SVELTE

SUCCINCTLY

BY **ED FREITAS**

SUCCINCTLY EBOOK SERIES

Syncfusion®

Svelte Succinctly

Ed Freitas

Foreword by Daniel Jebaraj

ISBN: 978-1-64200-230-0

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The *Succinctly* Series of Books

Daniel Jebaraj
CEO of Syncfusion, Inc.

When we published our first *Succinctly* series book in 2012, *jQuery Succinctly*, our goal was to produce a series of concise technical books targeted at software developers working primarily on the Microsoft platform. We firmly believed then, as we do now, that most topics of interest can be translated into books that are about 100 pages in length.

We have since published over 200 books that have been downloaded millions of times. Reaching more than 2.7 million readers around the world, we have more than 70 authors who now cover a wider range of topics, such as Blazor, machine learning, and big data.

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, you will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

We are absolutely thrilled with the enthusiastic reception of our books. We believe the *Succinctly* series is the largest library of free technical books being actively published today. Truly exciting!

Our goal is to keep the information free and easily available so that anyone with a computing device and internet access can obtain concise information and benefit from it. The books will always be free. Any updates we publish will also be free.

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctlyseries@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on social media and help us spread the word about the *Succinctly* series!

About the Author

Ed Freitas is a consultant on business process automation and a software developer focused on customer success.

He likes technology and enjoys learning, playing soccer, running, traveling, and being around his family.

Ed is available at https://edfreitas.me.

Acknowledgments

A huge thank you to the fantastic Syncfusion team that helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The manuscript managers and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Jacqueline Bieringer and Graham High from Syncfusion, and James McCaffrey from Microsoft Research. Thank you all.

I dedicate this book to my beloved "Chelin" and "Puntico." May your journeys be blessed.

Introduction

Let's begin by explaining what Svelte is. To do that, we first need to understand what Svelte isn’t.

In the traditional context of modern declarative JavaScript frameworks such as React and Vue, Svelte departs from the virtual DOM approach by compiling the code you write into native-browser JavaScript when you build your application.

Svelte allows you to do the same things as React and Vue, such as creating reusable UI components, data-binding or event-handling, and creating single-page applications.

But Svelte compiles your code during build time instead of using differences to update the DOM. The result is minimal and highly optimized pure JavaScript that the browser executes with no overhead (it ships no runtime, like React and Vue do).

When creating an application using React or Vue, the runtimes of these frameworks are included along with your application code build, resulting in a *heavier* app deployment. In other words, this runtime ships along with your app and executes your application code.

For example, when you write an application using Vue, your application contains code written in a way only the Vue framework can interpret and execute. This code will include one or more Vue components and other Vue-specific constructs.

When you build and deploy your Vue app, those Vue-specific code constructs and components get bundled into a deployable Vue application. Additionally, for your app to run, the Vue runtime is also deployed to the browser with your app. The same is true for React.

So, when the browser is running your Vue or React application, what is happening behind the scenes is that the browser executes the framework runtime, which then interprets and executes your application’s specific logic.

Svelte does not deploy a runtime, since that’s additional code for the browser to execute, and instead deploys the application’s optimized JavaScript code.

Svelte can achieve this because it establishes a convention for how you should write your application code so that the Svelte compiler can then take that code and convert it into optimized JavaScript, ready for the browser to run.

Therefore, the main difference between Svelte and traditional virtual DOM-oriented frameworks like React and Vue is that Svelte compiles your app code into JavaScript before it ships, behaving like a compiler and a wrapper around conventional JavaScript.

Svelte does a great job at working with the native languages of the web, such as HTML, CSS, and JavaScript.

Beyond that, Svelte also eliminates unused code (that never runs) from the final app build and includes a rich toolset.

Chapter 1 Getting Started

Svelte architecture

To understand how Svelte works, let’s explore the following diagram.

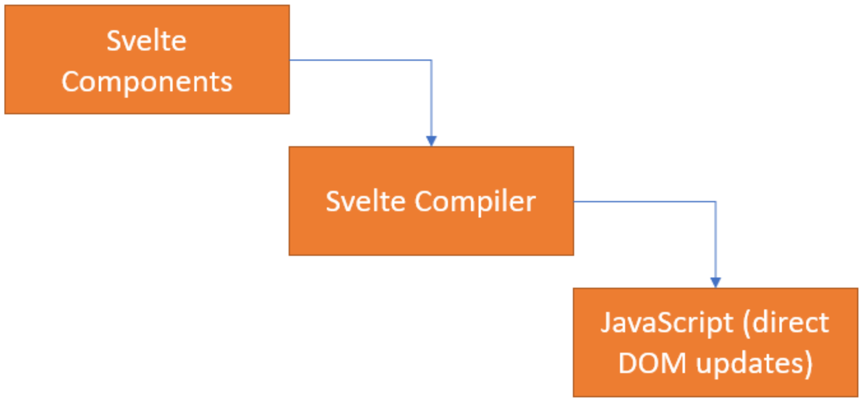


Figure 1-a: Svelte Architectural Design

A Svelte application, just like a React or Vue application, also includes components. The Svelte compiler reads these components, and the result is JavaScript that directly updates the DOM.

In contrast, a React or Vue application's architecture uses the virtual DOM approach, which we can observe in the following figure.

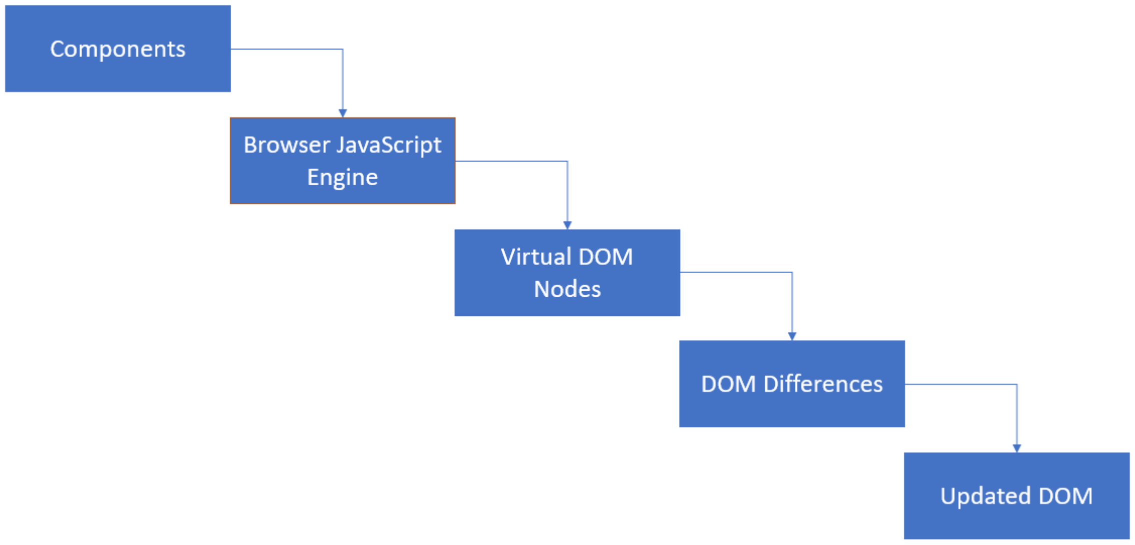


Figure 1-b: Virtual DOM Architectural Design

So we have an app that also has components, and there's a virtual representation of how the application's UI looks, kept in memory and synced with the current DOM.

Those UI differences between the virtual and actual DOM are reconciled, and then the DOM that the browser executes is updated as needed. In other words, with a React or Vue application, there's much more going on behind the scenes.

Technically speaking, there's nothing wrong with the virtual DOM approach used by frameworks like React and Vue—otherwise, they wouldn't be as popular as they are. It's just a different way of doing things with Svelte.

There are certain performance advantages and less code overhead when using a nonvirtual DOM approach. To better understand what inspired Svelte's creator to use a compiled method, take a look at this article that dives a bit deeper into the subject.

Now that we have a high-level understanding of Svelte's architecture, let's install it.

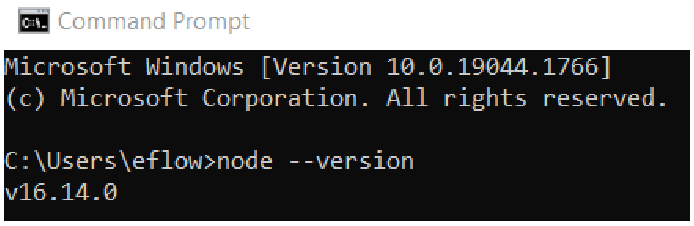
Installing Node.js

To get started with Svelte, we need to have Node.js installed. You can verify if you have Node.js installed by opening the command prompt or terminal and executing the following command.

Code Listing 1-a: Check if Node.js Is Installed

```
node --version
```

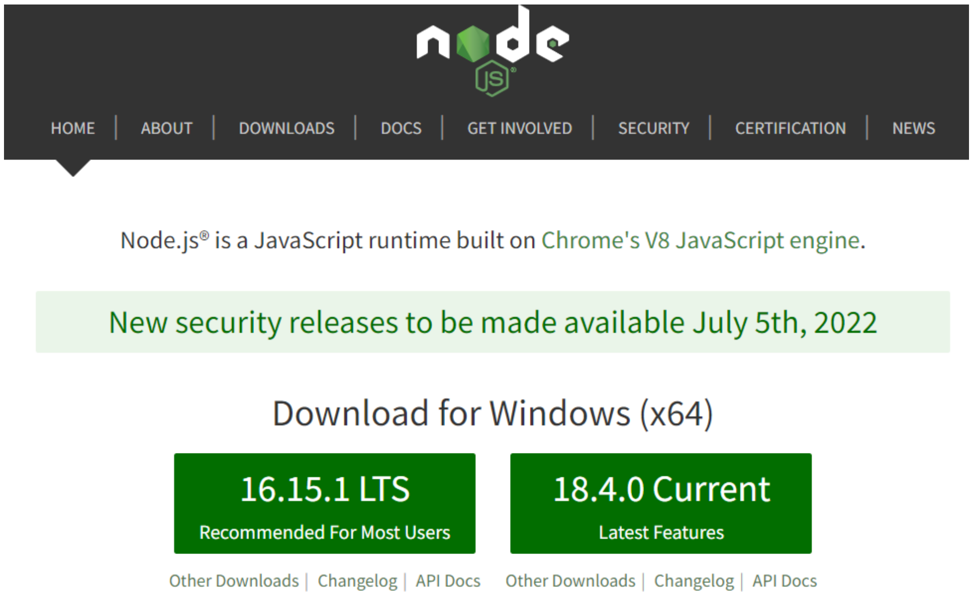
In my case, I have version 16.14.0 installed.



```
Microsoft Windows [Version 10.0.19044.1766]
(c) Microsoft Corporation. All rights reserved.

C:\Users\eflow>node --version
v16.14.0
```

Figure 1-c: Node.js Version Installed

Any version of Node.js later than version 8 is sufficient for working with Svelte. However, I encourage you to use one of the most recent versions, which you can download from the Node.js website. The long-term support (LTS) version is always a safe bet.


Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

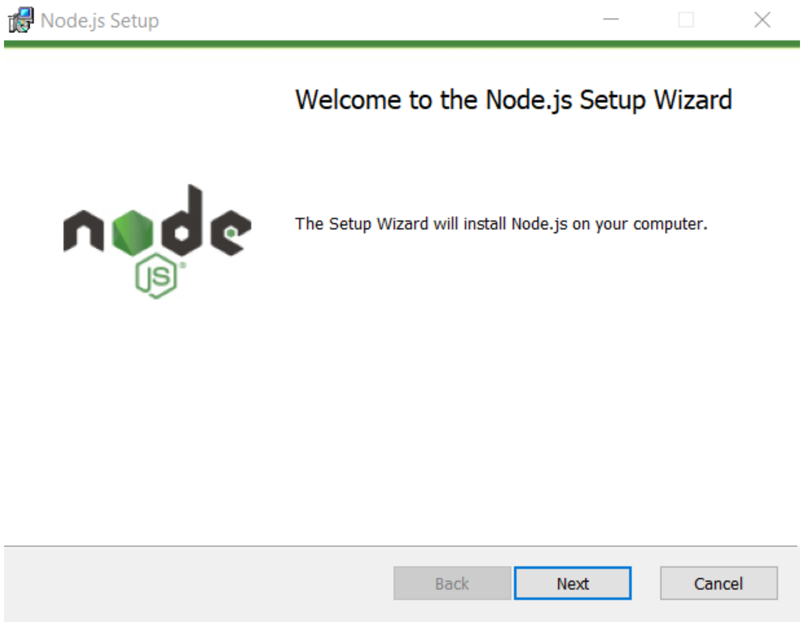
New security releases to be made available July 5th, 2022

Download for Windows (x64)

HOME | ABOUT | DOWNLOADS | DOCS | GET INVOLVED | SECURITY | CERTIFICATION | NEWS
16.15.1 LTS
Recommended For Most Users
18.4.0 Current
Latest Features
Other Downloads | Changelog | API Docs
Other Downloads | Changelog | API Docs

Figure 1-d: Node.js Website Front Page

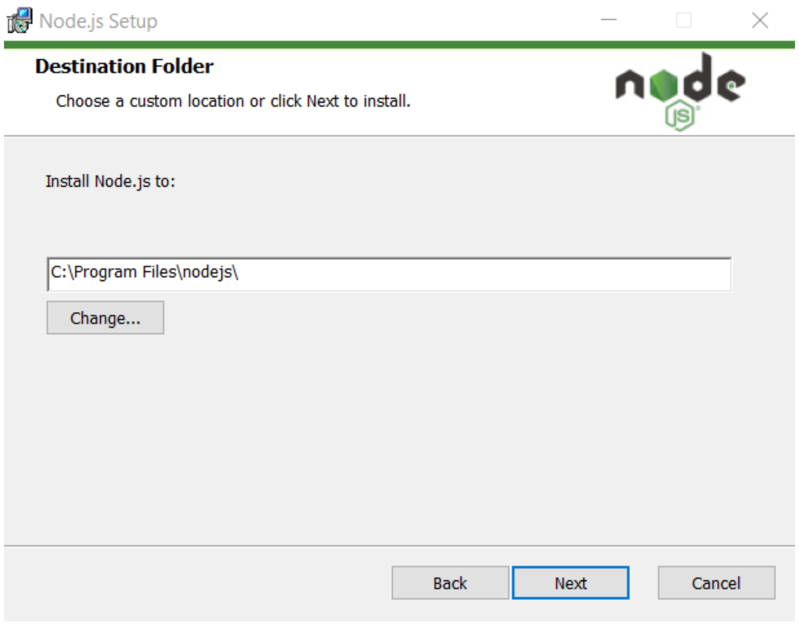
Once you have downloaded Node.js, execute the installer file. Once you have run the installer, you'll see the following screen. Click **Next** to continue the installation process.


Welcome to the Node.js Setup Wizard

The Setup Wizard will install Node.js on your computer.

Figure 1-e: Initial Node.js Installation Screen

You'll be asked to accept the license terms and click **Next** to carry on with the installation. At this stage, you'll see the screen where you can select the Node.js installation folder.


Node.js Setup

Destination Folder

Choose a custom location or click Next to install.

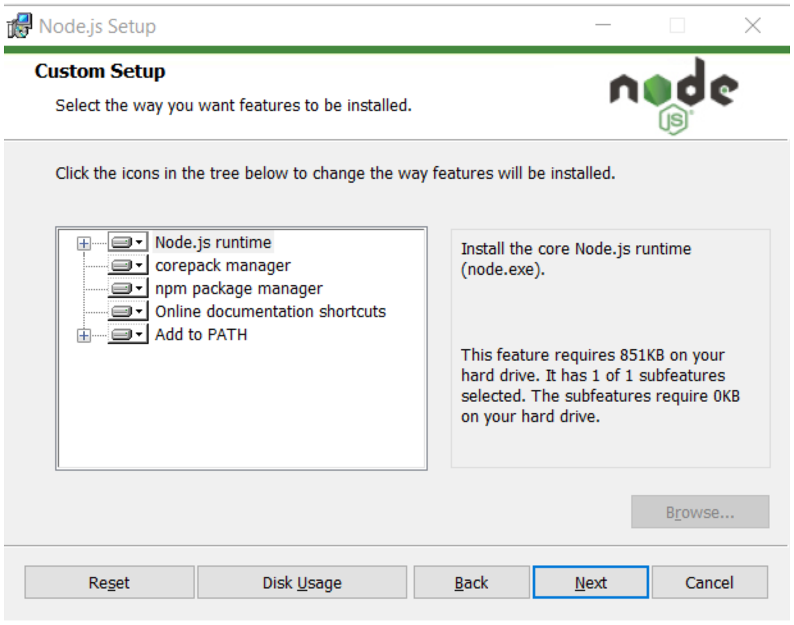
Install Node.js to:

C:\Program Files\nodejs\

Figure 1-f: Node.js Installation (Destination Folder Screen)

I usually leave the default installation folder and click **Next**; however, you can choose a different folder if you prefer. With that done, click **Next** to continue the installation.

At this point, you'll see the Custom Setup screen. In my case, I always use the default options, as you can see in the following figure.


Node.js Setup

Custom Setup

Select the way you want features to be installed.

Click the icons in the tree below to change the way features will be installed.

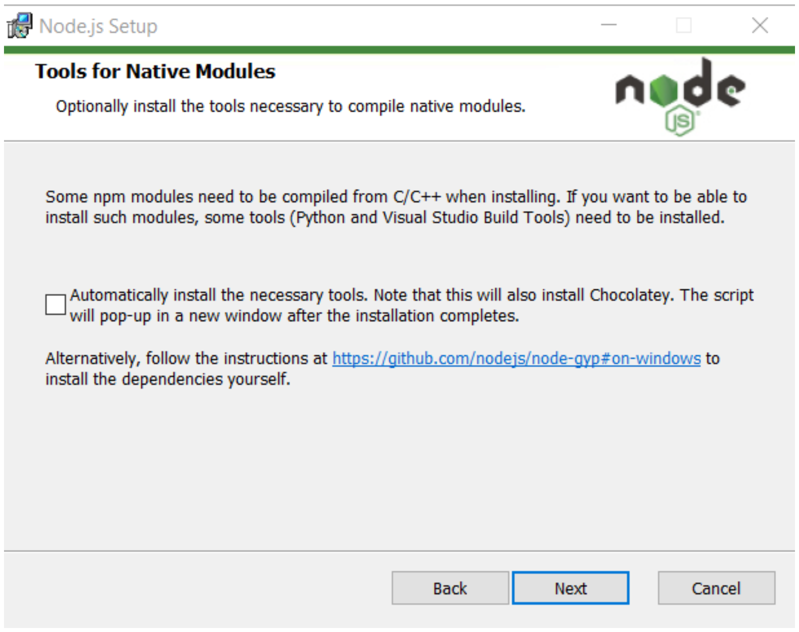
- Node.js runtime
- corepack manager
- npm package manager
- Online documentation shortcuts
- Add to PATH

Install the core Node.js runtime (node.exe).

This feature requires 851KB on your hard drive. It has 1 of 1 subfeatures selected. The subfeatures require 0KB on your hard drive.

Figure 1-g: Node.js Installation (Custom Setup Screen)

To continue the installation, click **Next**. After doing that, you should see the following screen.


Node.js Setup

Tools for Native Modules

Optionally install the tools necessary to compile native modules.

