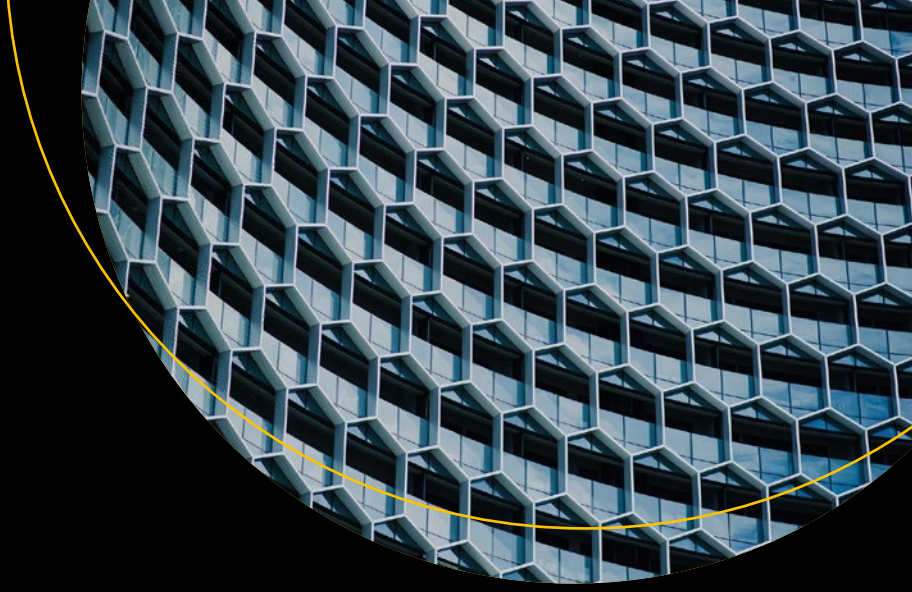

Building Offline Applications with Angular

Develop Reliable, Performant Web Applications for Desktop and Mobile Platforms

—

Venkata Keerti Kotaru

Apress®

Building Offline Applications with Angular

Develop Reliable, Performant Web Applications for Desktop and Mobile Platforms

Venkata Keerti Kotaru

Apress®

Building Offline Applications with Angular: Develop Reliable, Performant Web Applications for Desktop and Mobile Platforms

Venkata Keerti Kotaru
Hyderabad, Telangana, India

ISBN-13 (pbk): 978-1-4842-7929-8 ISBN-13 (electronic): 978-1-4842-7930-4
https://doi.org/10.1007/978-1-4842-7930-4

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Spandana Chatterjee
Development Editor: James Markham
Coordinating Editor: Divya Modi
Copy Editor: Kim Wimpsett

Cover designed by eStudioCalamar

Cover image designed by Pixabay

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at https://github.com/Apress/Building-Offline-Applications-with-Angular/upload. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my family

Table of Contents

About the Author

Venkata Keerti Kotaru has been in software development for almost two decades. He has helped design and develop scalable, performant, modern software solutions for multiple clients. He holds a master's degree in software systems from the University of St. Thomas, Minneapolis and St. Paul, USA.

He is the author of several books on Angular, and he also contributes to the developer community by blogging, writing articles, and speaking at technology events. He has written for Dotnet Curry (DNC Magazine). He has presented technology sessions at AngularJS Hyderabad, AngularJS Chicago, and Google Developer Groups at Hyderabad including the annual event Dev Fest. He is a three-time Microsoft MVP.

About the Technical Reviewer

Yogendra Sharma is a developer with experience in the architecture, design, and development of scalable and distributed applications, with a core interest in microservices and DevOps. Currently he works as a technical architect at Intelizign Engineering Services Pvt Pune. He also has hands-on experience in technologies such as AR/VR, CAD CAM, simulation, AWS, IoT, Python, J2SE, J2EE, NodeJS, VueJs, Angular, MongoDB, and Docker. He constantly explores technical novelties, and he is open-minded and eager to learn about new technologies and frameworks. He has reviewed several books and video courses published by Packt and Apress.

Introduction

Congratulations on beginning to read a new book. This book will help you learn and build modern web applications with Angular and TypeScript. Even if you are a beginner in JavaScript and Angular technologies, the book provides step-by-step instructions to easily follow along with and build next-generation web applications. It presents you with a consistent use case across the book. You will build a few pages for Web Arcade, an imaginary online arcade.

The book primarily focuses on building an application that is resilient to changes in network connectivity. Traditionally, web applications work well when online, but they can crash badly if the connectivity is lost. A decade ago, most applications operated on wired connections or on relatively stable WiFi networks. Connectivity was not a factor. Today, applications run on a variety of networks with varying connectivity and speed. Users expect data not to be lost while they are on the move switching between networks. The advent of new web technologies allows developers to build applications that provide better user experiences. Through the course of this book, you will learn about these aspects with the help of a single use case, Web Arcade.

The book provides step-by-step instructions to get started with an Angular application and add progressive web app and service worker features, and it explores the different configurations. It helps you learn how to build installable web applications with little effort.

Next, the book provides instructions to cache data in IndexedDB. It details the JavaScript API for creating and managing data in the database running on the browser's client machine. Finally, you will learn how to use Dexie.js, which is a wrapper for the IndexedDB API, to simplify the implementation.

Throughout the chapters in the book, you will work through Web Arcade scenarios, which begin with a simple Angular component for rolling dice. You will see how to build pages that list games, navigate to the details of a game, and add comments on a game. You will learn how to build these pages that work even when network connectivity is lost. You will also build features that create data offline. For example, users will be able to add comments while offline and synchronize once the connectivity is restored.

We are glad you picked up this book. Happy learning.

CHAPTER 1

Building Modern Web Applications

Welcome! Congratulations on picking this book to learn how to build offline applications with Angular. This introductory chapter sets the expectations and framework for the book. It provides a brief look at traditional web application development and why creating just another traditional web application is not enough.

Laying the Foundation

This book provides some perspective and elaborates on cutting-edge features like service workers and IndexedDB. It provides step-by-step instructions for creating an Angular application and incrementally adding features to build a sophisticated web application. The book uses an imaginary online gaming system, Web Arcade, to illustrate the techniques. It acts as a use case for building a modern web application that is resilient to network connectivity drops and speed changes.

Let's establish the context with a little bit of history. As you might know, web applications have been popular for more than two decades. There have been a tremendous number of applications built. In fact, many web applications are mission critical for running businesses.

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4_1

In the early 2000s, these applications replaced *thick clients*, which were installed on a device, a desktop, or a laptop. The thick clients posed a challenge because the applications had to be built by targeting an operating system. Most applications were not interoperable between Apple macOS and Microsoft Windows; however, there were exceptions. A few organizations and developers used Java-based technologies for building the thick clients, which ran on a Java Virtual Machine (JVM). Before such solutions gained traction, web applications took over. Largely, the thick clients were not compatible across different platforms and operating systems.

Web applications helped address this problem. A web application runs on a browser. The browsers interpret and execute HyperText Markup Language (HTML), JavaScript, and Cascading Style Sheets (CSS). The features are standardized by the European Computer Manufacturers Association (ECMA) and Technical Committee 39 (TC39).

The scenario changed with the advent of mobile apps, which are installed and run on mobile devices. This could be a mobile phone or a tablet device like an iPad. Apple's App Store, which started in 2008, had a major role in organizations' and developers' shift to apps. Today, Apple's iOS and Google's Android are the two major mobile platforms. iOS uses App Store, and Android uses Google Play (also called Play Store) to distribute apps for their respective platforms.

Original Problem

On mobile devices, the apps brought back the original problem: you need to develop apps for respective platforms. You build your native app once for iOS and repeat for Android. Of course, there are alternatives, which include workarounds with hybrid and cross-platform technologies. They suit the major use cases, but there are always a few workflows for which such a solution does not help. Moreover, there are compromises with

the native user experience; an iOS user may not feel the user interface matches the platform if originally built for Android. Many workflows and applications need large screens and the flexibility provided by desktop-class operating systems. There has only been minor traction bringing iOS and Android apps to the desktop. In most cases, the user experience is limited.

Caveats with the Web Application Solution

Organizations and developers build web applications that cater to all major platforms and browsers on iOS and Android and on macOS and Microsoft Windows. Remember, organizations adapted this solution originally while moving away from thick clients. At that time, the devices were largely stationary at a desk and not mobile. They were connected by either a cable or a WiFi network. The connectivity was stable. While building applications, connectivity was not a factor.

Mobile platforms have changed this scenario. Users are highly mobile, moving in and out of networks. Applications need to consider the connectivity factor. Imagine a user loses connectivity temporarily. When she attempts to launch a mobile web application, a traditional web application shows a message similar to "page cannot be displayed," and the application does not launch. The user cannot proceed. The problem might be worse if the user is in the middle of a transaction. Data is lost, and the user might have to retry the entire workflow.

Apps provide a ready-made solution. Remember, apps are installed on the devices. Users can launch the app and interact with past data or messages even when the user is disconnected. If a user submitted a transaction or a form, an app can cache temporarily and synchronize when online.

Take the example of a social application such as Twitter. When offline, it allows you to launch the application, see cached tweets, and even compose a new tweet and save it as a draft.

Modern web applications support such advanced caching features. This book details how to build a modern web application that allows the users to function when offline.

Use Case

The book uses Web Arcade, an online game system on the Web as a use case. You will build the application with Angular and TypeScript. The book provides step-by-step instructions for how to create the application, various components, services, and more.

Throughout the course of this book, you will learn how to do the following:

- Install and upgrade web applications.
- Cache the application so that it is accessible while offline.
- Cache the data retrieved.
- Enable the application to function while offline. With the help of the Web Arcade use case, we will detail how to add data to the system. When offline, data is cached on the device. Once back online, the app will provide an opportunity to synchronize with the server.

On desktop and mobile devices, modern browsers like Google Chrome, Microsoft Edge, Safari (on Mac and iOS), and Firefox allow users to install the application. The book provides step-by-step instructions to enable them to install the web application and create a shortcut for the application. The shortcut provides easy access to the web application and launches it in its own window (unlike a typical web application,

which always launches in a browser). Figure 1-1 shows the Web Arcade application installed on macOS. Notice the Web Arcade application icon in the Dock. You will see a similar icon in the Windows taskbar. The application is in its own separate window, not among the browser tabs.

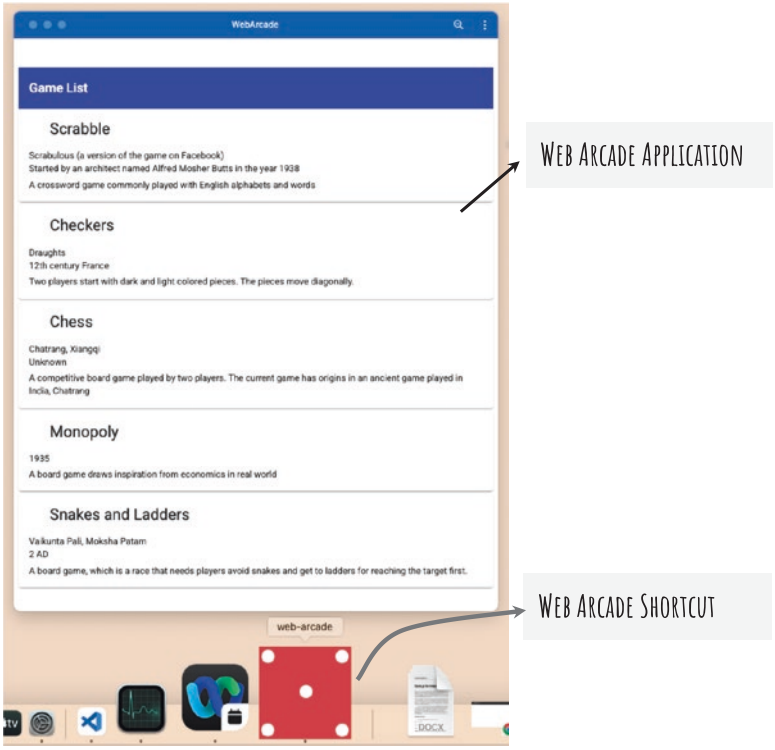

Figure 1-1. *Web Arcade application installed on macOS*

This book details how to use a service worker (with Angular) to cache the application. It provides step-by-step instructions to set up a development environment and test the caching features. At this stage the application loads even if the network is unavailable. In an example, you may use the feature to roll a die even when the application is offline. As you can imagine, rolling a die does not require server-side connectivity. It is a random number generated between 1 and 6. The application visualizes rolling the die (Figure 1-2).

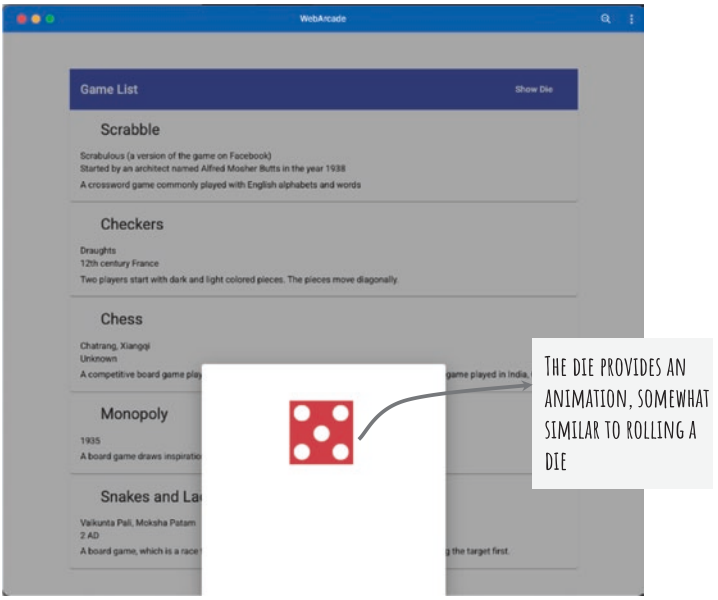

Figure 1-2. *Rolling a die in the Web Arcade application*

You'll start by caching the application data and using it while offline (or on a slow network). You will see how the service worker cache is utilized when the real server-side services are unavailable.

The book also details how to create data; specifically, the Web Arcade application will allow users to add comments even while offline. The comments are cached using IndexedDB, a local database. The application identifies once the connectivity is established. It synchronizes the cached offline comments with the server-side services and database.

Later, you'll need to create new versions of the application and the database. The book covers features and implementations to prompt users about an available upgrade. It details how to provide seamless transitioning to the new database and upgrade its structure.

Code Samples

This section explains how to download and run the code samples. Clone the Web Arcade repository from this GitHub location: https://git-scm.com/downloads. Open a terminal/command prompt and use the following command, which requires Git to be installed on your machine.

```
git clone https://github.com/kvkirthy/web-arcade.git
```

Note Git is a popular distributed source code management (SCM) tool. It has a small footprint and works with minimal resources and disk space on your machine. It is also free and open source.

By default, you have cloned the `master` branch. Check out a branch called `book-samples` for the samples of the examples in the book. First change the directory to `web-arcade`. Next, check out the branch `book-samples`.

```
cd web-arcade
git checkout book-samples
```

We anticipate enhancements and will incorporate feedback in the `master` branch accordingly. However, the branch `book-samples` is dedicated to matching the code samples in the book exactly.

Next, run the following command to install all the required packages for the code samples. Throughout the book, we provide instructions to use Node Package Manager (NPM) and Yarn. While NPM is the default package manager with Node.js, Yarn is an open source project backed by Facebook. It has received traction in the developer community due to its strengths in performance and security. We advise you to pick one and stick to it until the end of the book.

```
npm install
(or)
yarn install
```

Note This command needs Node.js, NPM, or Yarn installed on your machine. If they are unavailable, read along for the moment. Detailed instructions are provided in the next chapter.

Next, run the following command to start the Web Arcade sample application:

```
npm run start-pwa
(or)
yarn start-pwa
```

The previous command starts a full-fledged application running on a developer-class web server. It is useful to run the application while reading and understanding the code. However, if you are making changes and updating the code, the changes need to show up in the application. Typically, the page reloads, and the application updates with the changes. It is difficult to do this with the previous command. You might have to end the process and restart every time you make a change. Hence, consider using the following command while updating the code. It instantly updates the application with the changes.

```
npm start
(or)
yarn start
```

It is a good practice to leave the application running in the background. Throughout the book you will continuously create and update the code and run the samples. The scripts run by the previous code keep the application up and running. Do not terminate this script, unless directed.

Note At the time of writing, the command does not support caching and loading the application while it is offline. It is an important feature of the sample application and the concepts explained in this book. To test caching features with service workers, use the `start-pwa` command.

Summary

This chapter touched on the need for new implementations like service workers and IndexedDB, which are natively supported by most modern browsers. The remainder of the book will detail how to implement and integrate these technologies in a web application. We also introduced a use case called Web Arcade, an online gaming system on the Web, which will be used throughout the rest of the book.

CHAPTER 2

Getting Started

This chapter provides instructions for how to get started with an Angular application. It is the foundation for all the upcoming chapters. Follow the steps detailed in this chapter to set up your development environment. The upcoming chapters will use the software, libraries, and packages detailed in this guide.

Specifically, the chapter provides steps to create a sample application, namely, Web Arcade. The sample application will provide use cases and examples for all the concepts and their explanation in this book. In this chapter, you will start by creating the Web Arcade Angular application.

You will also add offline features to the Web Arcade Angular application. You will see the introductory details of how to access the Angular application, without network connectivity.

Prerequisites

To create, run, and test Angular applications, you need the list of software to install and set up on your computer. Luckily, all the software and libraries listed and described in this book are open source and free to use, at least for an individual developer. This section lists the minimum required software to begin creating an Angular application.

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4_2

Node.js and NPM

Node.js is a cross-platform JavaScript runtime that works on the JavaScript engine called V8. It is primarily used for running JavaScript on the server side and back end.

In this book, to a large extent you will use Node Package Manager (NPM), which comes with the Node.js installation. As the name describes, it is a package manager, which helps you install and manage libraries and packages. It keeps track of the entire dependency tree for a package. With simple commands, it helps download and manage the library and its dependencies.

For example, Lodash is a highly useful library of JavaScript utilities and functions. With a single command you can install and add the package as a dependency to your project. Others downloading your project do not need to perform additional steps.

Download and install Node.js from the official Node.js website, `https://nodejs.org`. Click the download link on the website. It lists installers for long-term support (LTS) and the latest version. Preferably choose LTS. Next, select an option based on your operating system and platform. It will download the installer.

It installs Node.js and NPM. At the time of authoring this book, the Node.js version is 14.17.0, and NPM is 6.14.13.

Once the download finishes, open the installer, and follow the steps to finish the installation. See Figure 2-1 for the introduction screen and version information.

Figure 2-1. *Node.js installer*

Yarn

While NPM is the default package manager with Node.js, Yarn is an open source project backed by Facebook. It has received traction in the developer community due to its strengths in performance and security. The examples in the book include Yarn and NPM commands. If you are new to Angular development, pick one and consistently use it for all the samples and the exercises. Including Yarn provides a choice to the reader. Sometimes, teams and organizations could be picky in selecting their toolset (for various reasons including important considerations like security). Hence, it helps to learn both NPM and Yarn.

To install Yarn, run the following command:

```
npm install -g yarn
```

Note Notice the option `-g`, which stands for "global." The package is available across all the projects and the directories. Hence, it might need elevated permissions to run and install.

On a Windows machine, run this command as an administrator.

On macOS, run the command as a super user. Consider the following snippet. The command will prompt for the root user password.

```
sudo npm install -g yarn
```

If you do not use the `-g` option, you may still use `yarn` (or any other tool installed without `-g`) at the directory level. You might need to re-install it for every new directory or a project. It is not a bad solution if you prefer to keep resources local to the project or a directory.

To verify the installation has been successful, run `yarn --version`. Ensure the `yarn` command is identified and returns version information.

```
% yarn --version
1.22.10
```

Angular CLI

While working with Angular applications, Angular CLI is a highly useful command-line tool. You will be using this tool for all Angular-related tasks including creating projects, adding new Angular components, using Angular services, running the Angular application, doing build-related tasks, etc.

Install Angular CLI with the following command:

```
 npm install -g @angular/cli
# (or)
 yarn global add @angular/cli
```

To verify the installation has been successful, run `ng --version`. See Listing 2-1. Ensure the Angular CLI command is identified and returns version information.

Listing 2-1. Verify Angular CLI Installation

```
% ng --version
```

```
Angular CLI: 12.0.1
Node: 14.16.1
Package Manager: npm 6.14.12
OS: darwin x64
Angular: undefined

Package                      Version
------------------------------------------------------
@angular-devkit/architect    0.1200.1 (cli-only)
@angular-devkit/core         12.0.1 (cli-only)
@angular-devkit/schematics   12.0.1 (cli-only)
@schematics/angular          12.0.1 (cli-only)
```

Note Notice in Listing 2-1 that Angular is undefined; however, Angular CLI has a version 12.0.1. The installation is successful. Angular will show a version once you create a project with the CLI.

Visual Studio Code

Strictly speaking, you can use any simple text editor to write Angular code and use a terminal or a command prompt to compile, build, and run the application. This may be true for most programming languages. However, for better productivity and ease of development, use an integrated development environment (IDE) like Visual Studio Code. The software is built by Microsoft, and it's free, light in footprint, easy to download, and easy to install. Its features are highly useful while working with Angular applications. However, it is a personal preference. You may choose to pick any IDE that you are comfortable with to create and run the Angular application described in this book.

Download Visual Studio Code from its website, `https://code.visualstudio.com`. Use the download link on the page. At the time of writing this content, the website automatically identifies your operating system and presents an appropriate download link.

See Figure 2-2 for a snapshot of Visual Studio Code.

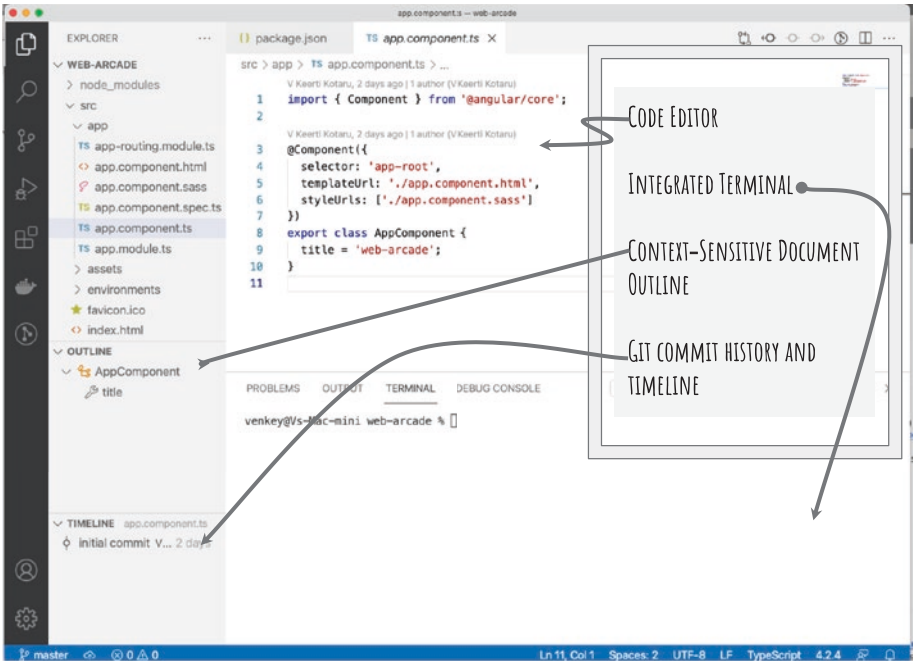

Figure 2-2. *Visual Studio Code snapshot*

The following are a few other alternatives:

- *Sublime Text*: This is shareware and a useful text editor. It is a good fit for JavaScript development for its smaller footprint, speedy response, and ease of use.
- *WebStorm*: This is a sophisticated IDE built by JetBrains for JavaScript development. It has custom features for many popular frameworks including Angular and Node.js. However, it is proprietary software that needs to be purchased.
- *Atom*: This is open source and a free text editor. It is a cross-platform application built with HTML and JavaScript.

Http-Server

Http-Server is a quick and efficient way to run a Node.js-based web server for static files. During development, you will be using this NPM package for working with the cached application.

Run the following command to install Http-Server globally on your machine:

```
 npm install -g http-server
# (or)
 yarn global add http-server
```

To verify the installation has been successful, run `http-server --version`. Ensure the `http-server` command is identified and returns version information.

```
% http-server --version
v0.12.3
```

Note If you are having trouble with `yarn global add` and a package is not found even after the global install, refer to the section "Working with yarn global add" later in the chapter.

Create an Angular Application

Now that all the prerequisites are available, you are ready to create a new Angular application. You will use Angular CLI, which you just installed (`@angular/cli`). Angular CLI's executable is named `ng`. In other words, you will run the tool with an `ng` command. It uses the option you provide with the `ng` command to perform a task.

Follow these instructions to create a new Angular application:

```
ng new web-arcade
```

`ng new` is an Angular CLI command to create a new application. As you create a new application, typically CLI prompts you to choose whether you want to use Angular routing and stylesheet format. See Listing 2-2.

For the sample application, choose to implement routing. As the sample application evolves, you will create multiple pages. Navigation between these pages need Angular routing.

Note Routing is an important feature of single-page applications (SPAs) because most web applications have more than one page. Users navigate between the pages. Each page will have a unique URL.

In an SPA, as users navigate between pages, an entire page does not reload. With the help of the routing implementation (Angular routing in this case), the SPA updates only the sections of the page that change between two URLs.

For the second prompt about selecting a stylesheet format, if you prefer to match the examples in the book, choose Sass. However, if you are comfortable with one of the other stylesheet formats, feel free to choose the other application.

Listing 2-2. Prompts While Creating a New Angular Application

```
 % ng new web-arcade
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? Sass
[ https://sass-lang.com/documentation/syntax#the-indented-
syntax ]
```

Angular CLI out of box provides a choice of the following stylesheet formats:

- *Cascading Style Sheets (CSS)*: This is the traditional approach to stylesheet development. It works well while working small units of code.
- *Syntactically Awesome Style Sheets (SCSS - Sassy CSS)*: This provides better programming-type features compared to CSS. The features include variables, functions, mixins, and nested rules. It is a superset of CSS.

  The stylesheet is written into a file with the extension `.scss`. It is preprocessed to CSS. It is fully compatible with CSS. Hence, all the valid CSS statements work in an `.scss` file as well.

  The SCSS syntax includes curly braces to indicate the beginning and end of a block and semicolons for the end of a stylesheet statement.
- *Syntactically Awesome Style Sheets (SASS)*: This is similar to SCSS, except that the stylesheet statements are indented instead of using curly braces and semicolons. To explicitly indicate the file format, SASS code is written into `.sass` files.
- *Leaner Style Sheets (LESS)*: This is another superset of CSS that allows you to use variables, functions, mixins, etc.

One of the advantages of using Angular CLI is that it scaffolds the build process to include preprocessing the stylesheets. While running or building the application, there is no additional effort involved in creating the scripts or running the preprocessors.

Next, Angular CLI copies files and installs the application (Listing 2-3).

Listing 2-3. Angular CLI: Create and Install Web Arcade

```
CREATE web-arcade/README.md (1055 bytes)
CREATE web-arcade/.editorconfig (274 bytes)
CREATE web-arcade/.gitignore (604 bytes)
CREATE web-arcade/angular.json (3231 bytes)
CREATE web-arcade/package.json (1072 bytes)
CREATE web-arcade/tsconfig.json (783 bytes)
CREATE web-arcade/.browserslistrc (703 bytes)
CREATE web-arcade/karma.conf.js (1427 bytes)
CREATE web-arcade/tsconfig.app.json (287 bytes)
CREATE web-arcade/tsconfig.spec.json (333 bytes)
CREATE web-arcade/src/favicon.ico (948 bytes)
CREATE web-arcade/src/index.html (295 bytes)
CREATE web-arcade/src/main.ts (372 bytes)
CREATE web-arcade/src/polyfills.ts (2820 bytes)
CREATE web-arcade/src/styles.sass (80 bytes)
CREATE web-arcade/src/test.ts (743 bytes)
CREATE web-arcade/src/assets/.gitkeep (0 bytes)
CREATE web-arcade/src/environments/environment.prod.ts
(51 bytes)
CREATE web-arcade/src/environments/environment.ts (658 bytes)
CREATE web-arcade/src/app/app-routing.module.ts (245 bytes)
CREATE web-arcade/src/app/app.module.ts (393 bytes)
CREATE web-arcade/src/app/app.component.sass (0 bytes)
CREATE web-arcade/src/app/app.component.html (23809 bytes)
CREATE web-arcade/src/app/app.component.spec.ts (1069 bytes)
CREATE web-arcade/src/app/app.component.ts (215 bytes)
⠋ Installing packages (npm)..
```

Add Service Worker

To add offline features to the Angular application, run the command in Listing 2-4.

Listing 2-4. Add Progressive Web App Features for Offline Access

```
ng add @angular/pwa

# This command install @angular/pwa on default project
# If you are running the above command on an existing Angular
# solution that has multiple projects, use
# ng add @angular/pwa --project projectname
```

Service workers are one of the features of progressive web app (PWAs). They run in the background in a browser and enable you to cache the application, including the scripts, assets, remote service responses, etc.

Traditionally, web applications are easy to deploy and manage. To add new features, developer or engineering teams deploy a new version on one or more web servers. Users access the new application and the new features when they next open the website. However, mobile apps and installed client applications (Windows or Mac) need regular updates. Installation needs to happen on thousands of client devices (even more in number, depending on the application).

However, mobile apps and client applications have an advantage that they are accessible even when the network is unavailable. For example, mobile apps in the social media space (Twitter or Facebook) may show posts even when there is no network available. A traditional web application shows a message to the effect of "page not found" when the network is unavailable. Users completely lose access to the application. They cannot view or interact with cached data.

Progressive web apps, specifically service workers, bridge this gap. You continue to take advantage of easy deployment and management of a web application. It allows you to install the application, cache scripts and assets, and more.

The command in Listing 2-4 allows you to cache scripts, assets, and data. It adds the needed configurations and registers the service worker in the Angular application module.

Run the Angular Application

So far, you have set up an environment for working with Angular applications, created a new application called Web Arcade, and installed PWA features. Next, run the bare-bones application to verify everything is working smoothly. To run the Web Arcade application, change the directory to the root of the application folder and execute the following command:

```
npm run build
#(or)
yarn build
```

The previous command runs an NPM script (namely, `build`) in the `package.json` file. This is one of the files created by Angular CLI while creating the new project. Open the file in Visual Studio Code or an IDE of your choice. You will see the scripts in Listing 2-5.

Listing 2-5. Scripts in the package.json File

```
"scripts": {
  "ng": "ng",
  "start": "ng serve",
  "build": "ng build",
  "watch": "ng build --watch --configuration development",
  "test": "ng test"
},
```

Note The build script provided just before Listing 2-5 runs `ng build`. Remember, Angular CLI's executable is `ng`. You may run `ng build` directly on the console or on the terminal. Both will result in the same outcome.

The previous build command outputs to `dist/web-arcade`. Next, run Http-Server in this directory. It starts a Node.js-based web server to render the Angular application. The web server runs on the default port 8080. See Listing 2-6.

Listing 2-6. Run Http-Server on Web Arcade Build Output

```
% http-server dist/web-arcade

Starting up http-server, serving dist/web-arcade
Available on:
  http://127.0.0.1:8080
  http://192.168.0.181:8080
```

The application is now accessible on localhost at port number 8080. Open the link `http://localhost:8080` on any modern browser. See Figure 2-3. This image is captured on Google Chrome.

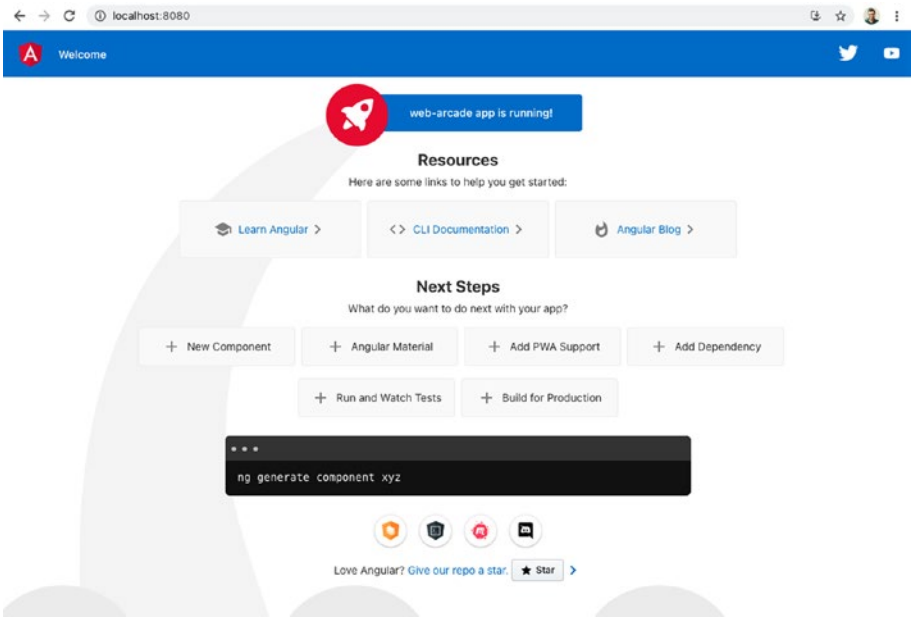

Figure 2-3. *New application running on Http-Server*

Note You can use the option `-p <new port number>` to start the Http-Server on a different port, for example `http-server dist/web-arcade -p 4100`.

Notice that it's the default content scaffolded by Angular CLI. It has useful links and documentation for continuing to build the application with Angular. In the upcoming chapters in the book and code snippets, we will enhance the code samples and create new components and services to create the Web Arcade application.

Next, open the developer tools and reload the application. On Google Chrome, you'll find Developer Tools under the menu More Tools. Navigate to the Network tab and look for `ngsw.json`. This is a configuration file for a service worker. This allows you to register the Web Arcade service worker with the browser. See Figure 2-4.

