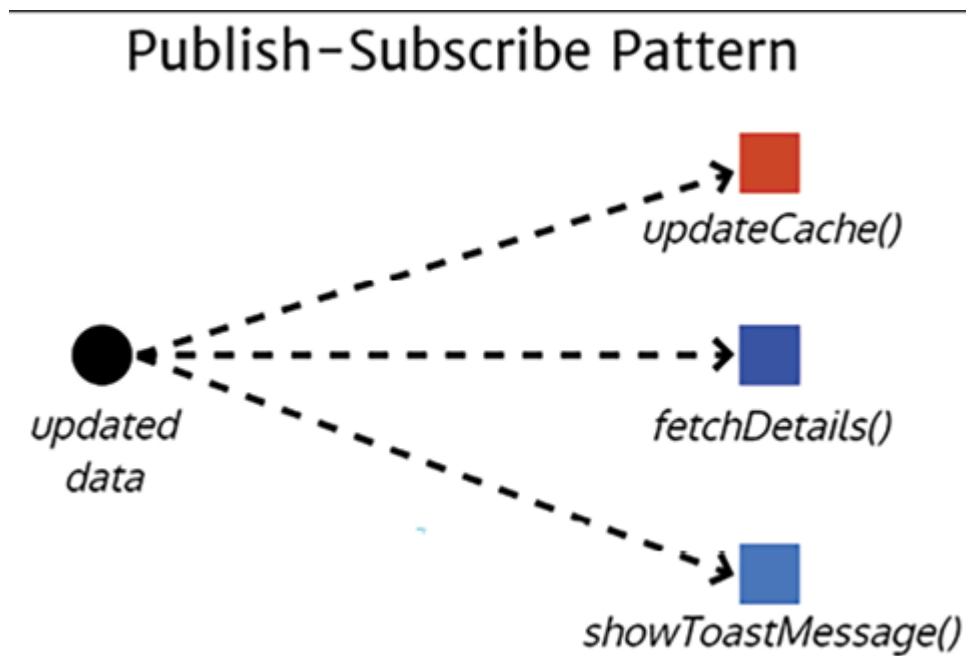


Figure 1.13: Pub/sub pattern implementation

All these events can happen asynchronously; however, the `fetchDetails` and `showToastMessage` functions will receive more data than they need, and it can get convoluted to try to compose these events in different ways to modify application behavior.

In reactive programming, everything is treated as a stream. A stream will contain events that happen over time, which can contain some or no data. The following diagram visualizes a scenario where your app is listening for mouse clicks from the user. Uncontrolled streams

of user clicks are meaningless. You exert some control over this stream by applying the `throttle` function, so you only get updates every 250 **milliseconds (ms)**. If you subscribe to this new event stream, every 250 ms, you will receive a list of click events. You may try to extract some data from each click event, but in this case, you're only interested in the number of click events that happened. Using the `map` function, we can shape the raw event data into the sum of all clicks.

Further down the stream, we may only be interested in listening for events with two or more clicks, so we can use the `filter` function to act only on what is essentially a double-click event. Every time our filter event fires, it means that the user intended to double-click, and you can act on that information by popping up an alert.

The true power of streams comes from the fact that you can choose to act on the event at any time as it passes through various control, transformation, and filter functions. You can choose to display click data on an HTML list using `@for` and Angular's `async` pipe so that the user can monitor the types of click data being captured every 250 ms.

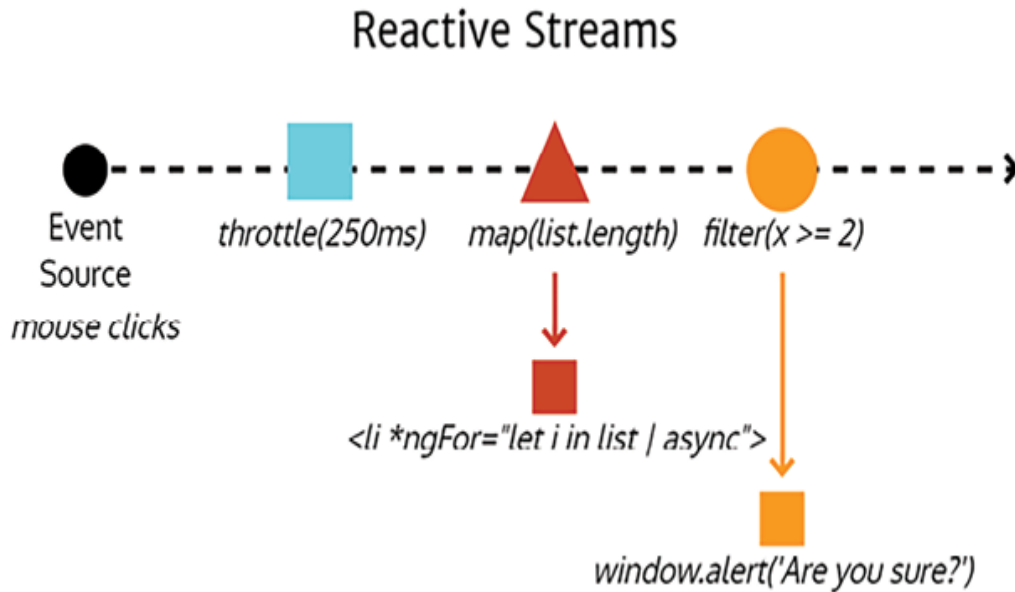


Figure 1.14: A reactive data stream implementation

Now let's consider some more advanced Angular architectural patterns.

Modular architecture

As mentioned earlier in the *Component architecture* section, if you create an `NgModule` project, Angular components, services, and dependencies are organized into modules. Angular apps are bootstrapped via their root module, as shown in the diagram that follows:

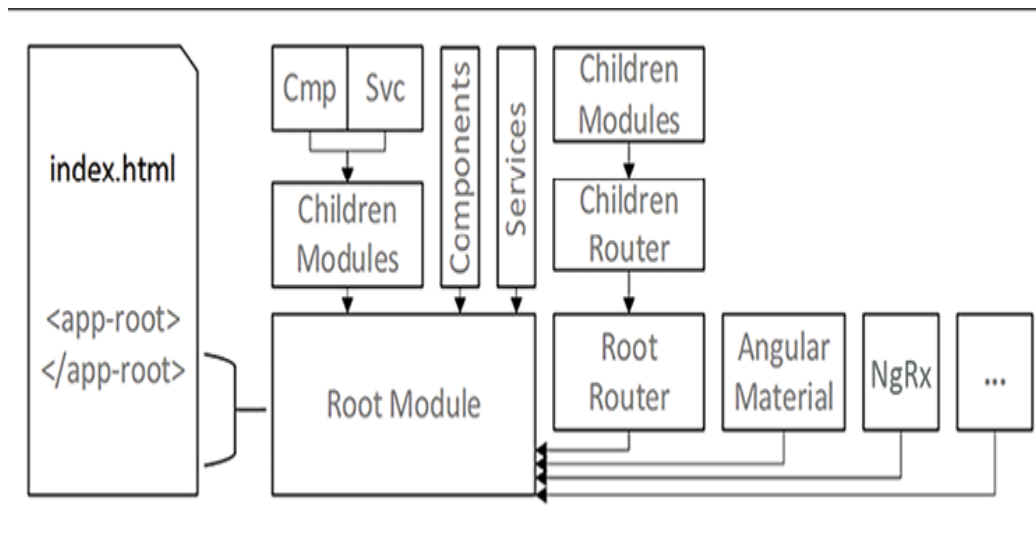


Figure 1.15: Angular Bootstrap process showing major architectural elements

The root module can import other modules, declare components, and provide services. As your application grows, you must create sub-modules containing their components and services. Organizing your application in this manner allows you to implement lazy loading, allowing you to control which parts of your application get delivered to the browser and when. As you add more features to your application, you import modules from other libraries, like Angular Material or NgRx. You implement the router to enable rich navigational experiences between your components, allowing your routing configuration to orchestrate the creation of components.

Chapter 4, Creating a Router-First Line-of-Business App, introduces router-first architecture, where I encourage you to start developing your application by creating all your routes ahead of time.

In Angular, services are provided as singletons to a module by default. You'll quickly get used to this behavior. However, you must remember that if you provide the same service across multiple modules, each module has its own instance of the provided service. In the case of an authentication service, where we wish to have only one instance across our entire application, you must be careful to provide that instance of the authentication service only at the root module level. Any service, component, or module provided at the root level of your application becomes available in the feature module.

Standalone architecture

If you create a standalone project, your dependencies will be provided at the root level `bootstrapApplication` function. First-party and third-party libraries are updated to expose provider functions instead of modules. These provider functions are inherently tree-shakable, meaning the framework can remove them from the final package if unused. The provider functions can be customized using “with” functions, where a function named `withFeature()` can enable a certain feature.

In standalone projects and while using standalone components in general, we must explicitly import the features they use that are not included in the providers. This means pipes, directives (including fundamental directives like `*ngIf` -- unless you're using `@if`, of course), and child components must be provided. This can feel more verbose and restrictive than an NgModule project, but the long-term benefits outweigh the short-term pain. The better information we can

provide to the framework about our projects, the better the framework can optimize our code and improve performance.

You can migrate existing NgModule projects to a standalone project using the following command:

```
$ npx ng g @angular/core:standalone
```

Beware - this is not a foolproof or entirely automated process. Read about it more at <https://angular.dev/reference/migrations/standalone>.

The router is the next most powerful technology you must master in Angular.

Angular Router

The Angular Router, shipped in the `@angular/router` package, is a central and critical part of building SPAs that act and behave like regular websites that are easy to navigate, using browser controls or the zoom or micro zoom controls.

The Angular Router has advanced features such as lazy loading, router outlets, auxiliary routes, smart active link tracking, and the ability to be expressed as an `href`, which enables a highly flexible Router-first app architecture leveraging stateless data-driven components, using RxJS `BehaviorSubject` or a `signal`.

A class (a component or a service in Angular) is stateless if it doesn't rely on instance variables in executing any of its behavior (via functions or property getters/setters). A class is data-driven when it's used to manage access to data. A stateless data-driven component can hold references to data objects and allow access to them (including mutations via functions) but would not store any bookkeeping or state information in a variable.

Large teams can work against a single code base, with each team responsible for a module's development, without stepping on each other's toes while enabling easy continuous integration. With its billions of lines of code, Google works against a single code base for a very good reason: integration after the fact is very expensive.

Small teams can remix their UI layouts on the fly to quickly respond to changes without having to rearchitect their code. It is easy to underestimate the time wasted due to late-game changes in layout or navigation. Such changes are easier for larger teams to absorb but costly for small teams.

Consider the following diagram; first off, depending on the bootstrap configuration, the app will either be a standalone or `NgModule` project. Regardless, you'll define a `rootRouter` at the root of your application; components `a`, `master`, and `detail`; services; pipes; directives; and other modules will be provided. All these components will be parsed and eagerly loaded by the browser when a user first navigates to your application.

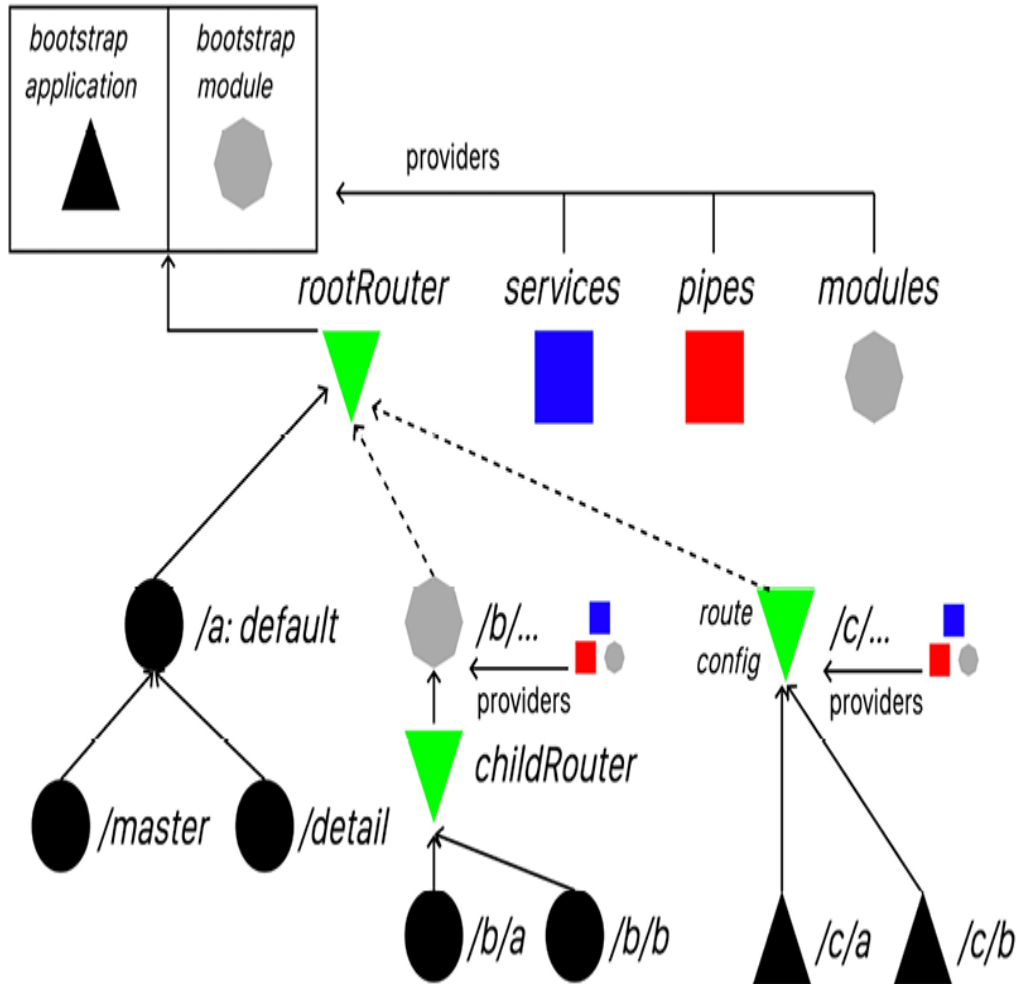


Figure 1.16: Angular architecture

If you were to implement a lazily loaded route, `/b`, you would need to create a feature module named `b`, which would have its `childRouter`; components `/b/a` and `/b/b`; services; pipes; directives; and other modules provided for it. During transpilation, Angular will package these components into a separate file or bundle, and this bundle will only be downloaded, parsed, and loaded if the user ever navigates to a path under `/b`.

In a standalone project, you can lazy load other standalone components represented by the triangles. You can organize

components in a route configuration file. The `/c/a` and `/c/b` components will have access to providers at the root level. You may provide an **environment injector** for a specific component in the route config file. Practically speaking, this is only useful if you want to provide a service only ever used by that component or one with a specific scope, e.g., a state that's only used by that component. In contrast to a `NgModule` app, you will have to declare the modules you're using in each component granularly. However, unlike an `NgModule` app, root-level providers not used by any component are tree-shakable. The combination of these two properties results in a small app bundle, and given each module can be individually lazy loaded, the size of each bundle will be smaller as well, leading to better overall performance.

Let's investigate lazy loading in more detail.

Lazy loading

The dashed line connecting `/b/...` to `rootRouter` demonstrates how lazy loading works. Lazy loading allows developers to achieve a sub-second First Meaningful Paint quickly. By deferring the loading of additional modules, we can keep the bundle size delivered to the browser to a minimum. The size of a module negatively impacts download and loading speeds because the more a browser has to do, the longer it takes for a user to see the app's first screen. By defining lazily loaded modules, each module is packaged as separate files, which can be downloaded and loaded individually and on demand.

The Angular Router provides smart active link tracking, which results in a superior developer and user experience, making it very easy to implement highlighting features to indicate to the user the current tab or portion of the currently active app. Auxiliary routes maximize components' reuse and help easily pull off complicated state transitions. With auxiliary routes, you can render multiple master and detail views using only a single outer template. You can also control how the route is displayed to the user in the browser's URL bar and compose routes using `routerLink` in the template and `Router.navigate` in the component class, driving complicated scenarios.

In *Chapter 4, Creating a Router-First Line-of-Business App*, I cover implementing router basics, and advanced recipes are covered in *Chapter 8, Recipes – Reusability, Forms, and Caching*.

Beyond routing, state management is another crucial concept to master if you want to build sophisticated Angular applications.

State management

An `EcmaScript` class backs every component and service in Angular. When instantiated, a class becomes an object in memory. As you work with an object, if you store values in object properties, you're introducing state to your Angular application. If unmanaged, the state becomes a significant liability to the success and maintainability of your application.

I'm a fan of stateless design both in the backend and frontend. From my perspective, state is evil, and you should pay careful attention to

not introduce state into your code. Earlier, we discussed how services in Angular are singletons by default. This is a terrible opportunity to introduce state to your application. You must avoid storing information in your services. In *Chapter 4, Creating a Router-First Line-of-Business App*, I introduce you to `readonly BehaviorSubject`, which acts as a data anchor for your application. In this case, we store these anchors in services to share them across components to synchronize data. The data anchor is a reference to the data instead of a copy. The service doesn't store any metadata or do any bookkeeping.

In Angular components, the class is a ViewModel acting as the glue code between your code and the template. Components are relatively short-lived compared to services, and it is okay to use object properties in this context.

However, beyond design, there are specific use cases for introducing robust mechanisms to maintain complicated data models in the state of your application. **Progressive web applications (PWA)** and mobile applications are cases where connectivity is not guaranteed. In these cases, being able to save and resume the entire state of your application is a must to provide a great UX for your end user.

The NgRx library for Angular leverages the Flux pattern to enable sophisticated state management for your applications. In *Chapter 2, Forms, Observables, Signals, and Subjects*, and *Chapter 9, Recipes – Master/Detail, Data Tables, and NgRx*, I provide alternative implementations for various features using NgRx to demonstrate the differences in implementation between more lightweight methods.