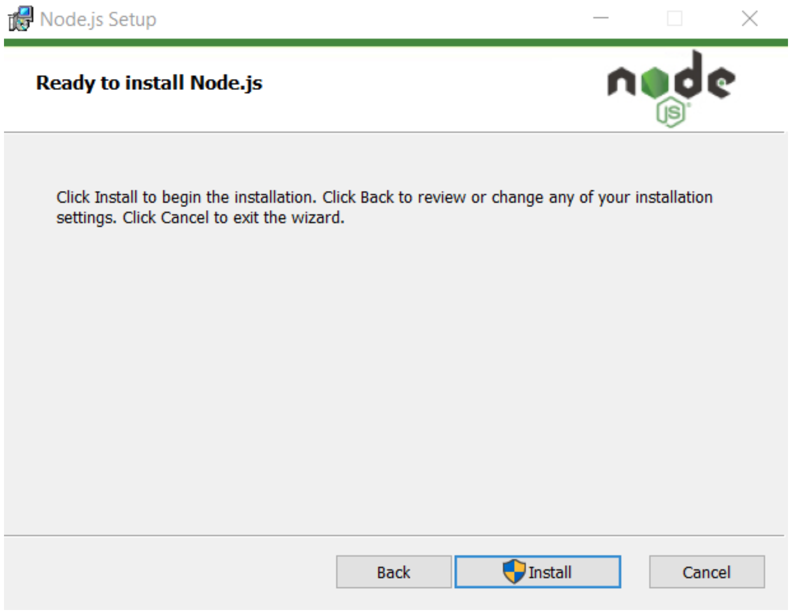
Some npm modules need to be compiled from C/C++ when installing. If you want to be able to install such modules, some tools (Python and Visual Studio Build Tools) need to be installed.

☐ Automatically install the necessary tools. Note that this will also install Chocolatey. The script will pop-up in a new window after the installation completes.

Alternatively, follow the instructions at https://github.com/nodejs/node-gyp#on-windows to install the dependencies yourself.

Figure 1-h: Node.js Installation (Tools for Native Modules Screen)

You may select the option **Automatically install the necessary tools**, which allows the Node.js installer to install any other dependency needed using Chocolatey. To continue the installation, click **Next**.


Node.js Setup

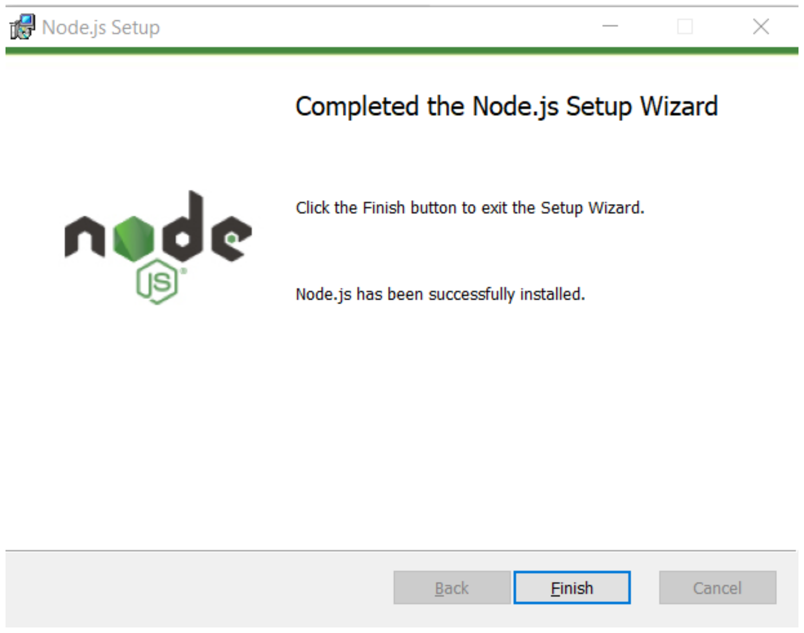
Ready to install Node.js

Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.

Figure 1-i: Node.js Installation (Ready to install Node.js Screen)

Click **Install** to deploy the Node.js runtime and files in the installation folder previously selected. The process is usually quick.

If a previous version of Node.js exists on your machine, that version gets removed before deploying the newer version. Once the new files have been installed, you'll see the following screen.


Completed the Node.js Setup Wizard

Click the Finish button to exit the Setup Wizard.

Node.js has been successfully installed.

Figure 1-j: Node.js Installation (Node.js Setup Wizard Screen—Finish)

To finalize the installation, all we need to do is click **Finish**. Now that Node.js is installed, the next step is to prepare the development environment.

Svelte with VS Code

I'll use Visual Studio Code (VS Code) as my editor and development environment of choice throughout this book. You should add the **Svelte for VS Code** extension, because it is a requirement and will provide the necessary tooling to work with Svelte components, templates, and syntax.

We can get this extension by clicking the extension's icon in VS Code, writing the word **Svelte** under the extensions search box, and then clicking the extension's **Install** button.

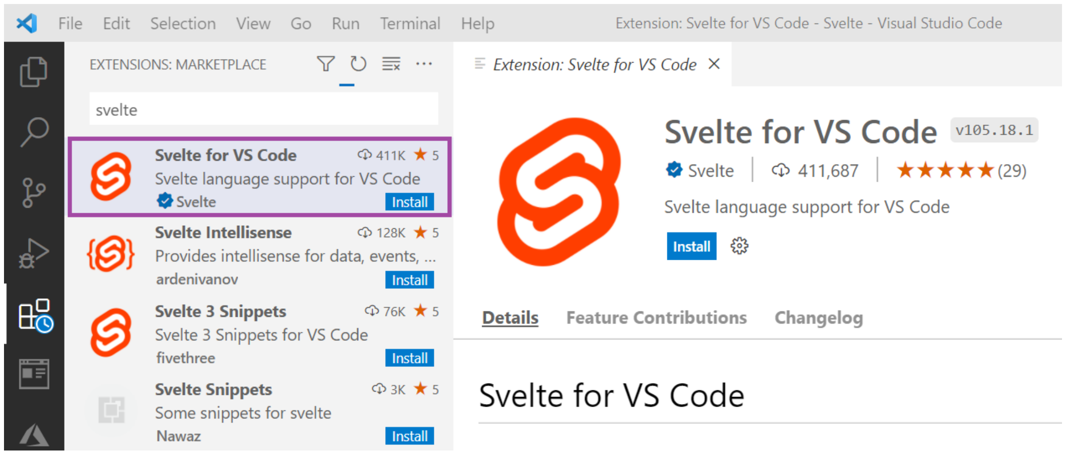

File Edit Selection View Go Run Terminal Help
Extension: Svelte for VS Code - Svelte - Visual Studio Code
EXTENSIONS: MARKETPLACE
svelte
Svelte for VS Code 411K 5
Svelte language support for VS Code
Svelte
Install
Svelte Intellisense 128K 5
Provides intellisense for data, events, ...
ardenivanov
Install
Svelte 3 Snippets 76K 5
Svelte 3 Snippets for VS Code
fivethree
Install
Svelte Snippets 3K 5
Some snippets for svelte
Nawaz
Install
Extension: Svelte for VS Code
Svelte for VS Code v105.18.1
Svelte 411,687 (29)
Svelte language support for VS Code
Install
Details Feature Contributions Changelog
Svelte for VS Code

Figure 1-k: Svelte for VS Code Extension

Creating a Svelte project

With VS Code and the Svelte extension ready, we can create a Svelte application. The easiest (but no longer maintained) way to do this is using the Svelte template—which you can do by executing the following commands.

Code Listing 1-b: Creating a Svelte Project (Using the Svelte Template)—No Longer Maintained

```
npx degit sveltejs/template svelte-app
cd svelte-app
```

Another way is to use Vite (pronounced "Veet") and select the **svelte** option by running the following command.

Code Listing 1-c: Creating a Svelte Project (Using Vite)

```
npm init vite
```

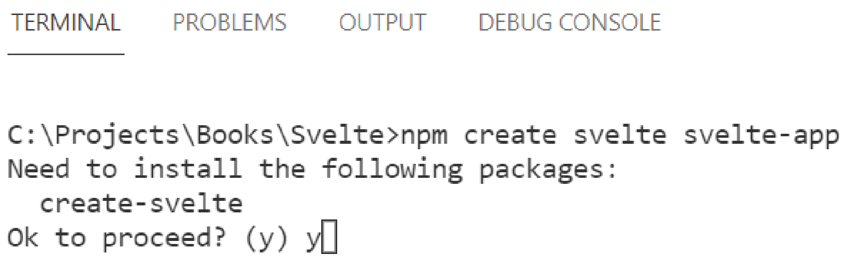
A third (and my preferred) way is to use SvelteKit, a Svelte framework for creating web apps with file system-based routing and server-side rendering, among other benefits. This is the option we'll be using.

Within VS Code, open the built-in terminal by clicking the **Terminal** menu option, then click **New Terminal** and enter the following command.

Code Listing 1-d: Creating a Svelte Project (Using the SvelteKit)

```
npm create svelte svelte-app
```

If the create-svelte package is not installed, you'll be prompted to install it. So, to continue, enter **y**.



```
C:\Projects\Books\Svelte>npm create svelte svelte-app
Need to install the following packages:
  create-svelte
Ok to proceed? (y) y
```

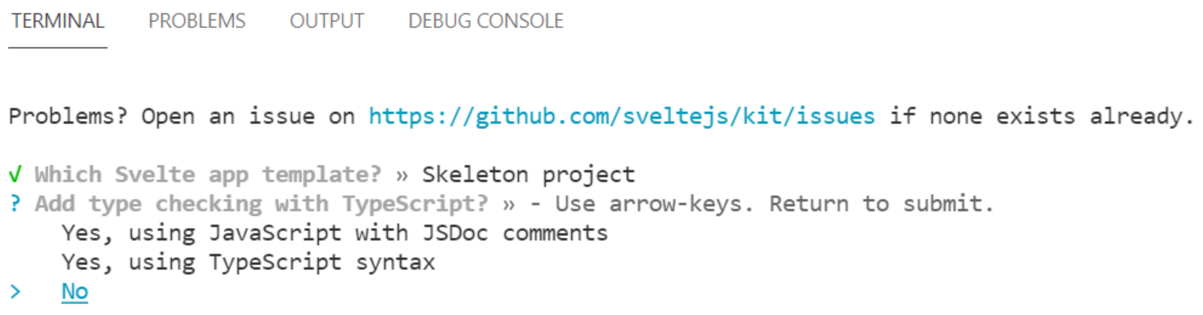
Figure 1-l: Installing create-svelte (SvelteKit)

Once create-svelte has been installed, you'll have to choose a Svelte template: SvelteKit demo app or the Skeleton project.

You can use the arrow keys to choose the template you're interested in establishing as the base for the project and then press Enter.

In my case, I choose the **Skeleton project**, which is the barebones Svelte project—for following along quickly, I suggest you choose the same one.