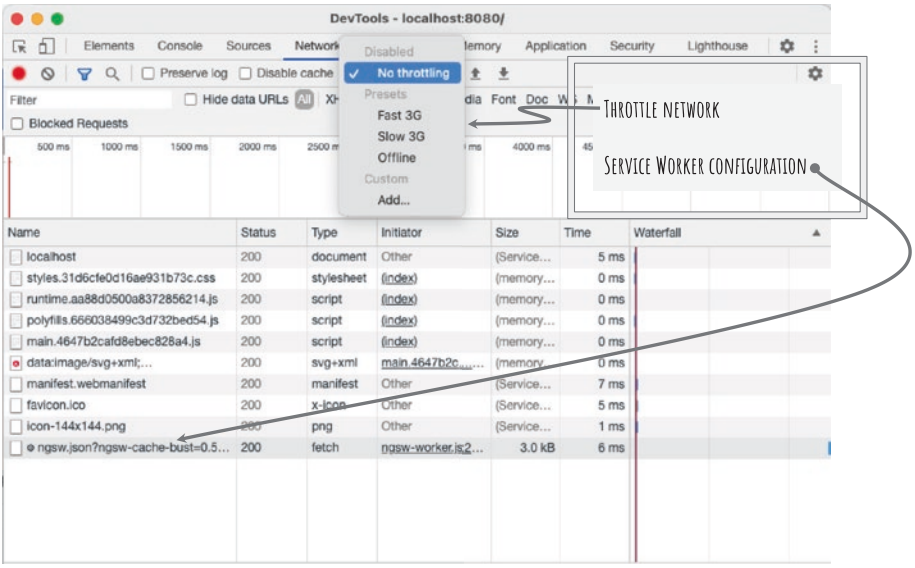

Figure 2-4. *Chrome Developer Tools for Web Arcade*

In the throttle options (on the Network tab of the developer tools), select the option Offline. Now, if you navigate to any online website in this browser, it does not load the page. However, localhost:8080 continues to work, rendered from the Service Worker cache.

Security: Service Workers Need HTTPS

A service worker is a powerful feature, which can act as a proxy to all the network requests from the application. A proxy implementation could result in security risks. Hence, browsers enforce a secure HTTPS connection. If a website without HTTPS implementation attempts to register a service worker, browsers ignore the feature.

However, notice that we did not implement an HTTPS certificate while deploying to Http-Server. Browsers do not perform this check for HTTPS on localhost. It is an exception to the rule. It helps develop the

applications easily. Implementing an HTTPS connection for development purposes could be a tedious task. And the applications accessible on localhost are installed and running on the local machine; hence, there is no security risk.

Notice that Listing 2-6 shows that the application is available on two IP addresses, shown here:

- 127.0.0.1. This stands for localhost.
- 192.x.x.x. This is the local IP address of the machine. It does not have a secure HTTPS certificate installed.

  Even though 192.x.x.x is a local IP address, browsers do not have a way to validate this. Considering the connection is not HTTPS, the browser does not register the service worker. See Figure 2-5. The browser is in offline mode. The service worker does not load the page in spite of the service worker. When you bring the application online, the service worker does not load in the Network tab.

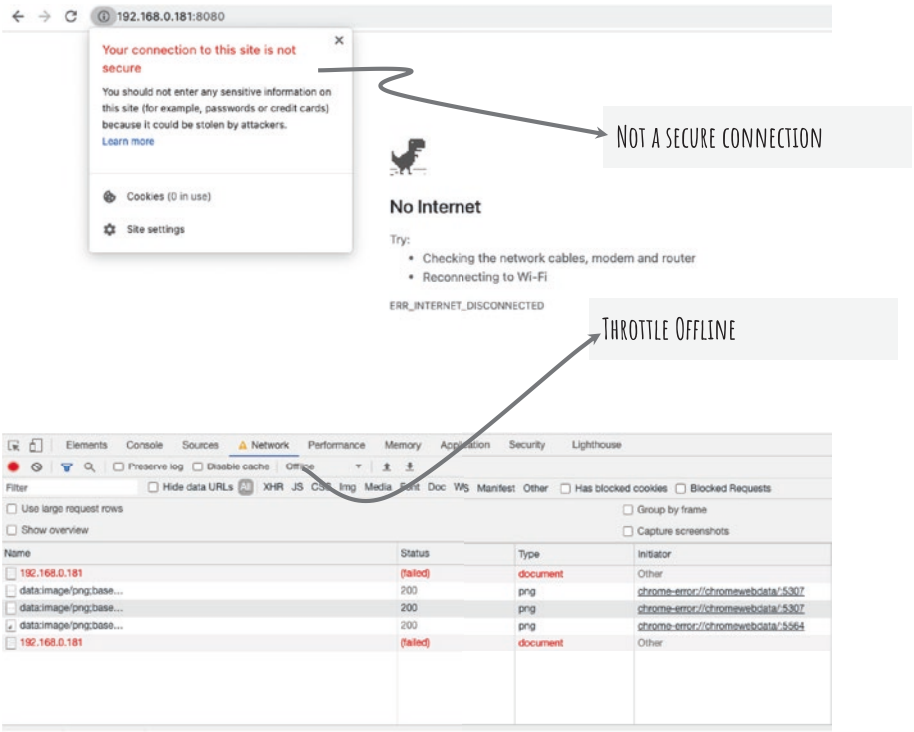

Figure 2-5. *The service worker does not work on an HTTP connection*

Working with yarn global add

Yarn provides a few commands including `add`, `list`, and `remove` to install a package, list packages, and remove packages, respectively. These packages are installed in the `node_modules` of the given directory. The directory is the scope of the command or the action.

However, with a prefix of `yarn global`, the command runs at a global level. The global level typically means for all the directories/projects of a logged-in user. Run the command in Listing 2-7 to see the global `bin` directory for Yarn. Notice that it is under the logged-in user's directory. This is the default path for a Yarn installation.

Listing 2-7. Show Global Yarn bin Directory

```
# macOS
% yarn global bin
/Users/logged-in-user/.yarn/bin

# Microsoft Windows
C:> yarn global bin
C:\Users\logged-in-user\AppData\Local\Yarn\bin
```

If this directory is not included in the environment variable `path`, packages do not work even after global installation. Set the path and retry the package installed. See Listing 2-8 to set the path.

Listing 2-8. Set Path to Yarn Global bin Directory

```
#On Microsoft Windows, set path with the following command
set PATH=%path%;c:\users\logged-in-user\AppData\Local\Yarn\bin

#On macOS, set path with the following command
export PATH="$(yarn global bin):$PATH"
```

Summary

The chapter was a getting-started guide for building an Angular application with the service worker for offline features. It provided a list of prerequisite software and libraries, along with download and installation instructions. Most software and libraries used in this book are open source and free.

EXERCISE: CREATING THE WEB ARCADE APPLICATION

Set up a development environment on your computer for Angular. Be sure to install Node.js.

1. Pick a package manager for installing and managing the Angular and TypeScript (or JavaScript) packages. Choose to use NPM or Yarn. Preferably, stay with your decision for all the future exercises.
2. Install and verify Angular CLI, preferably at the global level.
3. Install Http-Server, preferably at the global level.
4. Create a new Angular project and add `@angular/pwa`.
5. Build and deploy the first version of the project on Http-Server.
6. Review and ensure the service worker is available when the network is unavailable.

CHAPTER 3

Installing an Angular Application

This chapter begins by providing instructions to create new screens and components in an Angular application. It provides introductory details about a component and then provides enough details to build an offline Angular application. If you are looking for in-depth information about a component and Angular concepts, read the book *Angular for Material Design* or refer to the Angular documentation provided in the references at the end of the book. Toward the end of this chapter, you will package the application to be installed on a client device (desktop or mobile).

Angular Components

A web application is a composition of many web pages. In a web page, the view that the user interacts with, including labels, text fields, buttons, etc., is built with HTML. Document Object Model (DOM) nodes compose an HTML page. The DOM is organized as a tree. The HTML page starts with a root node, typically an `html` element (`<html></html>`). It has child nodes, and the child nodes have more child nodes.

As you can imagine, all the browsers know about these HTML elements. They have built-in implementations to render each element in the HTML page. For example, if a browser encounters an `input` element

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4_3

(`<input />`), it shows a text field; the browser will display the text in a `strong` element (`<strong>text</strong>`) in bold.

However, are we limited to predefined elements in HTML? What if you wanted to create new elements that encapsulate a view and behavior? Imagine that you want to build a die for Web Arcade.

Create a Component

Angular components enable developers to build custom elements. Components are the building blocks of an Angular application. This section provides instructions for how to create a new component.

As you saw in Chapter 2, you will use Angular CLI to create and manage an Angular application. When you set up the development environment in the previous chapter, you installed Angular CLI, so it should be ready to use.

To create a new component, run the following command:

```
% ng generate component components/dice
```

As you saw earlier, the `ng` executable runs Angular CLI.

- The `generate` command with Angular CLI creates files.
- Next, the `component` collection when used with `generate` specifies how to add component files.
- The third parameter specifies the component name, `dice`. Prefixing the value with `components/` creates it under the folder `components`. It creates the folder if it does not already exist.

Note Alternatively, you may use the following short form, which uses g for generate and c for component.

```
% ng g c components/dice
```

Angular CLI generates the following files in a new folder called `src/app/components/dice`. See Figure 3-1.

- `dice.component.html`: An HTML template file for the markup. To create a view in a web page, you use HTML. A side of a die is created with an HTML template.
- `dice.component.sass`: A stylesheet file contains SASS styles for the look and feel of the component. It includes colors, text decoration, margins, etc.
- `dice.component.spec.ts`: A TypeScript file for unit tests.
- `dice.component.ts`: A TypeScript file for the functional implementation of the `dice` component.

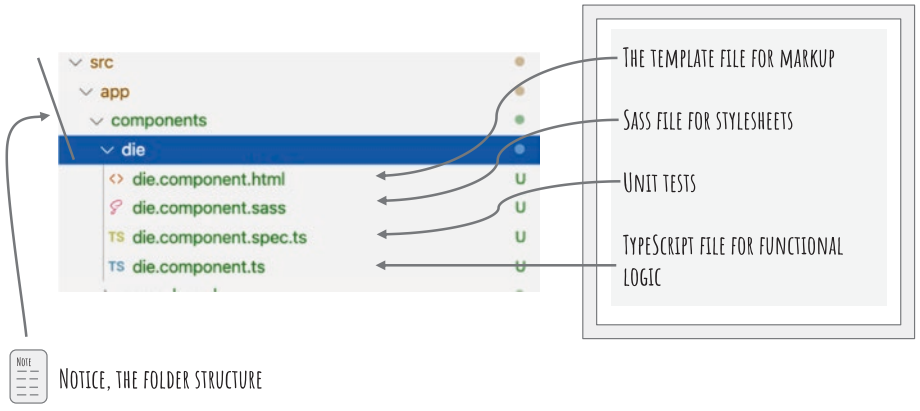

Figure 3-1. *Component files generated with Angular CLI*

Web Arcade: Create a Die

This section details the code for the `dice` component. It is an opinionated take on creating a `dice` component. Refer to the "Exercise" section for additional ideas on building a die.

Styles for the Component

Stylesheets manage the look and feel, colors, font, etc., for a component. The styles can be applied on elements of the component. It is a good practice to scope the stylesheet local to the component. This is the default behavior in an Angular application. This section details how to create a stylesheet for the `dice` component.

Notice that, in the Web Arcade code sample, the `src/assets` directory has six PNG files for each side of a die. Use these images in the stylesheets to show the sides of a die. Begin by creating variables in the stylesheet for each side. Consider Listing 3-1 for a list of variables with a URL reference to the sides of the die images. The left side of the colon (`:`) is a variable name. The right side of the colon is a value, in this case a PNG image of the die. Use the `url()` function to include the image file in CSS.

Listing 3-1. Sides of the Die Images

```
// Variables in SASS
$side1:url('/assets/side1.png')
$side2:url('/assets/side2.png')
$side3:url('/assets/side3.png')
$side4:url('/assets/side4.png')
$side5:url('/assets/side5.png')
$side6:url('/assets/side6.png')
```

Next, create CSS classes for each side of a die, usable on a `div` element. See Listing 3-2.

Listing 3-2. Code for the CSS Class That Defines Each Side of a Die

```
div.img-1
  background-image: $side1

div.img-2
  background-image: $side2

div.img-3
  background-image: $side3

div.img-4
  background-image: $side4

div.img-5
  background-image: $side5

div.img-6
  background-image: $side6
```

Each `img-x` class (for example, `img-6`) is a CSS class. It is prefixed with a `div`. In the HTML, the CSS class `img-6` can be applied only on a `div` element.

Note Remember, SASS files do not use curly braces or semicolons. Notice that the indentation of the CSS style under the element and class name. That is, the style background image named `$side6` relates to `div.img-6` as it is indented a tab further, indicating it relates to the CSS class.

TypeScript: Functional Logic for the Component

Each component has at least one TypeScript file and a TypeScript class. A component's functional/behavioral logic is written in this class. For example, consider the logic to roll a die and generate a random number between 1 and 6. That is the functional/behavioral logic of the `dice` component. This section covers how to create the TypeScript class for the `dice` component.

Output (Event Emitter) from the Component

So far, you have seen that the component has a stylesheet to show any one of the six sides of a die. When you roll a die, it draws one of the six numbers. The code making use of the `dice` component might need the result of rolling a die. Consider this the output. Create an `EventEmitter` object with an output decorator for the output. Use the `emit()` function on this object to output the value outside the component. Continue reading for further explanation and the code samples.

Input to the Component

The component might also allow a value to be set from outside the component. As long as it is a legal value (a value between 1 and 6), the component can show on the die. Consider this the input. Create a class-level variable and decorate it with `Input()`. This acts as an attribute to the component. To this attribute, an input value can be provided by the code using the component.

Consider the TypeScript shown in Listing 3-3 and Figure 3-2. Notice the highlighted text in bold, lines 7 and 8. The `Input()` decorator allows the variable value set from outside the component. The `Output()` decorator enables the `rollResult` value emitted from the component.

Listing 3-3. Dice Component TypeScript File

```
import { Component, Input, OnInit, Output, EventEmitter } from
'@angular/core';
1. @Component({
2.  selector: 'wade-dice',
3.  templateUrl: './dice.component.html',
4.  styleUrls: ['./dice.component.sass']
5. })
6. export class DiceComponent implements OnInit {
7.   @Input() draw: string = '';
8.   @Output() rollResult = new EventEmitter<number>();
9.   constructor() { }
10.  ngOnInit(): void { }
11. }
```

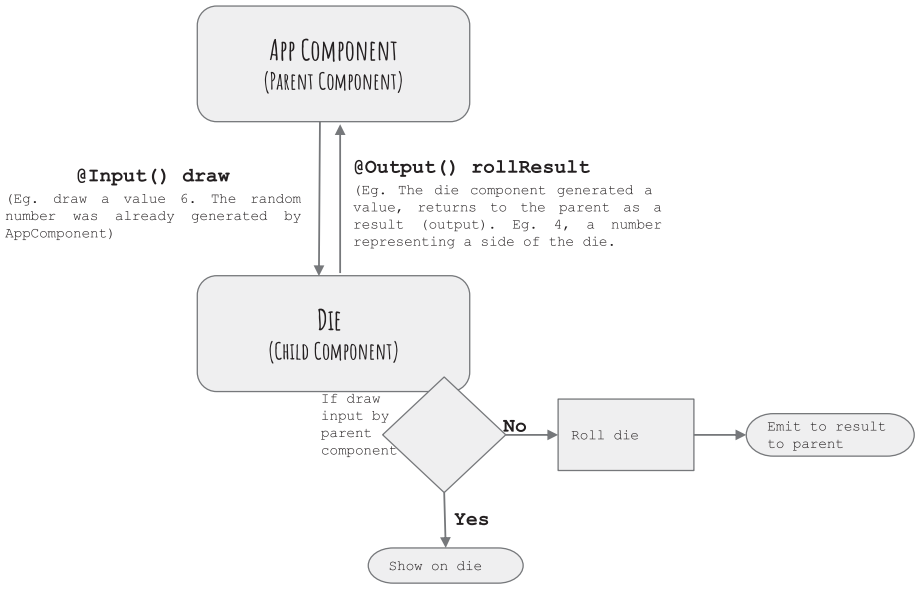

Figure 3-2. *Input and output between the dice and app components*

Also notice Listing 3-3, lines 1 to 5. The component decorator specifies metadata for the component.

- `selector`: As described earlier, components are reusable custom elements. While using the component in an HTML file, you will refer to the component using this value. In this example, see `<wade-dice> </wade-dice>`.

Note `wade-` is an arbitrary prefix chosen for all the components in the Web Archive code samples. The default used by Angular is `app`.

- `template-url`: This refers to the HTML file in use for the component. See Figure 3-1. It shows the HTML file created for the `dice` component.
- `style-urls`: This refers to the stylesheet files in use for the component. There can be more than one stylesheet. Hence, it is an array of values. See Figure 3-1. It shows the SASS file created for the `dice` component.

Notice the constructor shown in line 9. It instantiates the TypeScript class. The function in line 10, `ngOnInit()`, is an Angular lifecycle hook invoked after the constructor. Hence, when this function is invoked, the class variables are already instantiated. It is an ideal place for setting the context for Angular components.

In the current `dice` component case, we set the context by rolling the die, generate a random number between 1 and 6, and show that side of the die. Alternatively, the `dice` component also allows you to set a value externally, from outside the component. Consider Listing 3-4.

Listing 3-4. ngOnInit() Hook for the Component

```
01: @Input() draw: string = '';
02: @Output() rollResult = new EventEmitter<number>();
03:
04: constructor() { }
05:
06: ngOnInit(): void {
07:   if(this.draw){
08:     this.showOnDice(+this.draw);
09:   } else {
10:     this.rollDice();
11:   }
12: }
```

Note TypeScript class variables and methods (functions) are accessed with the `this` keyword.

Notice lines 1 and 7. `this.draw` is an input attribute to the component. If an input is provided to the component, you show the value on the die. The Angular code using the component explicitly provides a value. You do not need to roll the die to generate a random number. See Listing 3-5.

On the other hand, when no input is provided, roll the die, generate a random number, and show the value on die. See listing 3-6.

Note Notice the + prefix on line 8 of Listing 3-4. The input attribute is a string value. The + prefix typecasts the string value to a number.

Listing 3-5. Show the Given Number on a Die

```
/*
     At class level, a variable selectedDiceSideCssClass is
     declared.

selectedDiceSideCssClass: string = '';
/*
01:  // show the given number (draw parameter) on the dice
02:   showOnDice(draw: number){
03:     // the css class img-x show appropriate side on
           the dice.
04:     switch (draw) {
05:       case 1: {
06:         this.selectedDiceSideCssClass = 'img-1';
07:         break;
08:       }
09:       case 2: {
10:         this.selectedDiceSideCssClass = 'img-2';
11:         break;
12:       }
13:       case 3: {
14:         this.selectedDiceSideCssClass = 'img-3';
15:         break;
16:       }
17:       case 4: {
18:         this.selectedDiceSideCssClass = 'img-4';
19:         break;
20:       }
21:       case 5: {
22:         this.selectedDiceSideCssClass = 'img-5';
23:         break;
```

```
24:       }
25:       case 6: {
26:         this.selectedDiceSideCssClass = 'img-6';
27:         break;
28:       }
29:       default: {
30:         break;
31:       }
32:     }
33:   }
```

Remember the stylesheet in Listings 3-1 and 3-2. They define CSS classes `img-1` to `img-6`, which show images depicting the six sides of the die. Listing 3-5 sets an appropriate CSS class name to a variable called `selectedDiceSideCssClass`. You will use this CSS class in the HTML template.

To roll the die (when no input provided), use the function `rollDice()`.

Listing 3-6. Roll the Die

```
01: rollDice(){
02:     let i = 0;
03:
04:     // run the provided function 25 times depicting a
           rolloing dice
05:     const interval = setInterval(() => {
06:
07:       // random number generator for numbers
             between 1 and 6
08:       let randomDraw = Math.round((Math.random()*5) + 1);
09:       this.showOnDice(randomDraw);
10:
```

```
11:       // After 25, clear the interval so that the dice
             doesn't roll next time.
12:       if(i > 25) {
13:         clearInterval(interval);
14:         this.rollResult.emit(randomDraw);
15:       }
16:
17:       i += 1;
18:
19:     }, 100);
20: }
```

The function attempts to mimic a rolling die. Hence, it sets values on the die for 25 times (an arbitrary number). It runs the code every 100 milliseconds. See the lines between 5 and 19.

- A `setInterval` JavaScript function accepts a callback function as the first parameter.
- The second parameter indicates the number of milliseconds after which the first callback function runs.

See Listing-3-7 for a short and empty snippet for easier understanding.

Listing 3-7. The setInterval() Function

```
setInterval(() => { }, // first parameter, callback
 100 // second parameter, interval duration in
     // milliseconds
);
```

For generating a random number, refer to line 8 in Listing 3-6.

- `Math.random()` generates a value between 0 and 1.

- Multiplying this number by 5 limits the value to between 0 and 5.
- A die doesn't show a decimal value; hence, use the JavaScript function `Math.round()` to round the number.
- A die doesn't show a zero; hence, add 1.

Note When do you use `rollResult`? Imagine moving a piece in a board game based on a number drawn by the die. The board game component uses a random number result from the `dice` component. The `dice` component emits the number for the board game.

In an example, say the die draws 6. A piece on Monopoly should move six places. The `dice` component shows 6 and emits the number. The Monopoly component receives 6 and moves a piece by six positions.

Refer to the complete code snippet of the TypeScript class file. See Listing 3-8.

Listing 3-8. Dice Component, TypeScript File

```
import { Component, Input, OnInit, Output, EventEmitter } from
'@angular/core';

@Component({
  selector: 'wade-dice',
  templateUrl: './dice.component.html',
  styleUrls: ['./dice.component.sass']
})
export class DiceComponent implements OnInit {

  @Input() draw: string = '';
```

```
  @Output() rollResult = new EventEmitter<number>();

  selectedDiceSideCssClass: string = '';

  constructor() { }

  ngOnInit(): void {
    if(this.draw){
      this.showOnDice(+this.draw);
    } else {
      this.rollDice();
    }
  }

  // show the given number (draw parameter) on the dice
  showOnDice(draw: number){
    // the css class img-x show appropriate side on the dice.
    switch (draw) {
      case 1: {
        this.selectedDiceSideCssClass = 'img-1';
        break;
      }
      case 2: {
        this.selectedDiceSideCssClass = 'img-2';
        break;
      }
      case 3: {
        this.selectedDiceSideCssClass = 'img-3';
        break;
      }
      case 4: {
        this.selectedDiceSideCssClass = 'img-4';
        break;
      }
```

```
      case 5: {
        this.selectedDiceSideCssClass = 'img-5';
        break;
      }
      case 6: {
        this.selectedDiceSideCssClass = 'img-6';
        break;
      }
      default: {
        break;
      }
    }
  }

// generate a random number between 1 and 6
// and set on the dice
  rollDice(){
    let i = 0;

// run the provided function 25 times depicting a rolling dice
    const interval = setInterval(() => {

      // random number generator for numbers between 1 and 6
      let randomDraw = Math.round((Math.random()*5) + 1);
      this.showOnDice(randomDraw);

      // After 25, clear the interval so that the dice doesn't
         roll next time.
      if(i > 25) {
        clearInterval(interval);
        this.rollResult.emit(randomDraw);
      }
```

```
      i += 1;
    }, 100);
  }
}
```

HTML Template

Refer to the HTML template in Listing 3-9 for the view that the user interacts with.

Listing 3-9. Dice Component, HTML Template

```
<div
  class="dice"
  [ngClass]="selectedDiceSideCssClass"
></div>
```

Notice that the CSS class value `dice` is static. It provides padding, height, and width that does not change on the fly while the die rolls.

`ngClass`, on the other hand, uses property binding to set the CSS class dynamically. It is an Angular attribute directive. `ngClass` applies CSS classes provided through the variable `selectedDiceSideCssClass` on the `div` element. See the `showOnDice()` function in Listing 3-8. It conditionally selects a CSS class name.

Note An attribute directive allows you to change the appearance and behavior of a DOM element. `ngClass` is a built-in directive provided by Angular to update CSS classes on the fly. It's highly useful for controlling the look and feel of an element dynamically.

Property binding enables one-way data binding TypeScript variables on the HTML properties.

Now, the `dice` component is ready to use. Go to `app.component.html` and remove the default content that resulted in Figure 2-3 of Chapter 2. It was a placeholder for the application created with Angular CLI. Remember, the selector for the `dice` component is `wade-dice`. Use it in the app component of the HTML template. See Listing 3-10.

Listing 3-10. App Component of the HTML Template

```
<div class="container align-center">
   <wade-dice></wade-dice>
</div>
```

Note At this point, the `div` and the two CSS classes `container` and `align-center` do not have a major significance. They help present content better on the page.

Remember the code from Listing 3-4. The dice component generates a random number if you do not provide a value with the draw attribute (input). Hence, Listing 3-10 rolls the die, generates a random number, and sets it on the die. Instead, if you use the attribute draw, it does not roll the die. It just shows side 4 of the die.

```
<wade-dice draw="4"></wade-dice>
```

See Figure 3-3 for the result.

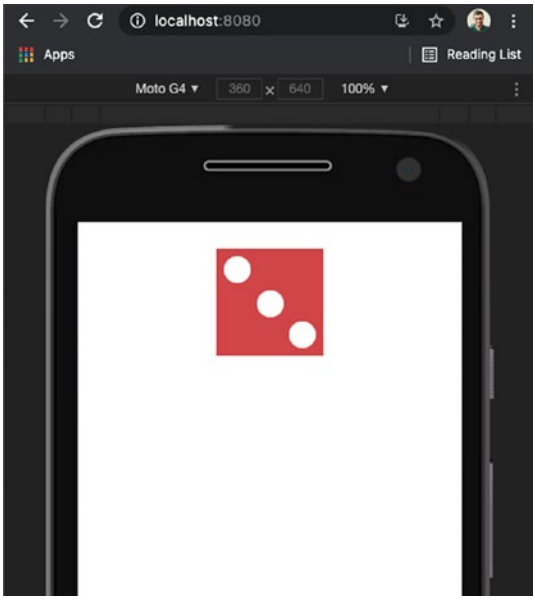

Figure 3-3. *Using the dice component in the app component*

Service Worker Configuration

Next, you're ready for the application to be installed. Once that's done, we can review how to install the application on a desktop.

In Chapter 2, when you installed `@angular/pwa`, it created the following configurations. The `ng add @angular/pwa` command adds the `@angular/service-worker` package. Consider the following updates to the application:

1. The command adds the `manifest.webmanifest` file to the application. When you load the application on a desktop or a mobile browser, it identifies a progressive web app with the help of this configuration file. The `ng add @angular/pwa` command updates `index.html` and adds a link to this configuration file.

2. The command adds the `ngsw-config.json` file to the application. This is an Angular-specific configuration file for a service worker. It is used by the CLI and build process. It configures the caching behavior for the application.
3. In Angular's application module, it imports the service worker module and registers it.

Note While loading the web application, the `ngsw-config.json` file is fetched every time from the web server. It is not cached with the service worker. It helps identify changes to the application and fetches a completely new version.

Create Icons

A PWA needs a variety of icon files. Remember, it is an installable application now. You need the launch icons with varying resolutions. These icons are used on mobile device home screens among apps, shortcuts on a desktop, on the Windows taskbar or macOS Dock, etc. The default icon is an Angular logo. You can use the icons in the Web Arcade code sample located at `web-arcade/src/assets/icons`.

The service worker application needs icons with the following resolutions (in pixels):

- 72 × 72
- 96 × 96
- 128 × 128
- 144 × 144
- 152 × 152

- 192 × 192
- 384 × 384
- 512 × 512

Copy the icons to the folder `<your-project-folder>/src/assets/icons`. Run the build command. Notice that the icons are copied over to the deployable directory along with the application bundle. Ensure Http-Server is running in the `dist/web-arcade` directory so that the URL, `http://localhost:8080`, continues to serve the application.

Launch the application in a new browser window that supports service workers. Notice that there is an install button for the Web Arcade application. Figure 3-4 shows the install option on Google Chrome on a desktop.

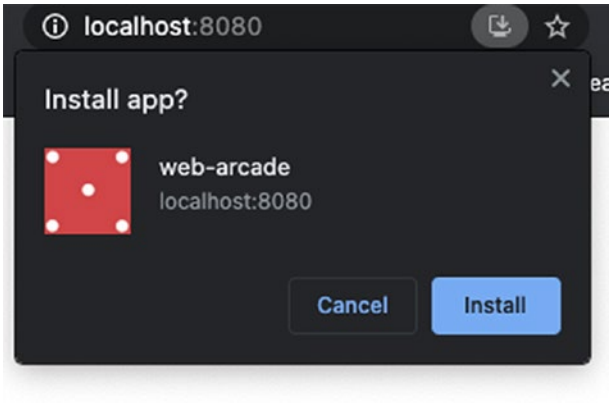

Figure 3-4. *Installing the service worker*

Note Launching the application in a new browser window (or a tab) ensures the old version of the app is not served from the service worker cache. Instead, it identifies an updated application bundle on the web server and downloads the new application.

Click Install. It's now available on the desktop. See Figure 3-5.

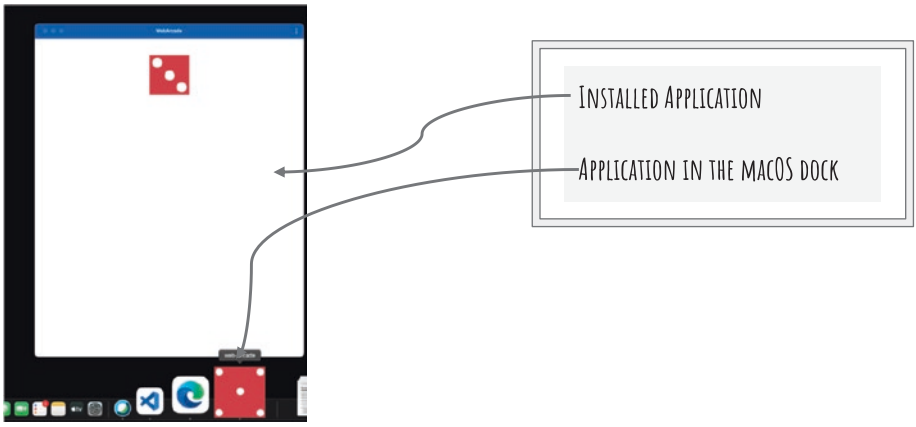

Figure 3-5. *Installed application on macOS*

Summary

The chapter continued to build on the Web Arcade sample application. In the process, the chapter detailed Angular and service worker concepts. It also developed the stylesheet for the die and concluded by installing the application on a desktop and mobile device and listing the configurations added by the package `@angular/pwa`.

EXERCISE

- The code sample described in the chapter does not show a button to repeatedly roll the die. Add a button to the `dice` component to roll again. Use click events to handle rolling the die on demand.
- Create sides of a die with CSS instead of using images.
- Create a die with 8 or 12 sides.
- Explore animations for rolling a die.

CHAPTER 4

Service Workers

Service workers run in the background on your browser. They provide a foundation for modern web applications and are installable, work offline, and are reliable in low-bandwidth situations. This chapter provides an introduction to service workers. It discusses the caching capabilities of service workers and how to use them in an Angular application. It details the lifecycle of a service worker. Next, the chapter discusses Angular's configurations and features while working with the service workers. It explains how to implement a cache for the Web Arcade sample application. Toward the end, it provides details about browser compatibility.

Service workers are a network proxy running on the browser. They can intercept outgoing network requests from the browser. These requests include an application's JavaScript bundle files, stylesheets, images, font, data, etc. You may program a service worker to respond to a request from the cache. This enables web applications to be resilient to network speeds and a loss of connectivity. Unlike a traditional web application, which returns a "page not found" error when you lose connectivity, service workers enable the application to utilize installed and cached resources. You can program the application to load cached data or show a graceful error message. Even in low-bandwidth scenarios, service workers enable you to build fluid and responsive applications with great user experiences.

The service workers are preserved even after the application or the browser is closed. To see a list of active service workers, navigate to

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4_4

the page `chrome://inspect/#service-workers` on Google Chrome or `edge://inspect/#service-workers` on Microsoft Edge. See Figure 4-1. Notice the popular websites including Angular's Angular.io among the applications that use the service workers. Also notice that the Web Arcade sample application's dev URL localhost:8080 has registered a service worker.

Figure 4-1. *Inspecting the service workers on Google Chrome*

Note To see all the service workers registered by various web applications you have accessed on your computer, launch the service worker internals page. Notice that the first URL, `chrome://inspect/#service-workers`, listed only the active service workers. Access the service worker internals by navigating to `chrome://serviceworker-internals` on Google Chrome (or `edge://serviceworker-internals` on Microsoft Edge).

Please note this page might be deprecated in the future. The `chrome://inspect/#service-workers` URL might include all the service worker debug features.

Service Worker Lifecycle

This section details the service worker lifecycle and its states running in the background (on the browser). See Figure 4-2, which depicts a service worker lifecycle. It begins by registering a new service worker. A web application using a service worker registers on loading in the browser. The "register" can occur every time a user loads the application. The browser ignores a fresh registration if the service worker has already been registered.

Successful registration of a service worker triggers an install event. A typical install event handles the caching logic. All the static resources including application bundles, images, fonts, and stylesheets are cached during the install event. These are configurable.

The service worker installation is atomic. A failure in downloading and caching one or more resources causes the event to error out completely. The next time the user accesses the website, it attempts to install it again. This is to ensure there are no partially installed applications causing unforeseen problems and bugs.

While the application is open, an installed service worker is activated. It runs in the background and acts as a proxy to all the network calls. Depending on the application logic and configuration, you may serve the data from the cache. If the data is not found in the cache, invoke the network service and retrieve data over the network.

If the application or the service worker is not in use, the service worker terminates to save memory. When needed, it activates the service worker. Notice the terminate button in Figure 4-1 (in the browser window) for manually terminating an active service worker. You may use this to force close the service worker. It helps relaunch the service worker afresh. If your computer is running low on resources, you may terminate a service worker to save memory. Also, notice the Inspect link, which launches the dev tools allowing you to explore network resources and the application source code.