The Flux pattern

Flux is the application architecture created by Facebook to assist in building client-side web applications. The Flux pattern defines a series of components that manage a store that stores the state of your application, via dispatchers that trigger/handle actions and view functions that read values from the store. Using the Flux pattern, you keep the state of your application in a store where access to the store is only possible through well-defined and decoupled functions, resulting in architecture that scales well because, in isolation, decoupled functions are easy to reason with and write automated unit tests for.

Consider the diagram that follows to understand the flow of information between these components:

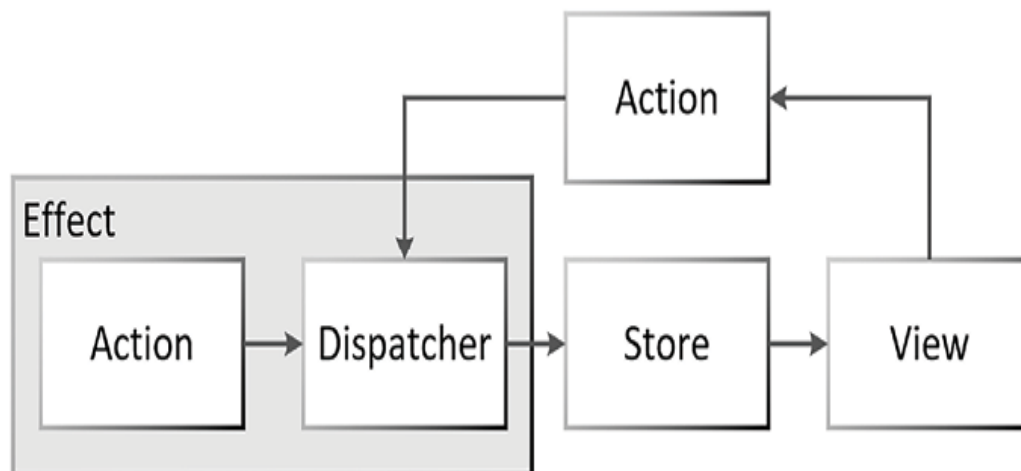


Figure 1.17: NgRx data flow

NgRx implements the Flux pattern in Angular using RxJS.

NgRx

The NgRx library brings Redux-like (a popular React.js library) reactive state management to Angular based on RxJS. State management with NgRx allows developers to write atomic, self-contained, and composable pieces of code, creating actions, reducers, and selectors. This kind of reactive programming allows side effects in state changes to be isolated and feels right at home with the general coding patterns of React.js. NgRx creates an abstraction layer over already complex and sophisticated tooling like RxJS.

There are excellent reasons to use NgRx, such as if you deal with 3+ input streams in your application. In such a scenario, the overhead of dealing with so many events makes it worthwhile to introduce a new coding paradigm to your project. However, most applications only have two input streams: REST APIs and user input. NgRx may make sense for offline-first **Progressive Web Apps (PWAs)**, where you may have to persist and restore complicated state information (or niche enterprise apps with similar needs).

Here's an architectural overview of NgRx:

Angular Application

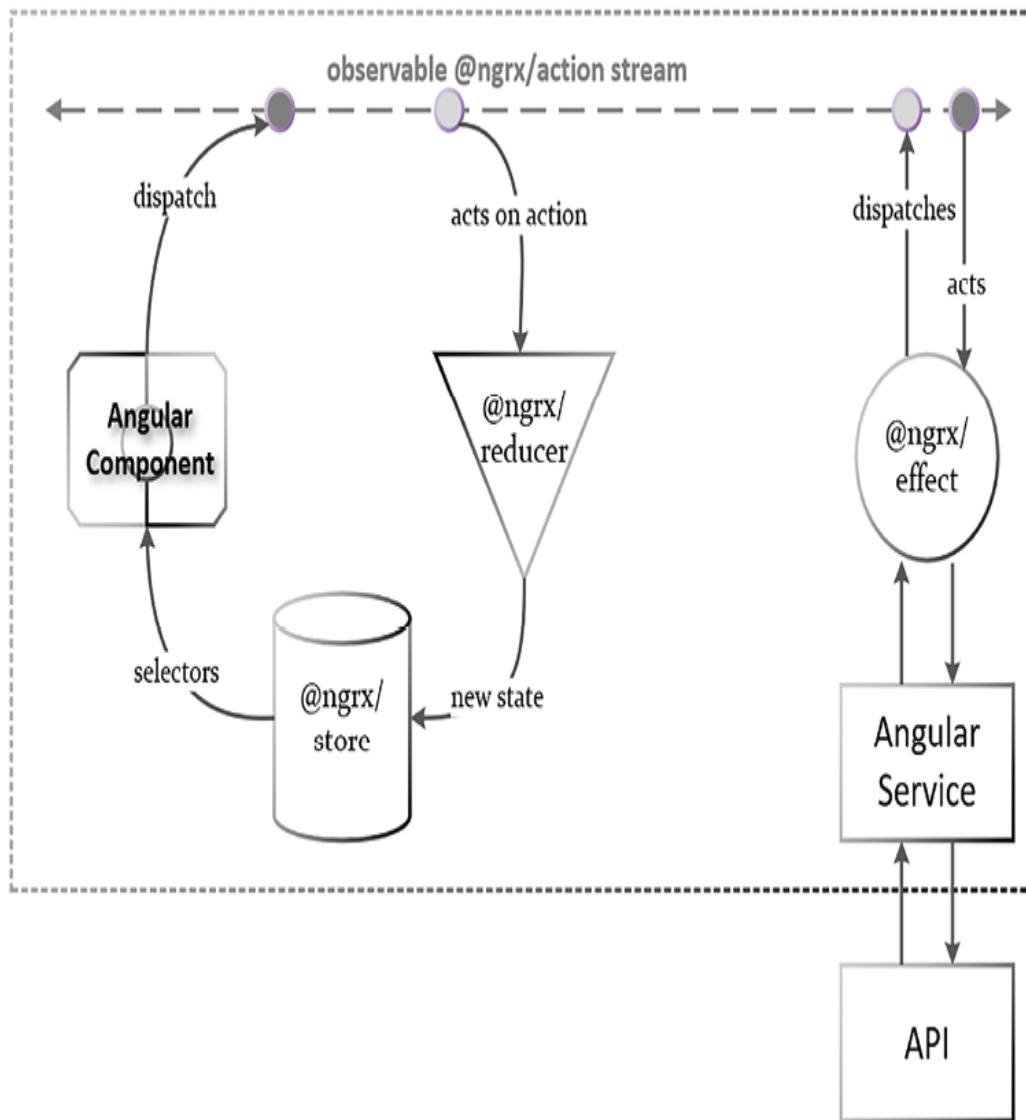


Figure 1.18: NgRx architectural overview

Consider the very top of the diagram as an observable action stream, where actions can be dispatched and acted upon as denoted by the circles. Effects and components can dispatch an action. Reducers and effects can act upon these actions to either store values in the store or trigger an interaction with the server. Selectors are leveraged by components to read values from the store.

Given my positive attitude toward minimal tooling and a lack of definite necessity for NgRx beyond the niche audiences previously mentioned, I do not recommend NgRx as a default choice.

`RxJS/BehaviorSubject` are powerful and capable enough to unlock sophisticated and scalable patterns to help you build great Angular applications, as is demonstrated in the chapters that lead up to *Chapter 9, Recipes – Master/Detail, Data Tables, and NgRx*.

You can read more about NgRx at <https://ngrx.io>.

NgRx component store

The NgRx component store, with the package name `@ngrx/component-store`, is a library that aims to simplify state management by targeting local/component states. It is an alternative to a reactive push-based subject-in-a-service approach. For scenarios where the state of a component is only changed by the component itself or a small collection of components, you can improve the testability, complexity, and performance of your code by using this library.

In contrast to global-state solutions like NgRx, the NgRx component store, with its limited scope, can automatically clear itself when its associated view is detached from the component tree. Unlike a singleton service, you can have multiple instances of a component store, enabling distinct states for different components. Additionally, the conceptual model for the component store is straightforward. One only needs to grasp the select, updater, and effect concepts, all operating within a confined scope. Hence, for those crafting a standalone Angular app or seeking component-specific storage, the

NgRx component store provides a sustainable and easily testable approach.

You can find out more about the NgRx component store at <https://ngrx.io/guide/component-store>.

React.js architecture

In contrast to Angular, React.js implements the Flux pattern holistically. Following is a router-centric view of a React application, where components/containers and providers are represented in a strict tree-like manner.

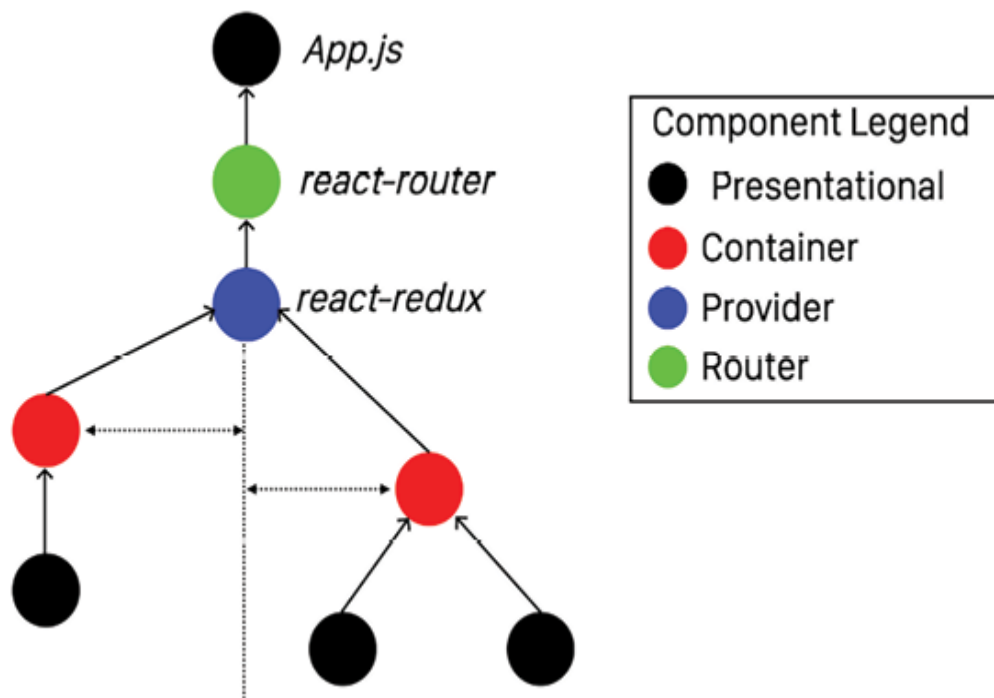


Figure 1.19: React.js architectural overview

In the initial releases of React, one had to laboriously pass values up/down the inheritance tree of every component for even the most

basic functionality to work. Later, `react-redux` was introduced, so each component can read/write values directly to the store without traversing the tree.

This basic overview should give you a sense of the significant architectural differences between Angular and React. However, keep in mind that just like Angular, React's community, patterns, and practices are continually evolving and getting better over time.

If you dig simplicity, check out Vue. It. Is. Simple. In a good way: <https://vuejs.org>.

You can learn more about React at <https://reactjs.org>.

Future of Angular

One of the biggest benefits of Angular is that you can count on major releases every 6 months. However, with a regular cadence comes the pressure to release meaningful and splashy updates with every major release. We can probably blame Google for creating this pressure. If you're not constantly producing, you're out. This has an unfortunate side effect of new features being released in preview or an unfinished state. While an argument can be made that releasing upcoming features in preview allows for feedback to be collected from the developer community, no guarantees are made that performance regressions will not be introduced.

If your team is not consuming every bit of Angular news coming out regularly, you may miss these nuances and roll out code into

production that negatively impacts your business, potentially impacting revenue. For example, some users have noticed performance regressions in Angular 16, and the Angular team knew about this and fixed it in Angular 17, but this posture puts businesses who've taken up the new version at risk.

The ambitious Angular Elements feature best exemplifies another aspect of this. Circa Angular 9, a big deal was made when announcing web component support for Angular. The promise was that you could create universally reusable components using your favorite framework. The team highlighted the great challenge of shipping a pared-down version of Angular along with the component – reducing the framework size from 150 KB to only a few KB. Instead of focusing on finalizing this feature, and despite making great incremental process, the team has found the task too daunting. So the team has moved on to different ideas to tackle this problem. But even those new ideas are being rushed and rolled out in a preview state, e.g., Angular signals adding to the pile of unfinished work in production software. In Angular 17, signals are partially out of preview and have the potential to transform how Angular apps are built in the future with the implementation of signal-based components. Signals do not easily leak memory compared to RxJS's leaky subscription concept. Signals can also work with async/await calls, avoiding many unnatural uses of reactive coding with RxJS. The stable delivery of all these features is probably due in Angular 19.

Find out up-to-date information about upcoming and in-preview features at

<https://angular.dev/roadmap>.

A large Angular application suffers from crippling performance issues just like Angular.js did, except the goalposts around the definition of *large* have moved significantly. The major trouble here is that it's impossible to resolve these performance issues, at least not without significant engineering investment that leaves you digging under the hood of the Angular rendering engine.

Further, in 2023, by leveraging ES2022 features, it is possible to build reactive and interactive web applications using pure JavaScript. Angular signals expose these ES2022 features to enable fine-grained reactivity by replacing Zone.js with native JavaScript. This means that only the parts of the DOM that need to get updated are updated, significantly reducing render times. This is a topic I further explore in *Chapter 3, Architecting an Enterprise App*. Combining these changes results in a more optimized change detection cycle, resulting in smoother FPS.

Every release of Angular seeks to improve **Time-to-Interactive (TTI)** for modern browsers. In the past, this meant improving bundle sizes, introducing lazy loading of modules, and now individual components. Angular now supports **Server-side Rendering (SSR)** with non-destructive hydration. This means that a server can compute the DOM of a view and transfer it to the client, and the client can update the DOM displayed to the user without completely replacing it.

Angular is also moving away from Jasmine to Jest. Jasmine has always been a great unit-testing framework. However, making it

work in a web application context always requires a lot of configuration and additional tools like Karma to execute the tests and get coverage reports. Jest includes all these features. The support is currently experimental, and it's unclear whether Vitest will be a better option than Jest. Angular is moving away from webpack to esbuild, which is about 40x faster than webpack. Once again, it is only available as a (developer) preview.

As you can see, some of the most exciting things happening in Angular are in preview features. The ground truth is that teams are heads down, working on delivering features for their projects and trying their best to keep up with all the latest changes. It's tough enough to keep updating dependencies continually; big changes in the mental model of the framework, combined with performance issues, risk losing the confidence of developers and businesses alike. Trust is hard to build and easy to lose.

The reality is the Angular team is doing great work, and the framework is making the necessary changes to evolve and meet ever-growing expectations. It bears repeating Google mandates that the 2,000+ Angular projects they have must all be on the same version of Angular. This means that every new update to Angular is well-tested, and there are no backward compatibility surprises.

Angular remains an exciting, agile, and capable framework. My motivation is to inform you of where the land mines are. I hope you are as excited as I am about the state of modern web development and the future possibilities it unlocks. Buckle up your seatbelt, Dorothy, 'cause Kansas is going bye-bye.

Summary

In summary, web technologies have evolved to a point where it is possible to create rich, fast, and native web applications that can run well on the vast majority of desktop and mobile browsers deployed today. Angular has become a mature and stable platform, applying lessons learned from the past. It enables sophisticated development methodologies that enable developers to create maintainable, interactive, and fast applications using technologies like TypeScript, RxJS, and NgRx-enabled patterns from object-oriented programming, reactive programming, the Flux pattern, and standalone components, along with the NgRx component store.

Angular is meant to be consumed in an evergreen manner, so it is a great idea always to keep your Angular up to date. Visit <https://AngularForEnterprise.com> for the latest updates and news.

Angular is engineered to be reactive through and through; therefore, you must adjust your programming style to fit this pattern. With signals, Angular even gains fine-grained reactivity. However, presentation layer reactivity is not the same as reactive programming. When signal-based components arrive circa Angular 19, Angular will no longer require reactive programming to achieve a reactive presentation layer. In *Chapter 9, Recipes – Master/Detail, Data Tables, and NgRx*, I provide an example of a nearly observable and subscription-free application using signals and NgRx SignalStore to show what's possible with Angular 17. Until then, the official documentation should be your bible, found at <https://angular.dev>.

In the next chapter, we will review the LocalCast Weather app as a standalone app; you will learn about capturing user input with reactive forms, keeping components decoupled, enabling data exchange between them using `BehaviorSubject` and how the NgRx component store and Angular signals differ from these concepts. In the following chapters, you will learn about advanced architectural patterns to create scalable applications and how your Angular frontend works within the context of a full-stack TypeScript application using minimal MEAN. The book wraps up by introducing you to DevOps and continuous integration techniques to publish your apps.

Further reading

- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994, Addison Wesley, ISBN 0-201-63361-2.
- *Human JavaScript*, Henrik Joreteg, 2013, <http://read.humanjavascript.com>.
- *What's new in TypeScript x MS Build 2017*, Anders Hejlsberg, 2017, <https://www.youtube.com/watch?v=0sMZJ02rs2c>.
- *The Pragmatic Programmer, 20th Anniversary Edition*, David Thomas and Andrew Hunt, 2019, Addison Wesley, ISBN 978-0135957059.
- *Thinking Reactively: Most Difficult*, Mike Pearson, 2019, <https://www.youtube.com/watch?v=-4cwkHNguXE>.
- *Data Composition with RxJS*, Deborah Kurata, 2019, <https://www.youtube.com/watch?v=Z76QlSpYcck>.