Following that, we can choose whether to add type checking with TypeScript. I select **No**, given that this is not an essential feature for beginners.



```
Problems? Open an issue on https://github.com/sveltejs/kit/issues if none exists already.

√ Which Svelte app template? » Skeleton project
? Add type checking with TypeScript? » - Use arrow-keys. Return to submit.
    Yes, using JavaScript with JSDoc comments
    Yes, using TypeScript syntax
>   No
```

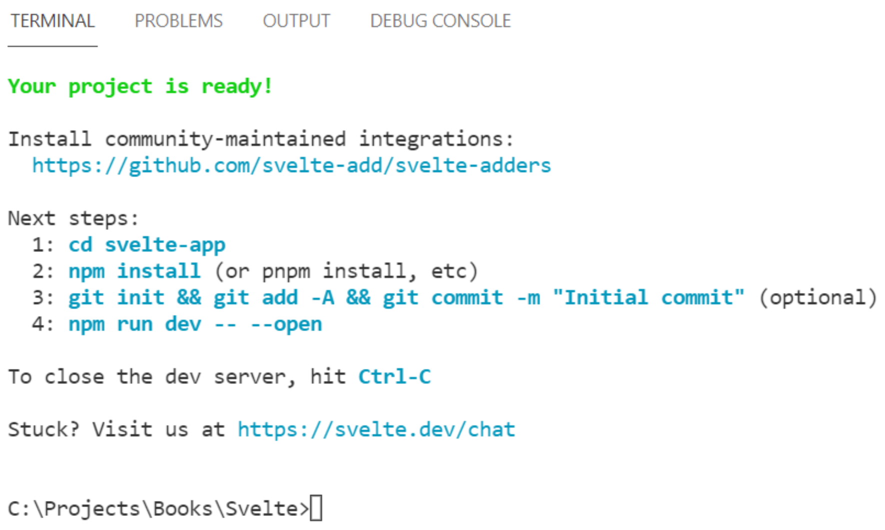
Figure 1-m: Selecting Type Checking

Next, we'll be asked whether or not to add ESLint for code linting, and I'll also select **No**.

Following that, we're asked whether we want to add Prettier for code formatting, and I choose **No**.

There's also an option to add Playwright for browser testing, and I select **No**.

With that done, the project is ready, and the following steps can be executed.



```
Your project is ready!

Install community-maintained integrations:
  https://github.com/svelte-add/svelte-adders

Next steps:
  1: cd svelte-app
  2: npm install (or pnpm install, etc)
  3: git init && git add -A && git commit -m "Initial commit" (optional)
  4: npm run dev -- --open

To close the dev server, hit Ctrl-C

Stuck? Visit us at https://svelte.dev/chat


C:\Projects\Books\Svelte>
```

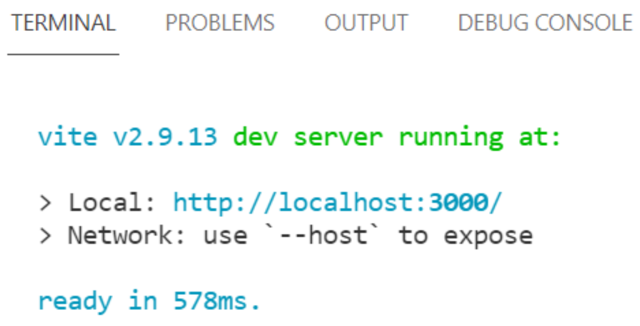
Figure 1-n: SvelteKit Skeleton Project Ready

To run the project as is, we can type in the **`cd svelte-app`** command within the built-in VS Code terminal, then run **`npm install`** to get all the required project dependencies installed, and then execute **`npm run dev`**.

Code Listing 1-e: Final Preparation Steps (SvelteKit Skeleton Project)

```
cd svelte-app
npm install
npm run dev
```

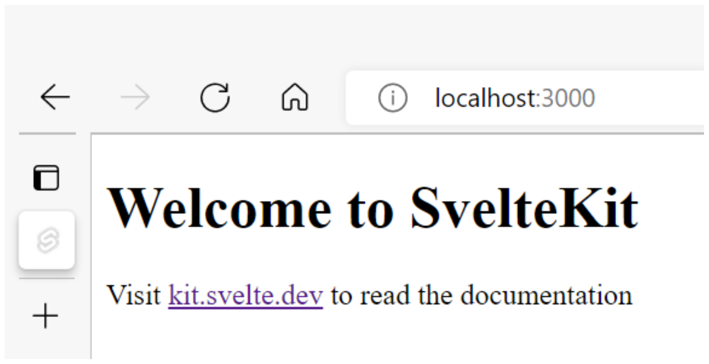
Once done, you'll be able to open your browser and point to the Local URL. In my case, this is **http://localhost:3000/**.



```
  vite v2.9.13 dev server running at:

  > Local: http://localhost:3000/
  > Network: use `--host` to expose

  ready in 578ms.
```

Figure 1-o: SvelteKit Skeleton Project Running (Built-in Terminal View)

When you open the browser, you'll see the following webpage.



Welcome to SvelteKit

Visit kit.svelte.dev to read the documentation

Figure 1-p: SvelteKit Skeleton Project Running (in the Browser)

Recap

If you've followed along until now, well done. We now have our barebones Svelte project created and ready, and we have everything we need to explore Svelte features and transform this into a practical application.

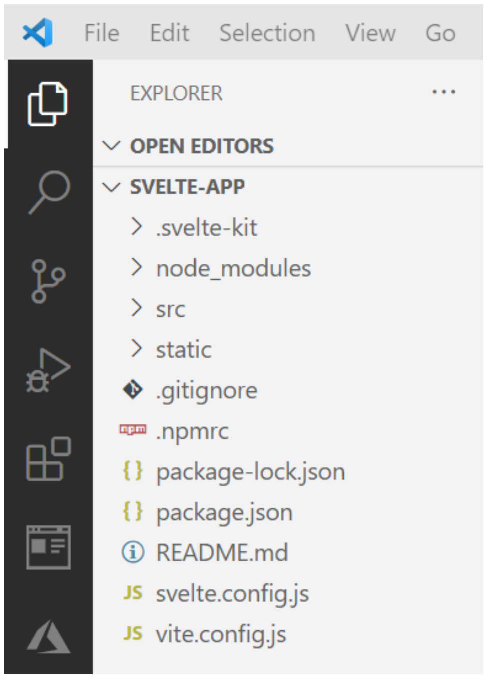
Chapter 2 Project Organization

Quick intro

Throughout this book, we'll be building a Svelte application using the barebones project we have just created, and this way, we'll also cover Svelte features as we go along. In this short chapter, we'll review the structure of the current skeleton project—this will help us organize our code later.

Project structure

By default, the Svelte project we created in the previous chapter has a project structure that we need to understand in order to add features and modify the application.



```
.svelte-kit
node_modules
src
static
.gitignore
.npmrc
package-lock.json
package.json
README.md
svelte.config.js
vite.config.js
```

Figure 2-a: Default Project Structure (VS Code)

As we add features to the application, we'll aim to keep the default project structure. In other words, we will keep all the folders and configuration files that the application creation or scaffolding process has created and add application-specific Svelte pages and component files as needed.

Several configuration files within the application's root folder allow the project to be built and executed. Let's explore these.

The vite.config.js file contains the required settings for Vite to bundle the app's code and serve it.

Code Listing 2-a: Vite Configuration (vite.config.js)

```
import { sveltekit } from '@sveltejs/kit/vite';

/** @type {import('vite').UserConfig} */
const config = {
    plugins: [sveltekit()]
};

export default config;
```

All this does is **`import`** SvelteKit (**`sveltekit`**) and add it as a Vite plugin (**`plugins: [sveltekit()]`**). The code is straightforward to grasp.

The svelte.config.js file exposes the Svelte core **`adapter`** to SvelteKit. As you can see in the following listing, the code is short and straightforward.

Code Listing 2-b: SvelteKit Configuration (svelte.config.js)

```
import adapter from '@sveltejs/adapter-auto';

/** @type {import('@sveltejs/kit').Config} */
const config = {
    kit: {
        adapter: adapter()
    }
};

export default config;
```

The README.md file, as its name implies, contains valuable information about building and deploying the SvelteKit application we've created and scaffolded.

The package.json file contains information about version dependencies and scripts used to execute and build the application, as well as the name and version of the app.

Code Listing 2-c: Project Dependencies and Scripts (package.json)

```
{
    "name": "svelte-app",
    "version": "0.0.1",
    "scripts": {
        "dev": "vite dev",
        "build": "vite build",
        "package": "svelte-kit package",
        "preview": "vite preview",
        "prepare": "svelte-kit sync"
    },
    "devDependencies": {
        "@sveltejs/adapter-auto": "next",
        "@sveltejs/kit": "next",
        "svelte": "^3.44.0",
        "vite": "^2.9.13"
    },
    "type": "module"
}
```

We can see that the script refers to Vite, which is responsible for creating the development version of the app, as well as making the final production build.

We can also see that Svelte and Vite are both development dependencies only—not actual dependencies—which means they do not get deployed with the actual application.

Beyond that, we can find the .npmrc (node package manager runtime configuration) file, a configuration file that can be used globally or at a user level to optimize the npm environment.

We can also see a .gitignore file, indicating folders and files for Git to ignore. If we look at this file, we can see what folders and files are ignored by Git.

Code Listing 2-d: The Git Ignore File (.gitignore)

```
.DS_Store
node_modules
/build
/.svelte-kit
/package
.env
.env.*
!.env.example
```

As for the specific project folders, we can see the static, src, node_modules, and .svelte-kit folders.

As its name implies, the static folder contains fixed assets that the application will use, such as the favicon.png file. The static folder is also used to include CSS files, images, and third-party JavaScript files.

We will be doing all our work in the src folder. Within the src folder, there's an app.html file, and you can also find the routes subfolder, which contains the index.svelte file, which is the application's main page.

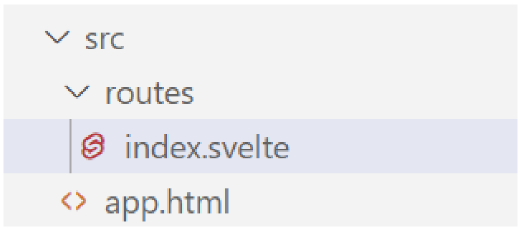


Figure 2-b: The src Folder Content

We'll come back to the src folder a bit later. For now, let's move our attention to the other remaining folders of our project. The nodes_modules folder contains all the packages that both Svelte and Vite require to work.