See Figure 4-2 for a depiction of the workflow and events.

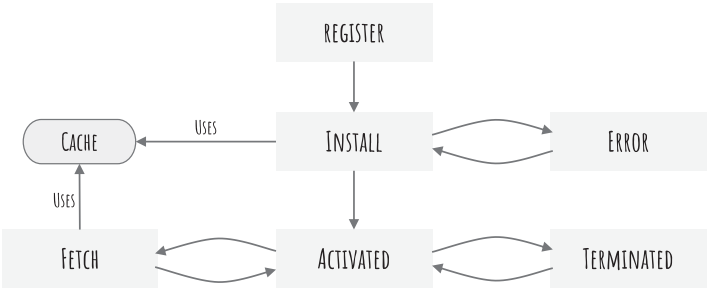

Figure 4-2. *Service worker lifecycle*

Service Worker in an Angular Application

Angular makes it easier to use the service worker and caching features in your application. Angular scaffolds a lot of the functionality described in the previous section "Service Worker Lifecycle," especially the caching features. This section details out-of-the-box Angular features for integrating the service workers.

Angular CLI generates `ngsw-config.json` when you add `@angular/pwa` to the project. It provides a service worker configuration for an Angular application. The Angular build process uses this configuration. One of the aspects of configuration is a list of static and dynamic resources to be cached and installed. Static resources include JavaScript bundle files that constitute the application, stylesheets, images, font, and icons. Typical dynamic resources include data responses.

A `ngsw-config.json` file includes the following sections:

- `appData`: Remember, the Web Arcade Angular application is installable and maintains versions. This field in the configuration provides a brief description of the application version. As you update the application,

use this field to provide meaningful details about the upgrade and the version of the software.

- `index`: This specifies the root HTML file for the Angular application and the single-page application (SPA). In the Web Arcade sample application, it is `index.html` in the `src` directory. With this field index, you are providing a link to the starting point of the application. As you will see next, Web Arcade caches this file using the service worker.
- `assetGroups`: This is a configuration for assets, typically JavaScript application bundles, stylesheets, font, images, icons, etc. The assets could be part of the Angular project or downloaded from a remote location like a content delivery network (CDN).
    - Notice Web Arcade's `ngsw-config` file in Listing 4-1. It includes files that constitute the application, i.e, `index.html`, all the JavaScript bundles, and the CSS files. It also includes assets such as images, icons, fonts, etc.

Note Remember, in Web Arcade the SASS files compile to CSS. Service workers are working on the build output of the Angular application. All the files including JavaScript bundles, compiled CSS, images, etc., are relative to the `dist` directory (output of the `yarn build` command).

---

1. You can configure multiple `assetGroups`. Notice that the field is an array. You can list JSON objects with the configuration details. An `assetGroup` object defines the following fields:

a. `name`: This is an arbitrary title for an asset group.

b. `resources`: The resources are the files or URLs to be cached by the service worker. As mentioned, the files could be JavaScript files, CSS stylesheets, images, icons, etc. On the other hand, for resources such as fonts (and a few other libraries), you may use CDN locations, which are URLs.

   i. `files`: This is an array of files configured for the service worker to cache.

   ii. `urls`: This is an array of URLs configured for the service worker to cache.

   At build time, it is not likely you will know every file to be cached. Hence, the configuration allows you to use file and URL patterns. See the section "Pattern Match Resources to Cache" for more details.

c. `installMode`: Install mode determines how to cache the resources for the first time when there is no existing version of the service worker on the browser. It supports two modes of caching.

   iii. `prefetch`: Cache all the resources, files, and URLs at the beginning. The service worker does not wait for the resource to be requested by the application. As and when the application requests, the resource is readily available in the cache.

   This approach is useful for the root `index.html` file, core application bundles, primary stylesheets, etc. However, the prefetch install mode could be bandwidth intensive.

Prefetch is the default install mode, when no configuration value is provided.

iv. `lazy`: Cache resources only when the application requests it for the first time. If a particular resource is configured but never requested, it is not cached. It is efficient. However, the resource is available offline only from the second use.

2. `updateMode`: Update mode determines how to cache the resources when a new version of the application is found. This is for a service worker (an Angular application) that is already installed in the browser. As you know, unlike a typical web application, the service worker enables caching an Angular application. It also allows you to discover and install updates as and when available. It supports two modes of caching.

   a. `prefetch`: Download and cache all the resources, files, and URLs while updating the application. The service worker does not wait for the resource to be requested by the application. As and when the application requests a resource, it is readily available in the cache.

      `Default`: When no configuration value is provided, it uses the value set for `installMode`.

   b. `lazy`: Cache resources only when the application requests it for the first time. If a particular resource is configured but never requested, it is not cached. This is efficient. However, the resource is available offline only from the second use.

      This configuration value is overridden if `installMode` is `prefetch`. For it to cache truly in lazy mode, `installMode` needs to be lazy as well.

3. `dataGroups`: While `assetGroups` enables caching application assets, largely static resources, `dataGroups` help cache dynamic data requests. It is an array of data group objects. You can configure multiple `dataGroups`. You can list JSON objects with the configuration details. A `dataGroup` object defines the following fields:

   a. `name`: This is an arbitrary title for a data group.

   b. `urls`: This is an array of strings that configure a list of URLs or a list of patterns matching the URLs. Unlike `assetGroups`, the pattern doesn't support matching with `?` as it is a common character for query strings in a URL.

   c. `version`: This helps identify the availability of a new version of `dataGroup` resources. The service worker discards old versions of the cache, fetches new data, and caches the new URL responses. If a version is not provided, it defaults to 1.

      Versioning data groups is useful especially when a resource is incompatible with the old URL responses.

   d. `cacheConfig`: This defines a configuration for the data cache policy. It includes the following fields:

      i. `maxSize`: This defines an upper limit for the size of the data to be cached. It is a good practice to limit the size by design. Browsers (like any other platform) manage and allocate memory for each application. If the application exceeds the upper limit, the entire dataset and the cache could be evicted. Hence, design a system to limit the cache size and prevent unforeseen results caused due to eviction.

ii. `maxAge`: `dataCache` is dynamic in nature. Often data changes at the source. Caching data for too long could cause the application to use obsolete fields and records. A service worker configuration provides a mechanism to automatically clear the data at periodic intervals, ensuring the application does not use stale data. In an example, imagine interest rates are updated once in a day. This means the cached interest rate values need to expire in 24 hours. On the other hand, users' profile pictures are rarely updated. Hence, they can be stored in the cache longer.

    You may qualify the max age value with one or more of the following:

    `d` stands for days. For example, use `7d` for seven days.

    `h` stands for hours. For example, use `12h` for 12 hours.

    `m` stands for minutes. For example, use `30m` for 30 minutes.

    `s` stands for seconds. For example, use `10s` for ten seconds.

    `u` stands for milliseconds, For example, use `500u` for half a second.

    You can mix and match to create a composite value. For example, `2d12h30m` stands for 2 days, 12 hours, and 30 minutes.

iii. `timeout`: Depending on the `dataCache` strategy (see the next item), often data requests attempt to use responses over the network. Only if the network request is taking too long (or fails), it uses the cached response.

The timeout defines a value after which the service worker ignores the network request and responds with a cached value.

You may qualify the timeout value with one or more of the following:

`d` stands for days. For example, use `7d` for seven days.

`h` stands for hours. For example, use `12h` for 12 hours.

`m` stands for minutes. For example, use `30m` for 30 minutes.

`s` stands for seconds. For example, use `10s` for ten seconds.

`u` stands for milliseconds, For example use `500u` for half a second.

You can mix and match to create a composite value. For example 2d12h30m stands for 2 days, 12 hours, and 30 minutes.

iv. `strategy`: A service worker can use one of the following two strategies:

- `performance`: For few data requests, the Angular application may prioritize performance, instructing the service worker to use cached responses. The response is returned faster as it comes from the local cache. A new network service request is sent only after `maxAge` (see the previous bullet point on `maxAge`). In an example, it is useful with the interest rates request that updates nightly. Imagine `maxAge` is set to `1d`, and the service worker uses the cache for 24 hours after which the cache expires.

- `freshness`: In many cases, the Angular application configures the service worker to fetch data over the network first, before using the cached data. Imagine on a slow network, if the data request timed out, the service worker uses the cache so that the application is still usable.

Web Arcade's Service Worker Configuration

Consider Listing 4-1. It is the default configuration file generated for the Web Arcade project.

Listing 4-1. ngsw-config.json File

```
--- ngsw-config.json ---

01: {
02:   "$schema": "./node_modules/@angular/service-worker/
      config/schema.json",
03:   "index": "/index.html",
04:   "assetGroups": [
05:     {
06:       "name": "app",
07:       "installMode": "prefetch",
08:       "resources": {
09:         "files": [
10:           "/favicon.ico",
11:           "/index.html",
12:           "/manifest.webmanifest",
13:           "/*.css",
14:           "/*.js"
```

```
15:         ]
16:       }
17:     },
18:     {
19:       "name": "assets",
20:       "installMode": "lazy",
21:       "updateMode": "prefetch",
22:       "resources": {
23:         "files": [
24:           "/assets/**",
25:           "/*.(eot|svg|cur|jpg|png|webp|gif|otf|ttf|woff|
              woff2|ani)"
26:         ]
27:       }
28:     }
29:   ]
30: }
```

- Notice the field `assetGroups` on line 4. Between lines 5 and 17 is the first asset group object. This object details resources to be cached by the service worker.
    a. The field `name` is an arbitrary title for the `assetGroup` (line 6). It uses an arbitrary name `app`, which is representative of primary application resources, like the JavaScript application bundles, stylesheets, `index.html` file, etc.
    b. The first few resource files between lines 9 and 12 include the following:
        i. A fav-icon that is shown along with the title of the application.

ii. The `index.html` root HTML file for the Web Arcade application.

iii. A web manifest configuration, which identifies the application as a progressive web app.

c. Notice asterisks for lines 13 and 14, instructing how to cache all the JavaScript and CSS files (file names ending with `js` and `css`). See the "Pattern Match Resources to Cache" section to find out more about pattern matching resources to the cache.

- Notice the install mode is `prefetch` on line 7. It enables the service worker to download all the assets at the beginning, regardless of whether they are utilized immediately or not. In an example, a few CSS or JS files may not be used on load. They may be used only after navigating to a different route or a page. However, prefetch install mode downloads the entire list of files.
- Considering these files constitute the application, it is appropriate to download the entire asset group at the beginning. Do not always use this install mode as it could cause a large number of network requests, slowing down the application and creating redundant network traffic.
- Notice the second asset group between lines 18 and 28.

  a. The asset group is named `assets` (line 19). These are static resources typically including images, icons, fonts, etc.

b. Resource files include all the files under the folder `/assets`. See line 24. Notice the usage of wildcard character asterisks. It refers to all the files and directories under the directory `assets`. Remember, Web Arcade has six die images for each side in the `assets` directory.

c. See line 25. It instructs the app to cache all the files in the given list of extensions. The list of extensions indicates the font and image files.

- Notice that install mode is `lazy` on line 20. It enables the service worker to download the files only when required. Unlike the first asset group, the service worker initiates the download of the files only when the application requests.

Note Update mode in line 21 is used while updating a new version of the service worker for the application. Chapter 6 details an approach and a strategy for updating service workers.

Pattern Match Resources to Cache

In Listing 4-1, notice the resource file path in lines 8, 9, 22, and 23. They follow a pattern. As you can imagine, it is not possible to list all the resources (files and URLs) individually while developing the application. The list could be dynamic. Even if they are all known, it is a tedious job to list every single asset in a large project.

Use pattern matching to list the resources. The following are a few syntaxes to pattern match a link to a file or a URL:

- Use two asterisks (**) to pattern match path segments. This is typically done to include all files and the child directories. In an example, the `assets` directory has another child directory called `icons` and a list of die images. See Figure 4-3. To include all the files and directories under `assets`, use `/assets/**`.

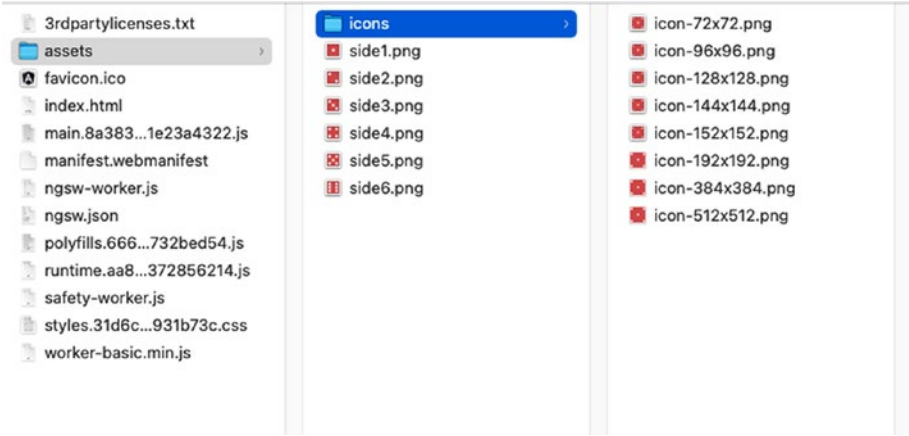

Figure 4-3. *Assets directory*

- To include any file name or any number of characters, use a single asterisk (*). This matches zero or more characters. It does not include child directories.

    a. In an example, `assets/*` includes all the files in the directory assets. However, it does not include icons directory. To explicitly include the `icons` directory, use `assets/icons/*`, which includes all the files under the directory `assets/icons/`.

b. In another example, imagine you need to include only the PNG files in the directory icons. You may use `assets/icons/*.png`. This will select all the icon files in Figure 4-3. If the directory has a file, say `icon.jpeg`, it will be excluded. This is hypothetical. Notice that lines 13 and 14 follow a similar pattern match that includes all `.js` (JavaScript) and `.css` (CSS) files.

c. You can rewrite line 24 as shown in Listing 4-2.

Listing 4-2. ngsw-config.json File

```
// Comment- rewriting "/assets/**",

"files": [
  "/assets/*.png",
  "/assets/icons/*.png"
]
```

This is a specific instruction to include all the files with the extension `.png` under the directories `/assets` and `/assets/icons`. The original statement was generic, which included everything under the `assets` directory.

Why write a specific pattern similar to Listing 4-2, when you can include everything under the `assets` directory? Remember the install phase of a service worker. It installs either everything or nothing if the download of a single file fails. Even though this does not affect the functionality of the application, the service worker installation is postponed until the next time the application reloads. Such situations can occur on low-bandwidth networks. Configuring a generic pattern might include unnecessary files, which when they fail to download can cause

problems with service worker installation. Hence, as much as you can, it is a good practice to be specific. However, if you know everything under the `assets` directory needs to be cached anyway, use the generic rule and simplify the configuration. More often than not, it depends on whether you use generic or specific patterns.

Note While matching patterns, it is possible that two line items match a file. The service worker caches or excludes a file once it finds a match. It does not continue to look for the pattern in the next few items.

Imagine that `/assets/**` is on top in the array. It matches all files under `assets`, and the specific rule never runs. Hence, specify the generic rules toward the bottom of the list; specific rules should be at the beginning in the array.

So far, you have seen the pattern to include files. You may use the exclamation point (`!`) to pattern match excluding files. In an example, say you want to exclude caching all *map* files. A map file contains symbols for JavaScript code, which helps debug a minified version of the file. It is used in debugging and adds no value to the user to cache these files with a service worker. Hence, exclude map files with the pattern `!/**/*.map`.

Notice that you are selecting map files with `*.map` and excluding them with an exclamation point at the beginning.

Note To pattern match a single character, use `?`. It is not often that we can be so specific that we know the number of characters in a file or directory name. Hence, it is rarely used in `ngsw-config.json`.

Browser Support

Consider Figure 4-4, which depicts the browser support for a service worker and its features. Notice that the data is captured on the Mozilla.org website, at `https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorker`. This is a reliable and open source platform for web technologies. Mozilla has been an advocate of the open web and has pioneered safe and free Internet technologies including the Firefox browser.

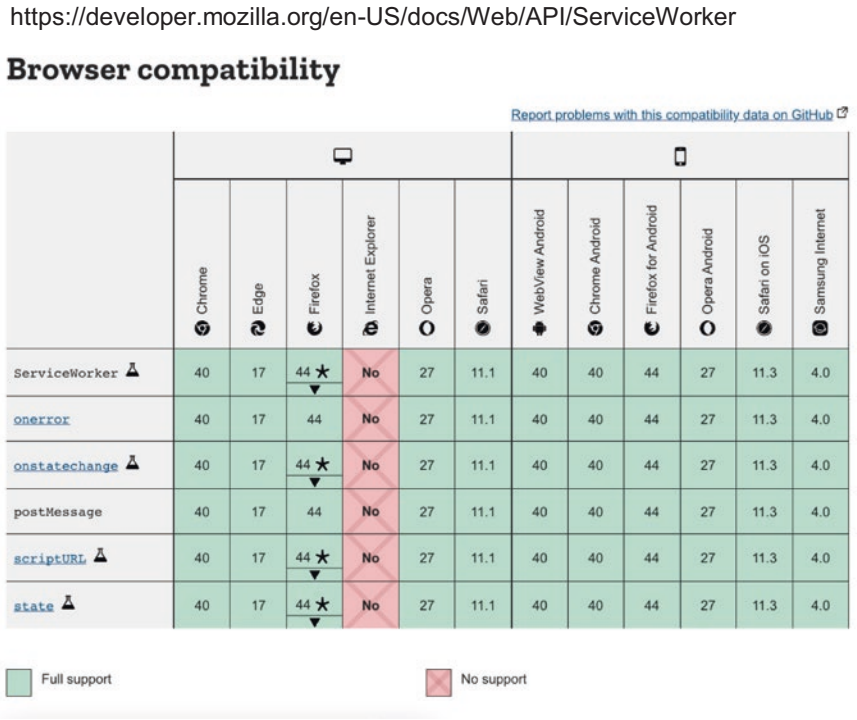

Figure 4-4. *Service workers browser support*

Note CanIUse.com is another quality source of information for browser compatibility. As an alternative to Mozilla, please try https://caniuse.com/serviceworkers to learn more.

Summary

This chapter introduced service workers and the service worker lifecycle. They are a proxy running in the background on the browser. A service worker intercepts all the network requests from the application. Service workers cache static and dynamic resources and programmatically use the cached application scripts, images, data, etc. They enable an application to function even while disconnected from the network and on slow speed networks.

EXERCISE

- Select and use a Google font in Web Arcade. Enable caching the font file with service workers. Do not copy the font to the project. Use the CDN location.
- Imagine a production application's releases are scheduled once a quarter. Regardless of how often the service worker features are updated, the icons, images, and stylesheets may change every quarter. Expire such resources' cache every 12 weeks.
- Explore and view all the service workers installed in your favorite browser.

CHAPTER 5

Cache Data with Service Workers

Service workers are used to cache data responses. So far, you have seen how to create a new Angular application, configure the application to be installable, and cache the application so that it is accessible even when offline. This chapter introduces how to cache a data response from an HTTP service.

This chapter begins by creating a new component to retrieve and show data from an HTTP service. Next, it discusses how to create an interface that acts as a contract between the service and the Angular application. Next, you will learn how to create a Node.js Express mock service that provides data to the Angular application. It runs in a separate process outside the Angular application. The chapter details how to create an Angular service, which uses an out-of-the-box `HttpClient` service to invoke the HTTP service.

Now that you have integrated with an HTTP service and accessed the data, the chapter details how to configure Web Arcade to cache data responses. It elaborates on the configuration and showcases a cached data response with a simulated offline browser.

Remember, Web Arcade is an online system for games. Imagine a screen that lists board games available on the application, as shown in Figure 5-1. Follow the instructions to build this component. It shows the data in an HTML table. On loading the page, the component invokes the service to retrieve the Web Arcade board games.

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4_5

Figure 5-1. *Board games list*

Adding a Component to List Board Games

Begin by creating a component for listing the board games. Remember the "Angular Components" section in Chapter 3. Create a new component by running the following command. It will scaffold the new component.

```
% ng generate component components/board-games
```

Earlier, you used the `dice` component in the `App` component. Update it to use the new component, as shown in Listing 5-1. Notice the `dice` component, called `wade-dice`, has been commented.

Listing 5-1. Use the Board Component

```
<div class="container align-center">
   <!-- <wade-dice></wade-dice> -->
   <wade-board-games></wade-board-games>
</div>
```

Note Angular single-page applications (SPAs) use routing to navigate between two pages with separate components. Listing 5-1 is temporary so that the focus stays on data caching in this chapter. Chapter 8 introduces Angular routing.

Define a Data Structure for Board Games

Next, define a data structure for the board games page. You create a TypeScript interface for defining the data structure. It defines a shape for the board games data objects. TypeScript uses an interface to define a contract, which is useful within the Angular application and with the external, remote service that serves the board games data.

The TypeScript interface enforces the required list of fields for board games. You will notice an error if a needed field is missing because of a problem in the remote service or a bug in the Angular application. An interface acts as a contract between the Angular application and the external HTTP service.

Run the following command to create the interface. It creates a new file called `board-games-entity.ts` in a new directory called `common`. Typically, data structures/entities are used across the Angular application. Hence, name the directory `common`.

```
ng generate interface common/board-games-entity
```

Listing 5-2 defines the specific fields for board games. The remote service is expected to return the same fields. The component uses this shape and structure for the data. Add the code to `board-games-entity.ts`.

Listing 5-2. Interfaces for Board Games

```
export interface BoardGamesEntity {
   title: string;
   description: string;
   age: string;
   players: string;
   origin: string;
   link: string;
   alternateNames: string;
}

/* Multiple games data returned, hence creating an Array */
export interface GamesEntity {
   boardGames: Array<BoardGamesEntity>;
}
```

`BoardGamesEntity` represents a single board game. Considering Web Arcade will have multiple games, `GamesEntity` includes an array of board games. Later, `GamesEntity` can extend to other categories of games in the Web Arcade system.

Mock Data Service

A typical service retrieves and updates data from/to a database or a back-end system, which is out of scope for this book. However, to integrate with a RESTful data service, this section details how to develop mock responses and data objects. The mock service returns board games data in JavaScript Object Notation (JSON) format. It can be readily integrated with the Angular component created in the earlier section "Adding a Component to List Board Games."

You will use Node.js's Express server to develop the mock service. Follow these instructions to create a new service.

Use the Express application generator to easily generate a Node.js Express service. Run the following command to install it:

```
npm install --save-dev express-generator

# (or)

yarn add --dev express-generator
```

Note Notice the `--save-dev` option with the `npm` command and the `--dev` option with the `yarn` command. It installs the package in `dev-dependencies` in `package.json`, qualifying it as a developer tool. It will not be included in the production builds, which helps reduce the footprint. See Listing 5-3, line 15.

Listing 5-3. Package.json dev-dependencies

```
01: {
02:  "name": "web-arcade",
03:  "version": "0.0.0", /* removed code for brevity */
```

```
04:  "dependencies": {
05:    "@angular/animations": "~12.0.1",
06:    /* removed code for brevity */
07:    "zone.js": "~0.11.4"
08:  },
09:  "devDependencies": {
10:    "@angular-devkit/build-angular": "~12.0.1",
11:    "@angular/cli": "~12.0.1",
12:    "@angular/compiler-cli": "~12.0.1",
13:    "@types/jasmine": "~3.6.0",
14:    "@types/node": "^12.11.1",
15:    "express-generator": "^4.16.1",
16:    "jasmine-core": "~3.7.0",
17:    "karma": "~6.3.0",
18:    "karma-chrome-launcher": "~3.1.0",
19:    "karma-coverage": "~2.0.3",
20:    "karma-jasmine": "~4.0.0",
21:    "karma-jasmine-html-reporter": "^1.5.0",
22:    "typescript": "~4.2.3"
23:  }
24: }
```

Next, create a new directory for mock services; name it `mock-services` (an arbitrary name). Change the directory to `mock-services`. Run the following command to create a new Express service. It scaffolds a new Node.js Express application.

```
npx express-generator
```

Note The npx command first checks the local node_modules for the package. If it is not found, the command downloads the package to the local cache and run the command.

The previous command runs, even without the dev-dependency installation in the previous step (npm install --save-dev express-generator). If you do not intend to run this command often, you may skip the dev-dependency installation.

Next, run npm install (or yarn install) in the mock-services directory.

Create and save board games data in a JSON file. The code sample saves it to [application-directory]/mock-services/data/board-games.json. The server-side Node.js service returns these fields and values to the Angular application. The structure matches the Angular interface structure defined in Listing 5-2. See Listing 5-4.

Listing 5-4. Board Games Mock Data

```
{
   "boardGames": [
       {
           "title": "Scrabble",
           "description": "A crossword game commonly played
           with English alphabets and words",
           "age": "5+",
           "players": "2 to 5",
           "origin": "Started by an architect named Alfred
           Mosher Butts in the year 1938",
           "link": "https://simple.wikipedia.org/wiki/
           Scrabble",
```

```
            "alternateNames": "Scrabulous (a version of the game
            on Facebook)"
        },
        {
            "title": "Checkers",
            "description": "Two players start with dark and
            light colored pieces. The pieces move diagonally.",
            "age": "3+",
            "players": "Two players",
            "origin": "12th century France",
            "link": "https://simple.wikipedia.org/wiki/
            Checkers",
            "alternateNames": "Draughts"
        }
/* You may extend additional mock games data*/
   ]
}
```

Next, update the mock service application to return the previous board games data. Create a new file called `board-games.js` under `mock-services/routes`. Add the code in Listing 5-5.

Listing 5-5. New API Endpoint That Returns Mock Board Games Data

```
01: var express = require('express'); // import express
02: var router = express.Router(); // create a route
03: var boardGames = require('../data/board-games.json');
04:
05: /* GET board games listing. */
06: router.get('/', function(req, res, next) {
07:     res.setHeader('Content-Type', 'application/json');
```

```
08:     res.send(boardGames);
09: });
10:
11: module.exports = router;
```

Consider the following explanation:

- Line 3 imports and sets the board games mock data on a variable.
- Lines 6 to 9 create the endpoint that returns the board games data.
- Notice the `get()` function in line 6. The endpoint responds to an HTTP GET call, which is typically used to retrieve data (as opposed to create, update, or delete).
- Line 7 sets the response content type to `application/json` ensuring the client browsers interpret the response format accurately.
- Line 8 responds to the client with the board games data.
- Line 11 exports a router instance that encapsulates the service endpoint.

Next, the endpoint needs to be associated with a route so that the previous code is invoked when the client requests data. Edit `app.js` in the root directory of the service application (`mock-services/app.js`). Add the lines of code in bold (lines 9 and 25) in Listing 5-6 to the file.

Listing 5-6. Integrate the New Board Games Endpoint

```
07: var indexRouter = require('./routes/index');
08: var usersRouter = require('./routes/users');
```

```
09: var boardGames = require('./routes/board-games');
10:
11: var app = express();
12:
13: // view engine setup
14: app.set('views', path.join(__dirname, 'views'));
15: app.set('view engine', 'jade');
16:
17: app.use(logger('dev'));
18: app.use(express.json());
19: app.use(express.urlencoded({ extended: false }));
20: app.use(cookieParser());
21: app.use(express.static(path.join(__dirname, 'public')));
22:
23: app.use('/', indexRouter);
24: app.use('/users', usersRouter);
25: app.use('/api/board-games', boardGames);
```

Consider the following explanation:

- Line 9 imports the board games route instance exported in the earlier Listing 5-5.
- Line 25 adds the route `/api/board-games` to the application. The new service is invoked when the client invokes this endpoint.

Run the mock service with the command `npm start`. By default, it runs the Node.js Express service application on port 3000. Access the new endpoint by accessing `http://localhost:3000/api/board-games`. See Figure 5-2.

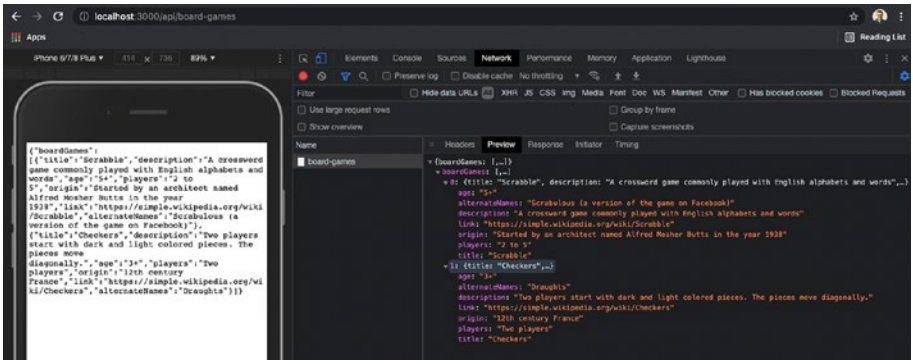

Figure 5-2. *Board games endpoint accessed on a browser*

Note Notice that you are running the service application on a separate port, which is 3000. Remember, in the earlier examples, the Angular application runs on ports 8080 (with Http-Server) and 4200 (with the `ng serve` command that uses Webpack internally). The Angular application running on one of these ports is expected to connect to the service instance running on port 3000.

Call the Service in the Angular Application

This section details how to update the Angular application to consume data from the Node.js service. In a typical application, Node.js services are server-side, remote services that access data from a database or another service.

Configure the Service in an Angular Application

Angular provides an easy way to configure various values including remote service URLs. In the Angular project, notice the directory `src/environment`. By default, you will see the following:

- `environment.ts`: This is for the debug build configuration used by the developer on localhost. Typically, the `ng serve` command uses it.
- `environment.prod.ts`: This is for production deployments. Running `ng build` (or `yarn build` or `npm run build`) uses this configuration file.

Edit the file `src/environments/environment.ts` and add the code in Listing 5-7. It has a relative path to the service endpoint.

Listing 5-7. Integrate the New Board Games Endpoint

```
1: export const environment = {
2:     boardGameServiceUrl: `/api/board-games`,
3:     production: false,
4:   };
5:
```

Consider the following explanation:

- Line 2 adds a relative path to the service endpoint. You will import and use the configuration field `boardGameServiceUrl` while making a call to the service.
- Line 3 set `production` to `false`. Remember, the file `environment.ts` is used with the `ng serve` command, which runs a debug build with the help of Webpack. It is set to `true` in the alternate environment file `environment.prod.ts`.

Create an Angular Service

Angular services are reusable units of code. Angular provides ways to create a service and instantiate and inject services into components and other services. The Angular services help separate concerns. Angular components primarily focus on the presentation logic. On the other hand, you may use services for other reusable functions that do not include presentation. Consider the following examples:

- A service can be used for sharing data among components. Imagine a screen with a list of users. Say the list is shown by a `UserList` component. Users can select a user. The application navigates to another screen, which loads another component, say `UserDetails`. The user details component shows additional information about the user in your system. The user details component needs data about the selected user so that it can retrieve and show the additional information.

  You may use a service to share the selected user information. The first component updates the selected user details to a common service. The second component retrieves the data from the same service.

Note A service is an easy and a simple way to share data among components. However, for a large application, it is advisable to adapt the Redux pattern. It helps maintain application state, ensures unidirectional data flows, provides selectors for easy access to the state in the Redux store, and has many more features. For Angular, NgRx is a popular library that implements the Redux pattern and its concepts.

How are the components sharing the same instance of a service? See the next section for details of how an Angular service is provided and how the service instances are managed in an Angular application.

- A service can be used to aggregate and transform JSON data. The Angular application might obtain data from various data sources. Create a service with a reusable function to aggregate and return the data. This enables a component to readily use the JSON objects for the presentation.
- A service is used to retrieve data from a remote HTTP service. In this chapter, you have already built a service to share the board games data with an Angular application. The Node.js Express server running in a separate process (ideally on a remote server) shares this data over an HTTP GET call.

Create a new service by running the following Angular CLI command. You will use this service to invoke the `api/board-games` service built in the previous section.

```
ng generate service common/games
```

The CLI command creates a new games service. It creates the following files in the directory `common`:

- `common/games.services.ts`: A TypeScript file for adding Angular service code that makes HTTP calls for games data
- `common/games.services.spec.ts`: A unit test file for the functions in `games.service.ts`

Consider Listing 5-8 for the games service. Add a new function called `getBoardNames()` to invoke the HTTP service.

Listing 5-8. Angular Service Skeleton Code

```
01: @Injectable({
02:     providedIn: 'root'
03: })
04: export class GamesService {
05:
06:     constructor() { }
07:
08:     getBoardGames(){
09:     }
10: }
```

Provide a Service

Notice the code statement in lines 1 to 3. Those lines contain the `Injectable` decorator, with `provideIn` at the root level. Angular shares a single instance for the entire application. The following are the alternatives:

- *Provide at the module level*: The service instance is available and shared within the module. Later sections give more details about Angular modules.
- *Provide at the component level*: The service instance is created and available for the component and all its child components.

Once a service is provided, it needs to be injected. A service can be injected into a component or another service. In the current example, the board games component needs data so that the games are listed for

users to view. Notice in the earlier Listing 5-8 that the code creates a new function called `getBoardGames()` intended to retrieve the list from the remote HTTP service.

Inject `GamesService` into `BoardGamesComponent`, as shown in Listing 5-9, line 5. The constructor creates a new field called `gameService` of type `GamesService`. This statement injects the service into the component.

Listing 5-9. Inject Games Service into a Component

```
01: export class BoardGamesComponent implements OnInit {
02:
03:     games = new Observable<GamesEntity>();
04:
05:     constructor(private gameService: GamesService) { }
06:
07:     ngOnInit(): void {
08:       this.games = this.gameService.getBoardGames();
09:     }
10:
11:   }
```

Note The `ngOnInit()` function on line 7 is an Angular lifecycle hook. It is invoked after the framework completes initializing the component and its properties. This function is a good place in a component for additional initializations, including service calls.

Line 8 in Listing 5-9 calls the service function that retrieves board games data. This data is required as part of component initialization as the primary functionality of the component is to show a list of games.

HttpClient Service

Next, invoke the remote HTTP service. Angular provides the `HttpClient` service as part of the package `@angular/common/http`. It provides an API to invoke various HTTP methods including GET, POST, PUT, and DELETE.

As a prerequisite, import `HttpClientModule` from `@angular/common/http`. Add it (`HttpClientModule`) to the imports list on the Angular module, as shown in Listing 5-10, lines 7 and 13.