- *Flux Pattern In-Depth Overview*, Facebook, 2019, <https://facebook.github.io/flux/docs/in-depth-overview>.
- *Developer experience: What is it and why should you care?*, GitHub, 2023, <https://github.blog/2023-06-08-developer-experience-what-is-it-and-why-should-you-care>.
- *Standalone Components*, Google, 2023, <https://angular.dev/reference/migrations/standalone>.
- *Built-in control flow*, Google, 2023, <https://angular.dev/guide/templates/control-flow>.

Questions

Answer the following questions as best as possible to ensure you've understood the key concepts from this chapter without googling anything. Do you know if you got all the answers right? Visit <https://angularforenterprise.com/self-assessment> for more:

1. What is the difference between a standalone and an NgModule project?
2. What is the concept behind Angular Evergreen?
3. Using the double-click example for reactive streams, implement the following steps using RxJS: listen to click events from an HTML target with the `fromEvent` function. Determine whether the mouse was double-clicked within a 250 ms timeframe using the `throttleTime`, `asyncScheduler`, `buffer`, and `filter` operators. If a double-click is detected, display an alert in the

browser. Hint: use <https://stackblitz.com> or implement your code and use <https://rxjs.dev/> for help.

4. What is NgRx, and what role does it play in an Angular application?
5. What is the difference between a module, a component, and a service in Angular?

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:

<https://packt.link/AngularEnterprise3e>



2

Forms, Observables, Signals, and Subjects

In this chapter, we'll work on a simple weather app, **LocalCast Weather**, using Angular and a third-party web API from [OpenWeatherMap.org](https://openweathermap.org). The source code for this project is provided on GitHub at <https://github.com/duluca/local-weather-app>, including various stages of development in the `projects` folder.

If you've never used Angular before and need an introduction to Angular essentials, I recommend checking out *What is Angular?* on Angular.dev at <https://angular.dev/overview> and going through the *Learn Angular Tutorial* at <https://angular.dev/tutorials/learn-angular>.

Feeling brave? Just type the following into your terminal:

```
$ npm create @angular
```


LocalCast Weather is a simple app that demonstrates the essential elements that make up an Angular application, such as components, standalone components, modules, providers, pipes, services, RxJS, unit testing, e2e using Cypress, environment variables, Angular Material, and **Continuous Integration** and **Continuous Delivery (CI/CD)** pipelines leveraging CircleCI.

I've created a Kanban board for this project on GitHub. You can access it at the following link to get more context about the project:

<https://github.com/users/duluca/projects/1>.

A Kanban board is a great way to document your plans for building an app. I touch on the importance of building a roadmap and creating information radiators for the status of your project in *Chapter 3, Architecting an Enterprise App*.

An information radiator is a physical or virtual display that is easily visible or accessible, conveying key information about a project or process. It typically includes metrics, progress charts, or status indicators and is designed to provide at-a-glance awareness without requiring the viewer to seek out information. The goal of an information radiator is to promote transparency, facilitate communication among team members, and enable stakeholders to get updates without interrupting the team's workflow.

As a bonus, I created a rudimentary wiki page on my repository at <https://github.com/duluca/local-weather-app/wiki>. Note that you can't upload images to `README.md` or wiki pages. To get around this limitation, you can create a new issue, upload an image in a comment, and copy and paste the URL for it to embed images in `README.md` or wiki pages. In the sample wiki, I followed this technique to embed the wireframe design into the page.

The source code for the sample projects in the book is divided into **stages** to capture snapshots of various states of development. In this chapter, we pick up the app development from `stage5` and evolve it into `stage6`. In `stage5`, the app is polished, but it can only pull weather information for one city, which is hardcoded into the app. As a result, it is not a very useful app.

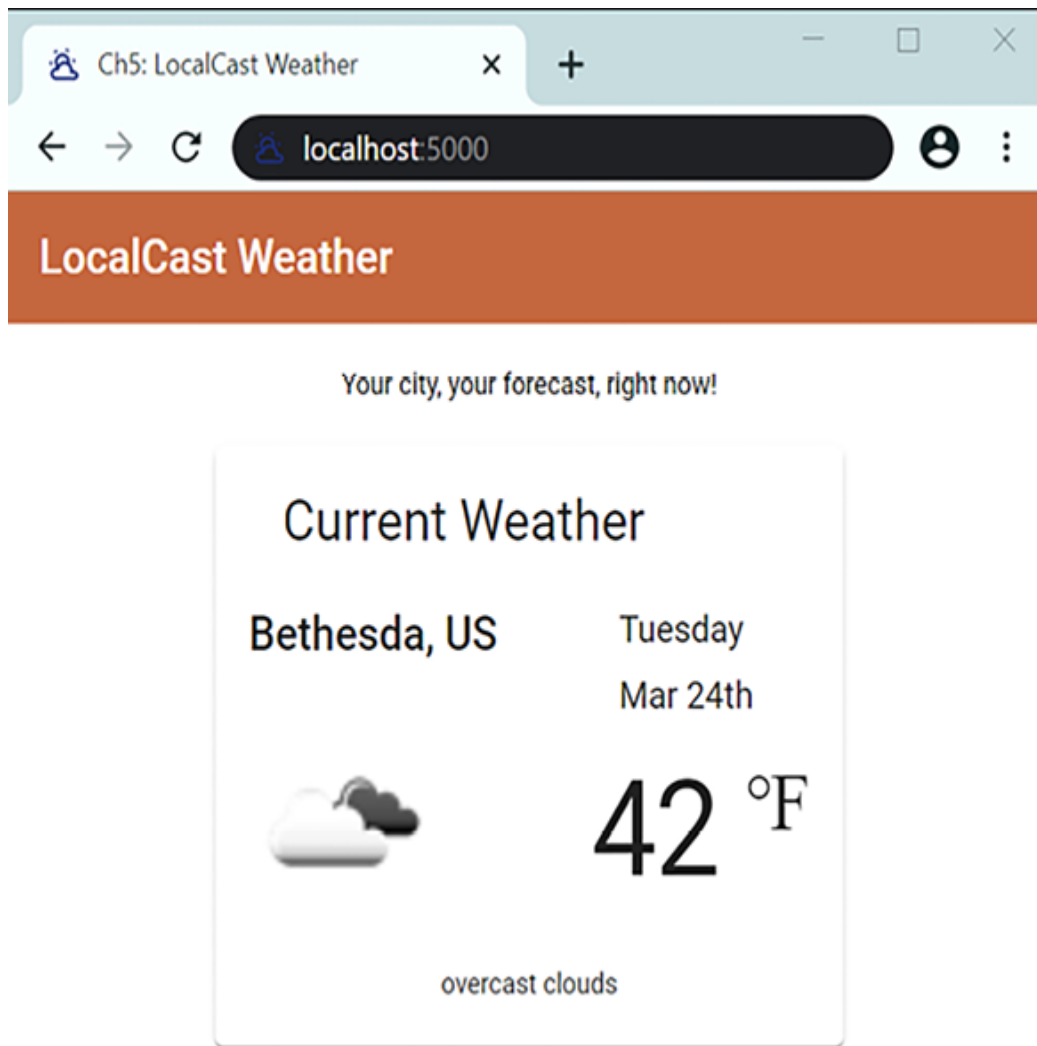


Figure 2.1: The LocalCast Weather app as in projects/stage5

You will inherit an existing project that is not interactive yet. To build an interactive app, we need to be able to handle user input. Enabling user input in your application opens possibilities for creating great user experiences.

We will cover the following main topics in this chapter:

- Great UX should drive implementation
- Reactive forms versus template-driven forms

- Component interaction with observables and RxJS/BehaviorSubject
- Managing subscriptions and memory leaks
- Coding in the reactive paradigm
- Chaining API calls
- Using Angular Signals
- Generating apps with ChatGPT

Technical requirements

The most up-to-date versions of the sample code for the book are on GitHub at the repository linked shortly. The repository contains the final and completed state of the code. You can verify your progress at the end of this chapter by looking for the end-of-chapter snapshot of code under the `projects` folder.

For *Chapter 2*:

1. Clone the <https://github.com/duluca/local-weather-app> repo.
2. Execute `npm install` on the root folder to install dependencies.
3. The beginning state of the project is reflected at:

```
projects/stage5
```

4. The end state of the project is reflected at:

```
projects/stage6
```

5. Add the stage name to any `ng` command to act only on that stage:

```
npx ng build stage6
```

Note that the `dist/stage6` folder at the root of the repository will contain the compiled result.

Beware that the source code provided in the book and the version on GitHub are likely to be different. The ecosystem around these projects is ever evolving. Between changes to how the Angular CLI generates new code, bug fixes, new versions of libraries, and side-by-side implementations of multiple techniques, there's a lot of variation impossible to account for. If you find errors or have questions, please create an issue or submit a pull request on GitHub.

By the end of the chapter, you should be comfortable leveraging observables and signals to build apps that provide a great UX. As a bonus, I'll touch on how you can leverage **Generative AI (GenAI)** tools like ChatGPT (<https://chat.openai.com/>) to build quick prototypes. But first, let's get back to UX because no matter how much you run, crawl, or scale the city walls, if you nail the UX, your app will be loved; but if you miss the mark, your app will be a dime a dozen.

Great UX should drive implementation

Creating an easy-to-use and rich **User Experience (UX)** should be your main goal. You shouldn't pick a design just because it's easiest to implement. However, often, you'll find a great UX that is simple to implement in the front end of your app but a lot more difficult on the back end. Consider google.com's landing page:

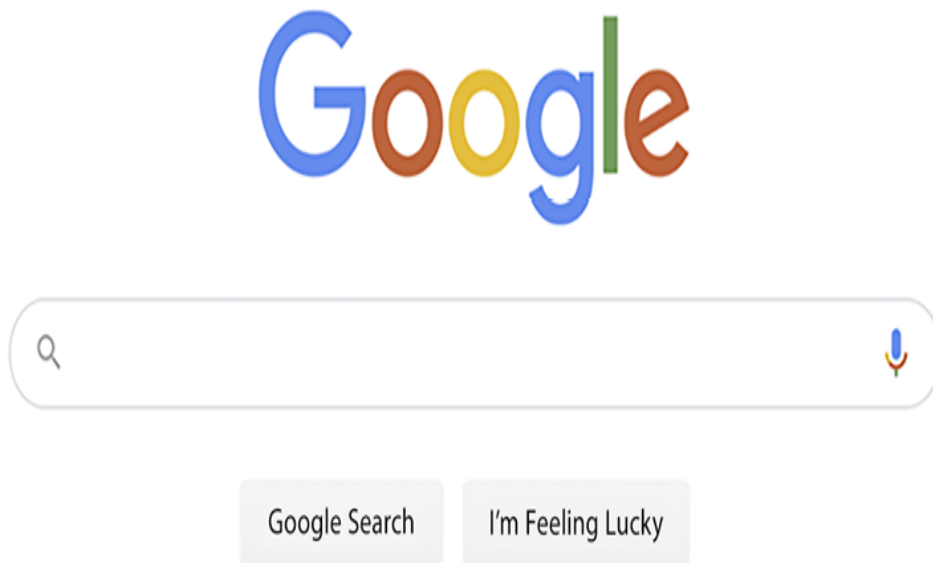


Figure 2.2: Google's landing page

In this context, Google Search is just a simple input field with two buttons. Easy to build, *right*? That simple input field unlocks some of the world's most sophisticated and advanced software technologies backed by a global infrastructure of custom-built data centers and **Artificial Intelligence (AI)**. It is a deceptively simple and insanely powerful way to interact with users. You can augment user input by leveraging modern web APIs like **GeoLocation** and add critical

context to derive new meaning from user input. So, when the user types in `Paris`, you don't have to guess whether they mean Paris, France, or Paris, Texas, or whether you should show the current temperature in Celsius or Fahrenheit. With `LocalStorage`, you can cache user credentials and remember user preferences to enable dark mode in your app.

In this book, we won't be implementing an AI-driven super app, but we will enable users to search for their cities using a city name or postal code (often called *zip codes* in the US). Once you realize how complicated it can get to implement something as seemingly simple as a search by postal code, you may gain a new appreciation for well-designed web apps.

To accomplish the UX goal, we need to build a UI centered around an input field. To do this, we need to leverage Angular forms with validation messages to create engaging search experiences with *search-as-you-type* functionality.

Behind the scenes, `RxJS/BehaviorSubject` or `signals` enables us to build decoupled components that can communicate with one another, and a reactive data stream allows us to merge data from multiple web APIs without increasing the complexity of our app. In addition, you will be introduced to Angular Signals and see how it differs from RxJS.

Next, let's see how to implement an input field using forms. Forms are the primary mechanism that we need to capture user input. In Angular, there are two kinds of forms: **reactive** and **template-driven**. We need to cover both techniques so that you're familiar with how forms work in Angular.

Reactive versus template-driven forms

Now, we'll implement the search bar on the home screen of the application. The next user story states **Display forecast information for current location**, which may be taken to imply an inherent GeoLocation functionality. However, as you may note, GeoLocation is a separate task. The challenge is that with native platform features such as GeoLocation, you are never guaranteed to receive the actual location information. This may be due to signal loss issues on mobile devices, or the user may simply refuse to give permission to share their location information.

First and foremost, we must deliver a good baseline UX and implement value-added functionality such as GeoLocation only afterward. In **stage5**, the status of the project is represented on the Kanban board, as captured in the following snapshot:

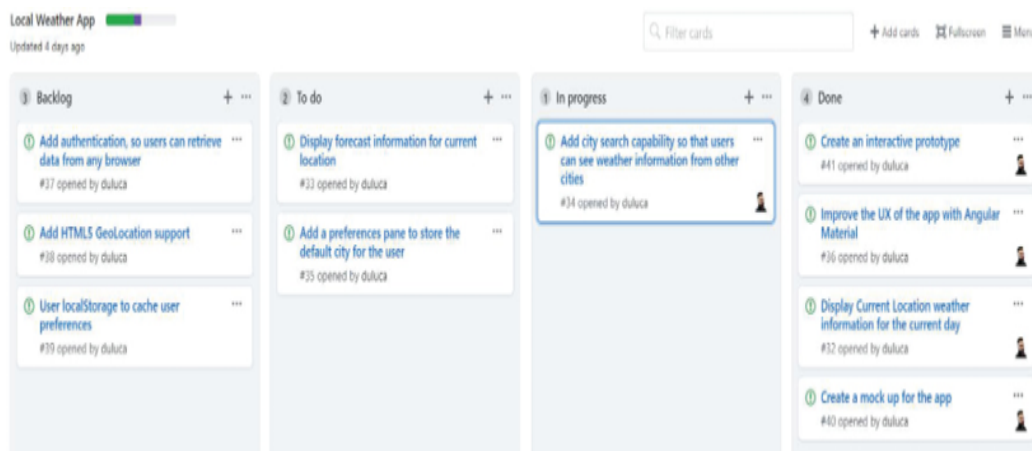


Figure 2.3: GitHub project Kanban board

We'll implement the **Add city search capability** card (which captures a user story), as shown in the **In progress** column. As part

of this story, we are going to implement a search-as-you-type functionality while providing feedback to the user if the service is unable to retrieve the expected data.

Initially, it may be intuitive to implement a type-search mechanism; however, `OpenWeatherMap` APIs don't provide such an endpoint. Instead, they provide bulk data downloads, which are costly and are in the multiples-of-megabytes range.

We will need to implement our application server to expose such an endpoint so that our app can effectively query while using minimal data.

The free endpoints for `OpenWeatherMap` do pose an interesting challenge, where a two-digit country code may accompany either a city name or zip code for the most accurate results. This is an excellent opportunity to implement a feedback mechanism for the user if more than one result is returned for a given query.

We want every iteration of the app to be a potentially releasable increment and avoid doing too much at any given time.

Before you begin working on a story, it is a good idea to break it into technical tasks. The following is the task breakdown for this story:

1. Add an Angular form control so that we can capture user input events.
2. Use an Angular Material input as documented at <https://material.angular.io/components/input> to improve the UX of the input field.
3. Create the search bar as a separate component to enforce the separation of concerns and a decoupled component architecture.

4. Extend the existing endpoint to accept a zip code and make the country code optional in `weather.service.ts` to make it more intuitive for end users to interact with our app.
5. Throttle requests so that we don't query the API with every keystroke but at an interval where users still get immediate feedback without clicking a separate button.

Let's tackle these tasks over the following few sections.

Adding Angular reactive forms

You may wonder why we're adding Angular forms since we've got just a single input field, not a form with multiple inputs. As a general rule of thumb, any time you add an input field, it should be wrapped in a `<form>` tag. The `Forms` module contains `FormControl` that enables you to write the backing code behind the input field to respond to user inputs and provide the appropriate data or the validation or message in response.

There are two types of forms in Angular:

- **Template-driven forms:** These forms are like what you may be familiar with in the case of AngularJS, where the form logic is mainly inside the HTML template. I'm not a fan of this approach because it is harder to test these behaviors, and fat HTML templates become challenging to maintain quickly.
- **Reactive forms:** The behavior of reactive forms is driven by TypeScript code in the controller. This means that your validation logic can be unit tested and, better yet, reused across your application. Reactive forms are the core technology that, in

the future, will enable the Angular Material team to write automated tools that can autogenerate an input form based on a TypeScript interface.

Read more about reactive forms at
<https://angular.dev/guide/forms/reactive-forms>.

In Angular, dependencies are encapsulated in modules provided by the framework. User-created modules are no longer mandatory, and our code sample is configured as a standalone app. For the component we define in the next section, you must import `FormsModule` and `ReactiveFormsModule` to be able to use these features in your template.

In a pure reactive form implementation, you only need `ReactiveFormsModule`. Note that `FormsModule` supports template-driven forms and other scenarios where you may only want to declare `FormControl` without `FormGroup`. This is how we implement the input field for this app. `FormGroup` is defined in the next section.

Note that reactive forms allow you to code in the reactive paradigm, which is a net positive when using observables. Next, let's add a city search component to our app.