On the other hand, the .svelte-kit folder, as its name implies, includes the Svelte compiler and runtime files.

Recap

Now that we know how our project's folders and files are organized, we can move on to greener fields by setting up a back end, learning about Svelte, and adding features to our application.

Chapter 3 Setting Up a Back End

Quick intro

If you've read some of my other *Succinctly* series books, you've probably noticed that I'm a frequent user (and big fan) of Google Firebase.

Firebase makes it easy to get a back end up and running for any web application (independently of the front-end framework used).

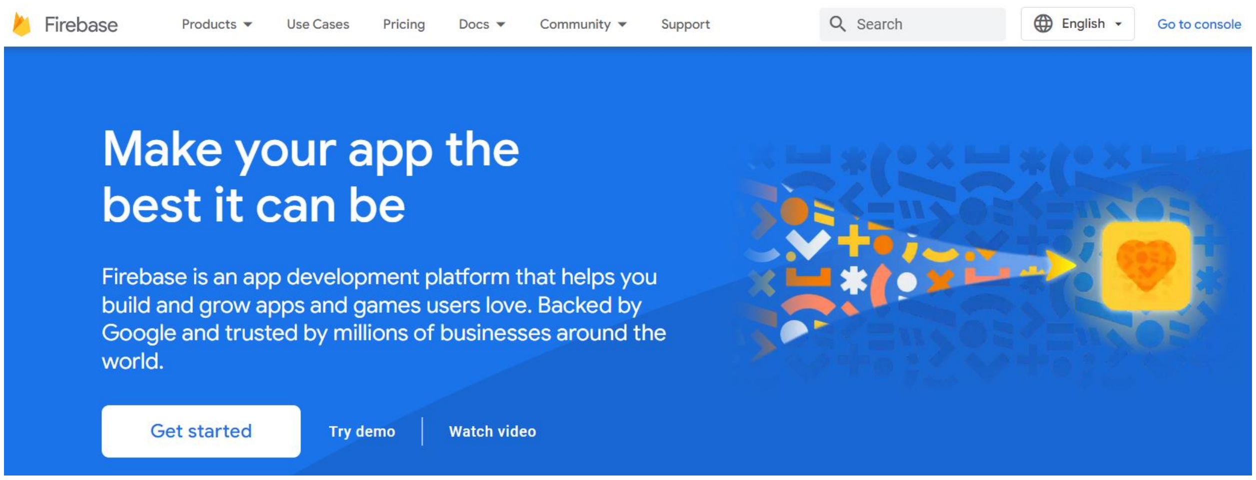

Firebase
Products
Use Cases
Pricing
Docs
Community
Support
Search
English
Go to console
Make your app the best it can be
Firebase is an app development platform that helps you build and grow apps and games users love. Backed by Google and trusted by millions of businesses around the world.
Get started
Try demo
Watch video

Figure 3-a: Firebase Front Page

Note: The Firebase webpages shown throughout this chapter might change over time. However, you should still be able to continue with the steps provided easily.

Most of the time, I leave the process of setting up the back end for the last part of my books, but this time around, I will do the opposite.

I'm going to focus on the back end and leave it ready so that in the remaining chapters, we can focus exclusively on the creation of the application with Svelte. Let's do that.

Getting started with Firebase

Getting started with Firebase is very easy. You need to be signed in with a Google Workspace or Gmail account.

On the Firebase home page, click **Get started**—this will take us to the Firebase console.

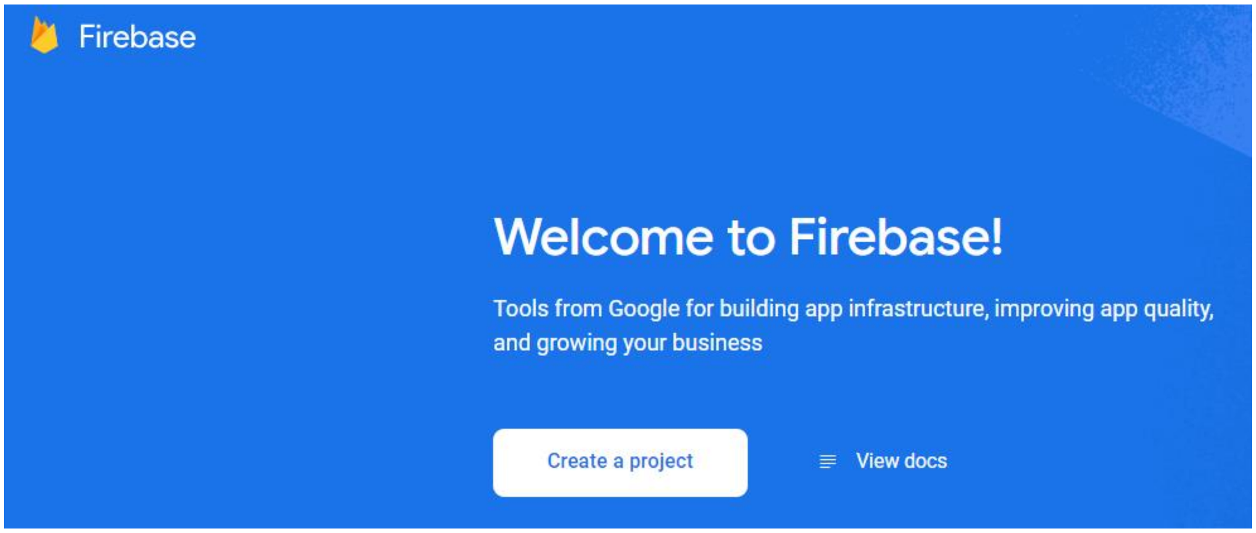

Firebase
Welcome to Firebase!
Tools from Google for building app infrastructure, improving app quality, and growing your business
Create a project
View docs

Figure 3-b: Firebase Console Front Page

Next, click **Create a project**. At this stage, we can indicate our Firebase project name. In my case, I'll be calling the project **SvelteSuccinctlyApp**.

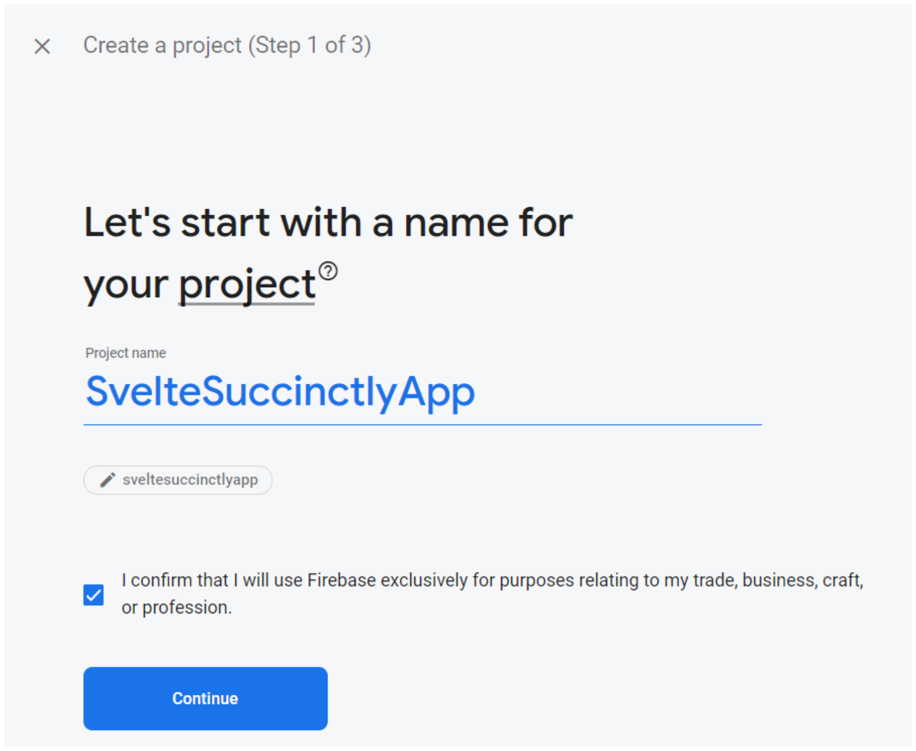
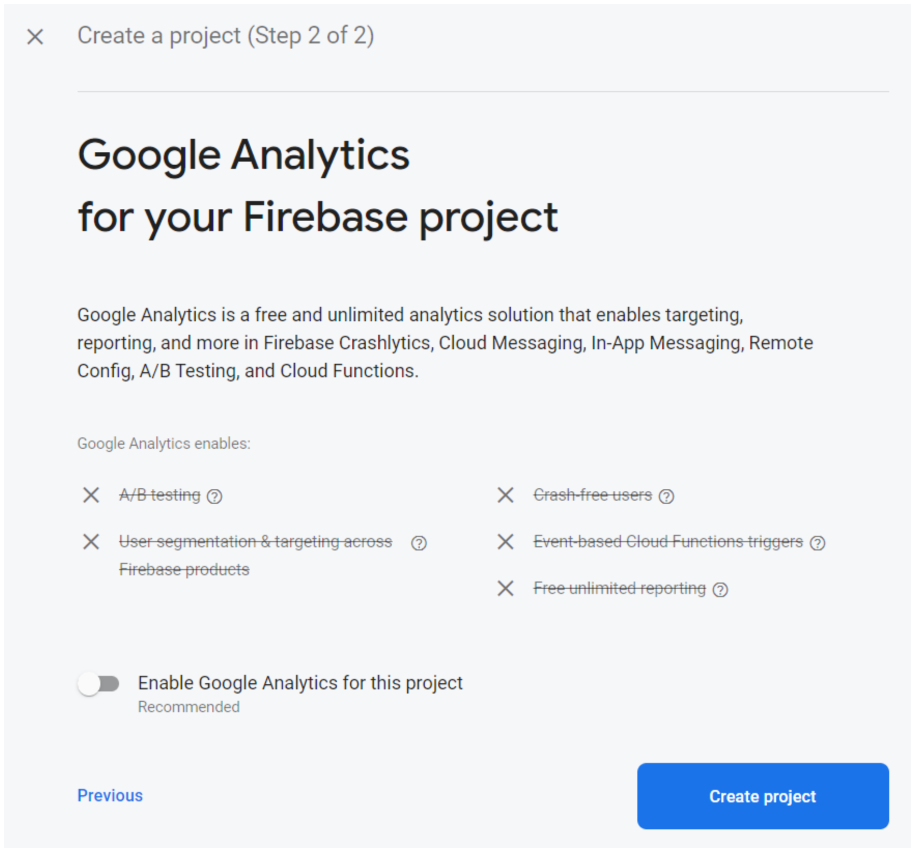

Create a project (Step 1 of 3)
Let's start with a name for your project
Project name
SvelteSuccinctlyApp
sveltesuccinctlyapp
I confirm that I will use Firebase exclusively for purposes relating to my trade, business, craft, or profession.
Continue

Figure 3-c: Creating a Firebase Project (Step 1 of 3)

Once you've entered the project name, click **Continue**.