Listing 5-10. Import HttpClientModule

```
01: import {HttpClientModule} from '@angular/common/http';
02:
03: @NgModule({
04:   declarations: [
05:  // pre-existing declaratoins
06:   ],
07:   imports: [
08:    // pre-existing imports
09:     BrowserModule,
10:     HttpClientModule,
11:     AppRoutingModule,
12:
13:   ],
14:   providers: [],
15:   bootstrap: [AppComponent]
16: })
17: export class AppModule { }
18:
```

Remember from Listing 5-5 (line 6) that the service returns data to the Angular application with a GET call. Hence, we will use the `get()` function on the `HttpClient` instance to invoke the service. Remember, we already created the function `getBoardGames()` as part of `GamesService` (see Listing 5-8, line 8).

Next, inject the `HttpClient` service into `GamesService` and use the `get()` API to make an HTTP call. See Listing 5-11.

Listing 5-11. GamesService Injects and Uses HttpClient

```
01: import { Injectable } from '@angular/core';
02: import { HttpClient } from '@angular/common/http';
03: import { environment } from 'src/environments/environment';
04: import { GamesEntity } from './board-games-entity';
05: import { Observable } from 'rxjs';
06:
07:
08: @Injectable({
09:   providedIn: 'root'
10: })
11: export class GamesService {
12:
13:   constructor(private client: HttpClient) { }
14:
15:   getBoardGames(): Observable<GamesEntity>{
16:     return this
17:       .client
18:       .get<GamesEntity>(environment.boardGameServiceUrl);
19:   }
20: }
21:
```

Consider the following explanation:

- Line 13 injects `HttpClient` into `GamesService`. Notice that the name of the field (an instance of `HttpClient`) is `client`. It is a private field and hence accessible only within the service class.
- The statement in lines 16 to 18 invokes the `client.get()` API. As a client is a field of the class, it is accessed using the `this` keyword.
- The `get()` function accepts one parameter, the URL for the service. Notice the import statement for the environment object on line 3. It imports the object exported from the environment configuration file. See Listing 5-7. It is one of the environment configuration files. Use the `boardGameServiceUrl` field from the configuration (Listing 5-11, line 18). You may have more than one URL configured in an environment file.
- Notice that the `get()` function is expected to retrieve `GamesEntity`. It was created in Listing 5-2.
- The `getBoardGames()` function returns an `Observable<GamesEntity>`. Observable is useful with asynchronous function calls. A remote service might take some time, such as a few milliseconds or sometimes a few seconds, to return the data. Hence, the service function returns an observable. The subscriber provides a function callback. The observable executes the function callback once data is available.

- Notice that line 16 returns the output of the `get()` function call. It returns an `Observable` of the type specified. You specified the type `GamesEntity` on line 18. Hence, it returns an `Observable` of type `GamesEntity`. It matches with the return type of `getBoardGames()` on line 15.

Now, the service function is ready. Review Listing 5-9 again, which is a component TypeScript class. It calls the service function and sets the return value of type `Observable<GamesEntity>` to a class field. The class field uses the returned object in the HTML template. The template file renders the board games list on a page. See Listing 5-12.

Listing 5-12. Board Games Component Template Shows List of Games

```
01: <div>
02:     <table>
03:         <tr>
04:             <th> Title </th>
05:             <th> History </th>
06:         </tr>
07:         <ng-container *ngFor="let game of (games | async)?.
            boardGames">
08:             <tr>
09:                 <td>
10:                     <strong>
11:                         {{game.title}}
12:                     </strong>
13:                     <span>{{game.alternateNames}}</span>
14:                 </td>
15:                 <td>{{game.origin}}</td>
16:             </tr>
```

```
17:             <tr >
18:                 <td class="last-cell" colspan="2">{{game.
                    description}}</td>
19:             </tr>
20:         </ng-container>
21:
22:     </table>
23: </div>
```

Consider the following explanation:

- The template renders the list as an HTML table.
- Notice, in line 7, that the *ngFor directive iterates through boardGames. See Listing 5-2. Notice that boardGames is an array on the interface GamesEntity.
- The template shows fields on each game in the entity. See lines 11, 13, 15, and 18. They show the fields title, alternateNames, origin, and description.
- Remember, the class field games is set with the values returned from the service. This field is used in the template. See line 7.
- Notice the pipe with async (| async) on line 7. It is applied on Observable. Remember, the service returns an Observable. As mentioned earlier, an Observable is useful with asynchronous function calls. A remote service might take time, a few milliseconds or sometimes a few seconds, to return the data. The template uses the field boardGames on games Observable, when the data is available, in other words, when it is obtained from the service.

Cache the Board Games Data

So far, we have created an HTTP service to provide board games data, created an Angular service to use the HTTP service to obtain the data, and added a new component to show the list. Now, configure the service worker to cache the board games data (and even other HTTP service responses).

Remember, in the previous chapter, we listed various configurations for Angular service workers. As you have seen, Angular uses a file called `ngsw-config.json` for service worker configurations. In this section, you will add a `dataGroups` section to cache the HTTP service data. See Listing 5-13 for the new configuration to cache the board games data.

Listing 5-13. Data Groups Configuration for a Service Worker in an Angular Application

```
"dataGroups": [{
    "name": "data",
    "urls": [
      "api/board-games"
    ],
    "cacheConfig": {
      "maxAge": "36h",
      "timeout": "10s",
      "maxSize": 100,
      "strategy":"performance"
    }
  }]
```

Consider the following explanation:

- Line 4 configures the service URLs to cache data. It is an array, and we can configure multiple URLs here.

- The URLs support matching patterns. For example, you may use `api/*` to configure all the URLs.
- As part of the cache configuration (`cacheConfig`), see line 10. Set `strategy` to `performance`. This instructs the service worker to use cached responses first for better performance. Alternatively, you may use `freshness`, which goes to the network first and uses the cache only when the application is offline.
- Notice that `maxAge` is set to 36 hours, after which the service worker clears the cached responses (of board games). Caching data for too long could cause the application to use obsolete fields and records. The service worker configuration provides a mechanism to automatically clear the data at periodic intervals, ensuring the application does not use stale data.
- The timeout is set to 10 seconds. This is dependent on `strategy`. Assuming `strategy` is set to `freshness`, after 10 seconds, the service worker uses cached responses.
- `maxSize` is set to 100 records. It is a good practice to limit the size by design. Browsers (like any other platform) manage and allocate memory for each application. If the application exceeds the upper limit, the entire dataset and the cache could be evicted.

Listing 5-13 has a single data groups configuration object. As we further develop the application, the additional services might have slightly different cache requirements. For example, the list of gamers might need to be the latest ones. If your friend joins the arcade, you prefer to see her listed instead of showing the old list. Hence, you might change the

strategy to `freshness`. Add this URL configuration as another object in the `dataGroups` array. On the other hand, for a service that fits the current configuration, add the URL to the `urls` field on line 4.

Run the Angular build and start Http-Server to see the changes. See the following command:

```
yarn build && http-server dist/web-arcade --proxy http://
localhost:3000
```

See Figure 5-3 for cached service response with a service worker.

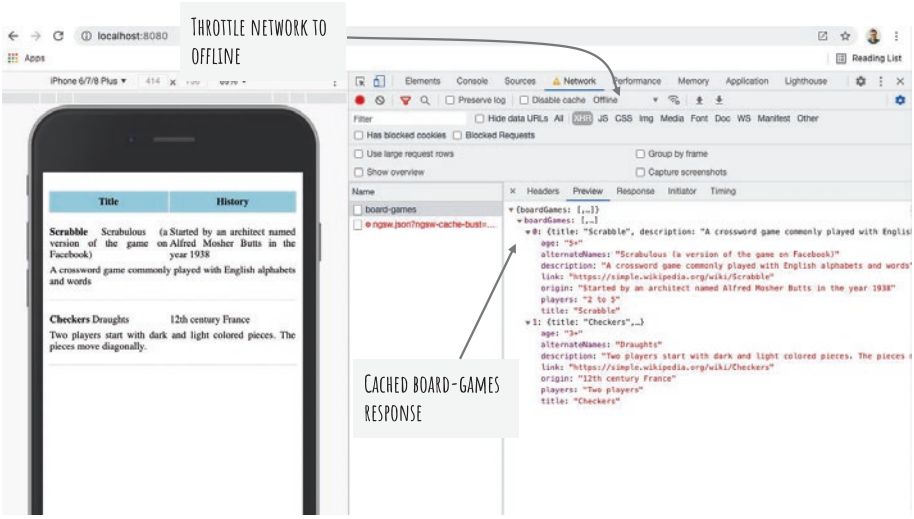

Figure 5-3. *Cached service responses with the service worker*

Angular Modules

Traditionally, Angular had its own modularity system. The new framework (Angular 2 and greater) uses NgModules to bring modularity to applications. An Angular module encapsulates directives including components, services, pipes, etc. Create Angular modules to logically group features. See Figure 5-4.

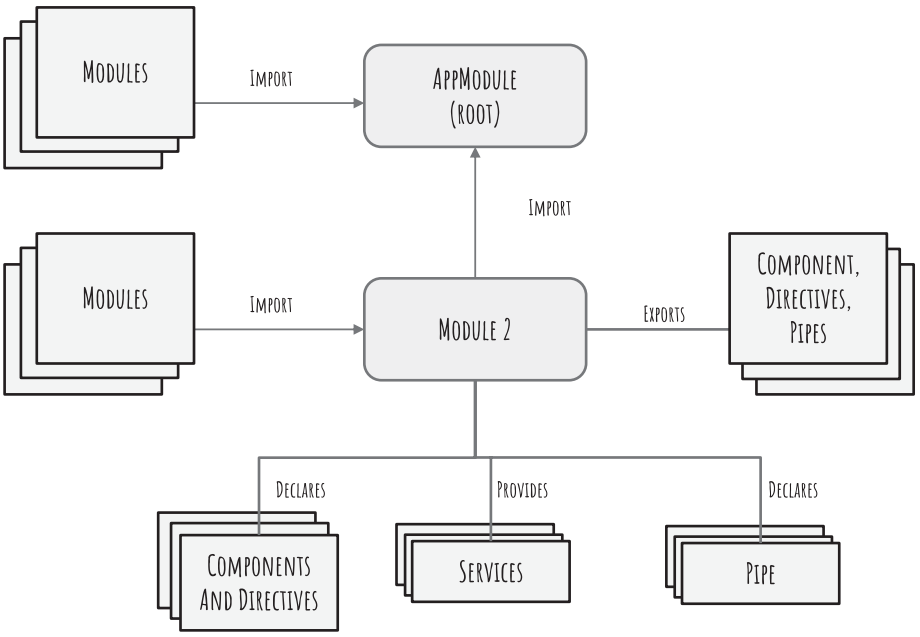

Figure 5-4. *Angular modules*

All Angular applications use at least one root module. Typically, the module is named `AppModule` and defined in `src/app/app.module.ts`. A module may export one or more functionalities. The other modules in the application can import the exported components and services.

Note Angular modules are separate from JavaScript (ES6) modules. They complement each other. An Angular application uses both JavaScript modules and Angular modules.

Summary

This chapter provided instructions for creating a new component for listing board games. With this code sample, it demonstrated how service workers cache data responses from an HTTP service. It provided instructions to create a board games component with Angular CLI. You also updated the application to use this new component instead of `dice`.

It also defined data contracts between the Angular application and the external HTTP service, detailed how to create a Node.js Express service for providing data to the Angular application, and introduced Angular services.

EXERCISE

- Create a new route in the Node.js Express application for exposing a list of jigsaw puzzles.
- Create an Angular service to use the new jigsaw puzzles service endpoint and retrieve data.
- Ensure the latest jigsaw puzzles data is available to the user. Cache only when the user is offline or lost connectivity.
- For the new service, configure to use data from the cache if the service does not respond after one minute.

CHAPTER 6

Upgrading Applications

So far you have created an Angular application, registered service workers, and cached application resources. This chapter details how to discover an update to the application, communicate with users, and handle events to gracefully upgrade to the next version.

The chapter extensively uses Angular's `SwUpdate` service, which provides ready-made features to identify and upgrade the application. It begins with instructions to include (import and inject) the `SwUpdate` service. Next, it details how to identify an available upgrade and activate the upgrade. It also details how to check at regular intervals for an upgrade. Toward the end, the chapter details how to handle an edge case, namely, an error scenario when browsers clean up unused scripts.

Considering the Web Arcade application is installable, you need a mechanism to look for updates, notify the user about new versions of the application, and perform an upgrade. A service worker manages installing and caching the Angular application. This chapter details working with `SwUpdate`, an out-of-the-box service provided by Angular to ease service worker communication. It gives access to events when a new version of the application is available, downloaded, and activated. You may use the functions in this service to do periodic checks for updates.

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4_6

Getting Started with SwUpdate

This section shows you how to get started with the `SwUpdate` service by importing and injecting the service. The `SwUpdate` service is part of the Angular module `ServiceWorkerModule`, which was referenced in the import list of `AppModule` already. Verify the code in the sample application in the `app.module.ts` file. It was included when you ran the Angular CLI `ng add @angular/pwa` command in Chapter 2. Consider Listing 6-1, lines 15 and 22.

Listing 6-1. ServiceWorkerModule Imported in AppModule

```
01: import { NgModule } from '@angular/core';
02: import { BrowserModule } from '@angular/platform-browser';
03: import { environment } from '../environments/environment';
04: import { ServiceWorkerModule } from '@angular/
    service-worker';
05: import { BrowserAnimationsModule } from '@angular/platform-
    browser/animations';
06:
07: import { AppComponent } from './app.component';
08:
09: @NgModule({
10:   declarations: [
11:     AppComponent,
12:   ],
13:   imports: [
14:     BrowserModule,
15:     ServiceWorkerModule.register('ngsw-worker.js', {
16:       enabled: environment.production,
17:       // Register the ServiceWorker as soon as the app
          is stable
```

```
18:       // or after 30 seconds (whichever comes first).
19:       registrationStrategy: 'registerWhenStable:30000'
20:     }),
21:     BrowserAnimationsModule
22:   ],
23:   providers: [],
24:   bootstrap: [AppComponent]
25: })
26: export class AppModule { }
27:
```

Ensure `ServiceWorkerModule` is imported as shown (in bold). This enables the `SwUpdate` service to be readily usable. Create a new Angular service to encapsulate the code for identifying a new version of the service worker, communicating with the user, and managing the details. This helps with code reusability and separation of concerns. To install the service, run the following command:

```
ng g s common/sw-communication
```

Note In the snippet, `g` is short for "generate," and `s` is for "service."

You can rewrite the previous command as `ng generate service common/sw-communication`.

The Angular CLI command creates a new service called `SwCommunicationService` in the directory `common`. By default it is provided at the root level. Import and inject the `SwUpdate` service into `SwCommunicationService`, as shown in Listing 6-2.

Listing 6-2. Scaffolded SwCommunicationService

```
01: import { Injectable } from '@angular/core';
02: import { SwUpdate } from '@angular/service-worker';
03:
04: @Injectable({
05:   providedIn: 'root'
06: })
07: export class SwCommunicationService {
08:   constructor(private updateSvc: SwUpdate)
09:   }
10: }
```

Lines 2 and 8 import and inject the `SwUpdate` Angular service. Line 5 provides the service at the root level. Although the service is provided at the root level, it is not yet used in the application. Unlike other services created in Web Arcade, it needs to run in the background, when you launch the application or at regular intervals. Hence, import and inject `swCommunicationService` in the root component, the `AppComponent`, as shown in Listing 6-3, lines 2 and 9.

Listing 6-3. Import and Inject SwCommunicationService in AppComponent

```
01: import { Component } from '@angular/core';
02: import { SwCommunicationService } from 'src/app/common/sw-
    communication.service';
03: @Component({
04:   selector: 'app-root',
05:   templateUrl: './app.component.html',
06:   styleUrls: ['./app.component.sass']
07: })
08: export class AppComponent {
```

```
09:   constructor(private commSvc: SwCommunicationService){
10:   }
11: }
```

Identifying an Update to the Application

Next, update `SwCommunicationService` to identify if an updated version of the application is available. The `SwUpdate` service provides an observable named `available`. Subscribe to this observable. It is invoked when the service worker identifies an updated version of the application.

Note At this point, you have the information that an upgrade is available on the server. You have not downloaded and activated it yet.

Consider Listing 6-4.

Listing 6-4. Identify a New Version of the Application

```
1: export class SwCommunicationService {
2:   constructor(private updateSvc: SwUpdate
3:    ){
4:     this.updateSvc.available.subscribe( i => {
5:       console.log('A new version of the application
         available', i.current, i.available);
6:     });
7:   }
8: }
```

Consider the following explanation:

- See line 4 for how to use the object `available` on `updateSvc` (an object of `SwUpdate`). It is an observable, which sends values when a new version of the application is available.
- Line 5 prints `current` and `available` objects. Consider the result in Figure 6-1. Notice the version number in the message.

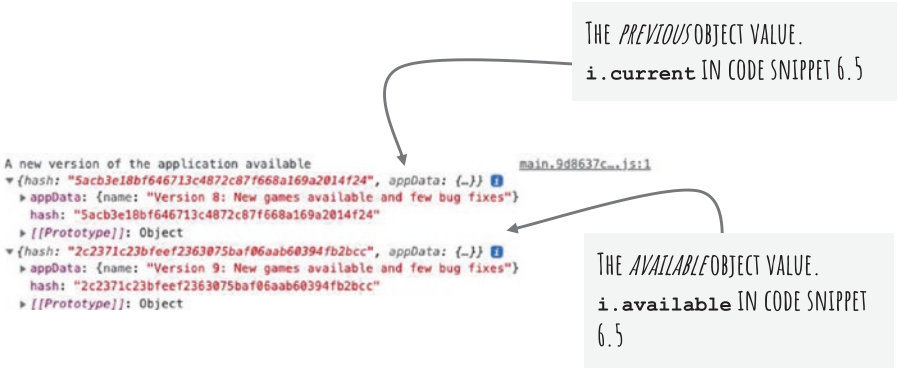

Figure 6-1. *Result on the available observable*

Note Listing 6-4 does not initiate a check for a new version. The subscribe callback (line 5) runs when a new version is identified. See the section "Checking for a New Version" to initiate a check for a new version at regular intervals.

Also, the new version is *not* downloaded and activated yet.

- Remember the `appData` field in `ngsw-config.json` for the application. (Refer to Chapter 4.) The objects `current` and `available` include data from `appData`. In the sample application, we added a single field name that describes changes to the application. The result in Figure 6-1 prints the `current` and `available` objects. They are the fields from `ngsw-config.json` in the current and new versions of the application. See Listing 6-5 for `ngsw-config.json`.

Listing 6-5. ngsw-config.json with appData

```
01: {
02:   "appData": {"name": "New games available and few bug
      fixes"},
03:   "$schema": "./node_modules/@angular/service-worker/
      config/schema.json",
04:   "index": "/index.html",
05:   "assetGroups": [
06:     {
07:       // Removed code for brevity. See code sample for the
          complete file.
42:     }]
43: }
```

As mentioned earlier, the `available` observer verifies if a new version of the service worker is ready and available. It does not mean it is in use yet. You may prompt the user to update the application. This gives the user a chance to complete the current workflow. For example, in Web Arcade, you do not want to reload and upgrade to a new version while the user is playing a game.

Identifying When an Update Is Activated

So far, you have identified if an upgrade is available. This step allows you to identify an activated upgrade. The `activated` observable is triggered once a service worker starts serving content from the new version of the application.

Consider Listing 6-6 in `SwCommunicationService` that uses the `activated` observable.

Listing 6-6. Activated Observable on SwUpdate

```
01: export class SwCommunicationService {
02:
03:     constructor(private updateSvc: SwUpdate) {
04:       this.updateSvc
05:         .available
06:         .subscribe( i => {
07:             console.log('A new version of the application
                available', i.current, i.available);
08:       });

09:       this.updateSvc
10:         .activated
11:         .subscribe( i =>
12:             console.log('A new version of the application
                activated', i.current, i.previous));
13:     }
14:
15: }
```

See lines 9 to 13. Notice that you subscribe to the `activated` observable. It is triggered on activating a new version of the application. See the `console.log` on line 12. This prints `current` and `previous`.

It is similar to the `current` and `available` objects on the `available` observable (see Listing 6-4), which is before activation. As the activated observable is triggered after activation, the available version is now the current version. See the result in Figure 6-2.

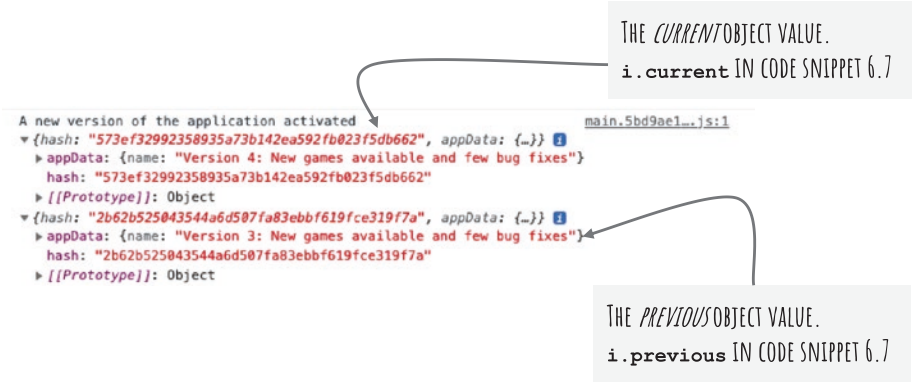

Figure 6-2. *Result on the activated observable*

Note Similar to the `available` observable, the `current` and `previous` objects include `appData` from `ngsw-config.json` for the respective versions of the application.

Activating with the SwUpdate Service

When the user opens the application in a new window, service workers check if a new version is available. The service worker might still load the application from the cache. This is because the new version may not have been downloaded and activated yet. It triggers the `available` event. The `subscribe` callback on the `available` observable will be invoked (similar to Listing 6-4). Typically, the next time a user attempts to open

the application in a new window, a newer version of the service worker and the application are served. The precise behavior depends on the configuration and a few other factors.

However, you may choose to activate the newer version as soon as you know a newer version is available. The `SwUpdate` service provides the `activateUpdate()` API to activate new versions of the application. The function returns a `promise<void>`. The success callback of the promise is invoked after activating the update. Consider Listing 6-7, which activates the update.

Note This section does not prompt the user to choose to update the new version yet. As you progress to the next section, you will add the code to alert the user about the availability of a new version of the application.

Listing 6-7. Activate Update with the SwUpdate Service

```
01: export class SwCommunicationService {
02:
03:     constructor(private updateSvc: SwUpdate) {
04:       this.updateSvc
05:         .available
06:         .subscribe( i => {
07:             console.log('A new version of the application
                available', i.current, i.available);
08:             this.updateSvc
09:                 .activateUpdate()
10:                 .then( () => {
11:                     console.log("activate update is
                        successful");
12:                     window.location.reload();
```

```
13:                 });
14:       });
15:
16:       this.updateSvc
17:         .activated
18:         .subscribe( i => console.log('A new version of the
            application activated', i.current, i.previous));
19:     }
20:   }
```

See lines 8 and 13. The `activateUpdate()` function is called in line 9. It returns a promise. The `then()` function is called on the returned promise. You provide a callback that is invoked after the promise is resolved. See line 11, which prints a message that the "activate update is successful."

Notice that `activateUpdate()` is called when an update is available. The `activateUpdate()` method is called within the `available` observable subscription. See lines 6 and 14. This is a subscription callback for the `available` observable.

Line 12 reloads the application (the browser window) after the update has been activated. It is a good practice to reload the browser on a successful update. In a few instances, routing with lazy loading may break if the window does not reload after the service worker and the cache are refreshed.

Checking for a New Version

The `SwUpdates` service can check for a new version of the application (service worker configuration) on the server. As mentioned earlier, so far you have received events if a new version was available or activated. You activated the new version if it was available. However, we are dependent on the browser and service worker built-in mechanisms to look for updates.

The function `checkForUpdates()` can be called on the instance of `SwUpdate` to look for updates on the server. It will trigger the `available` observable if a new version is available. The function is especially useful if you anticipate users keeping the application open for a long duration, sometimes days. It is also possible that you anticipate frequent updates and deployments to the application. You may set an interval and regularly check for updates.

Considering `checkForUpdates()` is used with an interval, it is important you ensure the Angular application is fully bootstrapped and stable before checking for an update. Using this function too often may not allow the Angular application to be stable. Hence, it is a good practice to check if the application is stable and use `checkForUpdates`, as shown in Listing 6-8. It is added to the `SwCommunicationService`.

Listing 6-8. Invoking CheckForUpdates at Regular Intervals

```
01: import { SwUpdate } from '@angular/service-worker';
02: import { ApplicationRef, Injectable } from '@angular/core';
03: import { first} from 'rxjs/operators';
04: import { interval, concat } from 'rxjs';
05:
06: @Injectable({
07:   providedIn: 'root'
08: })
09: export class SwCommunicationService {
10:
11:   constructor(private updateSvc: SwUpdate,
12:     private appRef: ApplicationRef) {
13:
14:     let isApplicationStable$ = this.appRef.isStable.
        pipe(first(isStable => isStable === true));
```

```
15:     let isReadyForVersionUpgrade$ = concat(
        isApplicationStable$, interval (12 * 60 * 60 * 1000));
        // //twelve hours in milliseconds
16:     isReadyForVersionUpgrade$.subscribe( () => {
17:       console.log("checking for version upgrade...")
18:       this.updateSvc.checkForUpdate();
19:     });
20:   }
21: }
22:
```

Consider the following explanation:

- You inject `ApplicationRef` to verify if the Angular app is stable. Lines 2 and 12 import and inject the class, respectively.
- Line 13 verifies if the application is stable. The field `isStable` is of type `Observable<boolean>`. When subscribed, it returns true when the application is stable. Line 14 assigns the observable to the local variable `isApplicationStable$`.
- In line 15, the `interval()` function returns an `Observable<number>`. The function accepts a time duration in milliseconds as a parameter. The subscribe callback is invoked after the specified time interval. Notice that the code snippet specifies 12 hours in milliseconds.
- Line 15 concats `isApplicationStable$` with the observable returned by `interval()`. The resultant observable is set to `isReadyForVersionUpgrade$`. Subscribe to this observable. The `success` callback is invoked when the application is stable and the specified interval (12 hours) has passed.

Note The `concat` function has now been deprecated. If you are on RxJS version 8, use the `concatWith()` function to concatenate observables.

- In line 18, in the `subscribe` callback for the observable `isReadyForVersionUpgrade$`, you check for updates using the `SwUpdate` instance.
- As a quick recap, you check for upgrades every 12 hours when the application is stable. The `checkForUpdate()` function may trigger the `available` subscriber, which calls `activateUpdate()` that triggers the `activate` subscriber after successfully activating the new version of the application.

Notifying the User About the New Version

Notice that the current code reloads the application when a new version of the service worker is identified. It does not yet alert the user and allow her to choose when to reload. To provide this option, integrate the application with a Snackbar component. This is a ready-made component available in the Angular Material library. Angular Material provides Material Design implementations for Angular applications.

Why did we choose the Snackbar component? A typical alert blocks the user workflow. Users cannot continue to use the application until an action is taken on the alert. Such a paradigm works well when a workflow cannot continue without the user's decision. For example, it could be an error scenario, which is important for a user to acknowledge and then make a correction.

On the other hand, when a new version of the service worker (and the application) is available, you do not want to disrupt the current user session. Users may continue to use the current version until the task at hand is complete. For example, if the user is playing a game on Web Arcade, continue until the game is complete. When the user deems it appropriate to reload the window, she may choose to respond to the alert.

A Snackbar component fits well with our scenario. It shows the alert in a corner of the application without interfering with the rest of the page. By default, the alert is shown at the bottom of the page, toward the center. See Figure 6-3. As mentioned earlier, you should allow users to click the button on the Snackbar component to install the new version of the application.

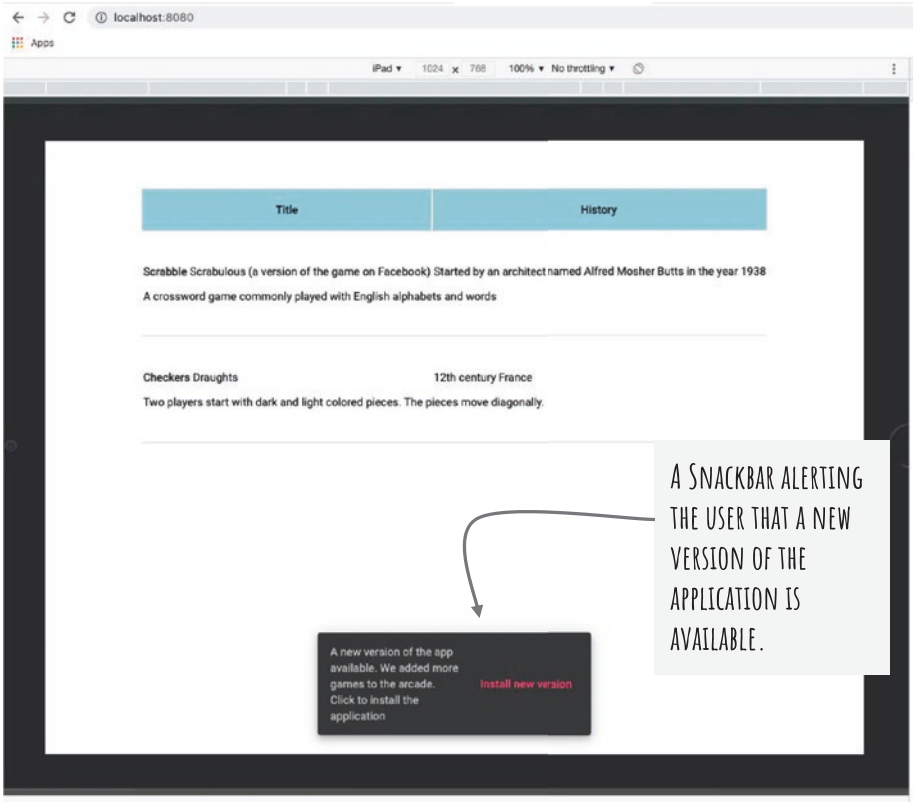

Figure 6-3. *A Snackbar component alerting the user that a new version of the application is available*

To install the Snackbar component, add Angular Material to the Web Arcade application. Run the following command:

```
ng add @angular/material
```

Angular Material is a UI component with other components and stylesheets conforming to Google's Material Design. To begin with, the CLI prompts you to select a theme. See Figure 6-4.

Figure 6-4. *Selecting an Angular Material theme*

Next, the CLI does the following:

1. Prompts you to choose Angular Material typography. This is a decision about using Angular Material fonts, default font sizes, etc.
2. Prompts you to include browser animations interacting with Angular Material components. Animations provide visual feedback for user actions such as animating the click a button, transitioning a tab of content, etc.

See Listing 6-9 for the result.

Listing 6-9. Result Installing Angular Material

```
✔ Package successfully installed.
? Choose a prebuilt theme name, or "custom" for a custom
theme: Indigo/Pink        [ Preview: https://material.angular.
io?theme=indigo-pink ]
? Set up global Angular Material typography styles? Yes
? Set up browser animations for Angular Material? Yes
UPDATE package.json (1403 bytes)
✔ Packages installed successfully.
UPDATE src/app/app.module.ts (1171 bytes)
UPDATE angular.json (3636 bytes)
UPDATE src/index.html (759 bytes)
UPDATE node_modules/@angular/material/prebuilt-themes/indigo-
pink.css (77575 bytes)
```

Adding Angular Material to the Web Arcade allows you to use various Angular Material components. You will import and use only the required components. The entire component library does not add to the bundle size.

To use the Snackbar component, import `MatSnackBarModule` to the `AppModule` in Web Arcade. Consider Listing 6-10.

Listing 6-10. Add the Snackbar Module to the Web Arcade Application

```
01: import { MatSnackBarModule } from '@angular/material/
    snack-bar';
02:
03: @NgModule({
04:   declarations: [
05:     AppComponent,
06:     // You may have more components
07:   ],
08:   imports: [
09:     BrowserModule,
10:     AppRoutingModule,
11:     MatSnackBarModule,
12:     // You may have additional modules imported
13:   ],
14:   providers: [],
15:   bootstrap: [AppComponent]
16: })
17: export class AppModule { }
```

See the first line, which imports `MatSnackBarModule` from the Angular Material module for the Snackbar component. Also see line 11 that imports the module to the `AppModule` on Web Arcade.

Next, import and inject the Snackbar component to the `SwCommunication` service. Remember, you show an alert to the user when a new version of the application is available. It is identified in the constructor of the `SwCommunication` service. See Listing 6-11.

Listing 6-11. Alert with a Snackbar that a new version of the application is available

```
01: import { Injectable } from '@angular/core';
02: import { SwUpdate } from '@angular/service-worker';
03: import { MatSnackBar } from '@angular/material/snack-bar';
04:
05: @Injectable({
06:   providedIn: 'root'
07: })
08: export class SwCommunicationService {
10:   constructor(private updateSvc: SwUpdate,
11:     private snackbar: MatSnackBar) {
13:     this.updateSvc.available.subscribe( i => {
14:       let message = i?.available?.appData as { "name":
          string };
15:       console.log('A new version of the application
          available', i.current, i.available);
16:       let snackRef = this.snackbar
17:         .open(`A new version of the app available.
          ${message.name}. Click to install the application`,
          "Install new version");
18:     });
19:   }
20: }
```

Consider the following explanation:

- Line 3 imports the Snackbar component from Angular Material's `snackbar` module. It is injected into the service on line 11.
- Notice the `success` callback between lines 14 and 18. It is on the `available` observable. As mentioned earlier, when an update is ready, this observer function is triggered.
- See line 16. The `open()` function shows a Snackbar component. You provide two parameters, the message to show on the alert (Snackbar component) and the title for the action (or the button) on the Snackbar component. Revisit Figure 6-3 to match the code to the result.
- Notice the open function returns a Snackbar reference object as well. You are using this object to perform an action when the user clicks the button on the Snackbar component.
- Notice `message.name` is interpolated on line 17. The `message` object is obtained on the prior line 14. Notice it is the `appData` object on `ngsw-config.json`. This is one way to provide a friendly message for each version upgrade of the application and show the information to the user while she chooses to reload and install the new version. See Figure 6-3. The message from `appData` says, "we added more games to the arcade."