Adding and verifying components

We will be creating a `citySearch` component using Angular Material form and input modules:

1. Create the new `citySearch` component:

```
$ npx ng g c citySearch
```

2. Import the form dependencies from the previous section and Material dependencies, `MatFormFieldModule` and `MatInputModule`:

```
src/app/city-search/city-search.component.ts
import { FormsModule, ReactiveFormsModule }
from '@angular/forms'
import { MatButtonModule } from
 '@angular/material/button'
import { MatFormFieldModule } from
 '@angular/material/form-field'
import { MatIconModule } from
 '@angular/material/icon'
import { MatInputModule } from
 '@angular/material/input'
...
@Component({
  ...
  standalone: true,
  imports: [
    FormsModule,
    ReactiveFormsModule,
    MatFormFieldModule,
    MatInputModule,
  ],
})
```

```
export class CitySearchComponent
...
```

We're adding `MatFormFieldModule` because each input field should be wrapped in a `<mat-form-field>` tag to get the most out of the Angular Material functionality.

At a high level, `<form>` encapsulates numerous default behaviors for keyboard, screen-reader, and browser extension users; `<mat-form-field>` enables easy two-way data binding, a technique that should be used in moderation, and also allows graceful label, validation, and error message displays.

3. Create a basic template, replacing the existing content:

```
src/app/city-search/city-search.component.html
<form>
  <mat-form-field appearance="outline">
    <mat-label>City Name or Postal Code</mat-label>
    <mat-icon matPrefix>search</mat-icon>
    <input matInput aria-label="City or Zip"
[formControl]="search">
  </mat-form-field>
</form>
```

4. Declare a property named `search` and instantiate it as an instance of `FormControl`:

```
src/app/city-search/city-search.component.ts
import { FormControl } from '@angular/forms'
...
export class CitySearchComponent implements
  OnInit {

  search = new FormControl()
  ...
}
```

Reactive forms have three levels of control:

- `FormControl` is the most basic element with a one-to-one relationship with an input field.
- `FormArray` represents repetitive input fields that represent a collection of objects.
- `FormGroup` registers individual `FormControl` or `FormArray` objects as you add more input fields to a form.

Finally, the `FormBuilder` object is used to orchestrate and maintain the actions of a `FormGroup` object more easily. `FormBuilder` and `FormGroup` are first used in *Chapter 6, Implementing Role-Based Navigation*, and all controls, including `FormArray`, are covered in depth in *Chapter 8, Recipes – Reusability, Forms, and Caching*.

5. In `app.component.ts`, import `CitySearchComponent`, then add `<app-city-search>` as a new `div` in between the row that

contains the tagline of the app and the row that contains `mat-card`:

```
src/app/app.component.ts
template: `
  ...
  </div>
  <div fxLayoutAlign="center">
    <app-city-search></app-city-search>
  </div>
  <div fxLayout="row">
  ...
  `
,
standalone: true,
imports: [
  FlexModule,
  CitySearchComponent,
  ...
],
})
export class AppComponent {
```

6. Launch your app from your terminal:

```
$ npm start
```

7. Test the integration of components by checking out the app in the browser, as shown:

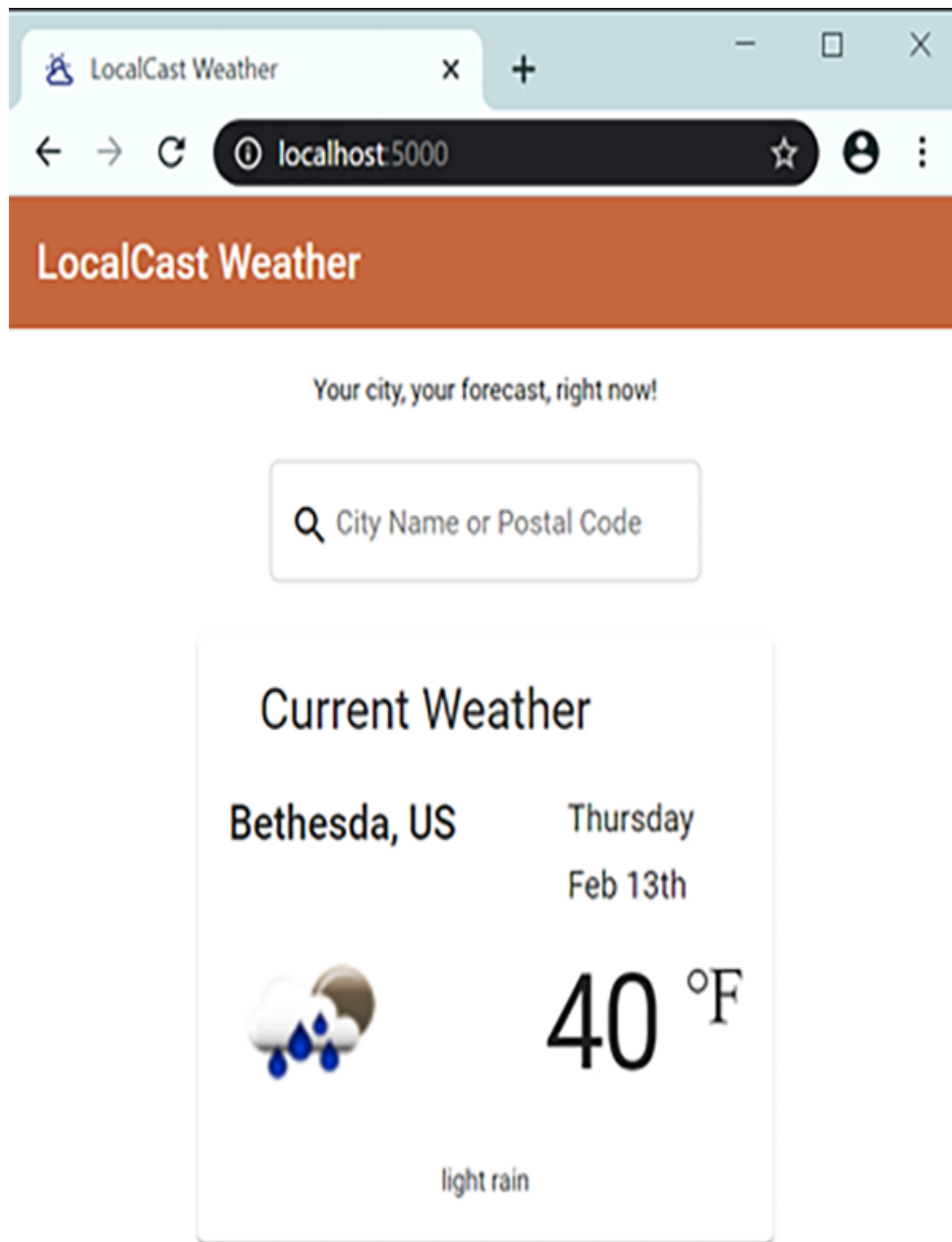


Figure 2.4: The LocalCast Weather app with a search field

If no errors occur, we can start adding the `FormControl` elements and wire them up to a search endpoint.

Adding a search option to the weather service

So far, we have been passing parameters to get the weather for a city using its name and country code. By allowing users to enter zip codes, we must make our service more flexible in accepting both types of inputs.

OpenWeatherMap's API accepts URI parameters, so we can refactor the existing `getCurrentWeather` function using a TypeScript **union type** and a **type guard**.

That means we can supply different parameters while preserving type checking:

1. Refactor the `getCurrentWeather` function in `weather.service.ts` to handle both `zip` and `city` inputs:

```
src/app/weather/weather.service.ts
getCurrentWeather(
  search: string | number,
  country?: string
): Observable<ICurrentWeather> {
  let uriParams = new HttpParams()
  if (typeof search === 'string') {
    uriParams = uriParams.set('q',
      country ? `${search},${country}` :
search
    )
  } else {
    uriParams = uriParams.set('zip',
'search')
```

```

    }

    uriParams = uriParams.set('appid',
environment.appId)
    return this.httpClient
        .get<ICurrentWeatherData>(

`${environment.baseUrl}api.openweathermap.org/
data/2.5/
        weather`,
        { params: uriParams }
    )
    .pipe(map(
        data =>
this.transformToICurrentWeather(data)))
    }

```

We renamed the `city` parameter to `search` since it can be a city name or a zip code. We then allowed its type to be either a `string` or a `number`, and depending on what the type is at runtime, we will either use `q` or `zip`. We also made `country` optional and only append it to the query if it exists.

`getCurrentWeather` now has business logic embedded into it and is thus a good target for unit testing. Following the single responsibility principle from the SOLID principles, we will refactor the HTTP call to its own function, `getCurrentWeatherHelper`.

If you're unfamiliar with SOLID principles, I cover them in the Agile engineering best practices section of *Chapter 3, Architecting an Enterprise App*. If you want to read more about it now, see the Wikipedia entry here:
<https://en.wikipedia.org/wiki/SOLID>.

2. Refactor the HTTP call into `getCurrentWeatherHelper`.

In the next sample, note the use of a backtick character, ```, instead of a single-quote character, `'`, which leverages the template literals' functionality that allows embedded expressions in JavaScript:

```
src/app/weather/weather.service.ts
getCurrentWeather(
  search: string | number,
  country?: string
): Observable<ICurrentWeather> {
  let uriParams = new HttpParams()
  if (typeof search === 'string') {
    uriParams = uriParams.set('q',
      country ? `${search},${country}` :
search
    )
  } else {
    uriParams = uriParams.set('zip',
'search')
  }
}
```

```

    return
  this.getCurrentWeatherHelper(uriParams)
}
private getCurrentWeatherHelper(uriParams:
HttpParams):
  Observable<ICurrentWeather> {
    uriParams = uriParams.set('appid',
environment.appId)
    return this.httpClient
      .get<ICurrentWeatherData>(
        `${environment.baseUrl}api.openweathermap.org/
data/2.5/
        weather`,
        { params: uriParams }
      )
      .pipe(map(
        data =>
this.transformToICurrentWeather(data)))
  }

```

As a positive side effect, `getCurrentWeatherHelper` adheres to the open/closed principle from SOLID. After all, it is open to extension by our ability to change the function's behavior by supplying different `uriParams` and is closed to modification because it won't have to be changed frequently.

To demonstrate the latter point, let's implement a new function to get the current weather by latitude and longitude.

3. Implement `getCurrentWeatherByCoords`:

```
src/app/weather/weather.service.ts
getCurrentWeatherByCoords(coords:
Coordinates): Observable<ICurrentWeather> {
    const uriParams = new HttpParams()
        .set('lat', coords.latitude.toString())
        .set('lon', coords.longitude.toString())
    return
this.getCurrentWeatherHelper(uriParams)
}
```

As you can see, the functionality of `getCurrentWeatherHelper` is extensible without modifying its code.

4. Ensure that you update `IWeatherService` with the changes made earlier:

```
src/app/weather/weather.service.ts
export interface IWeatherService {
    getCurrentWeather(
        search: string | number, country?: string
    ): Observable<ICurrentWeather>
    getCurrentWeatherByCoords(coords:
Coordinates): Observable<ICurrentWeather>
}
```

As a result of adhering to the SOLID design principles, we make it easier to robustly unit test flow-control logic and ultimately write code that is more resilient to bugs and cheaper to maintain.

Implementing a search

Now, let's connect the new service method to the input field:

1. Update `citySearch` to inject `weatherService` and subscribe to input changes:

```
src/app/city-search/city-search.component.ts
import { WeatherService } from
'../weather/weather.service'
...
export class CitySearchComponent implements
OnInit {
  search = new FormControl()

  constructor(private weatherService:
WeatherService) {}
  ...
  ngOnInit(): void {
    this.search.valueChanges.subscribe()
  }
}
```

We are treating all input as `string` at this point. The user input can be a city and zip code, city and country code, or a zip code and country code, separated by a comma. While a city or zip code is required, a country code is optional. We can use the `String.split` function to parse any potential comma-separated input and then trim any whitespace out from the beginning and the end of the string with `String.trim`. We then

ensure that we trim all parts of the string by iterating over them with `Array.map`.

We then deal with the optional parameter with the ternary operator `?:`, only passing in a value if it exists, otherwise leaving it undefined.

2. Implement the search handler:

```
src/app/city-search/city-search.component.ts
this.search.valueChanges
  .subscribe(
    (searchValue: string) => {
      if (searchValue) {
        const userInput =
searchValue.split(',').map(s => s.trim())
        this.weatherService.getCurrentWeather(
          userInput[0],
          userInput.length > 1 ? userInput[1] :
undefined
        ).subscribe(data => (console.log(data)))
      }
    })
```

3. Add a hint for the user under the input field, informing them about the optional country functionality:

```
src/app/city-search/city-search.component.html
...
<mat-form-field appearance="outline">
  ...
```

```
<mat-hint>Specify country code like  
'Paris, US'</mat-hint>  
</mat-form-field>  
...
```

At this point, the subscribe handler will call the server and log its output to the console.

Observe how this works using Chrome DevTools.
Note how often the `search` function is run and that we are not handling service errors.

Limiting user inputs with throttle/debounce

We currently submit a request to the server with every keystroke. This is not desirable behavior because it can lead to a bad user experience and drain battery life, resulting in wasted network requests and performance issues both on the client and server side. Users make typos; they can change their minds about what they are inputting, and rarely do the first few characters of information input result in useful results.

We can still listen to every keystroke, but we don't have to react to every stroke. By leveraging `throttle/debounce`, we can limit the number of events generated to a predetermined interval and maintain the type-as-you-search functionality.

Note that `throttle` and `debounce` are not functional equivalents, and their behavior will differ from framework to framework. In addition to throttling, we expect to capture the last input that the user has typed. In the `lodash` framework, the `throttle` function fulfills this requirement, whereas, in `RxJS`, `debounce` fulfills it.

It is easy to inject throttling into the observable stream using `RxJS/debounceTime`. Implement `debounceTime` with `pipe`:

```
src/app/city-search/city-search.component.ts
import { debounceTime } from 'rxjs/operators'
this.search.valueChanges
  .pipe(debounceTime(1000))
  .subscribe(...)
```

`debounceTime` will, at a maximum, run a search every second, but also run another search after the user has stopped typing. In comparison, `RxJS/throttleTime` will only run a search every second, on the second, and will not necessarily capture the last few characters the user may have input.

`RxJS` also has the `throttle` and `debounce` functions, which you can use to implement custom logic to limit input that is not necessarily time-based.

Since this is a time- and event-driven functionality, breakpoint debugging is not feasible. You may monitor the network calls within

the **Chrome Dev Tools | Network** tab, but to get a more real-time feel for how often your search handler is being invoked, add a `console.log` statement.

It is not a good practice to check in code with active `console.log` statements. These debug statements make it difficult to read the actual code, which creates a high cost of maintainability. Even if debug statements are commented out, do not check them.

Input validation and error messages

`FormControl` is highly customizable. It allows you to set a default initial value, add validators, or listen to changes on `blur`, `change`, and `submit` events, as follows:

example

```
new FormControl('Bethesda', { updateOn: 'submit'
})
```

We won't be initializing `FormControl` with a value, but we need to implement a validator to disallow single-character inputs:

1. Import `Validators` from `@angular/forms`:

```
src/app/city-search/city-search.component.ts
import { FormControl, Validators } from
'@angular/forms'
```

2. Modify `FormControl` to add a minimum length validator:

```
src/app/city-search/city-search.component.ts
search = new FormControl('',
[Validators.minLength(2)])
```

3. Modify the template to show a validation error message below the hint text:

```
src/app/city-search/city-search.component.html
...
<form style="margin-bottom: 32px">
  <mat-form-field appearance="outline">
    ...
    @if (search.invalid) {
      <mat-error>
        Type more than one character to search
      </mat-error>
    }
  </mat-form-field>
</form>
...
```

Note the addition of some extra margin to make room for lengthy error messages.

If you are handling different kinds of errors, the `hasError` syntax in the template can get repetitive. You may want to

implement a more scalable solution that can be customized through code, as shown:

example

```
@if (search.invalid) {  
  <mat-error>  
    {{getErrorMessage()}}  
  </mat-error>  
}  
getErrorMessage() {  
  return this.search.hasError('minLength') ?  
    'Type more than one character to search' :  
    '';  
}
```

4. Modify the `search` function to not execute a search with invalid input, replacing the condition in the existing `if` statement:

```
src/app/city-search/city-search.component.ts  
this.search.valueChanges  
  .pipe(debounceTime(1000))  
  .subscribe((search Value: string) => {  
    if (!this.search.invalid) {  
      ...  
    }  
  })
```

Instead of doing a simple check to see whether `searchValue` is defined and not an empty string, we can tap into the validation engine for a more robust check by calling `this.search.invalid`.

For now, we're done with implementing `search` functionality. Next, let's go over a what-if scenario to see how a template-driven form implementation would appear.

Template-driven forms with two-way binding

The alternative to reactive forms is template-driven forms. If you're familiar with `ng-model` from AngularJS, you'll find that the new `ngModel` directive is an API-compatible replacement for it.

Behind the scenes, `ngModel` implements `FormControl` that automatically attaches itself to `FormGroup`. `ngModel` can be used at the `<form>` level or the individual `<input>` level. You can read more about `ngModel` at <https://angular.dev/api/forms/NgModel>.

In the `stage6` example code of the LocalCast Weather app repository on GitHub, I have included a template-driven component in `app.component.ts` named `<app-city-search-tpldriven>` rendered under `<div class="example">`. You can experiment with this component to see what the alternate template implementation looks like:

```
projects/stage6/src/app/city-search-  
tpldriven/city-search-tpldriven.component.html  
...  
<input matInput aria-label="City or Zip"  
  [(ngModel)]="model.search"
```

```

        (ngModelChange)="doSearch($event)"
        minlength="2"
        name="search" #search="ngModel">
    ...
    @if(search.invalid) {
        <mat-error>
            Type more than one character to search
        </mat-error>
    }
    ...

```

Note the `[[()]]` “box of bananas” two-way binding syntax in use with `ngModel`.