Create a project (Step 2 of 2)

Google Analytics
for your Firebase project

Google Analytics is a free and unlimited analytics solution that enables targeting, reporting, and more in Firebase Crashlytics, Cloud Messaging, In-App Messaging, Remote Config, A/B Testing, and Cloud Functions.

Google Analytics enables:

- ~~A/B testing~~
- ~~User segmentation & targeting across Firebase products~~
- ~~Crash-free users~~
- ~~Event-based Cloud Functions triggers~~
- ~~Free unlimited reporting~~

Enable Google Analytics for this project
Recommended

Figure 3-d: Creating a Firebase Project (Step 2 of 3)

By default, Firebase enables Google Analytics for this project, but we'll disable it since we don't need it.

Now, click **Create project** to create the Firebase project.

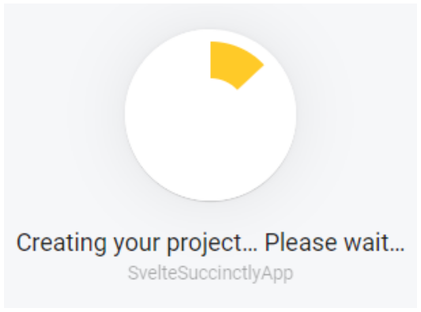


Figure 3-e: Creating a Firebase Project

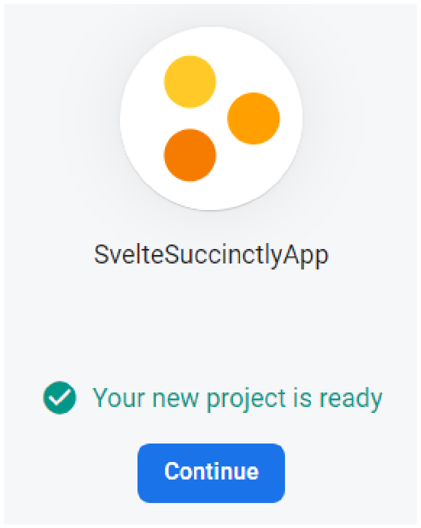


Figure 3-f: Creating a Firebase Project (Step 3 of 3)

Once the Firebase project is ready, click **Continue,** and we'll be directed to the project's overview page within the Firebase console.

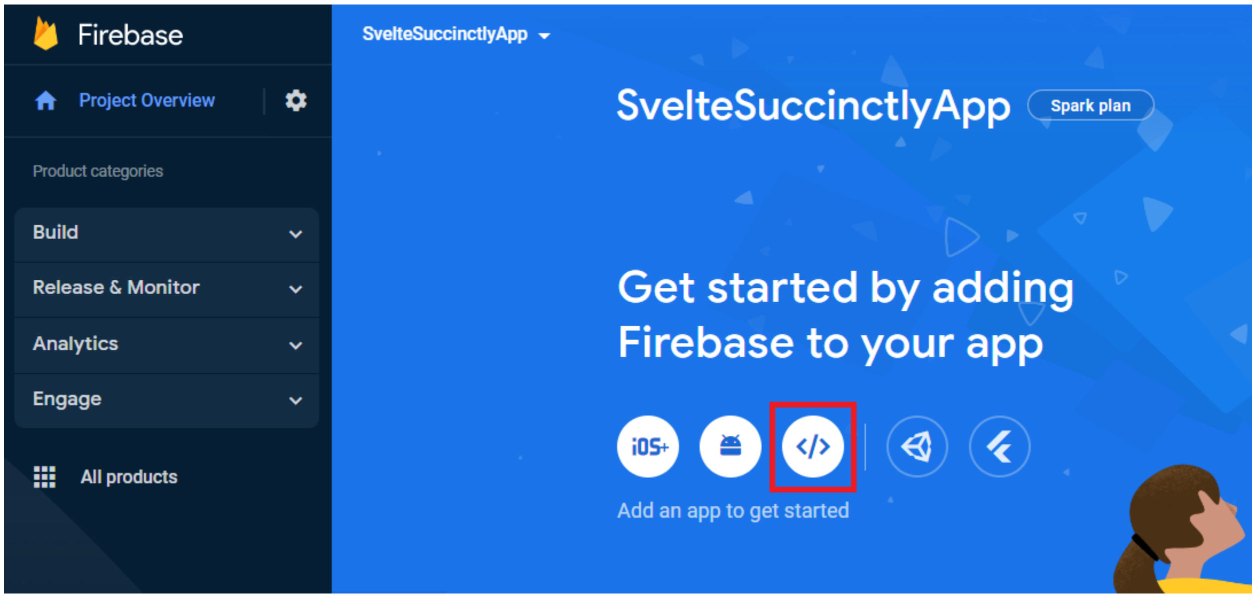
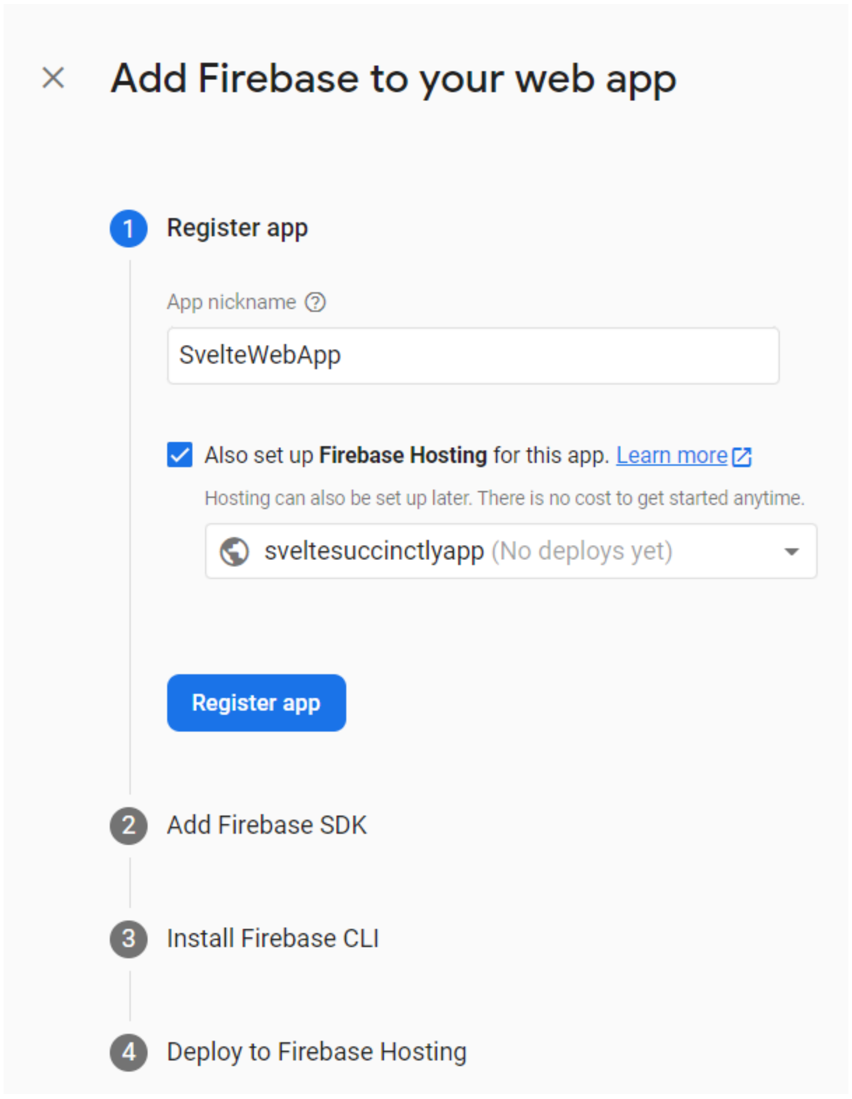


Figure 3-g: Firebase Project Overview Page

Click the web icon (highlighted in **red** in Figure 3-g) to add an app to the Firebase project. In our case, we will be building a web application using Svelte, so we need to click the web icon.

Next we'll see the **Add Firebase to your web app** page, where we can register our app. In my case, I'll name it **SvelteWebApp**.