Next, use the Snackbar component reference returned by the `open()` function, as shown in line 16. The reference object is named `snackRef`. You use the function `onAction()` on `snackRef`. This returns another

observable. As the name suggests, the observer callback function is triggered when the user performs an action on the Snackbar component. Notice, in the previous code sample, that you have a single action, the button "Install new version" on the Snackbar component. Hence, when this observer is invoked, you know the user clicked the button and can perform the install. See Listing 6-12. With the modified code, you perform the install only after the user chooses to install by clicking the button on the Snackbar component.

Listing 6-12. Add Snackbar Module to Web Arcade

```
01:
02: // include this snippet after snackRef created in the
    SwCommunicationService
03: snackRef
04: .onAction()
05: .subscribe (() => {
06:   console.log("Snackbar action performed");
07:   this.updateSvc.activateUpdate().then( () => {
08:     console.log("activate update invoked");
09:     window.location.reload();
10:   });
11: });
```

Consider the following explanation:

- Line 4 invokes the `onAction()` function on the `snackRef` object. You chain the `subscribe()` function on the returned object. As mentioned earlier, the `onAction()` function returns an observable.

- The success callback provided as an observer invokes the `activateUpdate()` function on the `SwUpdate` object. The observer callback between lines 6 and 10 are called when the user performs an action on the Snackbar component.
- Remember, the code between lines 6 and 10 is the same as Listing 6-6, which performed the install as soon as it identified a new version.

Managing Errors in Unrecoverable Scenarios

It is possible that users have not returned to the application for a while on a machine. Browsers will clear the cache and claim the disk space. When the user returns to the application (after the cache was cleaned), the service worker may not have all the scripts it needs. Imagine, meanwhile, that a new version of the application was deployed on the server. Now, the browser cannot obtain the deleted script (from the cache), not even from the server. This results in an unrecoverable state for the application. To the user, it leaves only one option: to upgrade to the latest version.

The `SwUpdate` service provides an observable called `unrecoverable` to handle such scenarios, as shown in Listing 6-13. Add an error handler when an unrecoverable state occurs. Notify the user and reload the browser window to clear the error. Similar to the earlier code samples, add the code in the `SwCommunicationService` constructor.

Listing 6-13. Handle the Unrecoverable State

```
01: export class SwCommunicationService {
02:
03:   constructor(private updateSvc: SwUpdate,
```

```
04:     private snackbar: MatSnackBar) {
05:     this.updateSvc.unrecoverable.subscribe( i => {
06:       console.log('The application is unrecoverable',
          i.reason);
07:       let snackRef = this.snackbar
08:       .open(`We identified an error loading the
          application. Use the following reload button.
          If the error persists, clear cache and reload the
          application`,
09:         "Reload");
10:     snackRef
11:       .onAction()
12:       .subscribe (() => {
13:         this.updateSvc.activateUpdate().then( () => {
14:           window.location.reload();
15:         });
16:      });
17:     });
18:   }
19: }
```

Consider the following explanation:

- See line 5. It subscribes to the `unrecoverable` observer on the instance of `SwUpdate` (namely, `updateSvc`). The success callback for the observable is between lines 6 and 17.
- You open a Snackbar component, which alerts the user about the error at the bottom center of the page. See Figure 6-5.

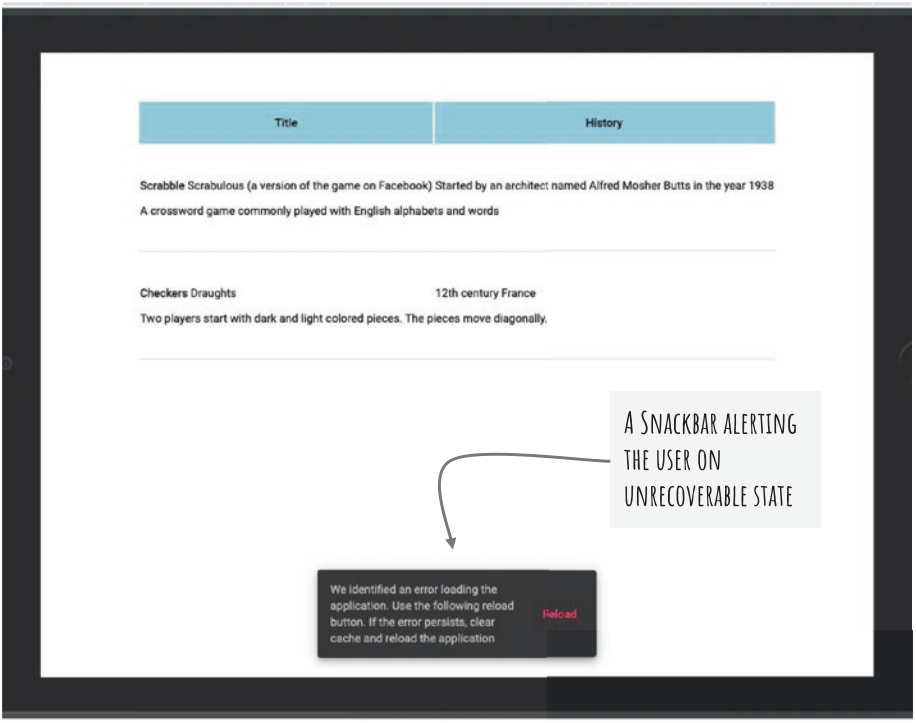

Figure 6-5. *A Snackbar component alerting the user of the unrecoverable error*

- Notice the reason field on line 6. It has additional information about the unrecoverable state. You may print and use this information in the browser console or log it to a central location to investigate further.

Summary

Installable and cached web applications are powerful. They enable users to access the application in low-bandwidth and offline scenarios. However, you also need to build features to seamlessly upgrade the applications. As you know, for Web Arcade, service workers manage caching and offline access features.

This chapter detailed how to seamlessly upgrade an Angular application and communicate with service workers. It extensively uses Angular's `SwUpdate` service, which provides many out-of-the-box features for identifying and activating a new version of the application. It uses observables; you subscribe when new versions of the applications are available or activated.

EXERCISE

- The chapter uses a Snackbar component to alert when a new version of the application is available. Show an alert after the upgrade is activated. Include information from `appData` in `ngsw-config.json`.
- Extend `ngsw-config.json` to show additional information about the release. Include details of enhancements and bug fixes. Take it close to the real-world use case. Users expect to see a summary of details about an upgrade.
- Explore positioning the Snackbar component at a different location on the page (as opposed to the default center bottom).
- Explore alerting users with more components other than a Snackbar component. Build an experience that suits a different use case better. Remember, we used the Snackbar component

as it does not interrupt and block an active user's workflow. However, a Snackbar component has been used for alerting the unrecoverable error scenario as well. Hence, choose an appropriate alert component and mechanism for such error conditions.

CHAPTER 7

Introduction to IndexedDB

So far you have cached the application skeleton and HTTP GET service calls. A RESTful service provides GET calls for data retrieval. However, HTTP also supports POST to create entities, PUT and PATCH for updates, and DELETE to remove entities. The sample application Web Arcade does not yet support offline access to service calls beyond the GET calls.

This chapter introduces IndexedDB for more advanced offline actions. In this chapter, you will get a basic understanding of IndexedDB, which runs on the client side on browsers. You will learn to get started with IndexedDB in an Angular application. JavaScript provides APIs to integrate with IndexedDB. You can create, retrieve, update, and delete data to/from IndexedDB, which is supported by most modern browsers. The chapter focuses on structuring the database including creating object stores, indices, etc. In the next chapter, you will work with data by creating and deleting records.

Traditionally, web applications used various features for client-side storage including cookies, session storage, and local storage. Even today they are highly useful for storing reasonably small amounts of data. IndexedDB, on the other hand, provides an API for more sophisticated client-side storage and retrieval. The JavaScript API is natively supported by most modern browsers. IndexedDB provides persistent storage for relatively large amounts of data including JSON objects. However, no

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4_7

database supports storing an unlimited amount of data. The browsers set an upper limit on the amount of data stored in IndexedDB relative to the size of the disk and the device.

IndexedDB is useful for persisting structured data. It saves data in key-value pairs. It works like a NoSQL database, which supports using object stores that contain records of data. The object stores are comparable to tables in a relational database. The traditional relational databases largely use tables with a predefined structure in terms of columns and constraints (primary key, foreign key, etc.). However, IndexedDB uses object stores to persist records of data.

IndexedDB supports high-performance searches. Data is organized with the help of indexes (defined on an object store), which help to retrieve data faster.

Terminology

Consider the following terminology working with IndexedDB:

- *Object store*: An IndexedDB may have one or more object stores. Each object store acts as a container for key-value pairs of data. As mentioned, an object store is comparable to tables in relational databases.
    - An object store provides structure to the IndexedDB. Create one or more object stores as logical containers of the application data. For example, you may create an object store named `users` to store user details and another named `games` to persist a list of game-related objects.
- *Transactions*: Data operations on IndexedDB are performed in the context of a transaction. This helps maintain data consistency. Remember, IndexedDB

stores and retrieves data on the client side in a browser. It is possible that more than one instance of the applications are open by the user. It can create scenarios, where create/update/delete operations are partially performed by each instance of the browser. One of the browsers may retrieve stale data while an update operation is in-progress.

  - Transactions help avoid the previously mentioned problems. A transaction locks data records until the operation is complete. The data access and modification operations are atomic. That is, an create/update/delete operation is either fully done or completely rolled back. A retrieve operation is performed only after a data modification operation is completed or rolled back. Hence, retrieve never returns inconsistent and stale data objects.

  - IndexedDB supports three modes of transactions, namely, `readonly`, `readwrite`, and `versionchange`. As you can imagine, `readonly` helps with retrieve operations and `readwrite` with create/update/delete operations. However, `versionchange` mode helps create and delete object stores on an IndexedDB.

- *Index*: An index helps retrieve data faster. An object store sorts data in the ascending order of the key. A key implicitly does not allow duplicate values. You can create additional indices that also act as a uniqueness constraint. For example, adding an index on a Social Security number or national ID ensures there are no duplicates in IndexedDB.

- *Cursor*: A cursor helps iterate over records in an object store. It is useful while iterating over the data records during the query and retrieval process.

Getting Started with IndexedDB

IndexedDB is supported by major browsers. The API enables applications to create, store, retrieve, update, and delete records in a local database, within the browser. This chapter details using IndexedDB with the native browser API.

The following are the typical steps when working with IndexedDB:

1. *Create and/or open a database*: For the first time, create a new IndexedDB database. As the user comes back to the web application, open the database to perform actions on the database.
2. *Create and/or use an object store*: Create a new object store the first time a user accesses the functionality. As mentioned earlier, an object store is comparable to a table in a relational database management system (RDBMS). You may create one or more object stores. You create, retrieve, update, and delete documents in an object store.
3. *Start a transaction*: Actions are performed on an IndexedDB object store as a transaction. It enables you to maintain a consistent state. For example, it is possible the user closes the browser while an action is being performed on the IndexedDB database. As the action is performed in the context of a transaction, if it is not complete, the transaction is aborted. A transaction ensures an error or an

edge condition does not leave the database in an inconsistent or unrecoverable state.

4. *Perform CRUD*: Like any database, you create, retrieve, update, or delete documents in an IndexedDB.

Consider the following Web Arcade use case, taking advantage of IndexedDB.

In an earlier chapter, you have seen a page showing a list of board games. Consider supporting a new use case with a game details page. Figure 7-1 details all the information about a game. Below the game description, you show a list of user comments and a form allowing users to add new comments. As a user types in a new comment and submits the form, you post this data to a remote HTTP service. The service ideally persists the user comment in a permanent storage/database like MongoDB or Oracle or Microsoft SQL Server. Considering the server-side code is not in the scope of this book, we will keep it simple. In the next chapter, the code samples showcase a service that stores the user comments in a file.

Figure 7-1 shows the section of the page with a current list of comments and a form that allows users to submit new comments.

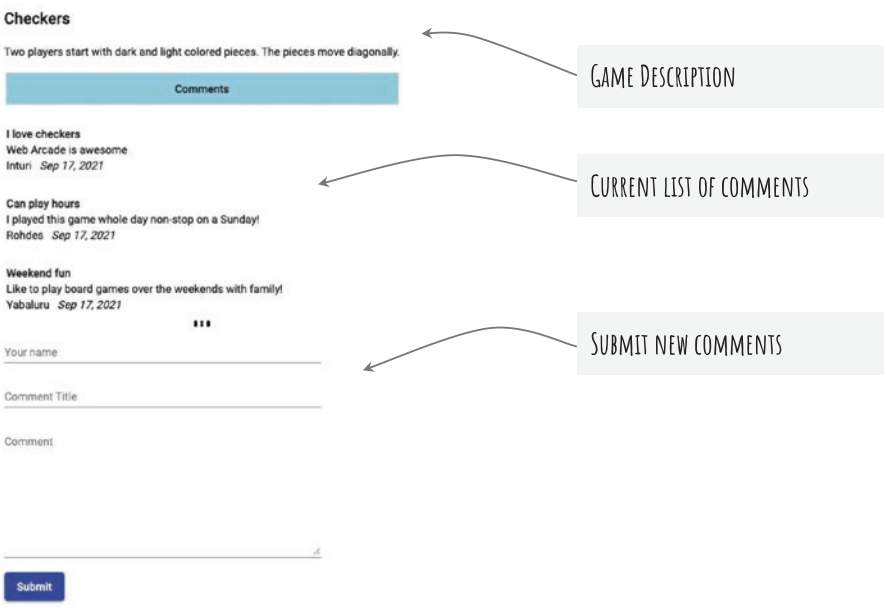

Figure 7-1. *List and submit comments on a game details page*

The submit action creates a new comment. The service endpoint is an HTTP POST method. As mentioned, the Web Arcade supports offline access on HTTP GET calls. Imagine losing connectivity as a user types in a comment and clicks Submit. A typical web application returns an error or a message similar to "Page can't be displayed." Web Arcade is designed to be resilient to the loss of network connectivity. Hence, Web Arcade caches user comments and synchronizes with a server-side service when the user returns to the application.

Angular Service for IndexedDB

Create a new service by running the following command:

```
ng generate service common/idb-storage-access
```

The command creates a new service called `IdbStorageAccessService` in the directory `src/app/common`. The service is to abstract code statements accessing IndexedDB. It is a central service that uses a browser API to integrate with IndexedDB. During initialization, the service performs one-time activities like creating a new IndexedDB store or opening the database if it already exists. See Listing 7-1.

Listing 7-1. Initialize IndexedDB with the IdbStorageAccessService

```
01: @Injectable()
02: export class IdbStorageAccessService {
03:
04:   idb = this.windowObj.indexedDB;
05:
06:   constructor(private windowObj: Window) {
07:   }
08:
09:   init() {
10:     let request = this.idb
11:       .open('web-arcade', 1);
12:
13:     request.onsuccess = (evt:any) => {
14:       console.log("Open Success", evt);
15:     };
16:
17:     request.onerror = (error: any) => {
18:       console.error("Error opening IndexedDB", error);
19:     }
20:   }
21:
22: }
23:
```

Note By default, the `ng generate service` command provides the service at the root level. In the context of the Web Arcade application, you may want to remove the `provideIn: 'root'` statement on line 1. Just leave the `inject()` decorator, as shown in the first line.

This is explained in detail in the following section along with Listing 7-2.

Consider the following explanation:

- Line 4 creates the class variable `idb` (short for IndexedDB). It is set to the `indexedDB` instance on the global window object. The `indexedDB` object has an API that helps open or create a new IndexedDB. Line 4 runs while initializing `IdbStorageAccessService`, similar to constructor.

Note Notice, the global window object is accessed through a `Window` service. See the constructor on line 6. It injects the window service. The instance variable is named `windowObj`. The `Window` service is provided in the `AppModule`.

- See lines 9 to 20 for the `init()` function initializing the service.
- See lines 10 and 11 that run the `open()` function on the `idb` object. If it is the first time a user opened the application on a browser, it creates a new database.
    a. The first parameter is the name of the database, `web-arcade`.

b. The second parameter (value 1) specifies a version for the database. As you can imagine, new updates to the application cause changes to the IndexedDB structure. The IndexedDB API enables you to upgrade the database as the version changes.

To return a user, the database was already created and available on a browser. The `open()` function attempts to open the database. It returns an object of the `IDBOpenDBRequest` object.

Figure 7-2 shows a new IndexedDB `web-arcade` that was created. The image was captured using Google Chrome's developer tools. Similar functionality is available for developers on all major browsers including Firefox and Microsoft Edge.

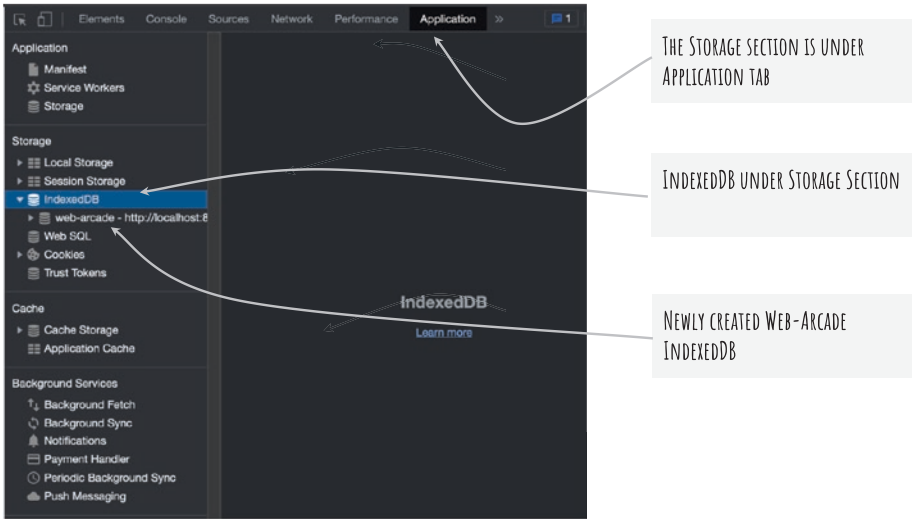

Figure 7-2. *IndexedDB in Google Chrome Dev Tools*

Almost all the IndexedDB APIs are asynchronous. An action like open does not attempt to complete the operation immediately. You specify a

callback function, which is invoked after completing the action. As you can imagine, the open action can be successful or error out. Hence, define a callback function for each outcome, `onsuccess` or `onerror`. See lines 13 to 15 and lines 17 to 19 in Listing 7-1. For the moment, you just print the result on the console (lines 14 and 18). We will further enhance handling the result in the upcoming code snippets.

When is the `init()` function invoked? It is one of the methods on the Angular service. You may invoke it in a component, which means IndexedDB is initialized only when you load (or navigate) to the component. On the other hand, an application like Web Arcade is highly dependent on IndexedDB. You may need to make use of the service from multiple components. The service needs to complete initialization and be ready for CRUD operations. Hence, it is a good idea to initialize the service along with the application, while the primary module `AppModule` starts. Consider Listing 7-2.

Listing 7-2. Initialize IndexedDB with IdbStorageAccessService

```
03: import { NgModule, APP_INITIALIZER } from '@angular/core';
15: import { IdbStorageAccessService } from './common/idb-
    storage-access.service';
18:
19: @NgModule({
20:   declarations: [
21:     AppComponent,
25:   ],
26:   imports: [
27:     BrowserModule,
40:   ],
41:   providers: [
42:     IdbStorageAccessService,
43:     {
44:       provide: APP_INITIALIZER,
```

```
45:       useFactory: (svc: IdbStorageAccessService) => () =>
          svc.init(),
46:       deps: [IdbStorageAccessService], multi: true
47:     }
48:   ],
49:   bootstrap: [AppComponent]
50: })
51: export class AppModule { }
52:
```

Considering the following explanation:

- See lines 42 to 48. The first line (line 42) in the block provides a newly created `IDBStorageAccessService`. Why do we need it? As you have seen, we did not provide the service at the root level. We removed the line of code `provideIn: 'root'` in `IdbStorageAccessService` (Listing 7-1).
- See lines 43 to 47, which provide `APP_INITIALIZER` and use the factory function, which invokes `init()`.
- In summary, we provide and initialize the `IdbStorageService` at a module level. In this example, you do it in `AppModule`. It could have been any module.

  It creates and/or opens the `Web-Arcade` IndexedDB on the browser. It keeps the database ready for further operations (e.g., CRUD). This code eliminates the need to inject the service into a component (or another service) and call the `init()` function. The service initializes along with the `AppModule`.

Creating Object Store

While the database is the highest level in IndexedDB, it can have one or more object stores. You provide a unique name to each object store within a database. An object store is a container that persists data. In the current example with Web Arcade, you will see how to save JSON objects. For ease of understanding, an object store is comparable to tables in a relational database.

Using "onupgradeneeded" Event

An event called onupgradeneeded is triggered after creating or opening an IndexedDB. You provide a callback function that is invoked by the browser when this event occurs. In the case of a new database, the callback function is a good place to create object stores. For a pre-existing database, if an upgrade is required, you may perform design changes here. For example, you may create new object stores, delete unused object stores, and modify an existing object store by deleting and re-creating it. Consider Listing 7-3.

Listing 7-3. onupgradeneeded Event Callback

```
01: init() {
02:     let request = this.idb
03:         .open('web-arcade', 1);
04:
05:     request.onsuccess = (evt: any) => {
06:         console.log("Open Success", evt);
07:     };
08:
09:     request.onerror = (error: any) => {
10:         console.error("Error opening IndexedDB", error);
```

```
11:     }
12:
13:     request.onupgradeneeded = function (event: any) {
14:         console.log("version upgrade event triggered");
15:         let dbRef = event.target.result;
16:         dbRef
17:             .createObjectStore("gameComments", {
                autoIncrement: true });
18:     };
19: }
```

Considering the following explanation:

- Notice that the code snippet repeated the `init()` function from Listing 7-1. In addition to the `onsuccess` and `onerror` callbacks, an event handler called `onupgradeneeded` is included. See lines 13 to 18.
- The event is provided as a parameter to the function callback.
- You can access a reference to IndexedDB on the event target on an object, namely, `target`.
- Use the `db` reference to create an object store. In this example, you name the object store `gameComments`. As explained earlier, you use IndexedDB and the object store to cache user comments if the user loses connectivity.

An object store persists data in key-value pairs. As you will see in the next few sections, data is retrieved using the key. It is a primary key uniquely identifying a value stored in IndexedDB. The following are the two options to create a key (for the values stored in an object store).

This is decided at the time of creating an object store. See line 17 in Listing 7-3. Notice the second parameter on the `createObjectStore()` function. You specify one of the following two options:

- *Auto increment*: IndexedDB manages the key. It creates a numeric value and increments for every new object added in the object store.

  ```
  dbRef.createObjectStore("gameComments", {
  autoIncrement: true });
  ```

- *Key path*: Specify a key path within the JSON object being added. As the key value is provided explicitly, ensure you provide unique values. A duplicate value causes the insert to fail.

  A field called `commentId` is provided as a keypath. If used, ensure you provide a unique value for `commentId`.

  ```
  dbRef.createObjectStore("gameComments", {
  keypath: 'commentId' });
  ```

Note A key path can be supplied only for a JavaScript object. Hence, creating an object store with a key path constrains it to store only the JavaScript objects. However, with auto increment, considering the key is managed by IndexedDB, you may store any type of object including a primitive type.

See Figure 7-3 with the newly created `gameComments` object store.

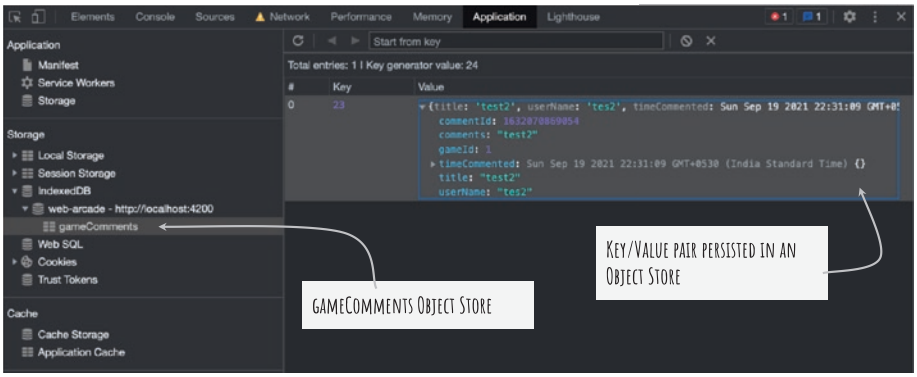

Figure 7-3. *gameComments object store and a sample value*

Creating Index

While defining an object store, you can create additional indices that also act as a uniqueness constraint. The index is applied on a field in the JavaScript object persisted in the object store. Consider the following code snippet. It explains the `createIndex` API on the object store reference.

```
objectStoreReference.createIndex('indexName', 'keyPath',
{parms})
```

Consider the following explanation:

- `Index name`: The first parameter is an index name (arbitrary).
- `Key path`: The second parameter, `keypath`, specifies that the index needs to be created on the given field.
- `Params`: You may specify the following parameters for creating an index:
  a. `unique`: This creates an uniqueness constraint on a field provided at the keypath.
  b. `multiEntry`: This is applied on an array.

If true, the constraint ensures each value in the array is unique. An entry is added to the index for each element in the array.

If false, the index adds a single entry for the entire array. The uniqueness is maintained at the array object level.

In the gameComments object store, imagine each comment will have an ID. To ensure the ID is unique, add an index. Consider Listing 7-4.

Listing 7-4. Create Index IdxCommentId for the Comment ID

```
1: request.onupgradeneeded = function(event: any){
2:     console.log("version upgrade event triggered");
3:     let dbRef = event.target.result;
4:     let objStore = dbRef
5:       .createObjectStore("gameComments", {
         autoIncrement: true })
6:
7:     let idxCommentId = objStore.createIndex('IdxCommentId',
       'commentId', {unique: true})
8:   };
```

Notice that line 7 creates an index using an object store reference, objStore. The index is named IdxCommentId. The index is added to the commentId field. You can see that the parameter unique is set to true, which ensures commentId is distinct for each record. Figure 7-4 showcases the object store with the new index.

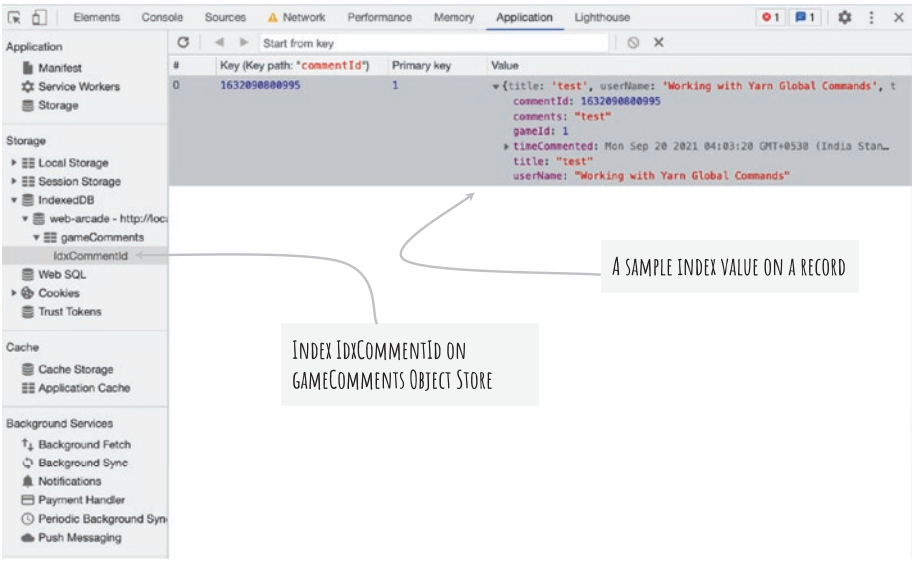

Figure 7-4. *Index IdxCommentId on an object store*

Browser Support

Figure 7-5 depicts the browser support for the global `indexedDB` object (`windowObj.indexedDB`). Notice that the data is captured on the Mozilla website, at `https://developer.mozilla.org/en-US/docs/Web/API/indexedDB`. It is a reliable and open source platform for web technologies. Mozilla has been an advocate of the open web and has pioneered safe and free Internet technologies including the Firefox browser.

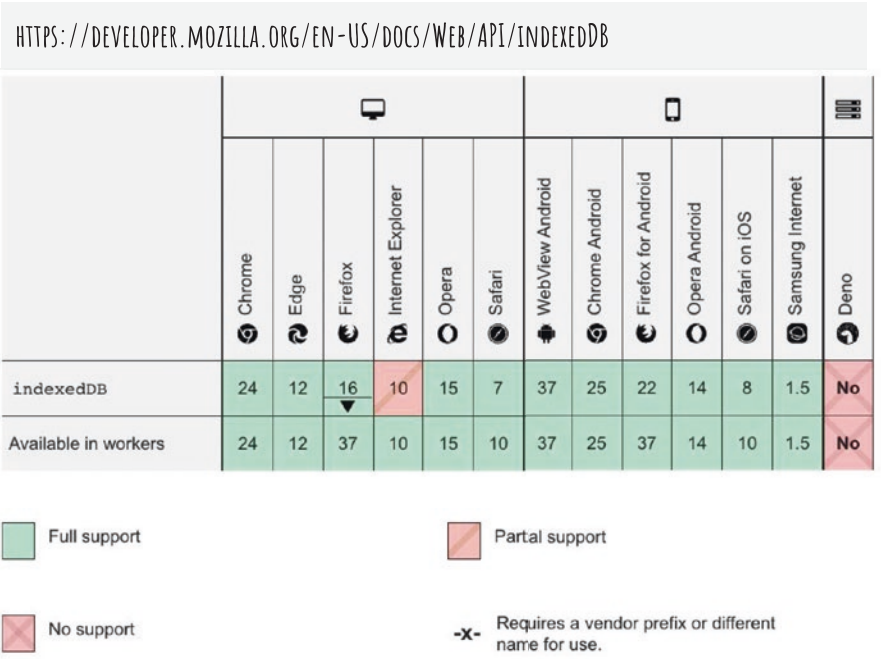

Figure 7-5. *window.indexedDB browser support*

Also refer to CanIUse.com, which is a reliable source of browser compatibility data. For IndexedDB, use the URL `https://caniuse.com/indexeddb`.

Limitations of IndexedDB

While IndexedDB provides a good solution for client-side persistence and querying in a browser, it is important to be aware of the following limitations:

- It does not support internationalized sorting, so sorting non-English strings could be tricky. Few languages sort strings differently from English. At the time of writing

this chapter, localized sorting is not fully supported by IndexedDB and all the browsers. If this feature is important, you might have to retrieve data from the database and write additional custom code to sort.

- There is no support for full-text search yet.
- IndexedDB cannot be treated as a source of truth for data. It is temporary storage. Data could be lost or cleared in the following scenarios:

    a. The user resets the browser or clears the database manually.

    b. The user launches the application in a Google Chrome Incognito window or a private browsing session (on other browsers). As the browser window is closed, considering it was a private session, the database will be removed.

    c. Disk quota for persistent storage is calculated based on a few factors including available disk space, settings, device platform, etc. It is possible the application crossed the quota limit and further persistence failed.

    d. Miscellaneous situations including corrupt databases, a bug upgrading the database caused by an incompatible change, etc.

Summary

This chapter provided a basic understanding of IndexedDB, which runs on the client side on browsers. JavaScript provides a native API to work with IndexedDB. It is supported by most modern browsers.

The chapter also explained how to initialize the Angular service along with `AppModule`. In the initialization process, you create or open IndexedDB store for Web Arcade. You create a new IndexedDB store if it is the first time a user is accessing the application on a browser. You open the pre-existing database if it is already there.

Next, the chapter explained how to use the `onupgradeneeded` function callback for creating an object store and indices. These are one-time activities for the first time a user accesses the application.

EXERCISE

- Create an additional object store for creating new games. Perform the action while loading the application (or the Angular module).
- Create the object store to use a designated ID as the key (primary). Do not use auto increment.
- Create an additional index on the game title. Ensure it is unique.

CHAPTER 8

Creating the Entities Use Case

While working with data, you began with data retrieval. You used remote HTTP services to make GET calls and show the data on the screen in an Angular application. You created the ability to cache resources including data calls. As you progress, applications need to create, update, and delete data. In an application that supports offline functionality, the create, update, and delete actions are complex.

This chapter establishes a use case for handling such scenarios. It details how to build Angular components and services that perform the create action. The example can be easily upgraded to edit and delete actions. The previous chapter introduced IndexedDB for persisting data in the browser. It is typically used for managing cache. The use case described in this chapter helps take full advantage of the IndexedDB implementation. As you proceed further in the book, the next chapter details how to perform offline actions with IndexedDB, so you need to understand the use case we will build in this chapter.

In Web Arcade, the create, update, and delete actions are performed on the game details page. The page will call remote HTTP services to save data. The chapter begins by explaining the HTTP methods for the previously mentioned actions. Next, it details how to create the component. It introduces Angular routing for navigation between the list component, which shows a list of board games and the details page. Next,

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4_8

it details the features we build on the game details page while developing the use case. Finally, the chapter details how to build mock server-side services to support the Angular application.

Web Arcade: Game Details Page

The game details page shows the details of a selected game. We use this page to showcase an example of how to synchronize offline actions with a remote HTTP service.

While invoking a remote HTTP service, the HTTP methods define a desired action to be performed. Consider the following most used HTTP methods:

- GET retrieves data. For example, it retrieves a list of board games.
- POST submits or creates an entity. For example, it creates a service to create a board game or add user comments on a game that implements the POST method.
- PUT replaces or fully updates an entity. For example, consider a board game entity with fields for a game ID, the game title, a game description, a web link with comprehensive details about the game, the origin, etc. For a given game ID, use a PUT method to replace all the fields. Even if a few fields do not change, you can provide the same values again while using the PUT method.
- PATCH replaces or updates a few fields on an entity. In the previous example, consider updating just the origin field. Develop an HTTP service with PATCH to update the origin field on an entity.
- DELETE removes or deletes an entity.