The differences in the components are implemented as follows:

```

projects/stage6/src/app/city-search-
tpldriven/city-search-tpldriven.component.ts
import { WeatherService } from
'../weather/weather.service'
export class CitySearchTpldrivenComponent {
    model = {
        search: '',
    }
    constructor(private weatherService:
WeatherService) {}

    doSearch(searchValue) {
        const userInput = searchValue.split(',').map(s
=> s.trim())

```

```
    this.weatherService
      .getCurrentWeather(userInput[0],
        userInput.length > 1 ?
          userInput[1] : undefined
      )
      .subscribe(data => console.log(data))
  }
}
```

As you can see, most of the logic is implemented in the template; as such, you are required to maintain an active mental model of the template and the controller. Any changes to event handlers and validation logic require you to switch back and forth between the two files.

Furthermore, we have lost input limiting and the ability to prevent service calls when the input is invalid. It is still possible to implement these features, but they require convoluted solutions and do not neatly fit into the new Angular syntax and concepts.

Overall, I do not recommend the use of template-driven forms. There may be a few instances where it may be very convenient to use the box-of-bananas syntax. However, this sets a bad precedent for other team members to replicate the same pattern around the application.

Component interaction with BehaviorSubject

To update the current weather information, we need the `citySearch` component to interact with the `currentWeather` component. There are four main techniques to enable component interaction in Angular:

- Global events
- Parent components listening for information bubbling up from children components
- Sibling, parent, or children components within a module that works off of similar data streams
- Parent components passing information to children components

Let's explore them in detail in the following sections.

Global events

This technique has been leveraged since the early days of programming in general. In JavaScript, you may have achieved this with global function delegates or jQuery's event system. In AngularJS, you may have created a service and stored variables within it.

In Angular, you can still create a root-level service, store values in it, use Angular's `EventEmitter` class, which is meant for directives, or use `RxJS/Subscription` to create a fancy messaging bus for yourself.

As a pattern, global events are open to rampant abuse, and rather than helping to maintain a decoupled application architecture, they lead to a global state over time. A global state or a localized state at the controller level, where functions read and write to variables in

any given class, is enemy number one of writing maintainable and unit-testable software.

Ultimately, if you're storing all your application data or routing all events in one service to enable component interaction, you're merely inventing a better mousetrap. A single service will grow large and complex over time. This leads to unforeseen bugs, side effects from unintentional mutations of unrelated data, continuously increasing memory usage because data from previous views can't be discarded, and low cohesion due to data stored from unrelated components of the application. Overusing a service is an anti-pattern that should be avoided at all costs. In a later section, you will find that, essentially, we will still be using services to enable component interaction; however, I want to point out that there's a fine line that exists between a flexible architecture that enables decoupling and the global or centralized decoupling approach that does not scale well.

Child-parent relationships with event emitters

Your child component should be completely unaware of its parent. This is key to creating reusable components.

We can implement the communication between `CitySearchComponent` and the `CurrentWeatherComponent`, leveraging `AppComponent` as a parent element and letting the `AppComponent` controller orchestrate the data.

Commit your code now! In the next two sections, you will be making code changes that you will need to discard.

Let's see how this implementation will look:

1. `CitySearchComponent` exposes `EventEmitter` through an `@Output` property:

```
src/app/city-search/city-search.component.ts
import { Component, OnInit, Output,
EventEmitter } from '@angular/core'
export class CitySearchComponent implements
OnInit {
  @Output() searchEvent = new
  EventEmitter<string>()
  ...
  this.search.valueChanges
    .pipe(debounceTime(1000))
    .subscribe((search Value: string) => {
      if (!this.search.invalid) {
        this.searchEvent.emit(searchValue)
      }
    })
  ...
}
```

2. `AppComponent` consumes that and calls `weatherService`, setting the `currentWeather` variable:

```

src/app/app.component.ts
import { WeatherService } from
  './weather/weather.service'
import { ICurrentWeather } from './interfaces'
...
template: `
  ...
  <app-city-search
(searchEvent)="doSearch($event)">
    </app-city-search>
  ...
`,
export class AppComponent {
  currentWeather: ICurrentWeather
  constructor(private weatherService:
WeatherService) { }
  doSearch(searchValue) {
    const userInput =
searchValue.split(',').map(
      s => s.trim())
    this.weatherService
      .getCurrentWeather(
        userInput[0], userInput.length > 1 ?
        userInput[1] : undefined
      )
      .subscribe(data => this.currentWeather =
data)
  }
}

```

Note that we are binding to `searchEvent` with the parenthesis syntax. The `$event` variable automatically captures the output from the event and passes it into the `doSearch` method.

We successfully bubbled the information up to the parent component, but we must also be able to pass it down to `CurrentWeatherComponent`.

Parent-child relationships with input binding

By definition, the parent component will know what child components it is working with. Since the `currentWeather` property is bound to the `current` property on `CurrentWeatherComponent`, the results are passed down for display. This is achieved by creating an `@Input` property:

```
src/app/current-weather/current-  
weather.component.ts  
import { Component, Input } from '@angular/core'  
...  
export class CurrentWeatherComponent {  
  @Input() current: ICurrentWeather  
  ...  
}
```

Note that the `ngOnInit` function of `CurrentWeatherComponent` is now superfluous and can be removed.

You can then update `AppComponent` to bind the data to the current weather:

```
src/app/app.component.ts
template: `
  ...
  <app-current-weather [current]="currentWeather">
  </app-current-weather>
  ...
`
```

At this point, your code should work! Try searching for a city. If `CurrentWeatherComponent` updates, then success!

The event emitter to input binding approach is appropriate in cases where you are creating well-coupled components or user controls, and the child is not consuming any external data. A good demonstrator for this might be by adding forecast information to `CurrentWeatherComponent` as shown:

Tue	Wed	Thu	Fri	Sat
80°F	☁	☁	☁	☁
☔	☁	☁	☁	☁
	☁	☁	☁	☁

Figure 2.5: Weather forecast wireframe

Each day of the week can be implemented as a component that is repeated using `@for`, and it will be perfectly reasonable for `CurrentWeatherComponent` to retrieve and bind this information to its child component:

example

```
@for (dailyForecast of forecastArray; track
dailyForecast) {
  <app-mini-forecast [forecast]="dailyForecast">
  </app-mini-forecast>
}
```

In general, if you're working with data-driven components, the parent-child or child-parent communication pattern results in an inflexible architecture, making it difficult to reuse or rearrange your components. A good example of tight coupling is when we imported `WeatherService` in `app.component.ts`. Note that `AppComponent` should have no idea about `WeatherService`; its only job is to lay out several components. Given ever-changing business requirements and design, this is an important lesson to remember.

*Discard changes from the two sections before moving on.
We will instead be implementing an alternate solution.*

Next, we cover a better way for two components to interact with each other without introducing additional coupling with subjects.

Sibling interactions with subjects

The main reason for components to interact is to send or receive updates to data either provided by the user or received from the server. In Angular, your services expose `RxJS/Observable` endpoints, which are data streams that your components can subscribe to. `RxJS/Observer` complements `RxJS/Observable` as a consumer of events emitted by `Observable`. `RxJS/Subject` brings the two functionalities together in an easy-to-work-with package.

You can essentially describe a stream that belongs to a particular set of data, such as the current weather data that is being displayed, with subjects:

example

```
import { Subject } from 'rxjs'
...
export class WeatherService implements
IWeatherService {
  currentWeather$: Subject<ICurrentWeather>
  ...
}
```

`currentWeather$` is still a data stream and does not simply represent one data point. You can subscribe to changes to `currentWeather$` data using `subscribe`, or you can publish changes to it using `next` as follows:

example

```
currentWeather$.subscribe(data => (this.current =
```

```
data))  
currentWeather$.next(newData)
```

Note the naming convention for the `currentWeather$` property, which is postfixed with `$`. This is the naming convention for observable properties.

The default behavior of `Subject` is very much like generic pub/sub mechanisms, such as jQuery events. However, in an asynchronous world where components are loaded or unloaded in unpredictable ways, using the default `Subject` is not very useful.

There are three advanced variants of subjects:

- `ReplaySubject` remembers and caches data points that occurred within the data stream so that a subscriber can replay old events at any given time.
- `BehaviorSubject` remembers only the last data point while listening for new data points.
- `AsyncSubject` is for one-time-only events that are not expected to reoccur.

`ReplaySubject` can have severe memory and performance implications on your application, so it should be used carefully. In the case of `CurrentWeatherComponent`, we are only interested in displaying the latest weather data received, but through user input or other events, we are open to receiving new data to keep

`CurrentWeatherComponent` up to date. `BehaviorSubject` would be the appropriate mechanism to meet these needs:

1. Add `currentWeather$` as a read-only property to `IWeatherService`:

```
src/app/weather/weather.service.ts
import { BehaviorSubject, Observable } from
'rxjs'
export interface IWeatherService {
  readonly currentWeather$:
  BehaviorSubject<ICurrentWeather>
  ...
}
```

`currentWeather$` is declared as read-only because its `BehaviorSubject` should not be reassigned. It's our data anchor or a reference, not a copy of the data itself. Any updates to the value should be sent by calling the `next` function on the property.

2. Define `BehaviorSubject` in `WeatherService` and set a default value:

```
src/app/weather/weather.service.ts
...
export class WeatherService implements
IWeatherService {
  readonly currentWeather$ =
    new BehaviorSubject<ICurrentWeather>({
      city: '--',
```

```

        country: '--',
        date: Date.now(),
        image: '',
        temperature: 0,
        description: '',
    })
    ...
}

```

3. Add a new function named `updateCurrentWeather`, which will trigger `getCurrentWeather` and update the value of `currentWeather$`:

```

src/app/weather/weather.service.ts
...
updateCurrentWeather(search: string | number,
  country?: string): void {
  this.getCurrentWeather(search, country)
    .subscribe(weather =>
      this.currentWeather$.next(weather)
    )
}
...

```

4. Update `IWeatherService` with the new function so that it appears as follows:

```

src/app/weather/weather.service.ts
...
export interface IWeatherService {
  readonly currentWeather$:

```

```

BehaviorSubject<ICurrentWeather>
  getCurrentWeather(city: string | number,
country?: string):
  Observable<ICurrentWeather>
  getCurrentWeatherByCoords(coords:
Coordinates):
  Observable<ICurrentWeather>
  updateCurrentWeather(
    search: string | number, country?:
string): void
}

```

5. Update `CurrentWeatherComponent` to subscribe to the new `BehaviorSubject`:

```

src/app/current-weather/current-
weather.component.ts
...
ngOnInit() {
  this.weatherService.currentWeather$
    .subscribe(data => (this.current =
data))
}
...

```

6. In `CitySearchComponent`, update the `getCurrentWeather` function call to utilize the new `updateCurrentWeather` function:

```

src/app/city-search/city-search.component.ts
...

```

```
this.weatherService.updateCurrentWeather(  
  userInput[0],  
  userInput.length > 1 ? userInput[1] :  
  undefined  
)  
...
```

7. Test your app in the browser; it should appear as follows:

LocalCast Weather

Your city, your forecast, right now!

City Name or Postal Code

Q bursa

Specify country code line 'Paris, US'

Current Weather

Bursa, TR

Wednesday

Mar 25th



45 °F

broken clouds

Figure 2.6: Weather information for Bursa, Turkey

When you type in a new city, the component should update to include the current weather information for that city. We can move the **Add city search capability...** task to the **Done** column, as shown on our Kanban board:

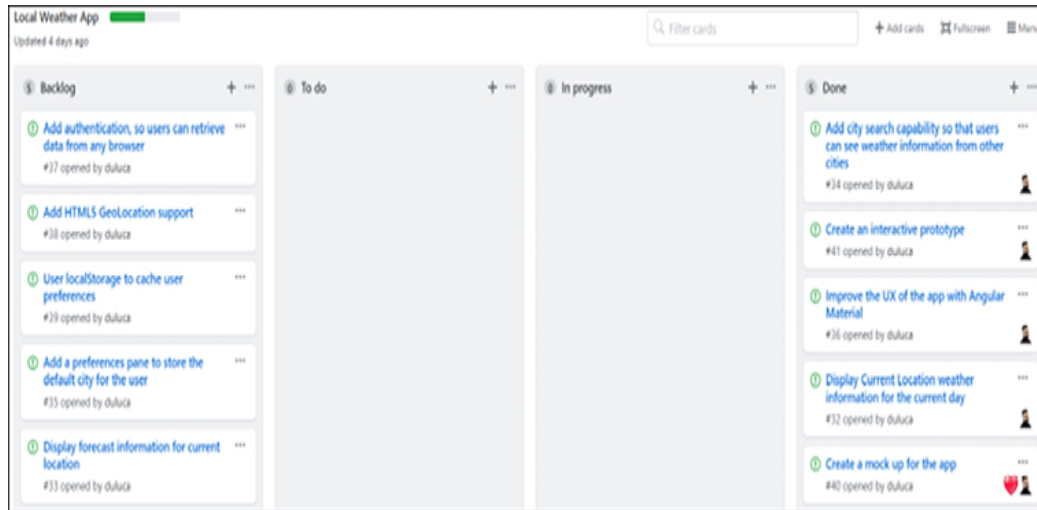


Figure 2.7: GitHub project Kanban board status

We have a functional app. However, we have introduced a memory leak in the way we handled the subscription to `currentWeather$`. In the next section, we'll review how memory leaks can happen and avoid them altogether by using `first` and `takeUntilDestroyed` operations.

Managing subscriptions

Subscriptions are a convenient way to read a value from a data stream for your application logic. If unmanaged, they can create memory leaks in your application. A leaky application will consume ever-increasing amounts of RAM, eventually leading the browser tab to become unresponsive, leading to a negative perception of your

app and, even worse, potential data loss, which can frustrate end users.

The source of a memory leak may not be obvious. In `CurrentWeatherComponent`, we inject `WeatherService` to access the value of `BehaviorSubject`, `currentWeather$`. If we mismanage subscriptions, `currentWeather$`, we can end up with leaks in the component or the service.

Lifecycle of services

By default, Angular services are **shared instance services** or singletons automatically registered to a **root provider**. This means that, once created in memory, they're kept alive as long as the app or feature module they're a part of remains in memory. See the following example of a shared instance service:

```
@Injectable({  
  providedIn: 'root'  
})  
export class WeatherService implements  
  IWeatherService  
...
```

From a practical perspective, this will mean that most services in your application will live in the memory for the application's lifetime. However, the lifetime of a component may be much shorter, or there could be multiple instances of the same component created repeatedly.

Additionally, there are use cases where a component needs its own instance or a copy of the service (e.g., caching values input into a form or displaying weather for different cities simultaneously). To create **multiple instance services**, see the example below:

```
@Injectable()
export class WeatherService implements
  IWeatherService
...

```

You would then provide the service with a **component provider**:

```
@Component({
  selector: 'app-current-weather',
  standalone: true,
  providers: [WeatherService]
})
export class CurrentWeatherComponent {
...

```

In this case, the service would be destroyed when the component is destroyed. But this is not protection against memory leaks. If we don't manage the interactions between long-lived and short-lived objects carefully, we can end up with dangling references between objects, leading to memory leaks.

Exposé of a memory leak

When we subscribe to `currentWeather$`, we attach an event handler to it so that `CurrentWeatherComponent` can react to value changes pushed to `BehaviorSubject`. This presents a problem when the component needs to be destroyed.

In managed languages such as JavaScript, memory is managed by the garbage collector, or GC for short, as opposed to having to allocate and deallocate memory by hand in unmanaged languages such as C or C++. At a very high level, the GC periodically scans the stack for objects not referenced by other objects.

If an object is found to be dereferenced, then the space it takes up in the stack can be freed up. However, if an unused object still has a reference to another object that is still in use, it can't be garbage collected. The GC is not magical and can't read our minds. When an object is unused and can't be deallocated, the memory taken up by the object can never be used for another purpose so long as your application is running. This is considered a memory leak.

My colleague, Brendon Caulkins, provides a helpful analogy:

Imagine the memory space of the browser as a parking lot; every time we assign a value or create a subscription, we park a car in that lot. If we happen to abandon a car, we still leave the parking spot occupied; no one else can use it. If all the applications in the browser do this, or we do it repeatedly, you can imagine how quickly the parking lot gets full, and we never get to run our application.

Next, let's see how we can ensure we don't abandon our car in the parking lot.

Unsubscribing from a subscription

Subscriptions or event handlers create references to other objects, such as from a short-lived component to a long-lived service. Granted, in our case, `CurrentWeatherComponent` is also a singleton, but that could change if we added more features to the app, navigating from page to page or displaying weather from multiple cities at once. If we don't unsubscribe from `currentWeather$`, then any instance of `CurrentWeatherComponent` would be stuck in memory. We subscribe in `ngOnInit`, so we must unsubscribe in `ngOnDestroy`. `ngOnDestroy` is called when Angular determines that the framework no longer uses the component.

Let's see an example of how you can unsubscribe from a subscription in the sample code in the following:

example

```
import { ..., OnDestroy } from '@angular/core'
import { ..., Subscription } from 'rxjs'
export class CurrentWeatherComponent implements
OnInit, OnDestroy {
  currentWeatherSubscription: Subscription
  ...
  ngOnInit() {
    this.currentWeatherSubscription =
```

```
        this.weatherService.currentWeather$  
            .subscribe((data) => (this.current =  
data))  
    }  
    ngOnDestroy(): void {  
        this.currentWeatherSubscription.unsubscribe()  
    }  
    ...
```

First, we need to implement the `OnDestroy` interface for the component. Then, we update `ngOnInit` to store a reference to the subscription in a property named `currentWeatherSubscription`. Finally, in `ngOnDestroy`, we can call the `unsubscribe` method.

Should our component get destroyed, it will no longer result in a memory leak. However, if we have multiple subscriptions in a component, this leads to tedious amounts of coding.

Note that in `CitySearchComponent`, we subscribe to the `valueChanges` event of a `FormControl` object. We don't need to manage the subscription to this event because `FormControl` is a child object of our component. When the parent component is dereferenced from all objects, all its children can be safely collected by the GC.