Add Firebase to your web app

1 Register app

App nickname

SvelteWebApp

✓ Also set up **Firebase Hosting** for this app. Learn more

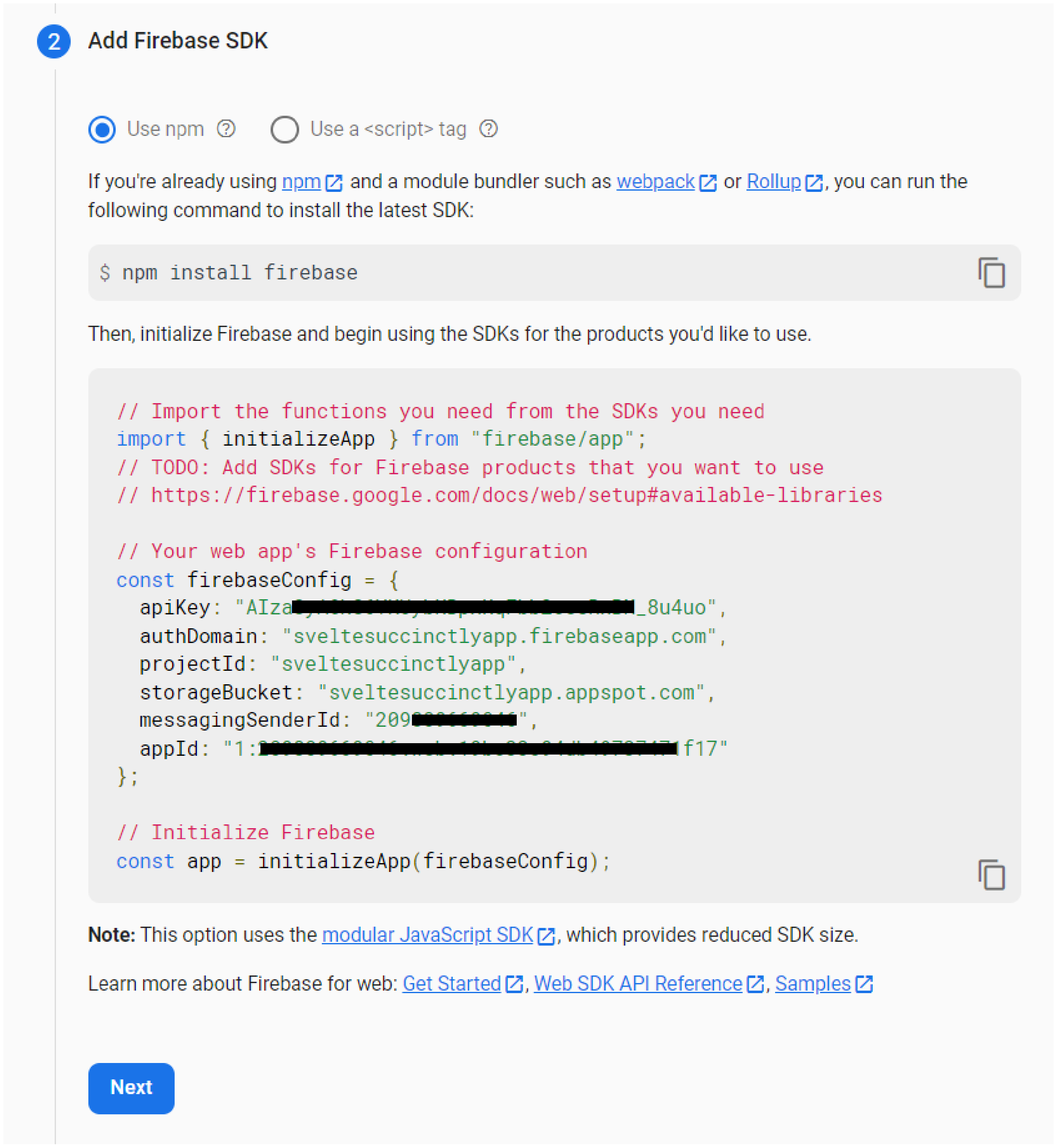
Hosting can also be set up later. There is no cost to get started anytime.

sveltesuccinctlyapp (No deploys yet)

Figure 3-h: Add Firebase to your web app

I'm also going to select the option **Also set up Firebase Hosting for this app**, which will later allow us to deploy the application to Firebase Hosting without breaking a sweat. To continue, click **Register app**.

We'll see the steps to include the Firebase SDK in our Svelte project, as shown in the following figure.



2 Add Firebase SDK

[✓ Use npm] Use a <script> tag

If you're already using npm and a module bundler such as webpack or Rollup, you can run the following command to install the latest SDK:

```
$ npm install firebase
```

Then, initialize Firebase and begin using the SDKs for the products you'd like to use.

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
const firebaseConfig = {
  apiKey: "AIza[redacted]_8u4uo",
  authDomain: "sveltesuccinctlyapp.firebaseapp.com",
  projectId: "sveltesuccinctlyapp",
  storageBucket: "sveltesuccinctlyapp.appspot.com",
  messagingSenderId: "209[redacted]",
  appId: "1:[redacted]f17"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
```

Note: This option uses the modular JavaScript SDK, which provides reduced SDK size.

Learn more about Firebase for web: Get Started, Web SDK API Reference, Samples

Figure 3-i: Add Firebase SDK

Note: I've hidden the `apiKey`, `messagingSenderId`, and `appId`, as these are specific to my Firebase environment and cannot be shared. For your Firebase environment, you'll have different values.

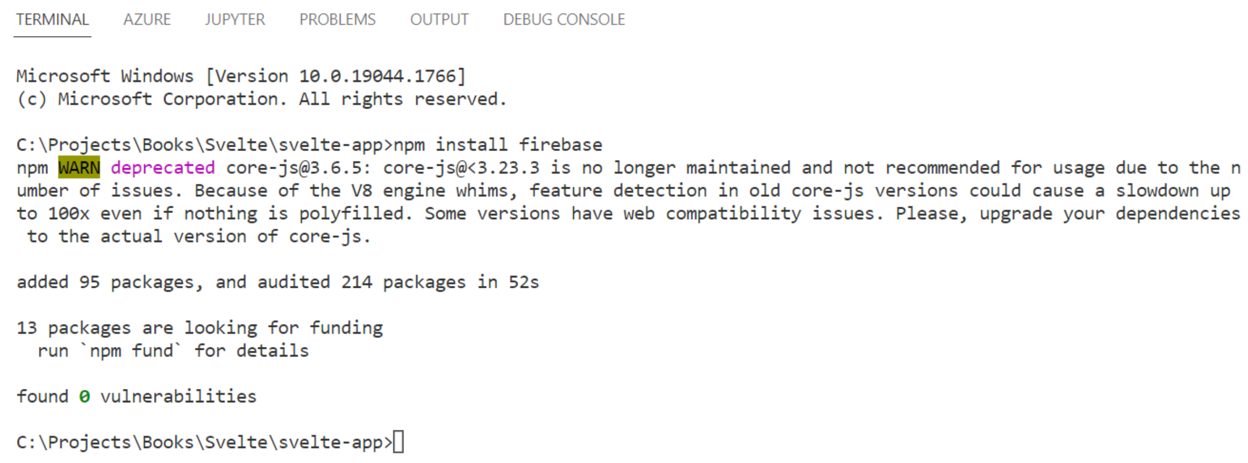
Adding the Firebase SDK includes two steps. The first step is to install Firebase in the project by executing the **`npm install firebase`** command.

Let's switch over to our project in VS Code and execute the following command using the built-in terminal.

Code Listing 3-a: Installing Firebase in Our Project

```
npm install firebase
```

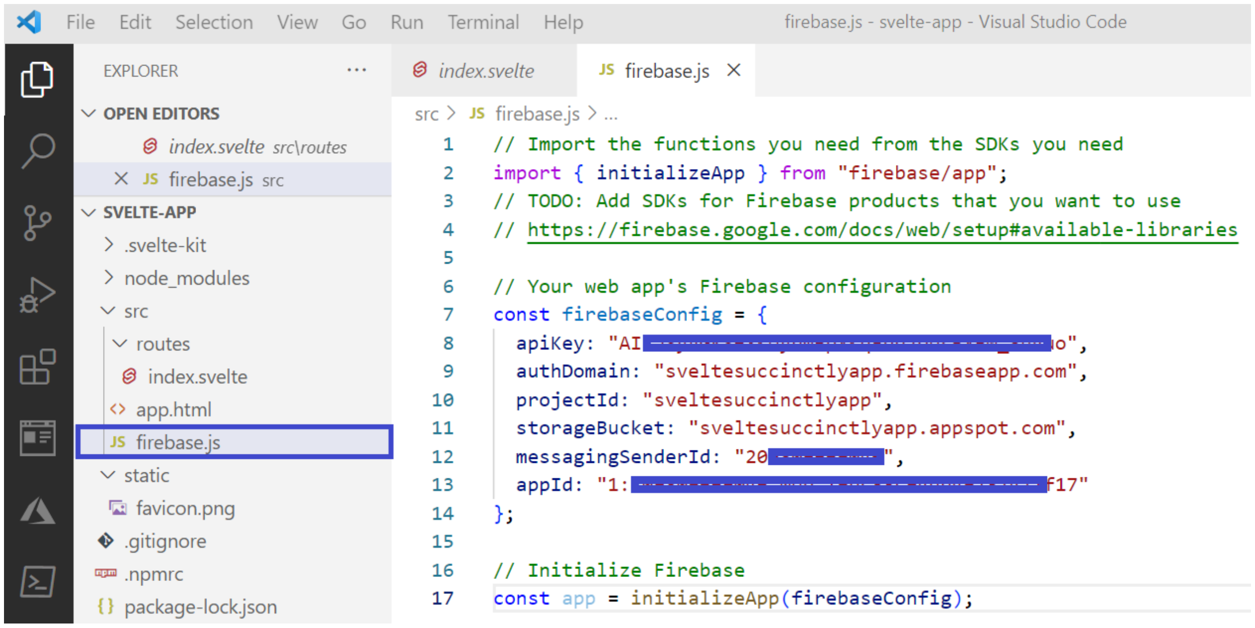
After executing this command, you'll see within the built-in terminal an output similar to the following.



```
Microsoft Windows [Version 10.0.19044.1766]
(c) Microsoft Corporation. All rights reserved.

C:\Projects\Books\Svelte\svelte-app>npm install firebase
npm WARN deprecated core-js@3.6.5: core-js@<3.23.3 is no longer maintained and not recommended for usage due to the n
umber of issues. Because of the V8 engine whims, feature detection in old core-js versions could cause a slowdown up
to 100x even if nothing is polyfilled. Some versions have web compatibility issues. Please, upgrade your dependencies
 to the actual version of core-js.

added 95 packages, and audited 214 packages in 52s

13 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

C:\Projects\Books\Svelte\svelte-app>
```

Figure 3-j: Installing Firebase within VS Code

With the Firebase SDK installed, let's initialize Firebase. To do that, let's create within the src folder of our Svelte project a new file called firebase.js. Copy the code shown in Figure 3-i. This is how it appears in VS Code on my environment.