Note In addition to the previous HTTP methods, there are other less used HTTP methods including HEAD, CONNECT, OPTIONS, and TRACE.

So far, offline access was provided to GET service calls. This chapter uses IndexedDB to cache and synchronize POST service calls. You may use a similar implementation on the remaining HTTP methods.

In an earlier chapter, you created a component to show a list of games. In this chapter, you will update the sample so that a click selects a game to navigate to the details page. See Figure 8-1.

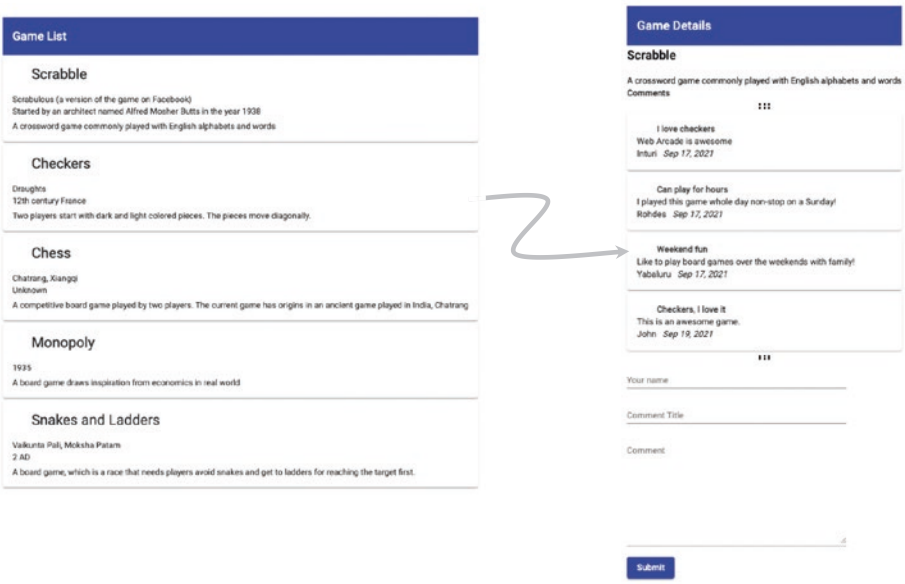

Figure 8-1. *Navigation to a game details page*

The game details page has a description and additional details about a game. It lists comments from all the users. It provides a form at the bottom to submit a new comment. See Figure 8-2.

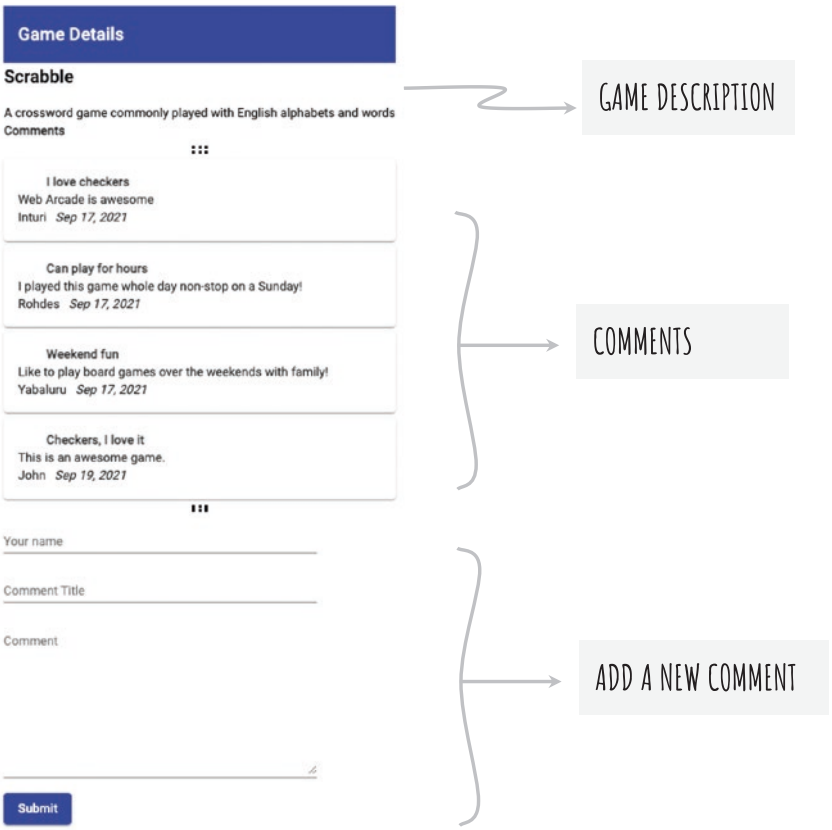

Figure 8-2. *Fields on game details page*

Offline Scenario

A user can submit comments. When online, the service calls an HTTP service to post new comments. However, if offline, use IndexedDB to save the comments temporarily. Once back online, post the comments to the remote service.

Creating a Component for Game Details

Run the following Angular CLI command to create a new component:

```
ng g c game-details
```

In the next few sections, you will update the component to show the game details. However, the game was selected in the earlier game list component. How does the game details component know about the selected game? Remember, you navigate to the game details as the user selects from a list of board games. The list component provides the selected game ID as a `query param` in the URL. Consider the following for a URL with the game ID parameter:

```
http://localhost:4200/details?gameId=1
```

Routing

Angular routing enables Angular applications to take advantage of the URL (see the address bar in a browser) and load content on the fly. You map a component to a path in the URL. The component loads as the user navigates to the respective path.

Remember, when you created a new application for Web Arcade with Angular CLI, routing was already set up. This includes an `AppRoutingModule` for configuring custom paths and URLs. Update the route configuration to show the game list component first and the game details component when navigating to the details page (as shown at the previously mentioned URL). Consider the route configuration in `app-routing.module.ts`, as shown in Listing 8-1.

Listing 8-1. Route Configuration

```
06: const routes: Routes = [{
07:   path: "home",
```

```
08:    component: BoardGamesComponent
09: }, {
10:    path: "details",
11:    component: GameDetailsComponent
12: }, {
13:    path: "",
14:    redirectTo: "/home",
15:    pathMatch: "full"
16: }];
17:
18: @NgModule({
19:    imports: [RouterModule.forRoot(routes)],
20:    exports: [RouterModule]
21: })
22: export class AppRoutingModule { }
23:
```

Notice that the board games component is configured to load with the path `/home`, and the game details component is configured to load with the path `/details`, for example, `http://localhost:4200/home` and `http://localhost:4200/details`.

The components load at `router-outlet` in the HTML template. Remember, `AppComponent` is the root component. Update the router outlet to load the previously mentioned components as and when the user navigates to the respective URLs (paths). Consider the following short snippet:

```
1: <div class="container align-center">
2:     <router-outlet></router-outlet>
3: </div>
```

Navigate to Game Details Page

Next, update the list component (`BoardGamesComponent`) to navigate to the details page. Edit the component's HTML template (`src/app/components/board-games/board-games.component.html`). See Listing 8-2.

Listing 8-2. Board Games Component Template

```
<mat-toolbar color="primary">
    <mat-toolbar-row>Game List</mat-toolbar-row>
</mat-toolbar>
<div>
    <ng-container *ngFor="let game of (games | async)?.
    boardGames">
        <a (click)="gameSelected(game)">
            <mat-card>
                <mat-card-header>
                    <h1>
                        {{game.title}}
                    </h1>
                </mat-card-header>
                <mat-card-content>
                    <span>{{game.alternateNames}}</span>
                    <div>{{game.origin}}</div>
                    <div>{{game.description}}</div>
                </mat-card-content>
            </mat-card>
        </a>
    </ng-container>
</div>
```

Consider the following explanation:

- See lines 6 and 19. Each game (card) is enclosed in a hyperlink element, `<a></a>`.
- Notice that line 5 iterates through a list of games with the `ngFor` directive. The variable `game` represents a game in the iteration.
- In line 6, the click event is handled by the `gameSelected()` function. Notice the game variable is passed in as a parameter. This is the variable with game data in the current iteration.

The `gameSelected` function (defined in the game details component's TypeScript file) navigates to the game details page, as shown in Listing 8-3.

Listing 8-3. Navigate to the Details Page

```
01:
02: export class BoardGamesComponent implements OnInit {
03:
04:   constructor(private router: Router) { }
05:
06:   gameSelected(game: BoardGamesEntity){
07:    this.router.navigate(['/details'], {queryParams:
       {gameId: game.gameId}})
08:   }
09:
10: }
```

Consider the following explanation:

- A router service instance is injected in line 4.
- Line 7 uses a router instance to navigate to the details page.

- Notice that a query param game ID is provided. The game object is passed in from the template. See the earlier Listing 8-2.
- The `game-id` selected in the current list component (`BoardGamesComponent`) will be used in the game details component. It retrieves complete details of the selected game.

Next, retrieve the game ID from the URL in the game details component, as shown in Listing 8-4.

Listing 8-4. Retrieve Game ID from Query Params

```
01: import { Component, OnInit } from '@angular/core';
02: import { ActivatedRoute } from '@angular/router';
03:
04: @Component({
05:   selector: 'wade-game-details',
06:   templateUrl: './game-details.component.html',
07:   styleUrls: ['./game-details.component.sass']
08: })
09: export class GameDetailsComponent implements OnInit {
10:   game: BoardGamesEntity;
      commentsObservable = new Observable<CommentsEntity[]>();
11:   constructor(private router: ActivatedRoute, private
      gamesSvc: GamesService) { }
12:
13:   ngOnInit(): void {
14:     this.router
15:       .queryParams
16:       .subscribe( r =>
17:         this.getGameById(r['gameId']));
18:   }
19:
```

```
20:   private getGameById(gameId: number){
21:     this.gamesSvc.getGameById(gameId).subscribe(
22:       (res: BoardGamesEntity) => {
23:         this.game = res;
24:         this.getComments(res?.gameId);
25:       });
26:   }
27: }
28:
29: private getComments(gameId: number){
30:     this.commentsObservable = this.gamesSvc.
        getComments(gameId);
31: }
32:
```

Consider the following explanation:

- The sample uses the `ActivatedRoute` service to read query params in the URL.
- Line 2 imports the `ActivatedRoute` service from Angular's router module. Next, line 11 injects the service into the component. The service instance is named `router`.
- See the `ngOnInit()` function between lines 13 and 18. This function is invoked after the constructor and during the component initialization.
- See the code between lines 14 and 17. Notice that the code uses `router.queryParams`. The `queryParams` is an observable. Subscribe to it to access the query params.

- The result of the `queryParams` subscription is named `r`. Access the game ID as a field on the result, `r['gameId']`. Now, you have access to the game ID provided by `BoardGamesComponent`.
- Pass the game ID as a function parameter to a private function, `getGameById()`. This function is defined in lines 20 to 27.
- The `getGameById()` function invokes another function with the same name `getGameById()` defined as part of `GamesService`. It returns an observable, and subscribing to it returns results from an HTTP service. The remote HTTP service provides game details by `GameId`.
- In line 23, you set the results from the HTTP service onto a game object on the component, which is used in the HTML template.
- The HTML template shows the game details to the user.
- Next, call the private function `getComments()`, which retrieves comments on the given board game. See lines 29 to 31. It invokes the `getComments()` function on the `GameService` instance, which obtains data from a remote HTTP service.
- Set results from the HTTP service onto a `commentsObservable` object on the component, which is used in the HTML template. The HTML template shows the comments.

In summary, Listing 8-4 retrieves the game details and comments and sets them on a class variable. In line 23, the class field `game` has selected the game title, description, etc. Next, on line 30, the class field

`commentsObservable` has a list of comments. These are comments made by various users on the selected game. Next, see the HTML template code that renders the game details and comments. Consider Listing 8-5.

Listing 8-5. Game Details Component HTML Template

```
01:
02: <!-- Toolbar to provide a title-->
03: <mat-toolbar [color]="toolbarColor">
04:     <h1>Game Details</h1>
05: </mat-toolbar>
06:
07:
08: <!-- This section shows game title and description-->
09: <div *ngIf="game">
10:     <h2>{{game.title}}</h2>
11:     <div>{{game.description}}</div>
12: </div>
13:
14:
15: <!-- Following section shows comments made by users -->
16: <div>
17:     <strong>
18:         Comments
19:     </strong>
20:     <hr />
21:     <mat-card *ngFor="let comment of commentsObservable
        | async">
22:         <mat-card-header>
23:             <strong>
24:                 {{comment.title}}
25:             </strong>
```

```
26:         </mat-card-header>
27:         <mat-card-content>
28:             <div>{{comment.comments}}</div>
29:             <div><span>{{comment.userName}}</span> <span
                class="date">{{comment.timeCommented | date}}
                </span></div>
30:         </mat-card-content>
31:     </mat-card>
32: </div>
```

Consider the following explanation:

- Lines 10 and 11 show the game title and description. Line 9 checks if the game object is defined (with an `ngIf` directive). This is to avoid errors with the component before the game data is obtained from the service. As you can imagine, when the component first loads, the service call is still in progress. The game title, description, and other fields are not yet available. Once retrieved from the service, the `ngIf` condition turns true, and the data is displayed.
- See line 21. It iterates through the comments. Line 24 shows the comment title. Lines 28 and 29 show the comment description, username, and comment timestamp.
- See Listing 8-4. `commentsObservable` is of type `Observable`. Hence, line 27 in Listing 8-5 uses `| async`.
- Notice the following HTML styling decisions:
  - The listing uses Angular Material's Toolbar component (`mat-toolbar`) to show the title. See lines 3 to 5.

- Each comment is shown on an Angular Material card. See the components `mat-card`, `mat-card-header`, and `mat-card-content` on lines 21 to 31.

Listing 8-4 uses two functions from a service: `getGameById()` and `getComments()`. As you can imagine, the Angular service function invokes a remote HTTP service to get the data.

Remember, we developed mock services to demonstrate remote HTTP service functionality. You returned mock JSON for board games. For the previous two functions, `getGameById()` and `getComments()`, you will extend the Node.js Express service. It is covered later in the chapter, in the section "Updates to Mock HTTP Services."

Note A real-world service integrates with and creates, retrieves, and updates data in mainstream databases such as Oracle, Microsoft SQL Server, or MongoDB. It is out of scope for this book. To ensure the code samples are functional, we created mock services.

However, as you have seen in the previous code samples, the components do not integrate directly with remote HTTP services. You use an Angular service, which uses other services to abstract this functionality from the components. The components purely focus on presentation logic for the application.

Remember, you created a service called `GamesService` for encapsulating the code that retrieves games data. Next, update the service to include the previous two functions `getGamesById()` and `getComments()`, as shown in Listing 8-6.

Listing 8-6. Game Service Invoking Remote HTTP Services

```
01: @Injectable({
02:     providedIn: 'root'
```

```
03:   })
04:   export class GamesService {
05:
06:     constructor(private httpClient: HttpClient) { }
07:
08:     getGameById(gameId: number):
        Observable<BoardGamesEntity>{
09:       return this
10:         .httpClient
11:         .get<BoardGamesEntity>(environment.
            boardGamesByIdServiceUrl,{
12:           params: {gameId}
13:         });
14:     }
15:
16:     getComments(gameId: number):
        Observable<CommentsEntity[]>{
17:         return this
18:           .httpClient
19:           .get<CommentsEntity[]>(environment.
              commentsServiceUrl,{
20:             params: {gameId}
21:           });
22:       }
23:
24: }
```

Consider the following explanation:

- Line 6 injects the `HttpClient` service. It is an out-of-the-box service provided by Angular to make HTTP calls.

- See lines 10 and 18. The functions use the `HttpClient` instance `httpClient.` It invokes the remote HTTP service.
- Both the functions use the GET HTTP method. The first parameter is the endpoint URL.
- It is advisable to configure URLs (instead of hard-coding them in the application). Hence, the URLs are updated in an environment file. See Listing 8-7 for the environment file.
- Notice that both the functions require `gameId` as a parameter. See lines 8 and 16.
- The game is passed as a query param to the remote HTTP service. See lines 12 and 20.
- Notice that `getGameById()` returns an observable of type `BoardGamesEntity (Observable<BoardGamesEntity>)`. The remote service is expected to return a JSON response adhering to the interface contract specified in `BoardGamesEntity`. See Listing 8-8(a) for the interface definition.
- `getComments()` returns an observable of type `CommentsEntity` (`Observable<CommentsEntity>`). As multiple comments are retrieved from the service, it is an array. The remote service is expected to return a JSON response adhering to the interface contract specified in `CommentsEntity`. See Listing 8-8(b) for the interface definition.

- The remote service calls return an observable because they are asynchronous. The service does not return the data as soon as the browser invokes it. The code does not wait until the result is returned. Hence, a subscriber callback function is invoked once the data is available from the remote service.

Listing 8-7. Environment File with Additional Endpoints

```
08: export const environment = {
09:   boardGameServiceUrl: `/api/board-games`,
10:   commentsServiceUrl: '/api/board-games/comments',
11:   boardGamesByIdServiceUrl: '/api/board-games/gameById',
12:   production: false,
13: };
14:
```

Listing 8-8(a). TypeScript Interface BoardGamesEntity

```
01: export interface BoardGamesEntity {
02:     gameId: number;
03:     age: string;
04:     link: string;
05:     title: string;
06:     origin: string;
07:     players: string;
08:     description: string;
09:     alternateNames: string;
10: }
```

Listing 8-8(b). TypeScript Interface CommentsEntity

```
1: export interface CommentsEntity {
2:     title: string;
```

```
3:     comments: string;
4:     timeCommented: string;
5:     gameId: number;
6:     userName:string;
7: }
```

Note The URLs are required in two files: `src/environments/environment.ts` and `src/environments/environment.prod.ts`. The `environment.ts` file is used for development builds (for example, `yarn start`). The `environment.prod.ts` file is used for production builds (for example, `yarn build` or `ng build`).

Adding Comments

See Figure 8-2. Notice the last section with a data form to add a comment. It enables users to add comments about the board game. So far you have largely worked with data retrieval. This is an example of creating an entity, namely, a comments entity. As mentioned earlier, you use the HTTP POST method for creating an entity in the back-end system.

Consider Listing 8-9, which shows the Add Comment HTML template.

Listing 8-9. Add Comment HTML Template

```
01: <div>
02:     <mat-form-field>
03:         <mat-label>Your name</mat-label>
04:         <input matInput type="text" placeholder="Please
            provide your name" (change)="updateName($event)">
05:     </mat-form-field>
06: </div>
```

```
07:
08: <div>
09:     <mat-form-field>
10:         <mat-label>Comment Title</mat-label>
11:         <input matInput type="text" placeholder="Please
            provide a title for the comment" (change)="updateTi
            tle($event)">
12:     </mat-form-field>
13: </div>
14:
15: <div>
16:     <mat-form-field>
17:         <mat-label>Comment</mat-label>
18:         <textarea name="comment" id="comment"
            placeholder="Write your comment here"
            (change)="updateComments($event)" matInput
            cols="30" rows="10"></textarea>
19:     </mat-form-field>
20: </div>
21:
22: <button mat-raised-button color="primary" (click)=
    "submitComment()">Submit</button>
```

Consider the following explanation:

- Notice lines 1 to 20. They create form fields for the username, title, and comment detail.
- The listing uses Material Design components and directives. Lines 4, 11, and 18 use `matInput` with the elements input and text area, respectively. These Angular Material elements need the Material Design input module. See Listing 8-10, lines 1 and 8.

- The `mat-form-field` component encapsulates the form field and the label. The component `mat-label` shows a label for the form field.
- Lines 4, 11, and 18 use change event data binding with the functions `updateName()`, `updateTitle()`, and `updateComments()`. Listing 8-11 sets the value of a form field to a variable in the component. The change event occurs every time a change occurs (a user types in a value) in the form field.
- In Listing 8-9, notice the click event data binding in the HTML template on line 22. The TypeScript function `submitComments()` is called as and when the user clicks the button.

Listing 8-10. Import Angular Material Input Module

```
01: import { MatInputModule } from '@angular/material/input';
02:
03: @NgModule({
04:   declarations: [
05:     AppComponent,
06:   ],
07:   imports: [
08:     MatInputModule,
09:     BrowserAnimationsModule
10:   ],
11:   bootstrap: [AppComponent]
12: })
13: export class AppModule { }
14:
```

Consider Listing 8-11 for the component's TypeScript code. It includes change event handlers and click event handlers in the comments form.

Listing 8-11. Comments Form Handlers

```
01: import { Component, OnInit } from '@angular/core';
02: import { MatSnackBar } from '@angular/material/snack-bar';
03: import { GamesService } from 'src/app/common/games.
    service';
04:
05: @Component({
06:   selector: 'wade-game-details',
07:   templateUrl: './game-details.component.html',
08:   styleUrls: ['./game-details.component.sass']
09: })
10: export class GameDetailsComponent implements OnInit {
11:
12:   name: string = "";
13:   title: string = "";
14:   comments: string = "";
15:
16:   constructor( private gamesSvc: GamesService,
17:     private snackbar: MatSnackBar) { }
18:
19:   updateName(event: any){
20:     this.name = event.target.value;
21:   }
22:
23:   updateTitle(event: any){
24:     this.title = event.target.value;
25:   }
26:
```

```
27:   updateComments(event: any){
28:     this.comments = event.target.value;
29:   }
30:
31:   submitComment(){
32:       this
33:         .gamesSvc
34:         .addComments(this.title, this.name, this.comments,
            this.game.gameId)
35:         .subscribe( (res) => {
36:           this.snackbar.open('Add comment successful',
              'Close');
37:         });
38:   }
39:
40: }
```

Consider the following explanation:

- See lines 19 to 29 for the functions `updateName()`, `updateTitle()`, and `updateComments()`. Remember, they are invoked on change events in the form fields. Notice the function definition uses `event.target.value`. The event's target refers to the form field (DOM element). The value returns the data typed in by the user.
- The values are set to the class variables `name` (for user name), `title` (for comment title), and `comments` (for comment description).

- The submit button's click event is data bound to the `submitComment()` function. See lines 32 to 38. Notice that it invokes the `addComments()` function on the service `GameService` instance (`gameSvc`). On line 16, `GameService` is injected to be used in the component.
- Notice the service function requires a list of parameters including the username, title, and description. The values captured earlier (with the change event handler) are passed into the service function.
- `addComments()` invokes the server-side HTTP service. If the add comment action is successful, the success callback for the observable is invoked. It shows a success message, providing the feedback on the add comment action.

Listing 8-12 shows the `GameService` implementation. The listing focuses on the `addComments()` action.

Listing 8-12. GameService Implementation

```
01: import { Injectable } from '@angular/core';
02: import { HttpClient } from '@angular/common/http';
03: import { environment } from 'src/environments/environment';
04:
05:
06: @Injectable({
07:   providedIn: 'root'
08: })
09: export class GamesService {
10:
11:   constructor(private httpClient: HttpClient) { }
12:
```

```
13:   addComments(title: string, userName: string, comments:
      string, gameId: number, timeCommented = new Date()){
14:     return this
15:       .httpClient
16:       .post(environment.commentsServiceUrl, [{
17:         title,
18:         userName,
19:         timeCommented,
20:         comments,
21:         gameId
22:       }]);
23:   }
24: }
```

Consider the following explanation:

- Line 11 injects the `HttpClient` service. This is an out-of-the-box service provided by Angular to make HTTP calls.
- In lines 14 and 22, the functions use the `HttpClient` instance called `httpClient.` This invokes the remote HTTP service.
- In line 16, notice you are making an HTTP POST call. The first parameter is the service URL. Considering that the URL is a configuration artifact, it is updated in the environment file.
- The second parameter is the POST method's request body. See Figure 8-5 to understand how the values translate to the request body on the network.

Note One comments URL is used for two actions, retrieving and creating comments. A RESTful service uses the HTTP method GET for retrieval. For a create action, the same URL is used with the POST HTTP method.

Updates to Mock HTTP Services

The new components need additional data and features from the remote HTTP services. This section details changes to the mock services. In a real-world application, such services and features are developed by querying and updating the database. As it is out of scope for the book, we will develop mock services.

Filtering Game Details by ID

The game details component needs one game detail at a time. Remember, in an earlier chapter, you developed a service to return all the board games. This section details how to retrieve game data by ID.

Remember, we use `mock-data/board-games.js` for all the board games related endpoints. Add a new endpoint, which retrieves a game by ID. Name it `/gameById`, as shown in Listing 8-13.

Listing 8-13. Filter Game by an ID

```
1: var express = require('express');
2: var router = express.Router();
3: var dataset = require('../data/board-games.json');
4:
5: router.get('/gameById', function(req, res, next){
```

```
6:     res.setHeader('Content-Type', 'application/json');
7:     res.send(dataset
           .boardGames
           .find( i => +i.gameId === +req.query.gameId));
8: });
```

Consider the following explanation:

- Line 7 filters board games by game ID. The statement `dataset .boardGames.find( i => +i.gameId === +req.query.gameId)` returns the game details with the given ID. Typically, we expect a single game with an ID. In a different scenario, if you anticipate more than one result, use the `filter()` function instead of `find()`.
- The results from the `find()` function are passed in as a parameter to the `send()` function on the response object (variable name `res`). This returns the results to the client (browser). See Figure 8-3.
    - See line 3. The mock service retrieves a list of board games from a mock JSON object in the data directory.
    - See line 5. The HTTP method for this filter endpoint is GET.
    - See line 6. The response content type is set to JSON, which is a ready-to-use format for the Angular services and components.

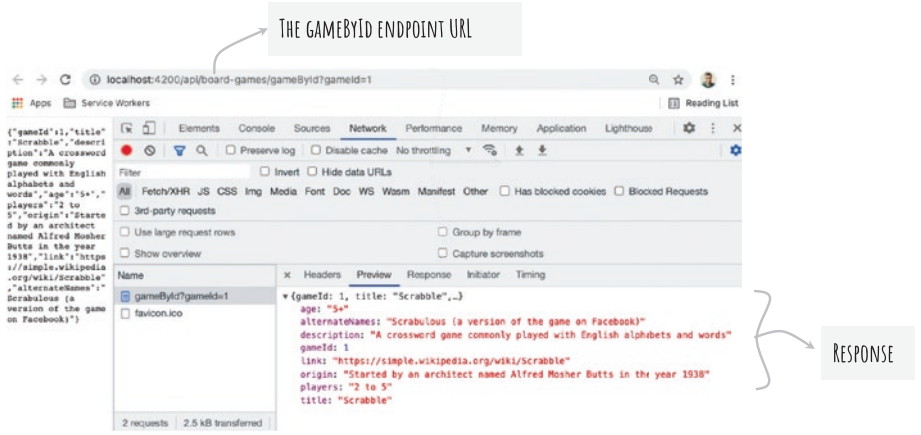

Figure 8-3. *Filtering the game by an ID*

Note Notice the + symbol on line 7. This is a way in JavaScript to type case a string to a number.

Retrieving Comments

The game details page lists comments, as shown in Figure 8-2. Notice the list of comments below the game description. This section details how to create a mock server-side service to retrieve comments.

The service returns comments on a given game. It uses a query param for the game ID. Consider an example URL, `http://localhost:3000/api/board-games/comments?gameId=1`. See Figure 8-4.

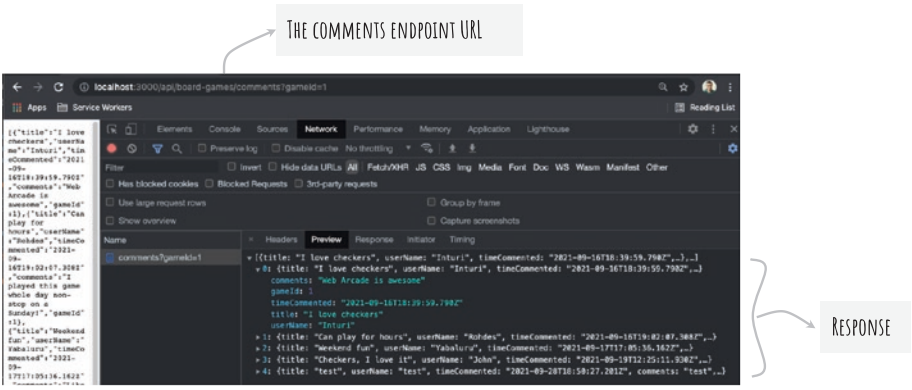

Figure 8-4. *The comments endpoint*

A real-world service integrates with a database to efficiently store and query data. As mentioned earlier, it is a mock service. Hence, it reads comments from a file. Consider Listing 8-14 in `mock_sevices/routes/board-games.js`. The `board-games.js` file is appropriate as it includes all the endpoints related to board games. The comments are on a board game.

Listing 8-14. An Endpoint to Retrieve Comments

```
var fs = require('fs');
var express = require('express');
var router = express.Router();

router.get('/comments', function(req, res){
    fs.readFile("data/comments.json", {encoding:
    'utf-8'},  function(err, data){
        let comments = [];
        if(err){
            return console.log("error reading from the
            file", err);
        }
        res.setHeader('Content-Type', 'application/json');
```

```
12:         comments = JSON.parse(data);
13:         comments = Object.values(comments).filter( i => {
14:
15:             return +i.gameId === +req.query.gameId
16:         });
17:         res.send(comments);
18:     });
19: });
```

Consider the following explanation:

- Line 5 creates an endpoint, which responds to an HTTP method, GET. The `get()` function on the express router instance enables you to create the endpoint.
- Line 6 retrieves a current list of comments from a file on disk, `data/comments.json`.
- The `readFile()` function on the `fs` module (for "file system") is asynchronous. You provide a callback function, which is invoked as the API successfully reads a file or errors out. In Listing 8-14, notice the callback function between lines 6 and 18.
- While the first parameter, `err`, represents an error, the second parameter, `data`, contains the contents of the file. See lines 8 to 10. If an error is returned, it is logged and returns the control out of the function.
- Assuming there are no errors reading the file, the file contents include an entire list of comments belonging to all the games in the system. The service is expected to return a comment only for the given game. Hence, you filter the comments by game ID. See the code on lines 13 to 16, which creates a filter. The `filter()`

function is defined on JavaScript array objects. The predicate on line 15 tests each item in the array. The comments with the given game ID are returned.

- See line 17, which responds with the filtered comments to the client (e.g., browser).

Adding Comments

The game details page allows users to comment, as shown in Figure 8-2. The form has fields for a username, a title, and a detailed comment. This section details how to create a mock server-side service to save comments.

Adding a comment is done through a create action. You are creating a new comment entity. Remember, the POST method is appropriate to create entities. A POST method has a request body, which includes a list of comments created by the Angular application (typed in by an user). See Figure 8-5.

Figure 8-5. *The create comments endpoint*

Consider Listing 8-15 in `mock_sevices/routes/board-games.js`. The `board-games.js` file is appropriate as it includes all the endpoints related to board games. Users are commenting on board games.

Listing 8-15. A POST Endpoint to Create Comments

```
var fs = require('fs');
var express = require('express');
var router = express.Router();

router.post('/comments', function(req, res){
    let commentsData = [];
    try{
        fs.readFile("data/comments.json", {encoding:
        'utf-8'},  function(err, data){
            if(err){
                return console.log("error reading from the
                file", err);
            }
            commentsData = commentsData.concat(JSON.
            parse(data));
            commentsData = commentsData.concat(req.body);

            fs.writeFile("data/comments.json", JSON.
            stringify(commentsData), function(err){
                if(err){
                    return console.log("error writing to
                    file", err);
                }
                console.log("file saved");
            });
        });
```

```
22:         res.send({
23:             status: 'success'
24:         });
25:     }catch(err){
26:         console.log('err2', err);
27:         res.sendStatus(200);
28:     }
29: });
```

Consider the following explanation:

- Line 5 creates an endpoint, which responds to the HTTP POST method. The `post()` function on the express router instance enables you to create the endpoint.
- The endpoint needs to append the new comments to the current list of comments. The file `data/comments.json` has an array of the current list of comments.
- The `readFile()` function on the `fs` module is asynchronous. You provide a callback function, which is invoked as the API successfully reads a file or errors out. In Listing 8-15, notice the callback function on lines 8 to 21.
- While the first parameter, `err`, represents an error, the second parameter, `data`, has contents of the file. See lines 9 to 11. If an error is returned, it is logged and returns the control out of the function.
- Assuming there is no error reading the file, line 12 adds comments in the file to a local variable called `commentsData`.

- Next, line 13 concatenates a new list of comments on the request object to the `commentsData` variable. As mentioned earlier, the POST method has a request body. It includes a list of comments provided by the Angular application.
- The consolidated list of comments are written back to the file. See line 15, which writes the entire list of comments to the file.

Summary

This chapter builds on the Web Arcade use case. It is crucial for understanding the offline function we will build in the next chapter. So far, while working with data, you performed data retrieval and caching. This chapter established a use case for the create, update, and delete scenarios.

EXERCISE

- The game details page just shows a title and a description for a board game. However, the mock service and the TypeScript interface include many additional fields including origin, alternate names, recommended number of players, etc. Include the additional fields on the game details page.
- The add comment functionality shows a Snackbar component message if the action is successful (see Listing 8-11, line 36). The sample does not show a message for an error. Update the code sample to show a Snackbar component alert for errors.
- Implement a back button on the game details page to navigate back to the list screen.

CHAPTER 9

Creating Data Offline

Earlier in the book, we started integrating IndexedDB in an Angular application. We also established a use case for creating data, namely, user comments, offline. Imagine that a user is on a game details page and attempts to add a comment. However, the user loses network access. The Web Arcade application is resilient because it saves the comments on the client device/browser temporarily. When the user is back online, the application synchronizes the comments online.

This chapter elaborates on how to create data offline. It begins with instructions to identify if the application is online or offline. You use the status to determine how to reach server-side services or use the local IndexedDB store. Next, the chapter details how to add a comment to IndexedDB when offline. It details how to provide feedback to the user that the application is offline but the data is temporarily saved.

Then, the chapter covers how to synchronize offline comments with the server-side services once the application is back online. Remember, the server-side database and the services are the source of truth for the data. IndexedDB is temporary and provides a seamless experience to the user.

Adding Comments Online and Offline

The previous chapter described how to add a comment. The submit action calls the server-side HTTP endpoint. If the device is offline and has lost network connectivity, a typical web application shows an error. Once

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4_9

online, users might have to retry the operation. As mentioned earlier, Web Arcade uses IndexedDB, persists data temporarily, and synchronizes when the remote service is available.