Subscribing to values in data streams itself can be considered an anti-pattern because you switch your programming model from reactive to imperative. But of course, we must subscribe at least once

to activate the data stream. In the next section, we will cover how you ensure you don't leak memory when subscribing.

Subscribe with first or takeUntilDestroyed

By default, an observable stream doesn't end. Given how engrained RxJS is within every Angular operation, this is rarely the desired outcome. There are two common strategies that we can apply at the time of subscribing to a resource so we can ensure that streams will complete predictably and won't lead to memory leaks.

The first strategy is, well, the first method. Observe the `updateCurrentWeather` method in `WeatherService`:

```
src/app/weather/weather.service.ts
import { map, switchMap, first } from
  'rxjs/operators'
export class WeatherService implements
  IWeatherService{
  ...
  updateCurrentWeather(searchText: string,
country?: string): void {
    this.getCurrentWeather(searchText, country)
      .pipe(first())
      .subscribe((weather) =>
this.currentWeather$.next(weather))
  }
  ...
}
```

In the example above, we intend to get the current weather and display it – and do this only once per request. By piping in a `first()` call into the observable stream, we instruct RxJS to complete the stream after it receives one result. This way, when a resource that utilizes this stream is being GC'd, the relevant RxJS objects will not cause a leak.

The second strategy is `takeUntilDestroyed`. The `first()` strategy doesn't make sense with components that will update multiple times, that will update multiple times. For example `CurrentWeatherComponent` can update after the user enters new search text, so we want to receive updates as long as the component exists. See the following example:

```
src/app/current-weather/current-  
weather.component.ts  
import { takeUntilDestroyed } from  
'@angular/core/rxjs-interop'  
export class CurrentWeatherComponent implements  
OnInit {  
  private destroyRef = inject(DestroyRef);  
  ...  
  ngOnInit(): void {  
    this.weatherService  
      .getCurrentWeather('Bethesda', 'US')  
      .pipe(takeUntilDestroyed(this.destroyRef))  
      .subscribe((data) => (this.current = data))  
  }  
  ...  
}
```

`takeUntilDestroyed` can only be used within an injector context, i.e., a constructor. When using it in lifecycle hook calls, like `ngOnInit`, we must inject `DestroyRef` and pass it into the function. It automatically registers itself, so when the component is destroyed, it completes the stream. This way, the component can receive messages while it's needed, but with no risk of leaking memory.

By applying these alongside the `subscribe` method, we don't have to rely on difficult-to-trace unsubscribe methods, and we can easily verify their implementation with a quick search of the word `subscribe`.

The best part is no part at all. Next, let's see how we can consume an observable component without subscribing to it all.

Coding in the reactive paradigm

As covered in *Chapter 1, Angular's Architecture and Concepts*, we should only subscribe to an observable stream to activate it. If we treat a `subscribe` function as an event handler, we implement our code imperatively.

Seeing anything other than an empty `subscribe()` call in your code base should be considered a red flag because it deviates from the reactive paradigm.

In reactive programming, when you subscribe to an event in a reactive stream, you shift your coding paradigm from reactive programming to imperative programming. There are two places in

our application where we subscribe, one in `CurrentWeatherComponent`, and the other in `CitySearchComponent`.

Let's start by fixing `CurrentWeatherComponent` so we don't mix paradigms.

Binding to an observable with an async pipe

Angular has been designed to be an asynchronous framework from the ground up. You can get the most out of Angular by staying in the reactive paradigm. It can feel unnatural to do so at first, but Angular provides all the tools you need to reflect the current state of your application to the user without having to shift to imperative programming.

You may leverage the `async` pipe in your templates to reflect the current value of an observable. Let's update `CurrentWeatherComponent` to use the `async` pipe:

1. Start by replacing `current: ICurrentWeather` with an observable property:

```
current$: Observable<ICurrentWeather>
```

2. In the constructor, assign `weatherService.currentWeather$` to `current$`:

```
src/app/current-weather/current-
weather.component.ts
import { Observable } from 'rxjs'
export class CurrentWeatherComponent {
  current$: Observable<ICurrentWeather>
  constructor(private weatherService:
WeatherService) {
    this.current$ =
this.weatherService.currentWeather$
  }
  ...
}
```

3. Remove all code related to `SubSink`, `ngOnInit`, and `ngOnDestroy`.
4. Update the template to so you can bind to `current$`:

```
src/app/current-weather/current-
weather.component.html
@if (current$ | async; as current) {
  <div> ... </div>
}
```

The `async` pipe automatically subscribes to the current value of `current$` and makes it available to the template to be used imperatively as the `current` variable. The beauty of this approach is that the `async` pipe implicitly manages the subscription, so you don't have to worry about unsubscribing.

5. Remove the `@else { <div>no data</div> }` block, which is no longer needed because `BehaviorSubject` is always

initialized.

So far, the reactive paradigm has allowed us to streamline and clean up our code.

The async pipe allows you to display a loading message with simple `if-else` logic. To display a message while your observable is resolved, see the following technique:

example

```
@if (current$ | async; as current) {  
  <div>{{current}}</div>  
} @else {  
  <div>Loading...</div>  
}
```

Next, let's further improve our code.

Tapping into an observable stream

The `CitySearchComponent` implements a callback within a `subscribe` statement when firing the `search` function. This leads to an imperative style of coding and mindset. The danger with switching programming paradigms is that you can introduce unintentional side effects to your code base by making it easier to store state or create bugs.

Let's refactor `CitySearchComponent` to be in the reactive functional programming style, as shown in the following example:

```
src/app/city-search/city-search.component.ts
import { debounceTime, filter, tap } from
  'rxjs/operators'
import { takeUntilDestroyed } from
  '@angular/core/rxjs-interop'
export class CitySearchComponent {
  search = new FormControl('',
    [Validators.required,
    Validators.minLength(2)])
  constructor(private weatherService:
    WeatherService) {
    this.search.valueChanges
      .pipe(
        takeUntilDestroyed(),
        filter(() => this.search.valid),
        debounceTime(1000),
        tap((searchValue: string) =>
this.doSearch(searchValue))
        takeUntilDestroyed()
      ).subscribe()
  }
  doSearch(searchValue: string) {
    const userInput = searchValue.split(',').map(s
=> s.trim())
    const searchText = userInput[0]
    const country = userInput.length > 1 ?
userInput[1] : undefined
```

```
this.weatherService.updateCurrentWeather(searchText, country)
}
}
```

In the preceding code, we removed the `OnInit` implementation and implemented our filtering logic reactively. The `tap` operator will only get triggered if `this.search` is valid.

`constructor` should only be used when working with properties and events local to the class context. In this case, `search` is initialized when defined, and `valueChanges` can only be triggered by a user interacting with the component. So, it's okay to set up the subscribe logic in it.

However, if you're referencing any properties within the template, `@Input` variables, or registering an external service call, you must use `ngOnInit`. Otherwise, you will run into render errors or unpredictable behavior. This is because template properties, including `@Input` variables, won't be accessible until `ngOnInit` is called. Further, external service calls may return a response before the component is initialized, leading to change detection errors.

Simply put, 99% of the time you should use `ngOnInit`.

In addition, `doSearch` is called in a functional context, making it very difficult to reference any other class property within the function. This reduces the chances of the state of the class impacting the outcome of our function. As a result, `doSearch` is a composable and unit-testable function, whereas in the previous implementation, it would have been very challenging to unit test `ngOnInit` in a straightforward manner.

Note that `subscribe()` must be called on `valueChanges` to activate the observable data stream. Otherwise, no event will fire.

The fact that we didn't need to implement `ngOnInit` reflects the truly asynchronous nature of our code, which is independent of the lifecycle or state of the application. However, you should stick with `ngOnInit` as a general best practice.

With our refactoring complete, the app should function the same as before but with less boilerplate code. Now, let's look into enhancing our app to handle postal codes from any country.

Chaining API calls

Currently, our app can only handle 5-digit numerical postal or zip codes from the US. A postal code such as `22201` is easy to differentiate from a city name with a simplistic conditional such as `typeof search === 'string'`. However, postal codes can vary widely from country to country, the UK being a great example, with

postal codes such as `EC2R 6AB`. Even if we had a perfect understanding of how postal codes are formatted for every country, we still couldn't ensure that the user didn't fat-finger a slightly incorrect postal code. Today's sophisticated users expect web applications to be resilient toward such mistakes. However, as web developers, we can't be expected to code up a universal postal code validation service by hand. Instead, we need to leverage an external service before we send our request to OpenWeatherMap APIs. Let's explore how we can chain back-to-back API calls that rely on each other.

After the first edition of this book was published, I received some passionate reader feedback on their disappointment that the sample app could only support US zip codes. I've implemented this feature because it demonstrates how simple requests can introduce unplanned complexity to your apps. As a bonus, the app now works worldwide.

Let's add a new item, **Support international zip codes**, to the backlog and move it to **In progress**:

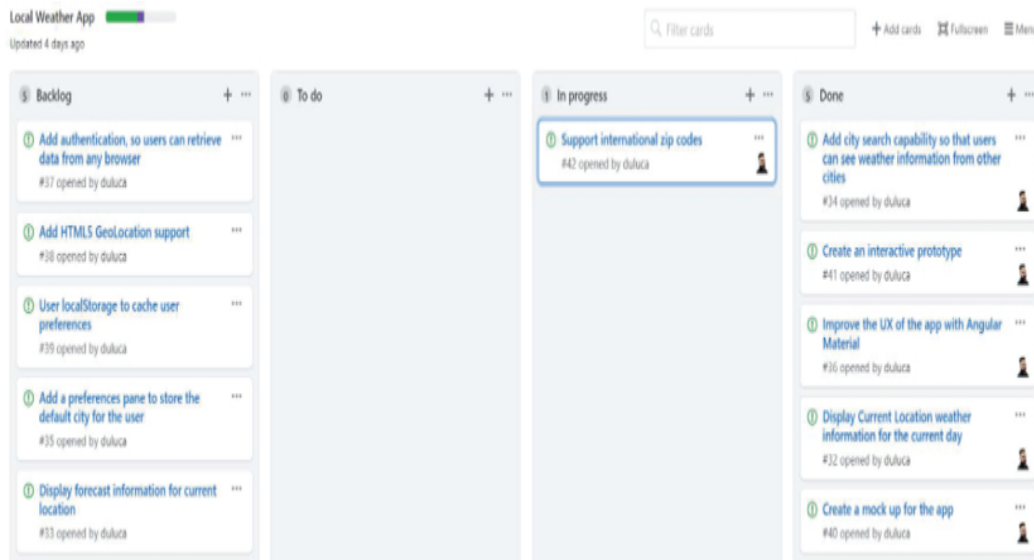


Figure 2.8: Adding an international zip codes story

Implementing a postal code service

To properly understand whether the user inputs a valid postal code versus the name of a city, we must rely on a third-party API call provided by [geonames.org](https://www.geonames.org). Let's see how we can inject a secondary API call into the search logic of our app.

You need to sign up for a free account on [geonames.org](https://www.geonames.org). Afterward, store username as a new parameter in `environment.ts` and `environment.prod.ts`.

You may experiment with the postal code API interactively at <https://www.geonames.org/postal-codes>.

We need to implement a service that adheres to the following interface:

```
interface IPostalCodeService {  
    resolvePostalCode(postalCode: string):  
    Observable<IPostalCode>  
}
```

Declaring an interface for your service is a useful practice when you're initially designing your app. You and your team members can focus on providing the right interaction model without being bogged down by implementation details. Once your interface is defined, you can quickly stub out functionality and have a walking skeleton version of your app in place. Stubbed-out functions help validate design choices and encourage early integration between components. Once in place, team members will no longer need to guess whether they are coding in the right spot. You should always export your interface, so you can use the type information for writing unit tests, creating test doubles or fakes.

Interfaces are key to practicing **Test-Driven Development (TDD)**.

Now implement `PostalCodeService` as shown below:

You may generate the service by executing `npx ng generate service postalCode --project=local-`

```
weather-app --no-flat.
```

```
src/app/postal-code/postal-code.service.ts
import { HttpClient, HttpParams } from
  '@angular/common/http'
import { Injectable } from '@angular/core'
import { Observable } from 'rxjs'
import { defaultIfEmpty, flatMap } from
  'rxjs/operators'
import { environment } from
  '../environments/environment'
export interface IPostalCode {
  countryCode: string
  postalCode: string
  placeName: string
  lng: number
  lat: number
}
export interface IPostalCodeData {
  postalCodes: [IPostalCode]
}
export interface IPostalCodeService {
  resolvePostalCode(postalCode: string):
  Observable<IPostalCode>
}
@Injectable({
  providedIn: 'root',
})
export class PostalCodeService implements
  IPostalCodeService {
  constructor(private httpClient: HttpClient) {}
```

```

    resolvePostalCode(postalCode: string):
Observable<IPostalCode> {
    const uriParams = new HttpParams()
        .set('maxRows', '1')
        .set('username', environment.username)
        .set('postalcode', postalCode)
    return this.httpClient
        .get<IPostalCodeData>(
            `${environment.baseUrl}${environment.geonamesApi}.
            geonames.org/
                postalCodeSearchJSON`,
            { params: uriParams }
        )
        .pipe(
            flatMap(data => data.postalCodes),
            defaultIfEmpty(null)
        )
    }
}

```

Note the new environment variable, `environment.geonamesApi`. In `environment.ts`, set this value to `api` and, in `environment.prod.ts`, to `secure`, so calls over HTTPS work correctly to avoid the mixed-content error, as covered in *Chapter 10, Releasing to Production Using CI/CD*.

In the preceding code segment, we implement a `resolvePostalCode` function that calls an API, which is configured to receive the first viable result the API returns. The results are then flattened and piped out to the subscriber. With `defaultIfEmpty`, we ensure that a null value will be provided if we don't receive a result from the API. If the call is successful, we will get back all the information defined in `IpostalCode`, making it possible to leverage `getCurrentWeatherByCoords` using coordinates.

Observable sequencing with `switchMap`

Let's update the weather service so that it can call the `postalCode` service to determine whether the user input was a valid postal code:

1. Start by updating the interface so we only deal with a string:

```
src/app/weather/weather.service.ts
...
export interface IWeatherService {
  ...
  getCurrentWeather(search: string, country?:
string):
    Observable<ICurrentWeather>
  updateCurrentWeather(search: string,
country?: string)
}
```

2. Inject `PostalCodeService` to the weather service as a private property:

```
src/app/weather/weather.service.ts
import {
  PostalCodeService
} from '../postal-code/postal-code.service'
...
constructor(
  private httpClient: HttpClient,
  private postalCodeService: PostalCodeService
) {}
```

3. Update the method signature for `updateCurrentWeather`.
4. Update `getCurrentWeather` to try and resolve `searchText` as a postal code:

```
src/app/weather/weather.service.ts
import { map, switchMap } from
'rxjs/operators'
...
getCurrentWeather(
  searchText: string,
  country?: string
): Observable<ICurrentWeather> {
  return this.postalCodeService.
    resolvePostalCode(searchText)
    .pipe(
      switchMap((postalCode) => {
        if (postalCode) {
          return
this.getCurrentWeatherByCoords({
          latitude: postalCode.lat,
          longitude: postalCode.lng,
```

```

        } as Coordinates)
    } else {
        const uriParams = new
HttpParams().set(
            'q',
            country ?
`${searchText},${country}` : searchText
        )
        return
this.getCurrentWeatherHelper(uriParams)
    }
})
)
}

```

If you run into TypeScript issues when passing the latitude and longitude into `getCurrentWeatherByCoords`, you may have to cast the object using the `as` operator. So, your code would look like this:

```

return
this.getCurrentWeatherByCoords({
    latitude: postalCode.lat,
    longitude: postalCode.lng,
} as Coordinates)

```


In the preceding code segment, our first call is to the `postalCode` service. We then react to postal codes posted on the data stream using `switchMap`. Inside `switchMap`, we can observe whether `postalCode` is null and make the appropriate follow-up call to either get the current weather by coordinates or by city name.

Now, LocalCast Weather should work with global postal codes, as shown in the following screenshot:

LocalCast Weather

Your city, your forecast, right now!

City Name or Postal Code


 EC2R 6AB

Specify country code like 'Paris, US'

Current Weather

Barbican, GB

Wednesday
Mar 25th



34 °F

clear sky

Figure 2.9: LocalCast Weather with global postal codes

We are done with implementing international zip code support. Move it to the **Done** column on your Kanban board:

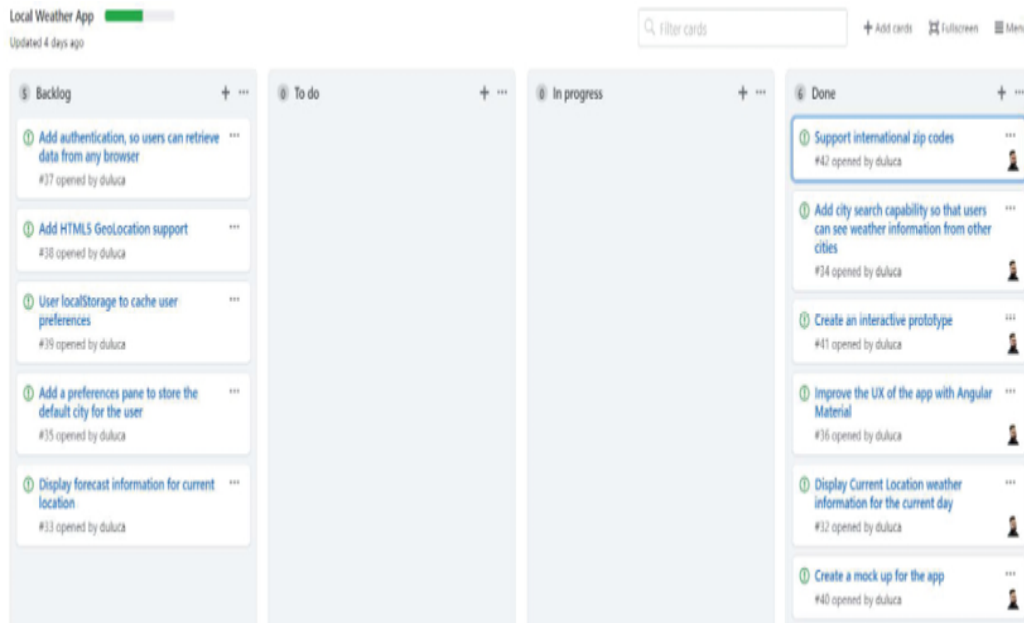


Figure 2.10: International zip code support done

As we complete our implementation of LocalCast Weather, there's still room for improvement. Initially, the app looks broken when it first loads because of the dashes and empty fields shown. There are at least two different ways to handle this. The first is to hide the entire component at the `AppComponent` level if there's no data to display. For this to work, we must inject `WeatherService` into `AppComponent`, ultimately leading to a less flexible solution.