Identifying the Online/Offline Status with a Getter

To identify whether the device (and the browser) is online, use the JavaScript API on the navigator object. It is a read-only property on the `window` object. The field `onLine` returns the current status, which is true if online and false if offline.

The developer tools on Google Chrome provide an option to throttle network speeds. This helps applications evaluate their performance and user experience. See Figure 9-1. The tools print the `onLine` field value on the navigator object. Notice the browser window is throttled offline.

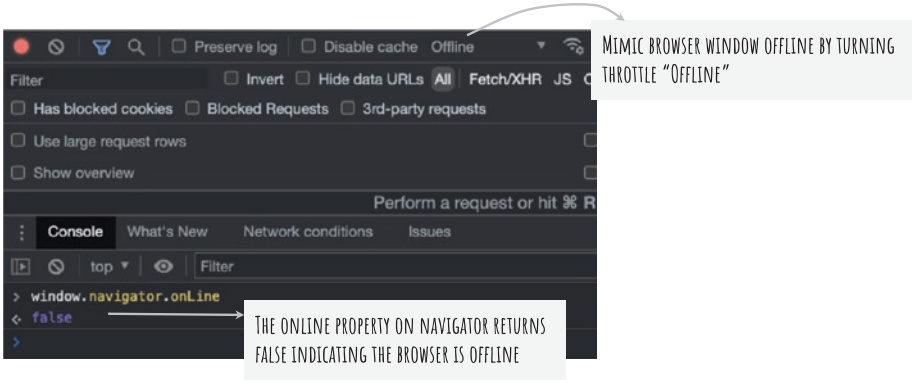

Figure 9-1. *Google Chrome Developer Tools, printing the onLine status on the console*

Note You can run a similar command on a browser of your choice. Figure 9-1 shows Google Chrome, which was chosen arbitrarily.

Remember, we created a service called `IdbStorageAccessService` to encapsulate access to IndexedDB. The online/offline status determines the components that can access IndexedDB. Hence, you should include lines of code to determine the online/offline status in the service.

Inject the `Window` service into `IdbStorageAccessService`, as shown in Listing 9-1, line 3.

Listing 9-1. Inject the Window Service

```
1: @Injectable()
2: export class IdbStorageAccessService {
3:   constructor(
         private windowObj: Window) {
4:     // this.create();
5:   }
6: }
```

Ensure the `Window` service is provided. See Listing 9-2, lines 10 and 15, in `AppModule` for the Web Arcade application. You provide the `Window` service with a global variable, `window`, as shown in line 14. This provides access to useful properties such as `document`, `navigator`, etc.

Listing 9-2. Provide the Window Service

```
01: @NgModule({
02:     declarations: [
03:       AppComponent,
04:       // ...
05:     ],
06:     imports: [
07:       BrowserModule,
08:       // ...
09:     ],
```

```
10:     providers: [
11:       IdbStorageAccessService,
12:       {
13:         provide: Window,
14:         useValue: window
15:       }
16:     ],
17:     bootstrap: [AppComponent]
18:   })
19:   export class AppModule { }
```

Create a getter function called `IsOnline` in `IdbStorageAccessService`. A service instance may use the `IsOnline` field to get the status of the browser. The code is abstracted in the service. See Listing 9-3.

Listing 9-3. IsOnline Getter as Part of IdbStorageAccessService

```
1: get IsOnline(){
2:     return this.windowObj.navigator.onLine;
3: }
```

Adding Online/Offline Event Listeners

It is possible that you will encounter a scenario where an action needs to be performed when the application goes online or offline. The window object (and hence the window service) provides the events `online` and `offline`. Add these events to `IdbStorageAccessService` at the time of initialization. The event handler callback function is invoked any time the event occurs.

Listing 9-4 prints a message on the browser console including the event data. You can perform an action when an event is triggered. See specifically lines 8 to 11 and lines 13 to 16.

Listing 9-4. Online and Offline Events

```
@Injectable()
export class IdbStorageAccessService {

  constructor(private windowObj: Window) {
  }

  init() {
    this.windowObj.addEventListener("online", (event) => {
      console.log("application is online", event);
      // Perform an action when online
    });

    this.windowObj.addEventListener('offline', (event)=> {
        console.log("application is offline", event)
        // Perform an action when offline
    });
  }
}
```

Figure 9-2 shows the results.

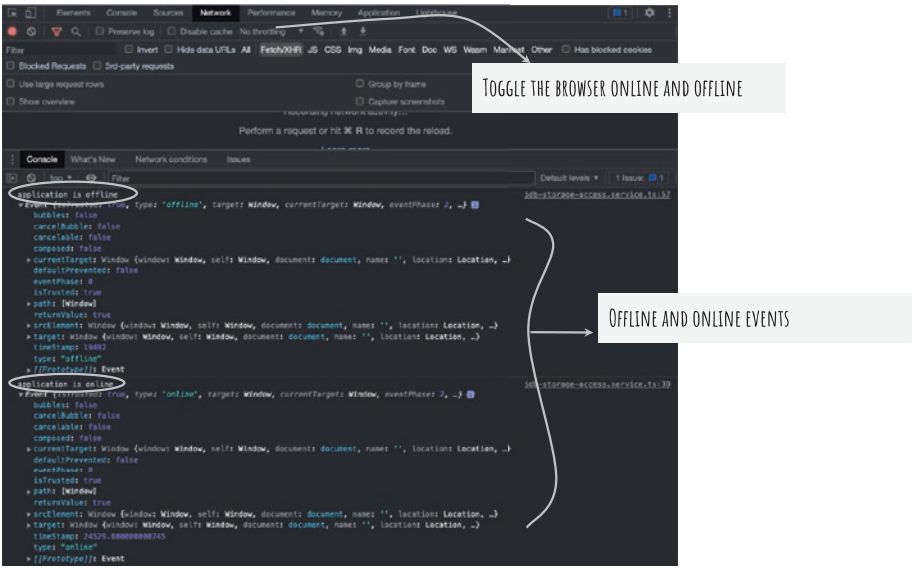

Figure 9-2. *Online and offline events*

Adding Comments to IndexedDB

Remember, when needed, we intend to cache comments in IndexedDB. Considering `IdbStorageAccessService` abstracts the task of accessing the database from the rest of the application, augment the service and add a function that caches comments in IndexedDB. But before we do that, Listing 9-5 shows a quick recap of the service so far.

Listing 9-5. IdbStorageAccessService

```
01: @Injectable()
02: export class IdbStorageAccessService {
03:
04:   idb = this.windowObj.indexedDB;
05:
06:   constructor(private windowObj: Window) {
```

```
07:   }
08:
09:   init() {
10:
11:     let request = this.idb.open('web-arcade', 1);
12:
13:     request.onsuccess = (evt:any) => {
14:       console.log("Open Success", evt);
15:
16:     };
17:
18:     request.onerror = (error: any) => {
19:       console.error("Error opening IndexedDB", error);
20:     }
21:
22:     request.onupgradeneeded = function(event: any){
23:       let dbRef = event.target.result;
24:       let objStore = dbRef
25:         .createObjectStore("gameComments", {
          autoIncrement: true })
26:
27:       let idxCommentId = objStore.
          createIndex('IdxCommentId', 'commentId',
          {unique: true})
28:     };
29:
30:     this.windowObj.addEventListener("online", (event) => {
31:       console.log("application is online", event);
32:       // Peform an action when online
33:     });
34:
```

```
35:     this.windowObj.addEventListener('offline', (event) => {
36:         console.log("application is offline", event)
37:         // Perform an action when offline
38:     });
39:
40:   }
41: }
```

So far, the service creates a reference to IndexedDB, opens a new database, and creates an object store and an index. Consider the following detailed explanation for Listing 9-5:

- In line 4, an IndexedDB reference is set on a class variable, namely, `idb`.
- Next, in the `init()` function (which initializes the service and appears on line 11), run the `open()` function on the `idb` object. It returns an object of the `IDBOpenDBRequest` object.
- If this is the first time a user has opened the application on a browser, it creates a new database.
    a. The first parameter is the name of the database, `web-arcade`.
    b. The second parameter (a value of 1) specifies a version for the database. As you can imagine, new updates to the application cause changes to the IndexedDB structure. The IndexedDB API enables you to upgrade the database as the version changes.

  For a returning user, the database was already created and available on a browser. The `open()` function attempts to open the database.

1. The IndexedDB APIs are asynchronous. The open action does not complete in line 11. You provide a function callback for the success and failure scenarios. They are invoked as a result of the open action.

    a. Notice the `onsuccess()` function callback on lines 13 to 16, which is invoked if the open database action is successful.

    b. The `onerror()` function callback on lines 18 to 20 is invoked if the open database action fails.

    c. The open function call returns `IDBOpenDBRequest`. The previous callback functions `onsuccess` and `onerror` are provided to this returned object.

2. See the code on lines 22 to 28, where `onupgradeneeded` is triggered after creating or opening IndexedDB. You provide a callback function, which is invoked by the browser when this event occurs. What is the significance of the `onupgradeneeded` event?

    a. In the case of a new database, the callback function is a good place to create object stores. In the current use case, you create an object store to save game comments. You name it `gameComments`.

    b. For a pre-existing database, if an upgrade is required, you may perform design changes here.

3. Finally, on lines 30 to 38, see the function callback for online and offline events when the browser goes online/offline.

The Angular service `IdbStorageAccessService` needs a reference to the `web-arcade` database. You use it to create a transaction. With

IndexedDB, you need a transaction to perform create, retrieve, update, and delete (CRUD) operations. The statement on line 11, the `this.idb.open('web-arcade',1)` function call, attempts to open a database, namely, `web-arcade`. If it's successful, you can access the database reference as part of the `onsuccess()` function callback. Consider Listing 9-6.

Listing 9-6. Access the web-arcade Database Reference

```
01: @Injectable()
02: export class IdbStorageAccessService {
03:
04:   idb = this.windowObj.indexedDB;
05:   indexedDb: IDBDatabase;
06:   init() {
07:
08:     let request = this.idb.open('web-arcade', 1);
09:
10:     request.onsuccess = (evt:any) => {
11:       console.log("Open Success", evt);
12:       this.indexedDb = evt?.target?.result;
13:     };
14:   }
15: }
```

Consider the following explanation:

- See line 5. `indexedDB` is a class variable accessible across the service.
- A value is assigned on the successful opening of the `web-arcade` database, as shown on line 12. The database instance is available in the `result` variable on the `target` property of the `event` object (`event.target.result`).

Next, add a function to create comments in IndexedDB. This creates an IndexedDB transaction, accesses the object store, and adds a new record. Consider Listing 9-7.

Listing 9-7. Add a New Record in IndexedDB

```
addComment(title: string, userName: string, comments:
    string, gameId: number, timeCommented = new Date()){
    let transaction = this.indexedDb
      .transaction("gameComments", "readwrite");

    transaction.objectStore("gameComments")
      .add(
        {
          title,
          userName,
          timeCommented,
          comments,
          gameId,
          commentId: new Date().getTime()
        }
      )


    transaction.oncomplete = (evt) => console.log("add
    comment transaction complete", evt);
    transaction.onerror = (err) => console.log("add
    comment transaction errored out", err);

  }
```

Consider the following explanation:

- First, create a new transaction with the class variable `indexedDB` (created in Listing 9-6). See the transaction function on line 3. It takes two parameters:
  a. One or more object stores in which a transaction needs to be created. In this case, you create a transaction on an object store `gameComments`.
  b. Specify the transaction mode, `readwrite`. IndexedDB supports three modes of transactions, namely, `readonly`, `readwrite`, and `versionchange`. As you can imagine, `readonly` helps with retrieve operations, and `readwrite` helps with create/update/delete operations. However, `versionchange` mode helps create and delete object stores on an IndexedDB.
- Next, perform the add record action on IndexedDB. Use the transaction object to access the object store on which the add action needs to be performed. See line 5, which uses the `objectStore` function to access the object `store()`.
- See lines 6 and 15. You store a JavaScript object including the fields for the comment title, username, time commented, comment description, game ID on which the comment was added, and a unique comment ID. To ensure uniqueness, you use a time value. You may use any unique value.
- As you have seen with IndexedDB, the database actions are asynchronous. The `add()` function does not immediately add a record. It eventually invokes a success or error callback function. A transaction has the following callback functions:

a. `oncomplete`: This is invoked on success. See line 18. It prints the status on the console.

b. `onerror`: This is invoked on error. See line 19.

Figure 9-3 shows a record in IndexedDB.

Figure 9-3. *A new record in IndexedDB*

The User Experience of Adding Comments

Remember from the previous chapter that `UserDetailsComponent` adds a comment by calling the `GameService` function named `addComments`. This invokes the server-side POST call to add a comment. If the application is offline, it will error out. You show an error feedback to the user and request the user to retry.

In this chapter, you have done the background work to cache comments in IndexedDB, if the browser is offline. Next, update the component to check if the application is online or offline and invoke the respective service function. Consider the code snippet in Listing 9-8, which comes from `GameDetailsComponent` (`app/components/game-details/game-details.component.ts`).

Listing 9-8. Add a Comment in the Game Details Component

```
01: @Component({ /* ... */ })
02: export class GameDetailsComponent implements OnInit {
03:
04:     constructor(private idbSvc: IdbStorageAccessService,
05:         private gamesSvc: GamesService,
07:         private snackbar: MatSnackBar,
08:         private router: ActivatedRoute) { }
09:
10:     submitComment() {
11:         if (this.idbSvc.IsOnline) {
12:             this
13:                 .gamesSvc
14:                 .addComments(/* provide comment fields */)
15:                 .subscribe((res) => {
16:
17:                     this.snackbar.open('Add comment
                        successful', 'Close');
18:                 });
19:         } else {
20:             this.idbSvc.addComment(this.title, this.name,
                this.comments, this.game.gameId);
21:             this.snackbar.open('Application is offline.
                We saved it temporarily', 'Close');
22:         }
23:     }
24: }
```

Consider the following explanation:

- At the beginning, inject `IdbStorageAccessService`. See line 4. The service instance is named `idbSvc`.

- Line 11 checks if the application is online. Notice that you use the `IsOnline` getter created in Listing 9-3.
  a. If true, continue to call the game service function, `addComments()`. It invokes the server-side service.
  b. If offline, use the `IdbStorageAccessService` function `addComment()`, which adds the comment to IndexedDB. See the implementation in Listing 9-7.
- Notice on line 21 that you show a Snackbar component message that the application is offline. Figure 9-4 shows the result.

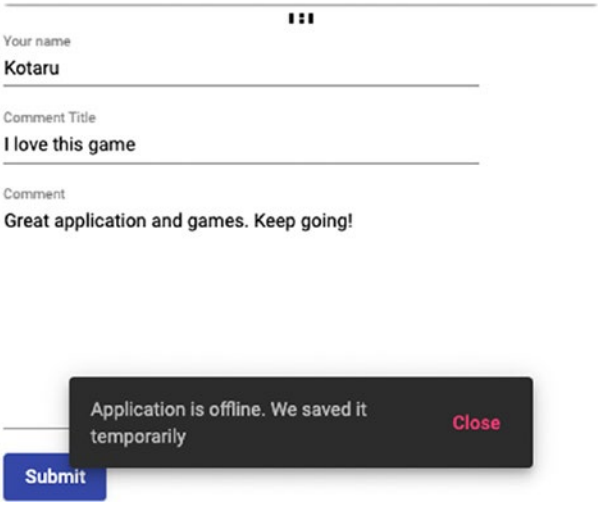

Figure 9-4. *Snackbar component alert indicating application is offline*

Synchronizing Offline Comments with the Servers

When the application is offline, you cache the comments within the browser in persistent storage using IndexedDB. Eventually, once the application is back online, when the user launches the application again, the comments need to be synchronized with the server side. This section details the implementation to identify that the application is online and synchronize the comment records.

The two events `online` and `offline` on `window` objects are triggered when the browser gains or loses connectivity. The `IdbStorageAccessService` service includes event handlers for the `online` and `offline` events. See Listing 9-4.

Next, update the online event handler. Consider the following steps to synchronize the data with the server-side databases. When the application is back online, you do the following:

1. Retrieve all the cached comments from IndexedDB.
2. Invoke a server-side HTTP service, which updates the primary database for the user comments.
3. Finally, clear the cache. Delete comments synchronized with the remote service.

Let's begin with the first step in the previous list, retrieving all the cached comments from IndexedDB. The following section details various options and the available API to retrieve data from IndexedDB.

Retrieving Data from IndexedDB

IndexedDB provides the following API for retrieving data:

`getAll()`: Retrieves all records in the object store

As mentioned earlier, the CRUD operations run in the scope of a transaction. Hence, you will create a read-only transaction (considering it is a data retrieval operation) on the object store. Call the `getAll()` API, which returns `IDBRequest`, as shown in Listing 9-9.

On the `IDBRequest` object, provide the `onsuccess` and `onerror` callback function definitions. As you know, almost all IndexedDB operations are asynchronous. The data retrieval with `getAll()` does not happen immediately. It calls back the provided callback function.

Listing 9-9. Using getAll()

```
1: let request = this.indexedDb
2: .transaction("gameComments", "readonly")
3: .objectStore("gameComments")
4: .getAll();
5:
6: request.onsuccess = resObject => console.log('getAll
   results', resObject);
7: request.onerror = err => console.error('Error reading
   data', err);
```

See Figure 9-5 for the result. Notice the success handler on line 6 in Listing 9-9. The result variable is named `resObject`. Result records are available on a `result` object on the `target` property on the `resObject` (`resObject.target.result`).

Figure 9-5. *The getAll() result*

`get(key)`: Retrieves a record by key. The `get()` function is run on an object store.

Similar to `getAll()`, create a read-only transaction for `get()` on the object store. The `get()` API returns `IDBRequest`, as shown in Listing 9-10.

Rest of the code handling the result or an error is the same. On the `IDBRequest` object, provide the `onsuccess` and `onerror` callback function definitions. As you know, almost all IndexedDB operations are asynchronous. The data retrieval with `get()` does not happen immediately. It calls back the provided callback function.

Listing 9-10. Using get()

```
let request = this.indexedDb
  .transaction("gameComments", "readonly")
  .objectStore("gameComments")
  .get(30);

```

```
6: request.onsuccess = resultObject => console.log('get()
   results', resultObject);
7: request.onerror = err => console.error('Error reading
   data', err);
```

Notice the success handler on line 6 in Listing 9-10. The result variable is named `resultObject`. Result records are available on a result object on the `target` property on the `resultObject` (`resultObject.target.result`).

> `openCursor()`: A cursor allows you to iterate through the results. It lets you act on one record at a time. We chose this option for the comments use case. It provides flexibility to transform the data format as and when you read from IndexedDB. The other two APIs, `getAll()` and `get()`, require an additional code loop to transform the data.

As mentioned earlier, the CRUD operations run in the scope of a transaction. Hence, you will create a read-only transaction (considering it is a data retrieval operation) on the object store. Call the `openCursor()` API, which returns `IDBRequest`.

Again, code handling the result or an error remains the same. On the `IDBRequest` object, provide the `onsuccess` and `onerror` callback function definitions. The data retrieval with `openCursor()` is asynchronous, which invokes above mentioned `onsuccess` or `onerror` callback functions.

Create a new private function to retrieve the cached comment records. Provide an arbitrary name of `getAllCachedComments()`. Add the private function shown in Listing 9-11 in `IdbStorageAccessService`.

Listing 9-11. Retrieve Cached Comments from IndexedDB

```
01: private getAllCachedComments() {
02:     return new Promise(
03:       (resolve, reject) => {
```

```
04:         let results: Array<{
05:           key: number,
06:           value: any
07:         }> = [];
08:
09:         let query = this.indexedDb
10:           .transaction("gameComments", "readonly")
11:           .objectStore("gameComments")
12:           .openCursor();
13:
14:           query.onsuccess = function (evt: any) {
15:
16:             let gameCommentsCursor = evt?.target?.result;
17:             if(gameCommentsCursor){
18:               results.push({
19:                 key: gameCommentsCursor.primaryKey,
20:                 value: gameCommentsCursor.value
21:               });
22:               gameCommentsCursor.continue();
23:             } else {
24:               resolve(results);
25:             }
26:           };
27:
28:           query.onerror = function (error: any){
29:             reject(error);
30:           };
31:
32:       });
33:   }
```

Consider the following explanation:

- The function creates and returns a promise. See line 2. Considering the data retrieval is asynchronous, you cannot instantly return comment records from the `getAllCachedComments()` function. The promise is resolved once the cursor finishes retrieving data from IndexedDB.
- Lines 9 and 12 create a read-only transaction, access object store `gameComments`, and open a cursor. This statement returns an `IDBRequest` object, which is assigned to a local variable `query`.
- Remember, the `onsuccess` callback is invoked if the cursor is able to retrieve data from the object store. Otherwise, the `onerror` callback is invoked (lines 28 and 30).
- See `onsuccesscallback()` defined in lines 14 to 26.
- Access the results at `event.target.result`. See line 16.

Note The `?.` syntax in `evt?.target?.result` performs a null check. If a property is undefined, it will return null, instead of throwing an error and crashing the entire function workflow. The previous statement may return the results or null.

- If the results are defined, transform the data to key-value pair format. Add the object to a local variable called `result`.

- Remember, the cursor works on a single comment record at a time (unlike `get()` and `getAll()`). To move the cursor to the next record, call the continue function on the query object. Remember, the `query` object is an `IDBRequest` object returned by `openCursor()`.
- The `if` condition on line 17 results in a true value until all the records in the cursor are exhausted.
- When false, as the entire dataset (comment records) is retrieved and added to the local variable `result`, resolve the promise. The calling function uses the results successfully resolved from `getAllCachedComments()`.

This completes the first step among the three described earlier, listed here:

1. **Retrieve all the cached comments from the IndexedDB.**

   Next, let's proceed to the other two steps:

2. Invoke a server-side HTTP service, which updates the primary database for the user comments.
3. Finally, clear the cache. Delete comments synchronized with the remote service.

Bulking Updating Comments on the Server Side

A user might have added multiple comments while the application was offline. It is advisable to upload all the comments in a single call. The server-side HTTP POST endpoint `/comments` accepts an array of comments.

Remember, the Angular service `GameService` (`src/app/common/game.service.ts`) encapsulates all the game-related service calls. Add a new function, which accepts an array of comments and makes an HTTP POST call. Similar to earlier service calls, the new function uses an `HttpClient` object to make a post call. See Listing 9-12 for the new function `addBulkComments` (the function name is arbitrary). See lines 9 and 18.

Listing 9-12. Add Bulk Comments

```
02: @Injectable({
03:   providedIn: 'root'
04: })
05: export class GamesService {
06:
07:   constructor(private httpClient: HttpClient) { }
08:
09:   addBulkComments(comments: Array<{title: string,
10:     userName: string,
11:     comments: string,
12:     gameId: number,
13:     timeCommented: Date}>){
14:     return this
15:       .httpClient
16:       .post(environment.commentsServiceUrl, comments);
17:
18:   }
19: }
```

Note The function `addBulkComments()` uses anonymous data type as a parameter. The comments variable is of type `Array<{title: string, userName: string, comments: string, gameId: number, timeCommented: Date}>`. The highlighted type is without a name. You may use this technique for one-off data types.

You may choose to create a new entity and use it.

The service function is now available, but it has not been called yet. However, you have the service function available to bulk update the cached comments. Before we start using this function, consider adding a function to delete.

This completes the second step as well. Now you have the code to retrieve cached comments from IndexedDB and call a server-side service for synchronizing the offline comments.

1. **Retrieve all the cached comments from the IndexedDB.**
2. **Invoke a server-side HTTP service, which updates the primary database for the user comments**.
3. Finally, clear the cache. Delete comments synchronized with the remote service.

Next, add code to clean up IndexedDB.

Deleting Data from IndexedDB

IndexedDB provides the following API for deleting data from IndexedDB:

`delete()`: Removes records in the object store. This selects the record to be deleted by its record ID.

As mentioned earlier, the CRUD operations run in the scope of a transaction. Hence, you will create a read-write transaction on the object store. Call the `getAll()` API, which returns `IDBRequest`.

On the `IDBRequest` object, provide the `onsuccess` and `onerror` callback function definitions. As mentioned earlier, almost all IndexedDB operations are asynchronous. The delete operation does not happen immediately. It calls back the provided callback function, as shown in Listing 9-13. Notice that it returns a promise. The promise is resolved if the delete action is successful. See line 10. If it fails, the promise is rejected. See line 14.

Listing 9-13. Using delete()

```
01: deleteComment(recordId: number){
02:     return new Promise( (resolve, reject) => {
03:       let deleteQuery = this.indexedDb
04:             .transaction("gameComments", "readwrite")
05:             .objectStore("gameComments")
06:             .delete(recordId);
07:
08:       deleteQuery.onsuccess = (evt) => {
09:         console.log("delete successful", evt);
10:         resolve(true);
11:       }
12:       deleteQuery.onerror = (error) => {
13:         console.log("delete successful", error);
14:         reject(error);
15:       }
16:     });
17:   }
```

Include the previous function in `IdbStorageAccessService`. Remember, this service encapsulates all actions related to IndexedDB. Now, you have the code for all three steps described for synchronizing offline comments.

1. **Retrieve all the cached comments from the IndexedDB.**
2. **Invoke a server-side HTTP service, which updates the primary database for the user comments.**
3. **Finally, clear the cache. Delete comments synchronized with the remote service.**

Notice that these service functions are available, but they are not yet triggered when the application comes back online. Earlier in the chapter, the service `IdbStorageAccessService` includes an event handler for the `online` event. It is called when the application comes back online. Update this event handler to synchronize offline comments. Consider Listing 9-14 to be updated in `IdbStorageAccessService`.

Listing 9-14. The Online Event Handler

```
this.windowObj.addEventListener("online", (event) => {
    this.getAllCachedComments()
    .then((result: any) => {
      if (Array.isArray(result)) {
        let r = this.transformCommentDataStructure(result);
        this
          .gameSvc
          .addBulkComments(r)
          .subscribe(
            () => {
              this.deleteSynchronizedComments(result);
            },
```

```
13:             () => ({/* error handler  */})
14:           );
15:       }
16:     });
17:   });
```

Consider the following explanation:

- First, you retrieve all cached comments. See line 2, which calls the `getAllCachedComments()` service function. See Listing 9-11 to review retrieving cached comments from IndexedDB.
- The function returns a promise. When the promise is resolved, you have access to the comment records from IndexedDB. You use this data to add comments in the back end, synchronizing server-side services and databases.
- Before you call the server-side service, transform the comment record to the request object structure. You loop through all the comments and change the field names as required by the server-side service.
    a. Listing 9-15 defines a private function called `transformCommentDataStructure()`. Notice the `forEach()` on the array of comments obtained from the IndexedDB object store. The comments are transformed and added to a local variable, `comments`. This is returned at the end of the function.
- Next call the `GameService` function `addBulkComments()`, which in turn calls the server-side service. To review the `addBulkComments()` function, see Listing 9-12.

- Remember, the function `addBulkComments()` returns an observable. You subscribe to the observable, which has handlers for success and failure. The success handler indicates the comments are added/synchronized with the server side. Hence, you can now delete cached comments in IndexedDB.
- Invoke a private function `deleteSynchronizedComments()` defined as part of the service `IdbStorageAccessService`. It loops through each comment record and deletes the comments from the local database. See Listing 9-16 for the `deleteSynchronizedComments()` function definition.
    a. Notice that the `forEach` loop uses an anonymous type with a key-value pair. See line 3 (`r: {key: number; value: any}`). It defines the expected structure for the comments data.
    b. `deleteComment()` deletes each record by its ID. To review the function again, see Listing 9-13.

Listing 9-15. Transform Comments Data

```
01: private transformCommentDataStructure(result: Array<any>){
02:     let comments: any[] = [];
03:     result?.forEach( (r: {key: number; value: any}) => {
04:         comments.push({
05:           title: r.value.title,
06:           userName: r.value.userName,
07:           comments: r.value.comments,
08:           gameId: r.value.gameId,
09:           timeCommented: new Date()
```

```
10:         });
11:     });
12:     return comments ;
13: }
```

Listing 9-16. Delete Synchronized Comments

```
1: private deleteSynchronizedComments(result: Array<any>){
2:     result
3:       ?.forEach( (r: {key: number; value: any}) => this.
         deleteComment(r.key));
4:   }
```

Now, you have synchronized offline comments with the server side. See Listing 9-17, which includes the event handler for handling the online event and private functions that orchestrate the synchronization steps.

Listing 9-17. Synchronized Comments with Online Event Handler

```
01: @Injectable()
02: export class IdbStorageAccessService {
03:
04:   idb = this.windowObj.indexedDB;
05:   indexedDb: IDBDatabase;
06:
07:   constructor(private gameSvc: GamesService, private
      windowObj: Window) {
08:   }
09:
10:   init() {
11:     let request = this.idb
12:       .open('web-arcade', 1);
13:
```

```
14:     request.onsuccess = (evt:any) => {
15:       this.indexedDb = evt?.target?.result;
16:     };
17:
18:     request.onupgradeneeded = function(event: any){
19:         // Create object store for game comments
20:     };
21:
22:     this.windowObj.addEventListener("online", (event) => {
23:       console.log("application is online", event);
24:       this.getAllCachedComments()
25:       .then((result: any) => {
26:         if (Array.isArray(result)) {
27:           let r = this.transformCommentDataStructur
              e(result);
28:           this
29:             .gameSvc
30:             .addBulkComments(r)
31:             .subscribe(
32:               () => {
33:                 this.deleteSynchronizedComments(result);
34:               },
35:               () => ({/* error handler  */})
36:             );
37:         }
38:       });
39:     });
40:
41:     this.windowObj.addEventListener('offline', (event) =>
        console.log("application is offline", event));
42:
```

```
43:   }
44:
45:   private deleteSynchronizedComments(result: Array<any>){
46:     result?.forEach( (r: {key: number; value: any}) => {
47:       this.deleteComment(r.key);
48:     });
49:   }
50:
51:   private transformCommentDataStructure(result:
      Array<any>){
52:       let comments: any[] = [];
53:       result?.forEach( (r: {key: number; value: any}) => {
54:           comments.push({
55:             title: r.value.title,
56:             userName: r.value.userName,
57:             comments: r.value.comments,
58:             gameId: r.value.gameId,
59:             timeCommented: new Date()
60:           });
61:       });
62:       return comments ;
63:   }
64:
65:   deleteComment(recordId: number){
66:     // Code in the listing 9-13
67:   }
68:
69:   private getAllCachedComments() {
70:     // Code in the listing 9-11
71:   }
72:
73: }
```

Updating Data in IndexedDB

IndexedDB provides the following API for updating data in IndexedDB:

`put()`: Updates records in the object store. This selects the record to be updated by its record ID.

As mentioned earlier, the CRUD operations run in the scope of a transaction. Hence, you will create a read-write transaction on the object store. Call the `put()` API, which returns `IDBRequest`.

On the `IDBRequest` object, provide the `onsuccess` and `onerror` callback function definitions. As mentioned, almost all IndexedDB operations are asynchronous. The data retrieval with `put()` does not happen immediately. It calls back the provided callback function, as shown in Listing 9-18.

Listing 9-18. Update Records in IndexedDB

```
updateComment(recordId: number, updatedRecord:
    CommentEntity){
    /* let updatedRecord = {
        commentId: 1633432589457,
        comments: "New comment data",
        gameId: 1,
        timeCommented: 'Tue Oct 05 2021 16:46:29 GMT+0530
        (India Standard Time)',
        title: "New Title",
        userName: "kotaru"
    } */

    let update = this.indexedDb
        .transaction("gameComments", "readwrite")
        .objectStore("gameComments")
```

```
14:         .put(updatedRecord, recordId);
15:
16:     update.onsuccess = (evt) => {
17:         console.log("Update successful", evt);
18:     }
19:     update.onerror = (error) => {
20:         console.log("Update failed", error);
21:     }
22: }
```

Consider the following explanation:

- You create a new function to update comments. Imagine a form that allows users to edit a comment. The previous function can perform this action.

Note The current use case does not include an edit comment use case. The previous function is for demonstrating the `put()` API on IndexedDB.

- Notice the commented lines of code between lines 2 and 9. This provides an arbitrary structure for updated comment data. However, the calling function provides the updated comment in an `updatedRecord` variable.
- See line 14. The put function takes two parameters.
    a. `updatedRecord`: This is the new object to replace the current one.
    b. `recordId`: This identifies the record to be updated by the second parameter, `recordId`.

Summary

This chapter provided an elaborate explanation for adding records to IndexedDB. In the Web Arcade use case with the game details page, the application allows users to add comments offline. The data is temporarily cached in IndexedDB, which is eventually synchronized with server-side services.

EXERCISE

- You have seen how to use the `put()` API to update a record in IndexedDB. Add the ability to edit comments. If the application is offline, provide the ability to temporarily save edits in IndexedDB.
- Notice that the `deleteComment()` function deletes records one at a time. Provide error handling to identify and correct failures.
- Provide a visual indicator when the application is offline. You may choose to change the color of the toolbar and the title.

CHAPTER 10

Dexie.js for IndexedDB

So far, you have seen use cases and implementations for using a database on the client side. You learned about and implemented IndexedDB. The browser API enables you to create a database, performing create/retrieve/update/delete (CRUD) operations. The functions are native to the browser. All the latest versions of the major browsers support IndexedDB. However, arguably, the IndexedDB API is complex. An everyday developer might need a simplified version.

Dexie.js is a wrapper for IndexedDB. It is a simple and easy-to-use library that is installable in your application. It is an open source repository with an Apache 2.0 license. The license allows commercial use, modifications, distribution, patent use, and private use. However, it has limitations with respect to trademark use and has no liability and warranty. Understand the agreement better while using the library for a business application you might be working on.

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4_10

The chapter is an introduction to Dexie.js. It provides an overview of the library within the parameters of the Web Arcade use cases. It begins with instructions to install Dexie.js in the Web Arcade application. Next, it details how to use the library among TypeScript files. You create a new class and a service to encapsulate data access logic to IndexedDB using Dexie.js. The chapter also details how to create transactions, performing CRUD operations on the data. Toward the end, the chapter lists a few additional libraries and wrappers on top of IndexedDB.

Installing Dexie.js

Install the Dexie package.

```
npm i -S dexie
or
yarn add dexie
```

Note The command `npm i -S dexie` is a short form of `npm install --save dexie`.

`-S` or `--save` is an option to add Dexie to the Web Arcade package. An entry will be added to `package.json`. This will ensure that future installs include Dexie.

Yarn does not need this option. It is implicit; it will always add the package to Web Arcade.

Web Arcade Database

Create a TypeScript class encapsulating the Web Arcade IndexedDB connection. Use this class to access the IndexedDB database `web-arcade` with the Dexie API. Run this command to create a TypeScript class:

```
ng generate class common/web-arcade-db
```

Use the class `WebArcadeDb` to specify the IndexedDB database to create and connect. You will also use this class to define object stores, indexes, etc. Add the code shown in Listing 10-1 to the new class `WebArcadeDb`.

Listing 10-1. A TypeScript Class for the Web Arcade DB

```
01: import { Dexie } from 'dexie';
02: import { CommentsEntity } from './board-games-entity';
03:
04: const WEB_ARCADE_DB_NAME = 'web-arcade-dexie';
05: const OBJECT_STORE_GAME_COMMENTS = 'gameComments';
06: export class WebArcadeDb extends Dexie {
07:     comments: Dexie.Table<CommentsEntity>;
08:
09:     constructor() {
10:       super(WEB_ARCADE_DB_NAME);
11:       this.version(1.0).stores({
12:         gameComments: '++IdxCommentId,timeCommented,us
            erName'
13:       });
14:       this.comments = this.table(OBJECT_STORE_GAME_
          COMMENTS);
15:      }
16:   }
```

Consider the following explanation:

- Line 6 creates a new TypeScript class, `WebArcadeDb`. It extends the `Dexie` class, which provides many out-of-the-box features including opening a database, creating stores, etc.
- Notice that the `Dexie` class is imported from the `dexie` ES6 module (part of the Dexie library) on line 1.
- Provide the `web-arcade` database name to the superclass. See line 10, the first line in the constructor. In this code sample, the TypeScript class `WebArcadeDb` is dedicated to one IndexedDB, `web-arcade`. The database name is assigned to a constant on line 4. It is used while opening a connection to the database.

Object Store/Table

Consider the following explanation that details how to use the `stores()` and `table()` APIs between lines 11 and 14:

- The constructor also defines the object store structure. In the current example, it creates a single store called `gameComments`. See the string value in line 5. You may create additional object stores by including additional fields in the JSON object. It is passed in as a parameter to the `stores()` function.
- The `gameComments` object store defines two fields, `IdxCommentId` and `timeCommented`.
- You prefix (or postfix) `++` on the primary key. This field identifies each comment uniquely. It auto-increments for each record added to the object store.

- The object store includes one or more fields. In this example, the object store includes two fields: `timeCommented` and `userName`. This statement creates the object store with the listed fields.
- While inserting records into the object store, you may include many more fields. However, indexes are created only on the fields specified with the `stores()` API (line 12). A query is limited to the fields indexed with the object store. Hence, include any field that you may query in the future.
- Notice that the stores function is in a `version()` API, which defines a version for the Web Arcade IndexedDB. As you will see in the next section, you may create additional versions of the database and upgrade.
- Dexie uses a TypeScript class called `Table` to refer to an object store. See line 7 for the class variable comments. You create the variable of type `Table` (`Dexie.Table`).
- Notice a generic type `CommentsEntity` passed to the `Table` class. The class variable `comments` is confined to the interface `CommentsEntity`. Remember, the comment entity includes all the fields related to a user comment. Revisit `CommentsEntity` on `src/app/common/comments-entity.ts`. See Listing 10-2.
- Next, see line 14. The `this.table()` function returns an object store reference. The `table()` function is inherited from the parent classes. Notice that you provide an object store name to the `table()` function. It uses the name to return that particular object store, for example, a `gameComments` object store.

- The returned object store is set to the class variable `comments`. Accessing this variable on the `WebArcadeDb` instance refers to the object store `gameComments`. For example, `webArcadeDbObject.comments` refers to the `gameComments` object store.

Listing 10-2. Comments Entity

```
1: export interface CommentsEntity {
2:     IdxCommentId?: number;
3:     title: string;
4:     comments: string;
5:     timeCommented: string;
6:     gameId: number;
7:     userName:string;
8: }
```

IndexedDB Versions

As your application evolves, anticipate changes to the database. IndexedDB supports versions to transition between the upgrades. Dexie uses the underlying API and provides a clean way to version IndexedDB.

Listing 10-3 creates the `web-arcade` database with one object store and three fields (one primary key and two indexes). See line 12. Imagine you need to add an additional field `gameId` to the index and create a new object store for board game comments.

Before you make this database change, increment the version number. Consider updating it to 1.1.

Note In version number 1.0, the number before the decimal point is called the *major version*. The number after the decimal point is the *minor version*. As the names indicate, consider updating the major version number if there is a major change to the database structure. For a minor addition of a single field, index, or object store, update the minor version.

Next, add a new index for `gameId`. Include a new object store called `boardGameComments` with a primary key, `commentId`. Consider Listing 10-3. See Figure 10-1 for the result. This is an IndexedDB view using Google Chrome Developer Tools.

Listing 10-3. Upgrade Web Arcade to a New Version

```
this.version(1.1).stores({
    gameComments: '++IdxCommentId,timeCommented, userName,
    gameId',
    boardGameComments: '++commentId'
  });
```

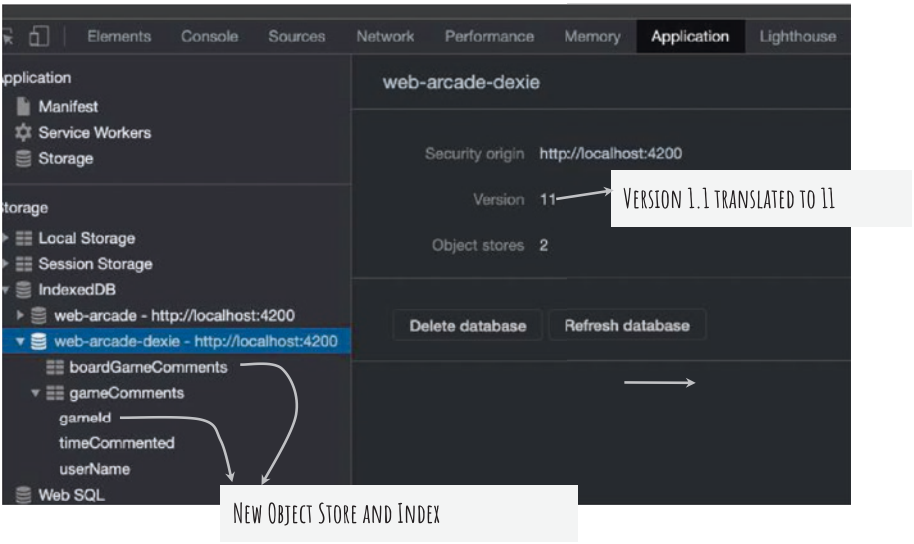

Figure 10-1. *New object store, index on version 11 (1.1)*

Next, consider a scenario where you need to delete an object store and remove an index. Consider removing the index on the username and deleting the `boardGameComments` object store. Follow these instructions:

1. Update the version number. Consider using 1.2. This will translate to 12 on the IndexedDB.
2. Set the object store to be deleted to null. In the current example, set `boardGameComments` to null. See line 3 in Listing 10-4.
3. To make changes to an object store, use the `upgrade()` API on the database object. In the current example, we remove an index called `userName` on the object store `gameComments` and provide a callback function. The function parameter is a reference variable to the database. Consider Listing 10-4.

Listing 10-4. Remove Object Store and Index

```
1: this.version(1.2).stores({
2:     gameComments: '++IdxCommentId,timeCommented, userName,
       gameId',
3:     boardGameComments: null
4:   }).upgrade( idb =>
5:     idb.table(OBJECT_STORE_GAME_COMMENTS)
6:       .toCollection()
7:       .modify( comments => {
8:         delete comments.userName;
9:       }) );
```

- Line 8 deletes the user on the `comments` object. The `comments` reference is obtained while modifying the object store `gameComments`. Remember, Dexie's table class (and the instance) refers to an object store.

Connecting with Web-Arcade IndexedDB

Remember the thought process creating the `IdbStorageAccessService`. It abstracts the IndexedDB API from the rest of the application. If you choose to use Dexie instead of the native browser API, follow a similar approach and create a service. Run the following command to create a service. Provide the arbitrary name `dexie-storage-access` to the service.

```
ng g s common/dexie-storage-access
```

Note The command `ng g s common/dexie-storage-access` is a short form of `ng generate service common/dexie-storage-access`.

g- generate

s- service

Similar to `IdbStorageAccessService`, initialize the `DexieStorageAccessService` at application startup. Include an `init()` function with the code to initialize. Use Angular's `APP_INITIALIZER` and include it in the `AppModule`. Consider Listing 10-5. See lines 11 to 16. Notice that the app initializer invokes the `init()` function (line 13).

Listing 10-5. Initialize DexieStorageAccessService at Application Startup

```
01: @NgModule({
02:     declarations: [
03:       AppComponent,
04:       /* More declarations go here */
05:     ],
06:     imports: [
07:       BrowserModule,
08:       /* additional imports go here */
09:     ],
10:     providers: [
11:       {
12:         provide: APP_INITIALIZER,
13:         useFactory: (svc: DexieStorageAccessService) => ()
            => svc.init(),
14:         deps: [DexieStorageAccessService],
15:         multi: true
```

```
16:       }
17:       /* More providers go here */
18:     ],
19:     bootstrap: [AppComponent]
20:   })
21:   export class AppModule { }
```

Initializing IndexedDB

DexieStorageAccessService initializes IndexedDB using an instance of the WebArcadeDB class (created in Listing 10-3). Use the open() function, which opens a connection to the database if it already exists. If not, it will create a new database and open the connection. Consider Listing 10-6.

Listing 10-6. Dexie Storage Access Service

```
01: import { Injectable } from '@angular/core';
02: import { WebArcadeDb } from 'src/app/common/web-arcade-db';
03: import { CommentsEntity } from 'src/app/common/
    comments-entity';
04:
05: @Injectable({
06:   providedIn: 'root'
07: })
08: export class DexieStorageAccessService {
09:   webArcadeDb = new WebArcadeDb();
10:   constructor() {}
11:   init(){
12:     this.webArcadeDb
13:     .open()
14:     .catch(err => console.log("Dexie, error opening DB"));
15:   }
16: }
```

Consider the following explanation:

- Create a new class-level instance of `WebArcadeDb` and instantiate. It encapsulates the Web Arcade IndexedDB. See line 9 in Listing 10-9.
- Remember that you invoked the `init()` function from the app module with the help of `APP_INITIALIZER`. Notice the definition on lines 11 to 15. This initializes by invoking `open()` on the IndexedDB. As mentioned earlier, it creates a database for Web Arcade if it doesn't exist. It will open a connection to IndexedDB.
- After initialization, the IndexedDB is open for the database operations including CRUD.
- The open function returns a promise. If it fails, the promise is rejected. Notice line 14. This is an error handling statement when the promise is rejected. In the current example, you log a message and the error to the browser console.

Transactions

It is important to include database operations in a transaction. A transaction ensures all enclosed operations are atomic, that is, performed as a single unit. Either all the operations are performed or none is performed. This is useful to ensure the consistency of the data.

In an example, imagine you are transferring data from object store 1 to object store 2. You read and deleted data from object store 1. Imagine the user closed the browser before the update to object store 2 is complete.

Without a transaction, data is lost. A transaction ensures the deletion from object store 1 is reverted if there is a failure before adding the data to object store 2. This ensures data is not lost.

Create a transaction on a `WebArcadeDb` object, as shown in Listing 10-7.

Listing 10-7. Create a Transaction

```
1: this.webArcadeDb.transaction("rw",
2:     this.webArcadeDb.comments,
3:     () => {
4:
5:     })
```

Consider the following explanation:

- See line 1. A transaction is created on an instance of the `WebArcadeDb` object. It is a class-level variable on `DexieStorageAccessService`.
- The first parameter on the transaction function is a transaction mode. Two values are possible.
  - `Read`: The value to the first parameter is `r`. The transaction can only perform read operations.
  - `Read-Write`: The value of the first parameter is `rw`. See line 1 in Listing 10-10. The transaction can perform read and write operations.
- The second parameter is an object store reference. See line 2. The `comments` field points to the object store `gameComments`. See line 14 in Listing 10-3.
- You may include more than one object stores in a transaction.

- The final parameter is a function callback. It includes code to perform create, retrieve, update, or delete operations on a database.

Add

Remember, so far, that you created an object store called gameComments. Listing 10-8 adds a record to the object store.

Listing 10-8. Add a Comment Record to the Object Store

```
addComment(title: string, userName: string, comments:
    string, gameId: number, timeCommented = new Date()){
        this.webArcadeDb
          .comments
          .add({
            title,
            userName,
            timeCommented: `${timeCommented.
            getMonth()}/${timeCommented.
            getDate()}/${timeCommented.getFullYear()}`,
            comments,
            gameId,
          })
          .then( id => console.log(`Comment added successfully.
          Comment Id is ${id}`))
          .catch( error => console.log(error))
          ;
}
```

Consider the following explanation:

- See line 2. You use an instance of the `WebArcadeDb` object. It is a class-level variable on `DexieStorageAccessService`.
- The `add()` function inserts a record into the object store (line 4). The record includes various comment fields including the title, username, comment date and time, comment description, and ID of the game on which the comment is added.
- The `add()` function returns a promise. If the add action is successful, the promise is resolved. See line 11, which logs the comment ID (primary key) to the browser console. If the add action fails, the promise is rejected. The `catch()` function on line 12 runs, which prints the error information on the browser console.

Delete

Perform the delete action by using the database object, `webArcadeDb`. Call the `delete()` API on the database. It needs a comment ID, the primary key as an input parameter. Consider Listing 10-9.

Listing 10-9. Delete a Comment Record in the Object Store

```
1: deleteComment(id: number){
2:     return this.webArcadeDb
3:       .comments
4:       .delete(id)
5:       .then( id => console.log(`Comment deleted
         successfully.`))
```

```
6:        .catch(err => console.error("Error deleting", err));
7: }
8:
```

Consider the following explanation:

- See line 2. You use an instance of the `WebArcadeDb` object. It is a class-level variable on `DexieStorageAccessService`.
- The `delete()` function deletes a record from the object store (line 4). The record to be deleted is identified by the comment ID, a primary key.
- The `delete()` function returns a promise. If the delete action is successful, the promise is resolved. See line 5, which logs a success message to the browser console. If the delete action fails, the promise is rejected. The `catch()` function on line 6 runs, which prints the error information on the browser console.

Update

Perform the update action by using the database object, `webArcadeDb`. Call the `update()` API on the database. It needs a comment ID, which is the primary key, as the first input parameter. It selects the record to be updated using the comment ID. It uses an object with a key path and a new value to be updated. Consider Listing 10-10.

Listing 10-10. Update a Comment Record

```
1: updateComment(commentId: number, newTitle: string,
   newComments: string){
2:     this.webArcadeDb
```

```
3:        .comments
4:        .update(commentId, {title: newTitle, comments:
          newComments})
5:        .then( result => console.log(`Comment updated
          successfully. Updated record ID is ${result}`))
6:        .catch(error => console.error("Error updating",
          error));
7:    }
```

Consider the following explanation:

- See line 2. You use an instance of the `WebArcadeDb` object. It is a class-level variable on `DexieStorageAccessService`.
- The `update()` function updates a record on the object store (line 4). The record to be updated is identified by the comment ID, a primary key. The second parameter is an object with key-value pairs of the values to be updated. Notice that a key identifies the field to be updated on the record, in the object store.
- For example, the following code snippet updates a record with the comment ID 1, a primary key. The next two parameters are the new title and description, respectively.

  ```
  this.updateComment(1, "New title", "new comment
  description");
  ```

Figure 10-2 shows the result.

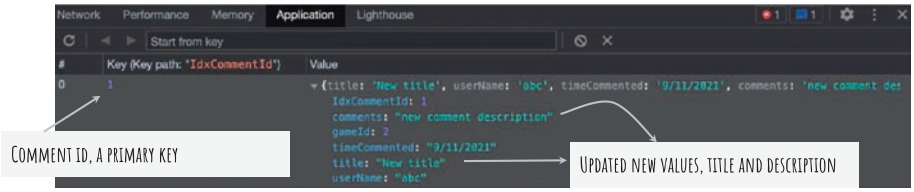

Figure 10-2. *Update result*

- The `update()` function returns a promise. If the update action is successful, the promise is resolved. See line 5, which logs a success message to the browser console. If the update action fails, the promise is rejected. The `catch()` function on line 6 runs, which prints the error information on the browser console.

Note The `update()` function updates specific fields on a record in an object store. To replace the entire object, use `put()`.

Retrieve

Dexie provides a comprehensive list of functions to query and retrieve data from IndexedDB. Consider the following:

- `get(id)`: Selects a record in the object store using an ID/primary key. The ID is passed in as a parameter. The `get()` function returns a promise. On successful get, the `then()` callback function returns results.
- `bulkGet([id1, id2, id3])`: Selects multiple records in the object store. IDs are passed in as a parameter. The `bulkGet()` function returns a promise. On a successful get, the `then()` callback function returns results.

- `where({keyPath1:value, keyPath2: value…, keyPath: value})`: Filters records by fields specified with `keyPath` and the given value.
- `each(functionCallback)`: Iterates through the objects in an object store. The API invokes the provided function callback asynchronously. Consider Listing 10-11.

Listing 10-11. Get All Cached Comments from the gameComments Object Store

```
1: getAllCachedComments(){
2:     this.webArcadeDb
3:       .comments
4:       .each( (entity: CommentsEntity) => {
5:         console.log(entity);
6:       })
7:       .catch(error => console.error("Error updating", error));
8: }
9:
```

Consider the following explanation:

- Line 2 uses an instance of the `WebArcadeDb` object. It is a class-level variable on `DexieStorageAccessService`.
- Line 4 iterates through each record in the `gameComments` object store.
- The callback function uses the parameter of type `CommentsEntity`. As and when the callback is invoked asynchronously, data confining to the `CommentsEntity` interface is expected to be returned.
- Line 5 prints the entity to the browser console.

More Options

In this book, you have seen the IndexedDB API supported natively by the browser. This chapter provided an introduction to Dexie.js, a wrapper intended to simplify access to the database.

The following are a few additional options to consider. While the implementation details are out of the scope of this book, consider reading and learning further about these libraries. All these libraries use IndexedDB underneath.

- *Local Forage*: This provides simple API and functions. The API is similar to local storage. On the legacy browsers that do not support IndexedDB, Local Forage provides a polyfill. It has the ability to fall back to WebSQL or local storage. It is an open source library with an Apache 2.0 license.
- *ZangoDB*: This provides a simple and easy-to-use API that mimics MongoDB. The library uses IndexedDB. The wrapper profiles an easy API for filtering, sorting, aggregation, etc. It is an open source library with an MIT license.
- *JS Store*: This provides Structured Query Language (SQL) like API for IndexedDB. It provides all the features IndexedDB provides in an easy-to-understand API similar to traditional SQL. It is an open source library with an MIT license.
- *PouchDB*: This provides an API to synchronize client-side, offline data with CouchDB. It's a highly useful library for applications using a CouchDB server-side back end. It is an open source library with an Apache 2.0 license.

Summary

This chapter introduced Dexie.js. It provided a basic understanding of the library within the parameters of the Web Arcade use cases. It listed instructions to install the Dexie.js NPM package to the web arcade application.

Also, the chapter listed a few additional libraries and wrappers on top of IndexedDB. While the implementation details are out of scope of this book, it lists the names and one-liner introduction for further learning.

EXERCISE

- Update the game details component to use the Dexie storage access service for caching comments while the application is offline.
- Update the online event to use the Dexie storage access service to retrieve records when the application returns online. Integrate with the server-side service to synchronize the data and delete local records using the Dexie.js API.
- Provide the ability to update a comment using the Dexie storage access service.

Addendum

Please read this addendum for a few minor details that did not fit into the Web Arcade use case descriptions of the book. We begin by detailing how to create a proxy for accessing the mock services (in the Angular application). The code sample already comes with the proxy configured. However, this section explains the purpose and implementation details.

The code sample for rolling a die was enhanced by launching in a bottom sheet, so we also detail how to use a bottom sheet in the Web Arcade application.

Also, we cover the hash location strategy using the traditional approach for routing in a single-page application. Hash routing prefixes the Angular route with a hash (#). Please note that it is advisable to stick to the default strategy used so far in the book. However, with the current configuration of Http-Server (a developer-class Node.js web server), the code samples do not work if you refresh the page. It forces you to start over from the beginning. Addendum provides hash routing as an alternative, which circumvents the refresh problem for Http-Server.

Creating a Proxy for Mock Services

Remember, in the code samples, the mock services run on a different port and instance. The Angular application runs on localhost (127.0.0.1) on port 4200. The mock web server runs on localhost on port 3000. Typically, web browsers do not allow users to access services on different domains. On a developer machine, consider the solution shown in Listing A-1 to access the mock services.

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4

Configure a proxy for the Angular application, and create a new file, called `proxy.conf.json`.

Listing A-1. Angular Application Proxy

```
{
    "/api": {
        "target": "http://localhost:3000",
        "secure": false
    }
}
```

Notice that an API call with the prefix `/api` is proxied to a target of `localhost:3000`. The Angular application in the browser invokes API calls on port 4200. The reroute to port 3000 is abstracted from the browser.

Next, specify the proxy configuration on the `ng serve` command. Consider the following code snippet. The option `--proxy-config` provides the configuration file.

```
"start": "npm run start-api & ng serve --proxy-config proxy.
conf.json",
```

You may specify a proxy with Http-Server (used while running the bundled application). Consider the following snippet:

```
"start-http-server": "yarn build && http-server dist/web-arcade
--proxy http://localhost:3000",
```

Using the Bottom Sheet for a Die Roll

When we started the book, one of the first components developed was a die. The component files are in the `src/components/dice` directory. Considering it was one of the first components in the sample project, routing was not introduced. Later, we extended the solution.

In the final solution, you launch the die within a bottom sheet, as shown in Figure A-1, using the button on the toolbar. As you know, the die animates, giving the impression of rolling a die.

Notice that the die is launched on the Game List page. Remember, the component for the game list is `BoardGamesComponent`. The code is located in the directory `src/app/components/board-games`.

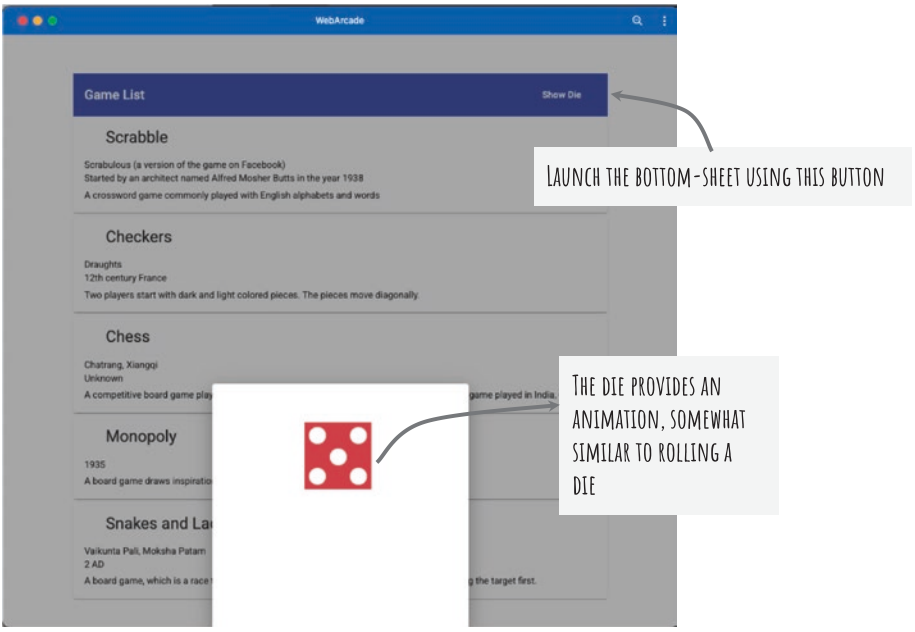

Figure A-1. *Rolling a die in the Web Arcade application*

Adding the Bottom Sheet in Web Arcade

To begin using a bottom sheet, import the Angular Material module for the bottom sheet in the App Module, as shown in Listing A-2. See lines 1 and 11, which import `MatBottomSheetModule` into `AppModule`.

Listing A-2. Import MatBottomSheetModule

```
01: import { MatBottomSheetModule } from '@angular/material/
    bottom-sheet';
02:
03: @NgModule({
04:     declarations: [
05:         /* Component declarations go here */
06:     ],
07:     imports: [
08:       BrowserModule,
09:       MatCardModule,
10:       MatInputModule,
11:       MatBottomSheetModule,
12:    ],
13:     providers: [
14:         // Services and providers declared here
15:     ],
16:     bootstrap: [AppComponent]
17:   })
18:   export class AppModule { }
```

Showing the Bottom Sheet with the Die Component

Next, import and inject the bottom sheet in `BoardGamesComponent` (for the game list). Consider Listing A-3.

Listing A-3. Import and Inject the Bottom Sheet in a Board Games Component

```
01: import { Component, OnInit } from '@angular/core';
02: import { MatBottomSheet } from '@angular/material/
    bottom-sheet';
03: import { DiceComponent } from '../dice/dice.component';
04:
05: @Component({
06:   selector: 'wade-board-games',
07:   templateUrl: './board-games.component.html',
08:   styleUrls: ['./board-games.component.sass']
09: })
10: export class BoardGamesComponent implements OnInit {
11:   constructor(
12:     private bottomSheet: MatBottomSheet) { }
14: }
```

See line 2, which imports the bottom sheet's TypeScript module in the Angular Material package. Next, see lines 11 and 12 in the code. They inject the bottom sheet into the component.

Also, notice line 3. It imports `DiceComponent` that encapsulates the die functionality. Next, include the function in Listing A-4, which draws the bottom sheet and shows a die on it.

Listing A-4. Open a Bottom Sheet with Dice

```
1: showDice(){
2:     this.bottomSheet.open(DiceComponent);
3: }
```

Notice that the function uses `this.bottomSheet`, an instance injected into the component constructor. The `open()` function draws the bottom sheet. You pass `DiceComponent` as a parameter. The bottom

sheet is a container drawn at the bottom of the page. The container hosts `DiceComponent` that is passed in as a parameter. See Figure A-1 for the result.

Invoke this function by clicking the Show Dice menu button. Consider Listing A-5 for the board games HTML template with the menu and the button.

Listing A-5. Board Games Template Launching Bottom Sheet

```
1: <mat-toolbar color="primary">
2:     <mat-toolbar-row>
3:       <span>Game List</span>
4:       <span class="flexExpand"></span>
5:       <button mat-button (click)="showDice()">Show Dice
         </button>
6:     </mat-toolbar-row>
7: </mat-toolbar>

8: <!-- rest of the board games template goes here -->
```

See line 5 with the click handler, which invokes the `showDice()` function. See Figure A-1 for the result.

Using a Hash Location Strategy

Angular provides the following two location strategies:

- `PathLocationStrategy`: The default location strategy. The screenshots and figures in the book use this strategy.
- `HashLocationStrategy`: Traditional location strategy, which uses the # prefix a path. See Figure A-2.

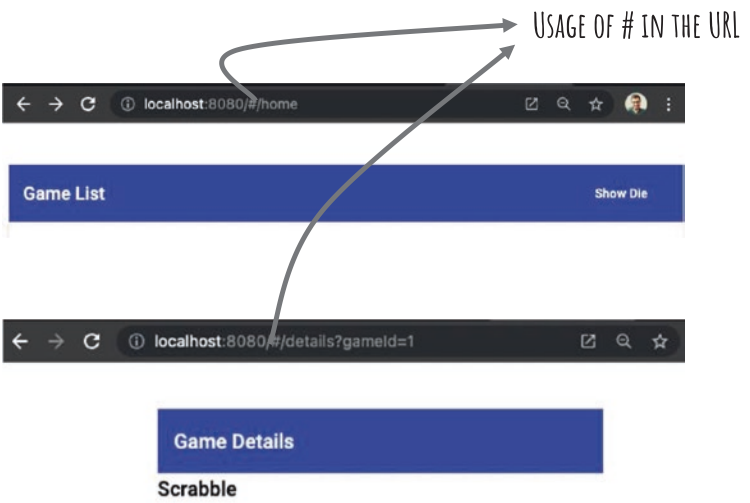

Figure A-2. *Hash routing*

Most modern applications use the path location strategy. It does not include any special characters in the URL. When an Angular application uses the path location strategy, the web server is expected to return `index.html` to load the Angular application for all the routes in the application. If a particular web server does not support this behavior (or is not configured to do so), you might fall back to the traditional hash routing strategy.

Note Remember that Web Arcade uses a developer-class web server, Http-Server. With the following configuration that we used in the sample application, reloading a page at a particular route (for example `http://localhost:8080/home`) returns an error. As mentioned earlier, the web server needs to return `index.html` for all the routes so that the Angular application in the browser manages a route. Considering that we don't have such a configuration, you may switch to hash routing for Web Arcade.

```
http-server dist/web-arcade --proxy http://
localhost:3000
```

To enable hash routing, update `AppRoutingModule` in `src/app/app-routing.module.ts`. See Listing A-6, line 19.

Listing A-6. Enabling Hash Routing

```
01: import { NgModule } from '@angular/core';
02: import { RouterModule, Routes } from '@angular/router';
03: import { BoardGamesComponent } from './components/board-
    games/board-games.component';
04: import { GameDetailsComponent } from './components/game-
    details/game-details.component';
05:
06: const routes: Routes = [{
07:   path: "home",
08:   component: BoardGamesComponent
09: }, {
10:   path: "details",
11:   component: GameDetailsComponent
12: }, {
13:   path: "",
```

```
14:   redirectTo: "/home",
15:   pathMatch: "full"
16: }];
17:
18: @NgModule({
19:   imports: [RouterModule.forRoot(routes, {useHash: true})],
20:   exports: [RouterModule]
21: })
22: export class AppRoutingModule { }
```

Summary

This addendum detailed content that did not fit into the main chapters of the book. First, we detailed how to configure a proxy for mock services. It is typical of Angular and browsers to add a proxy to access resources on different domains than that of the browser application.

Next, we detailed how to enhance the roll-a-die feature, by launching it in a bottom sheet. We covered how to add a bottom sheet to the Web Arcade application and integrate it with the die component.

Finally, we detailed a hash location strategy, an alternative to the default path location strategy. We specifically addressed the problem of refreshing pages deployed on Http-Server.

References and Links

The book and code samples use the following extensively:

- Angular framework
- Angular Material Library, which is Angular's implementation of Material Design

Use the following link for Angular's official documentation on the framework:

https://angular.io/docs

To learn how to build applications with Angular and Material Design, read the following book:

https://www.apress.com/us/book/9781484254332

The following provides documentation on SASS and SCSS for stylesheet development. Web Arcade and the code samples use the indented syntax provided with SASS. Please use the following URL to learn more about SASS and SCSS:

https://sass-lang.com/documentation/syntax

Node Package Manager (NPM) is a default tool for installing node modules, packages, and libraries. The book uses it extensively. To learn more about Node.js and NPM, use the following links:

- *NPM documentation*: https://docs.npmjs.com/about-npm
- *Node.js documentation*: https://nodejs.dev/learn

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4

Yarn is another package manager and a popular alternative. To learn more about Yarn, use the following links:

- *Website*: https://yarnpkg.com/
- *Yarn documentation*: https://classic.yarnpkg.com/en/docs/

The book and the code samples use Angular CLI extensively to create and maintain the Angular application. The book uses the following documentation as a reference:

https://angular.io/cli

The code samples use the Http-Server package for running a developer class web server.

- *Use the following link for NPM documentation on the package*: https://www.npmjs.com/package/http-server
- *Use the following link for the GitHub code repository*: https://github.com/http-party/http-server

The code sample uses url() in SASS (stylesheets). Please use the following link for documentation on the usage:

https://developer.mozilla.org/en-US/docs/Web/CSS/url()

Service workers are a core concept in the book. The book extensively uses documentation provided by Google developer pages. Refer to the following link:

https://developers.google.com/web/fundamentals/primers/service-workers

While using HTTP services, the API uses HTTP methods, and there are conventions for selecting one to use. The code samples and the book take references from the following documentation:

https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

The book uses Node.js and ExpressJS for developing mock HTTP services. Use the following documentation for learning more about Express:

https://expressjs.com/

Use the following link for a guide:

https://expressjs.com/en/guide/routing.html

IndexedDB is a core concept detailed in the book, which takes references from the API and an implementation from the following documentation and the links:

- *W3C Recommendation (at the time of writing)*: https://www.w3.org/TR/IndexedDB-2/
- *W3C Working Draft (at the time of writing)*: https://www.w3.org/TR/IndexedDB/

The IndexedDB documentation on the Mozilla.org website is at the following link:

https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

A Wikipedia article on IndexedDB can be found here:

https://en.wikipedia.org/wiki/Indexed_Database_API

Browser support for IndexedDB can be found here:

https://caniuse.com/indexeddb

The book explains how to use the Dexie.js library as an alternative API for the default IndexedDB API. It simplifies IndexedDB access. Refer to the documentation for Dexie.js at the following link:

https://dexie.org/docs/Dexie.js

Verify browser support for a feature at CanIUse.com, found here:

https://caniuse.com/indexeddb

The book mentions the following integrated development environments (IDEs) for the code:

- *Visual Studio Code*: https://code.visualstudio.com/
- *Sublime Text*: https://www.sublimetext.com/
- *Atom*: https://atom.io/
- *WebStorm*: https://www.jetbrains.com/webstorm/

Index

A

B

C

V. K. Kotaru, *Building Offline Applications with Angular*,
https://doi.org/10.1007/978-1-4842-7930-4

D

E

F

G

H

I

J, K

L

M

N

O

P, Q

R

S

T

U

V

W, X

Y, Z