Another way is to enhance `CurrentWeatherComponent` so that it is better able to handle missing data.

You improve the app further by implementing geolocation to get the weather for the user's current location upon launching the app. You

can also leverage `window.localStorage` to store the city that was last displayed or the last location retrieved from `window.geolocation` upon initial launch.

We are done with the LocalCast Weather app until *Chapter 9, Recipes – Master/Detail, Data Tables, and NgRx*, where I demonstrate how a state store like NgRx compares to using `RxJS/BehaviorSubject`.

Using Angular Signals

A signal is a reactivity primitive that keeps track of its value changing over time. Angular Signals implements this primitive to granularly sync the application state with the DOM. By focusing on granular changes in state and only the relevant DOM nodes, the number and severity of change detection operations are significantly reduced. As covered in *Chapter 1, Angular's Architecture and Concepts*, change detection is one of the most expensive operations that the Angular framework performs. As an app grows in complexity, change detection operations may be forced to traverse or update larger parts of the DOM tree. As the number of interactive elements increases in your app, change detection events occur more frequently. App complexity combined with the frequency of events can introduce significant performance issues, resulting in slow or choppy rendering of the app. Usually, there's no quick fix for a problem like this. So, it is critical to understand how signals work and implement them in your app to avoid costly performance issues.

As of this publication, Angular Signals is in preview. This means that the functionality and performance

characteristics of the feature set can and likely will change. Refer to the following guide for the latest information: <https://angular.dev/guide/signals>.

Angular Signals provides a few simple functions to interact with it:

- **signal**: A wrapper around a value. It works like a value getter or setter in a class and is conceptually similar to how **BehaviorSubject** works:

```
const mySignal = signal('Hello')
console.log(mySignal()) // outputs: Hello
                           (only once)
mySignal.set('Goodbye') // updates the value.
                           Update and mutate methods have subtle
                           differences in setting a new value.
                           // To display the new value, you must call
                           console.log again.
```

- **computed**: A computed signal. It utilizes one or more signals to modify the outcome:

```
const someSignal = computed(() =>
  `${mySignal()}, World`)
console.log(someSignal()) // outputs: Hello,
                           World. If needed, it lazily updates when
                           mySignal is set to a new value.
```

- **effect**: An event that triggers when a signal changes:

```
effect(() => {
  console.log(`A robot says: ${someSignal}`)
})
// console.log will be called any time
mySignal changes.
```

Signals are a new foundational concept, and they change how we think about observables, binding data, and syncing state between components. They are performant, surgical in their nature, and best of all, they're memory safe. No subscriptions to worry about here.

Let's start by covering a simple example of using signals.

Implementing dark mode

For our app to be considered cool by techies, we must implement a dark mode for it. Let's use signals to implement this feature and go a step further by remembering the user's selection in `localStorage`:

```
src/app/app.component.ts
const darkClassName = 'dark-theme'
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [...],
  template: `
    <mat-toolbar color="primary">
      <span data-testid="title">LocalCast
Weather</span>
    <div fxFlex></div>
```



```

    <mat-icon>brightness_5</mat-icon>
    <mat-slide-toggle
      color="warn"
      data-testid="darkmode-toggle"
      [checked]="toggleState()"
      (change)="toggleState.set($event.checked)"></mat-
slide-toggle>
    <mat-icon>bedtime</mat-icon>
  </mat-toolbar>
  <div fxLayoutAlign="center">
    <div class="mat-caption vertical-margin">
      Your city, your forecast, right now!
    </div>
  </div>
  <div fxLayoutAlign="center">
    <app-city-search></app-city-search>
  </div>
  <div fxLayout="row">
    <div fxFlex></div>
    <mat-card appearance="outlined"
fxFlex="300px">
      <mat-card-header>
        <mat-card-title>
          <div class="mat-headline-5">Current
Weather</div>
        </mat-card-title>
      </mat-card-header>
      <mat-card-content>
        <app-current-weather></app-current-
weather>
      </mat-card-content>
    </mat-card>
  </div>

```

```

        </mat-card>
        <div fxFlex></div>
    </div>
    `
  },
  })
export class AppComponent {
  readonly toggleState =
    signal(localStorage.getItem(darkClassName) ===
      'true')
  constructor() {
    effect(() => {
      localStorage.setItem(darkClassName,
        this.toggleState().toString())
      document.documentElement.classList.toggle(
        darkClassName, this.toggleState()
      )
    })
  }
}

```

If this were production code, I would not use this terse line of code:

```

document.documentElement.classList.toggle(
  darkClassName, this.toggleState())

```

Here, I wanted to keep the lines of code to a minimum, and the toggle function provided by the DOM API contains the logic needed to make this work

correctly. The line should be refactored to adhere to the single responsibility principle.

Observe the `readonly` property named `toggleState`. This is our signal. It holds a Boolean value. We can initialize it by reading a value from `localStorage`; if it doesn't exist, it will default to `false`.

In the toolbar, we define `mat-slide-toggle` and assign its `[checked]` state to `toggleState()`. This binds the value of the signal to the component. By assigning `(change)="toggleState.set($event.checked)"`, we ensure that when the user flips the toggle, its value will be written back to the signal.

Finally, we implement the `effect` method to react to the changes in the value of the signal. In the constructor, we can define the behavior we want within the `effect` function. First, we update `localStorage` with the current value of `toggleState`, and second, we set the `dark-theme` class on the DOM to toggle the dark mode state.

We leverage Angular Material's built-in dark theme functionality to define a dark theme and attach it to a CSS class named `dark-theme`. Refer to `styles.scss` to see how this is configured.

We could've implemented this functionality at least a half-dozen different ways, but signals do offer a very economical way of doing

it.

We can build on these concepts and replace the uses of `BehaviorSubject` and `[(ngModel)]` throughout our application. Doing so greatly simplifies how our Angular app works, while also reducing package size and complexity.

Replacing BehaviorSubject with signals

Now, let's see what it looks like to use signals instead of `BehaviorSubject`. Implementing a signal means we must change the end-to-end pipeline of how a value is retrieved and displayed. A signal is a synchronous pipeline, whereas RxJS is asynchronous.

You may wonder, isn't asynchronous better than synchronous? Yes, but not when the synchronous code can run in a non-blocking manner. Asynchronous is expensive, and due to the fundamental technologies that are being leveraged under the hood, signals are way cheaper and faster. This is due to great features that are now built into JavaScript. See <https://www.arrow-js.com> by Justin Schroeder as an example of this. Certain kinds and sizes of projects no longer need full-fat frameworks like Angular, React, or Vue.

We will need to update `WeatherService`, `CitySearchComponent`, and `CurrentWeatherComponent`:

1. First replace `currentWeather$` with `currentWeatherSignal` in `WeatherService`:

```
src/app/weather/weather.service.ts
import { signal } from '@angular/core'
export class WeatherService implements
IWeatherService{
    ...
    readonly currentWeatherSignal =
signal(defaultWeather)
    ...
}
```

2. Implement a new `getCurrentWeatherAsPromise` function to convert the observable to a `Promise` and a new `updateCurrentWeatherSignal` function to await the result of the call and assign the result to the signal:

```
src/app/weather/weather.service.ts
import { ..., firstValueFrom } from 'rxjs'
getCurrentWeatherAsPromise(
    searchText: string,
    country?: string
): Promise<ICurrentWeather> {
    return firstValueFrom(
        this.getCurrentWeather(searchText,
country)
    )
}
async updateCurrentWeatherSignal(searchText:
string,
```

```

country?: string): Promise<void> {
  this.currentWeatherSignal.set(
    await this.getCurrentWeatherAsPromise(
      searchText, country
    )
  )
}

```

Note that we use `firstValueFrom` to make sure the stream completes as intended.

3. Next, replace the `current$` property with `currentSignal` in `CurrentWeatherComponent`:

```

src/app/current-weather/current-
weather.component.ts
export class CurrentWeatherComponent {
  readonly currentSignal:
WritableSignal<ICurrentWeather>
  constructor(private weatherService:
WeatherService) {
    this.currentSignal =
this.weatherService.currentWeatherSignal
...

```

4. Update the template to use the signal:

```

src/app/current-weather/current-
weather.component.html
@if (currentSignal(); as current) {

```

```
...  
}
```

5. Finally, update `CitySearchComponent` to trigger the new service call:

```
src/app/city-search/city-search.component.ts  
export class CitySearchComponent {  
  ...  
  
  this.weatherService.updateCurrentWeatherSignal  
  (  
    searchText, country  
  )  
  ...  
}
```

We have transformed our app to use a signal to communicate between components. A signal is less sophisticated than `BehaviorSubject`, but most of the time, the extra capabilities aren't used. Signals are memory safe, lightweight, and allow novel applications by leveraging computed signals, like the RxJS merge technique discussed earlier in this chapter.

Ultimately, RxJS and signals are complementary technologies. For example, we wouldn't consider replacing the debounce logic in the search input away from RxJS. Angular also ships with `toSignal` and `fromSignal` interoperability functions; however, I would caution against mixing paradigms. To get the full benefit of signals, always prefer an end-to-end refactor, as this section covers.

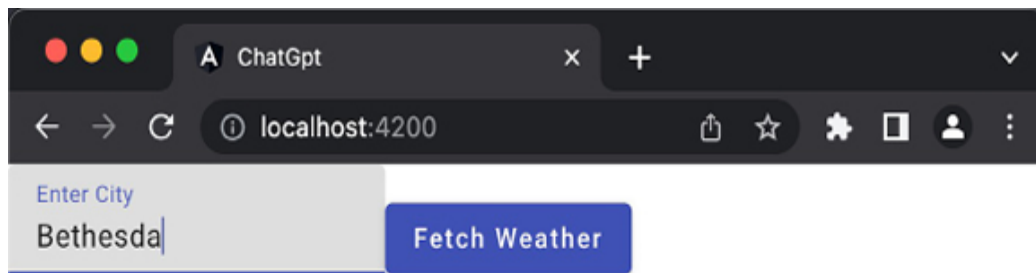
With so many options, paradigms, techniques, and gotchas, you may be wondering if you can just generate this code using AI. I did just that for you. Read on.

Generating apps with ChatGPT

Let's see what result we get if we ask ChatGPT to generate a weather app. In August 2023, I asked ChatGPT to generate a weather app using GPT-4 with the `CodeInterpreter` plugin. I gave it the following prompt:

Write an Angular app that displays real-time weather data from openweathermap.org APIs, using Angular Material, with a user input that accepts city name, country, or postal code as input.

After making a few minor corrections, this is the result I got:



Weather in Bethesda

Temperature: 287.95°C

Conditions: clear sky

Figure 2.11: ChatGPT weather app – August 2023

ChatGPT created a very simple and straightforward app for me, with a weather-display component using two-way binding for the input field. The service call was correctly implemented in a dedicated weather service triggered by the **Fetch Weather** button. To achieve similar results to the `LocalCast` app we built, we would have to provide a prompt with far more technical details. Non-technical people won't know to ask for specific implementation details, and developers may simply find it easier to iteratively develop their solution. Nevertheless, the results are impressive.

Four months later, I questioned my premise from the paragraph above.

What if developers were okay with providing one or two more prompts?

In December 2023, I provided the same prompt from above to ChatGPT using GPT-4 without using any plugins, and after it generated the code, I provided an additional prompt:

Can you rewrite weather.component.html and style it in a way that looks like a professional design on desktop and mobile devices alike?

And boom, I got a result that looked a lot better!

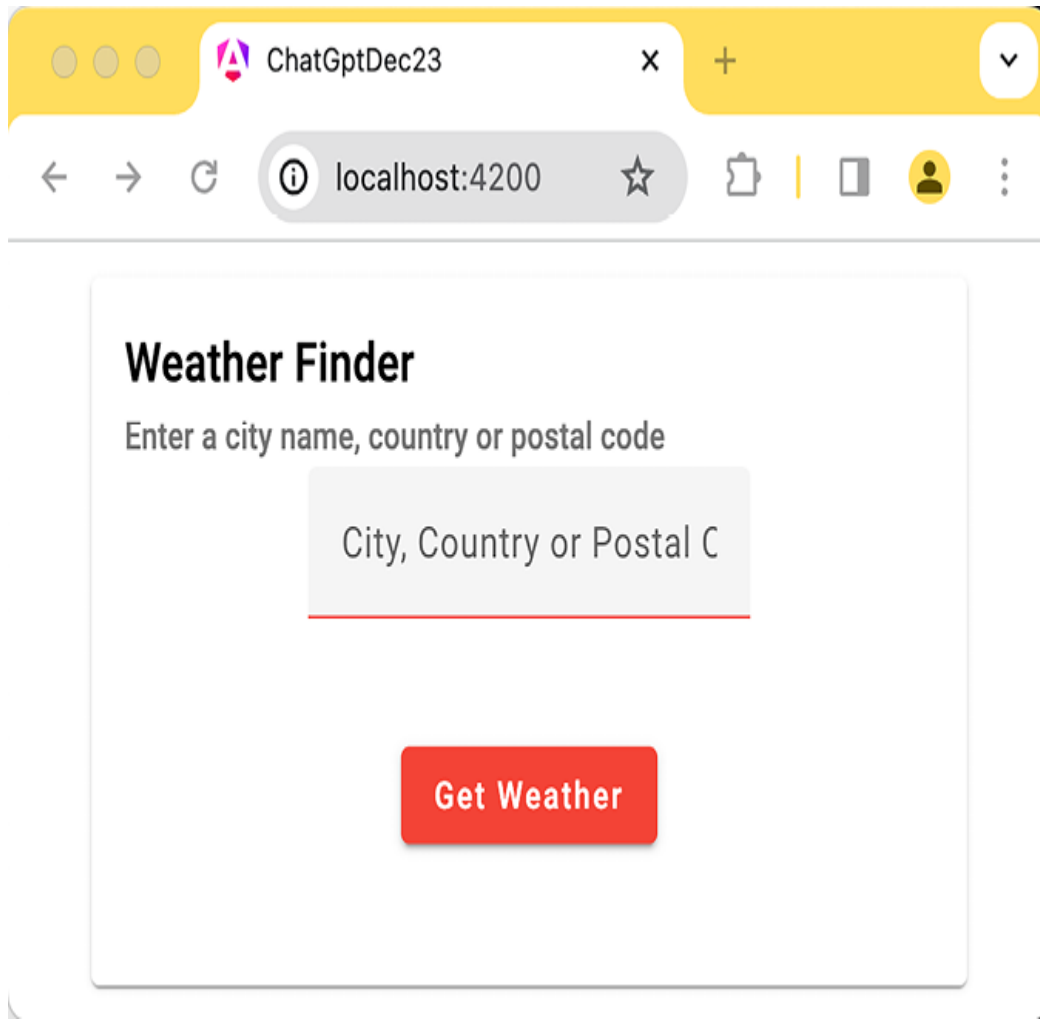


Figure 2.12: ChatGPT Weather app initial version – December 2023

Still, this output doesn't look like my design. Of course, ChatGPT has no idea what my design is, and it's too cumbersome to meticulously describe it in writing. Then I remembered I had a hand-drawn mockup of the weather app I created for the 1st edition in 2018.

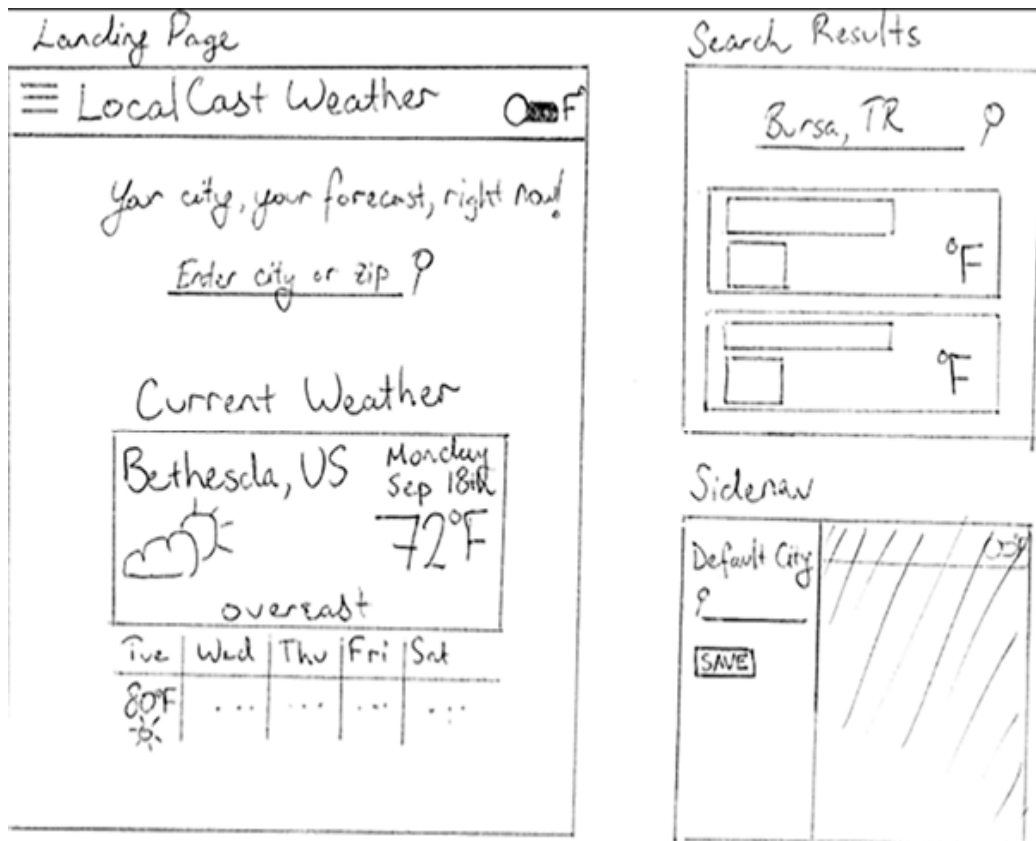


Figure 2.13: Hand-drawn wireframe for LocalCast

Yes, I did use a ruler!

In August 2023, ChatGPT couldn't see, but since then, it has gained computer vision. I uploaded the mockup as is and said, "Redesign the UI to follow this mockup." Remember that my mockup has three screens and difficult-to-read handwriting in it.

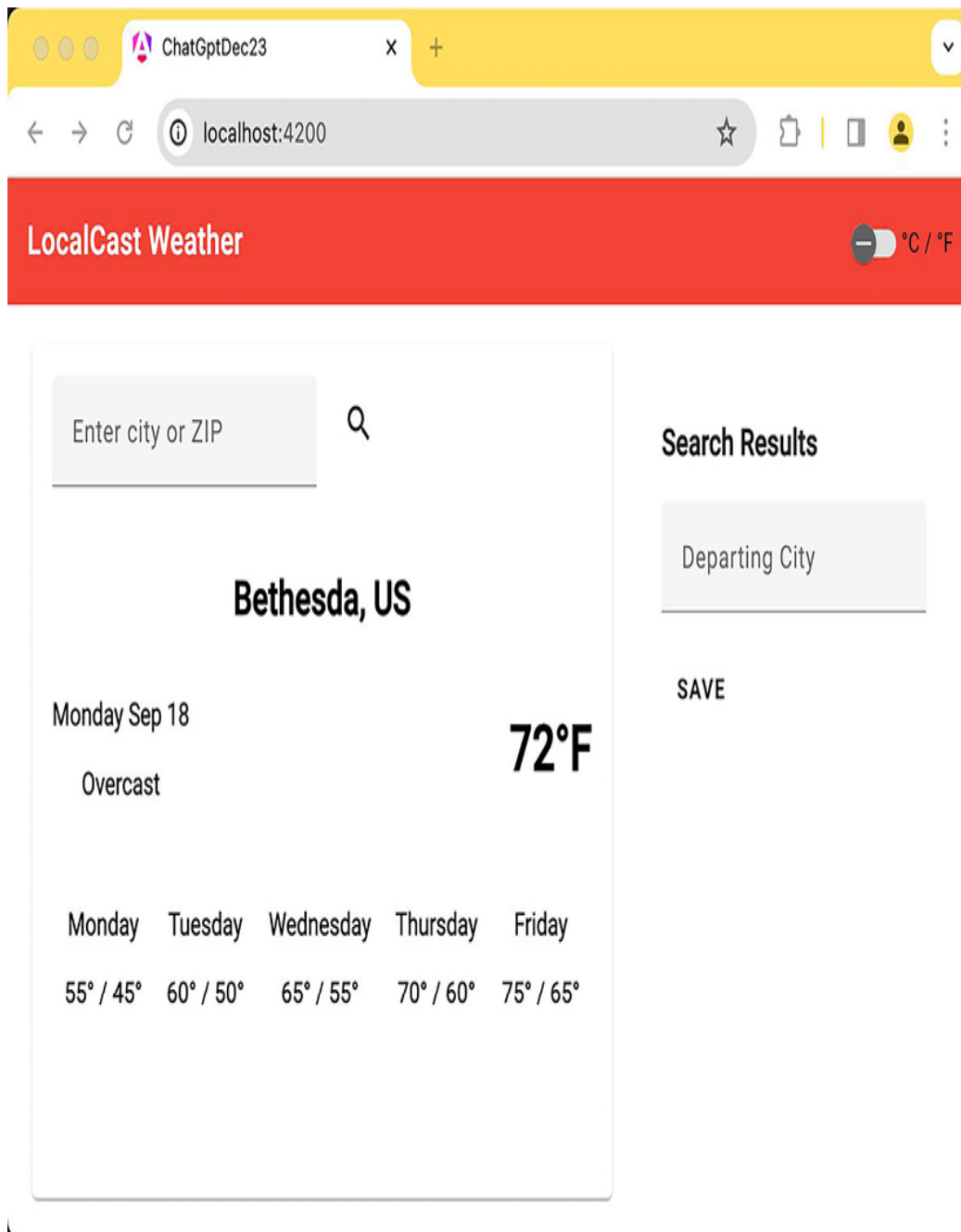


Figure 2.14: ChatGPT weather app second version – December 2023

I'm shocked that it picked up on `SideNav` and incorporated it using proper Material components and `FlexLayout` media queries to make it responsive – never mind the misinterpretation of my handwriting.

I updated the generated UI code to make it interactive and included it as a project named `chat-get-dec23` in the repo. Here's the result:

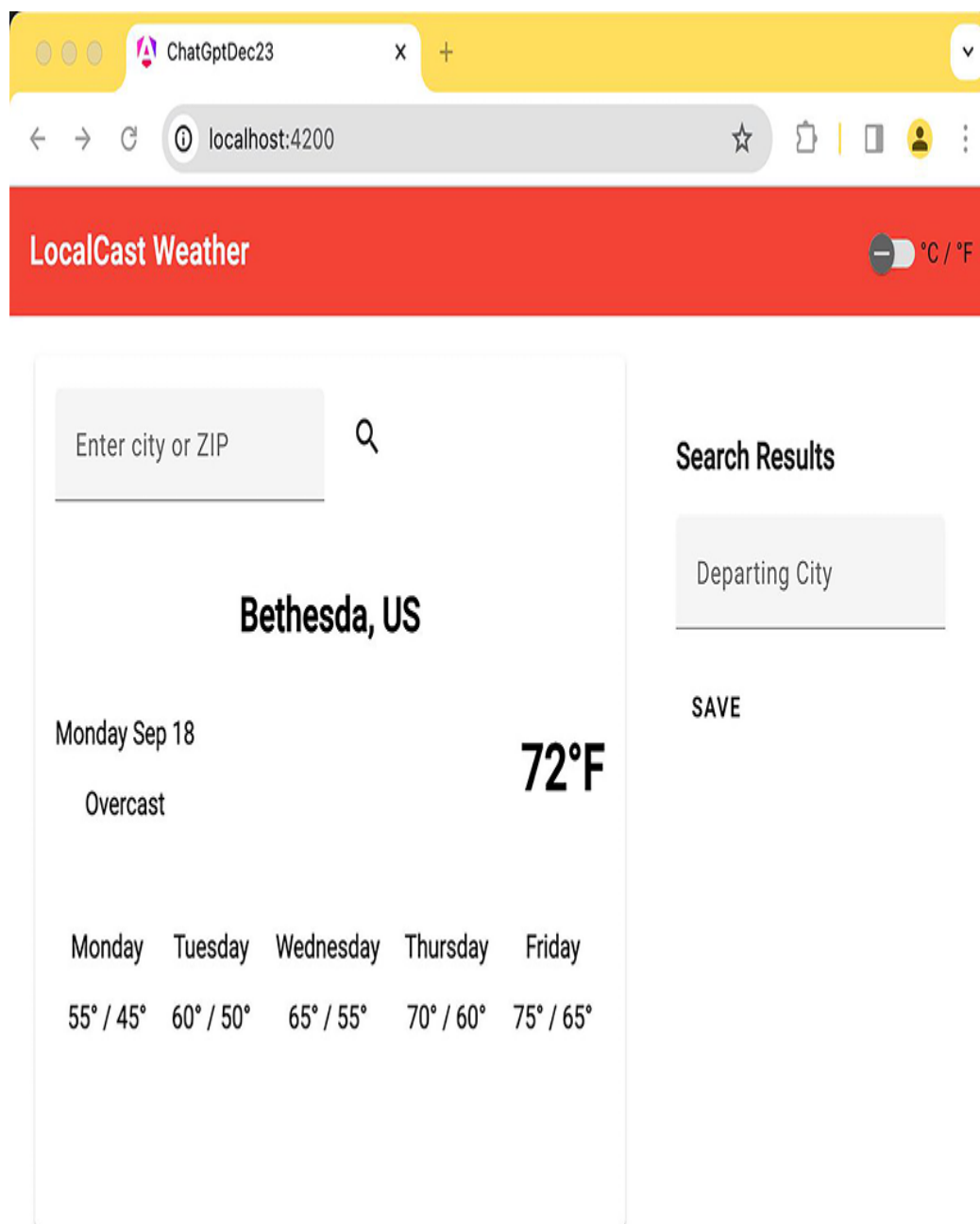


Figure 2.15: ChatGPT weather app final version – December 2023

This is beyond impressive. By the time the next edition of this book is published, this chapter may be only a few pages long and filled

with tips, highlighting the crucial need to use a ruler when drawing your mockups.

Summary

In this chapter, you learned how to create search-as-you-type functionality using `MatInput`, validators, reactive forms, and data-stream-driven handlers. You became aware of two-way binding and template-driven forms. You also learned about different strategies to enable inter-component interactions and data sharing. You dove into understanding how memory leaks can be created and the importance of managing your subscriptions.

You can now differentiate between imperative and reactive programming paradigms and understand the importance of sticking with reactive programming where possible. Finally, you learned how to implement sophisticated functionality by chaining multiple API calls together. You learned about the signal primitive and how you can use it to build simpler and more performant applications.

LocalCast Weather is a straightforward application that we used to cover the basic concepts of Angular. As you saw, Angular is great for building such small and dynamic applications while delivering a minimal amount of framework code to the end user. You should consider leveraging Angular for quick and dirty projects, which is always great practice for building larger applications. You also learned you can use GenAI tools like ChatGPT to give yourself a quick start when beginning a new project.

In the next chapter, we will dive into considerations around architecting a web application in an enterprise app and learn where all the monsters are hidden. We will go over how you can build a **Line-of-Business (LOB)** application using a router-first approach to designing and architecting scalable Angular applications with first-class authentication and authorization, user experience, and numerous recipes that cover a vast majority of requirements that you may find in LOB applications.

Exercises

After completing the **Support international zip codes** feature, did we switch coding paradigms here? Is our implementation above imperative, reactive, or a combination of both? If our implementation is not entirely reactive, how would you implement this function reactively? I'll leave this as an exercise for the reader.

Don't forget to execute `npm test`, `npm run e2e`, and `npm run test:a11y` before moving on. It is left as an exercise for the reader to fix the unit and end-to-end tests.

Visit GitHub to see the unit tests I implemented for this chapter at <https://github.com/duluca/local-weather-app/tree/master/projects/stage6>.

Questions

Answer the following questions as best as possible to ensure you've understood the key concepts from this chapter without googling anything. Do you know if you got all the answers right? Visit

<https://angularforenterprise.com/self-assessment> for more:

1. What is the `async` pipe?
2. Explain how reactive and imperative programming is different and which technique we should prefer.
3. What is the benefit of `BehaviorSubject`, and what is it used for?
4. What are memory leaks and why should they be avoided?
5. What is the best method for managing subscriptions?
6. How are Angular signals different than RxJS streams?
7. What are ways you can use Angular Signals to simplify your application?

3

Architecting an Enterprise App

In *Chapter 2, Forms, Observables, Signals, and Subjects*, we used the LocalCast Weather app to demonstrate various features of Angular to learn as well as experiment with and inform us if these features are suitable for more complex enterprise applications. Building enterprise applications is as much about the people building them as it is the technology used to build them. An over-eager approach to consuming and rolling out unproven tech is guaranteed to create the **sinkhole effect** in your project. If you're unfamiliar with sinkholes, they are a natural phenomenon that occurs due to the dissolution of underlying ground material. At some point, usually suddenly, the ground collapses to devastating effect, revealing ...

4

Creating a Router-First Line-of-Business App

As you read in *Chapter 3, Architecting an Enterprise App*, **Line-of-Business (LOB)** applications are the bread and butter of the software development world.

In this and the remaining chapters of the book, we'll set up a new application with rich features that can meet the demands of an LOB application with scalable architecture and engineering best practices that will help you start small and be able to grow your solution quickly if there's demand. We will follow the router-first design pattern, relying on reusable components to create a grocery store LOB application named LemonMart. We'll discuss designing around major data entities and the importance of completing high-level mock-ups for your application ...

5

Designing Authentication and Authorization

Designing a high-quality **authentication** and **authorization** system without frustrating the end user is a difficult problem to solve.

Authentication is the act of verifying the identity of a user, and authorization specifies the privileges that a user must have to access a resource. Both processes, **auth** for short, must seamlessly work in tandem to address users' needs with varying roles, needs, and job functions.

On today's web, users have a high baseline level of expectations from any auth system they encounter through the browser, so this is an important part of your application to get right the first time. The user should always know what they can and can't do in your application. If there are errors, ...

6

Implementing Role-Based Navigation

In *Chapter 5, Designing Authorization and Authentication*, we covered how designing an effective authentication and authorization system is challenging but crucial for user satisfaction. Users expect a high standard from web authentication systems, and any errors should be clearly communicated. As applications grow, their authentication backbone should be easily maintainable and extensible to ensure a seamless user experience.

In this chapter, we will discuss the challenges of creating a great auth UX and implementing a solid baseline experience. We will continue the router-first approach to designing SPAs by implementing the auth experience of LemonMart. In *Chapter 4, Creating a Router-First Line-of-Business ...*

7

Working with REST and GraphQL APIs

In *Chapter 1, Angular's Architecture and Concepts*, I introduced you to the wider architecture in which web applications exist, and in *Chapter 3, Architecting an Enterprise App*, we discussed various performance bottlenecks that can impact the success of your app. However, your web app can only perform as well as your full-stack architecture performs. If you're working with an inadequate API design or a slow database, you will spend your time implementing band-aid solutions instead of addressing the root cause of the issues. The moment we move away from the minimalist mindset and start patching holes, we are on our way to constructing a fragile tower that is at risk of collapsing or very expensive to maintain. ...

8

Recipes - Reusability, Forms, and Caching

In the next two chapters, we will complete most of the implementation of LemonMart and round out our coverage of the router-first approach. In this chapter, I will reinforce the idea of a decoupled component architecture by creating a *reusable* and *routable* component that supports data binding. We will use **Angular directives** to reduce boilerplate code and leverage classes, interfaces, enums, validators, and pipes to maximize code reuse with TypeScript and ES features.

In addition, we will create a **multi-step form** that architecturally scales well and supports a responsive design. Then, we will differentiate between user controls and components by introducing a **lemon rater** and a reusable form part that ...

9

Recipes - Master/Detail, Data Tables, and NgRx

In this chapter, we complete the router-first architecture implementation on LemonMart by implementing the top three most used features in business applications: master/detail views, data tables, and state management. I will demonstrate data tables with server-side pagination, highlighting the integration between the frontend and backend using LemonMart and LemonMart Server.

We will leverage the router orchestration concept to orchestrate how our components load data or render. We will then use resolve guards to reduce boilerplate code when loading data before navigating to a component. We will use auxiliary routes to lay out components through the router configuration and reuse the same component ...

10

Releasing to Production with CI/CD

Ship it or it never happened! If you don't publish your code, you create zero value. This motivation to ship your work is prevalent in many industries. However, delivering a piece of work to someone else or opening it up to public scrutiny can be terrifying. In software engineering, delivering anything is difficult; delivering something to production is even more difficult.

Check out my 2018 talk, *Ship It or It Never Happened: The Power of Docker, Heroku, and CircleCI*, at <https://bit.ly/ship-it-or-it-never-happened>.

We live in an era of moving fast and breaking things. However, the latter part of that statement rarely works in an enterprise. You can live on the edge and adopt the YOLO lifestyle, but this ...

Appendix A

Setting Up Your Development Environment

Sharing a consistent development environment between you and your team members is important. Consistency helps to avoid many IT-related issues, including ongoing maintenance, licensing, and upgrade costs. Further, you want to ensure the entire team has the same development experience. This way, if a team member runs into a configuration issue, other team members can help resolve the issue. Creating a frustration-free and efficient onboarding experience for a new team member is also essential.

Easy and well-documented onboarding procedures ensure that new team members can quickly become productive and be integrated into the team. On an ongoing basis, achieving a consistent and minimal development ...



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical ...

Index

Symbols

80-20 solution [127](#), [128](#)

disciplined and balanced approach [132](#), [133](#)

line-of-business apps [129-131](#)

#fill-area [398](#)

A

abstract auth service

implementing [213-216](#)

abstract form component [385](#), [386](#)

Agile engineering

best practices [104](#), [105](#)

Agile Manifesto

reference link [104](#)

Ahead-of-Time (AOT) compilation [508](#)

Akita [452](#), [455](#)

reference link [456](#)

Analog [106](#)

Angular [1](#), [11](#), [106](#), [298](#)

community support [107](#)

deterministic releases [12](#), [13](#)

diverse coding paradigm support [106](#)

Firebase auth provider, adding to [280-286](#)

first-class upgrades [13](#), [14](#)

future [35-37](#)

maintainability [14-16](#)

URL [41](#), [107](#)

Angular 2+ [11](#)

Angular app

initializing [542](#), [543](#)

rewriting, with NgRx/SignalStore [465](#), [466](#)

Angular CLI

installing [542](#)

project setup [540](#)

used, for configuring server ...