```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
const firebaseConfig = {
  apiKey: "AI[redacted]o",
  authDomain: "sveltesuccinctlyapp.firebaseapp.com",
  projectId: "sveltesuccinctlyapp",
  storageBucket: "sveltesuccinctlyapp.appspot.com",
  messagingSenderId: "20[redacted]",
  appId: "1:[redacted]f17"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
```

Figure 3-k: The firebase.js File (VS Code)

Note: I've hidden the `apiKey`***,*** `messagingSenderId`***, and*** `appId`***, as these are specific to my Firebase environment and cannot be shared. For your Firebase environment, you'll have different values.***

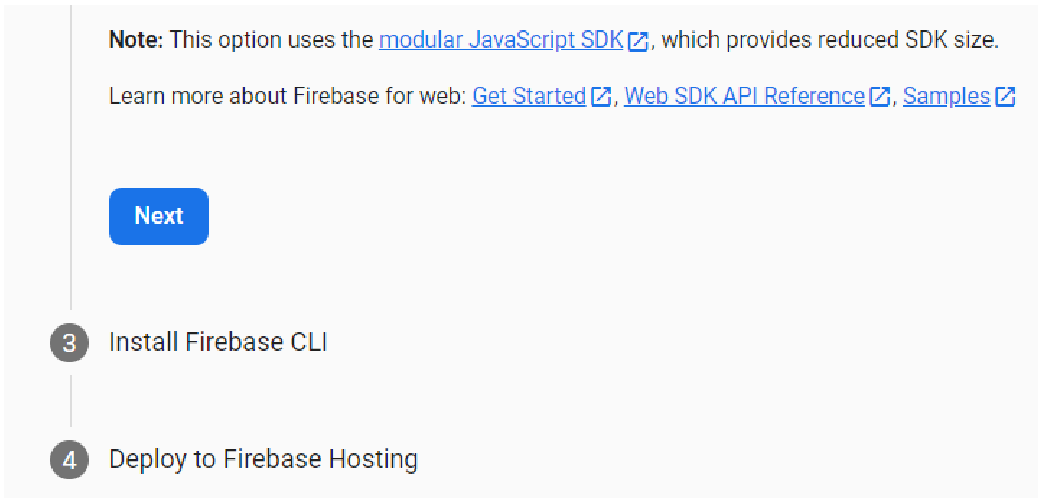
Here is the actual firebase.js code.

Code Listing 3-b: The firebase.js Code

```
import { initializeApp } from "firebase/app";

const firebaseConfig = {
  apiKey: "Aiz...",
  authDomain: "sveltesuccinctlyapp.firebaseapp.com",
  projectId: "sveltesuccinctlyapp",
  storageBucket: "sveltesuccinctlyapp.appspot.com",
  messagingSenderId: "209...",
  appId: "1:209..."
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
```

Note: I've removed the `apiKey`***,*** `messagingSenderId`***, and*** `appId` ***values (highlighted in yellow), as these are specific to my Firebase environment and cannot be shared. For your Firebase environment, you'll have different values.***

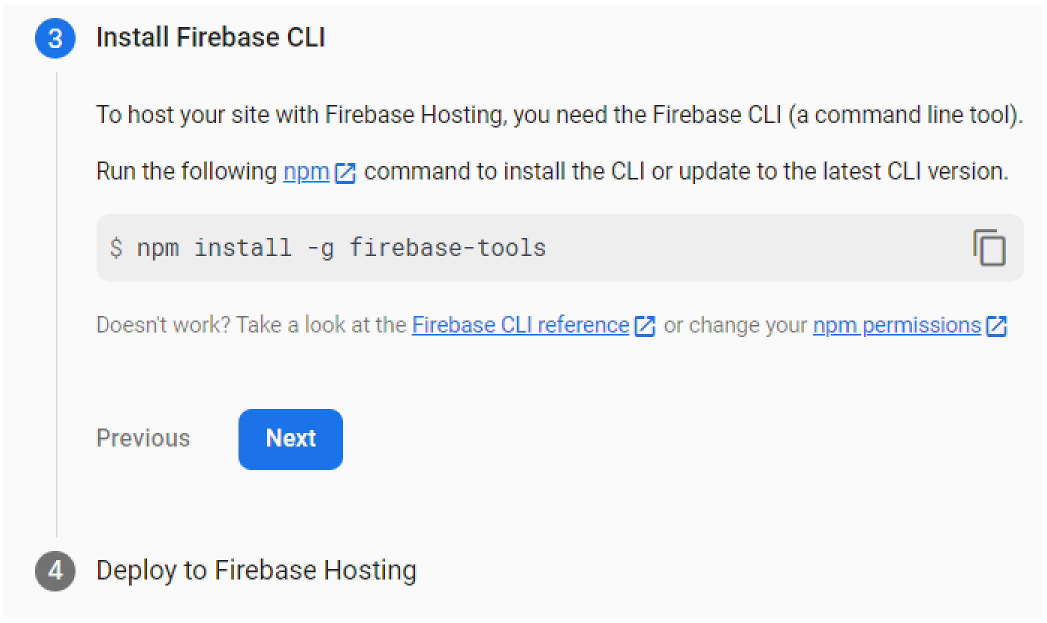
Now that we've set up the Firebase SDK and created the firebase.js file within VS Code, we can click **Next** within the Firebase console.


Note: This option uses the modular JavaScript SDK, which provides reduced SDK size.

Learn more about Firebase for web: Get Started, Web SDK API Reference, Samples

Figure 3-l: Add Firebase SDK (Final Step)

The next step is to install the Firebase CLI, which will allow you to deploy the application to Firebase Hosting later (if you wish to do that).


3 Install Firebase CLI

To host your site with Firebase Hosting, you need the Firebase CLI (a command line tool).

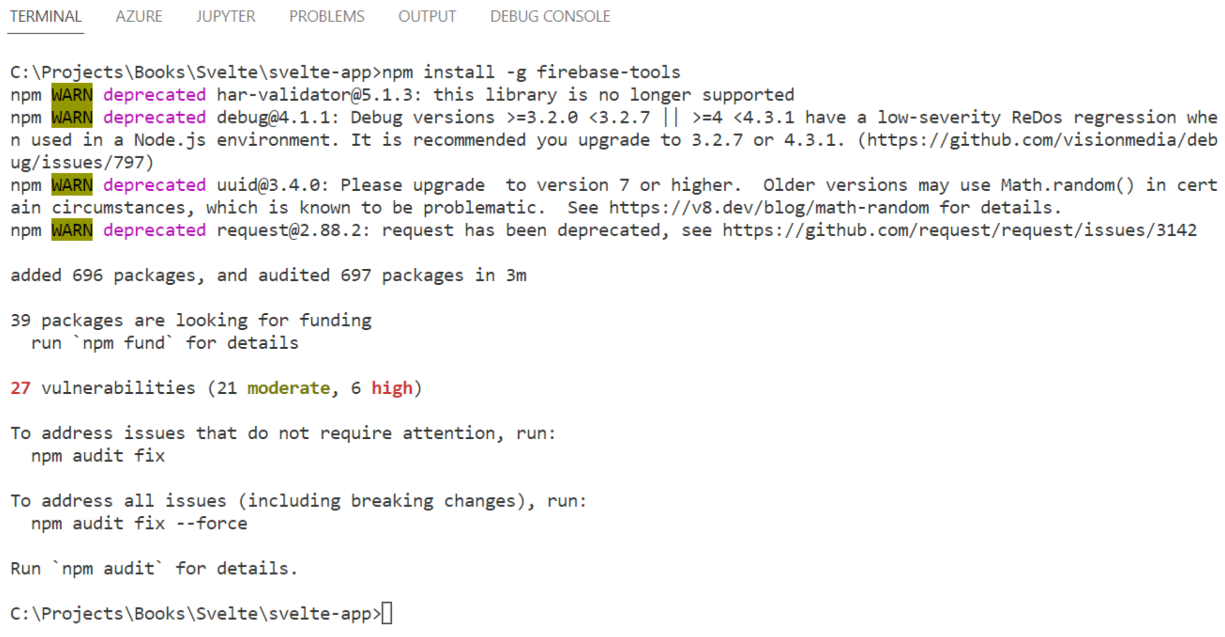
Run the following npm command to install the CLI or update to the latest CLI version.

```
$ npm install -g firebase-tools
```

Doesn't work? Take a look at the Firebase CLI reference or change your npm permissions

Figure 3-m: Install Firebase CLI

Let's switch back to VS Code, and within the built-in terminal, execute the **`npm install -g firebase-tools`** command.



```
C:\Projects\Books\Svelte\svelte-app>npm install -g firebase-tools
npm WARN deprecated har-validator@5.1.3: this library is no longer supported
npm WARN deprecated debug@4.1.1: Debug versions >=3.2.0 <3.2.7 || >=4 <4.3.1 have a low-severity ReDos regression whe
n used in a Node.js environment. It is recommended you upgrade to 3.2.7 or 4.3.1. (https://github.com/visionmedia/deb
ug/issues/797)
npm WARN deprecated uuid@3.4.0: Please upgrade  to version 7 or higher.  Older versions may use Math.random() in cert
ain circumstances, which is known to be problematic.  See https://v8.dev/blog/math-random for details.
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142

added 696 packages, and audited 697 packages in 3m

39 packages are looking for funding
  run `npm fund` for details

27 vulnerabilities (21 moderate, 6 high)

To address issues that do not require attention, run:
  npm audit fix

To address all issues (including breaking changes), run:
  npm audit fix --force

Run `npm audit` for details.

C:\Projects\Books\Svelte\svelte-app>
```

Figure 3-n: Installing the Firebase CLI

Once the Firebase CLI is installed, let's click **Next** in the Firebase console.

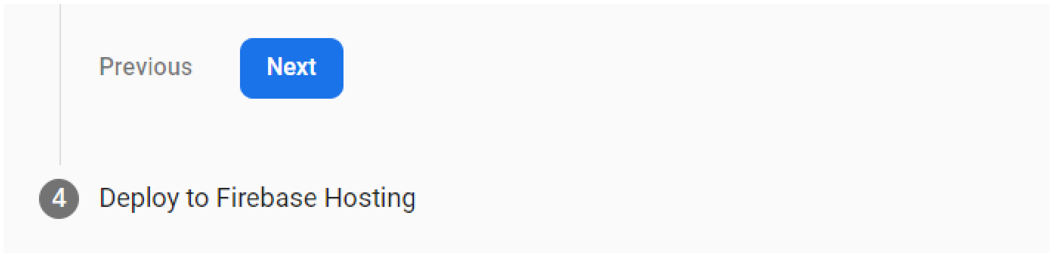
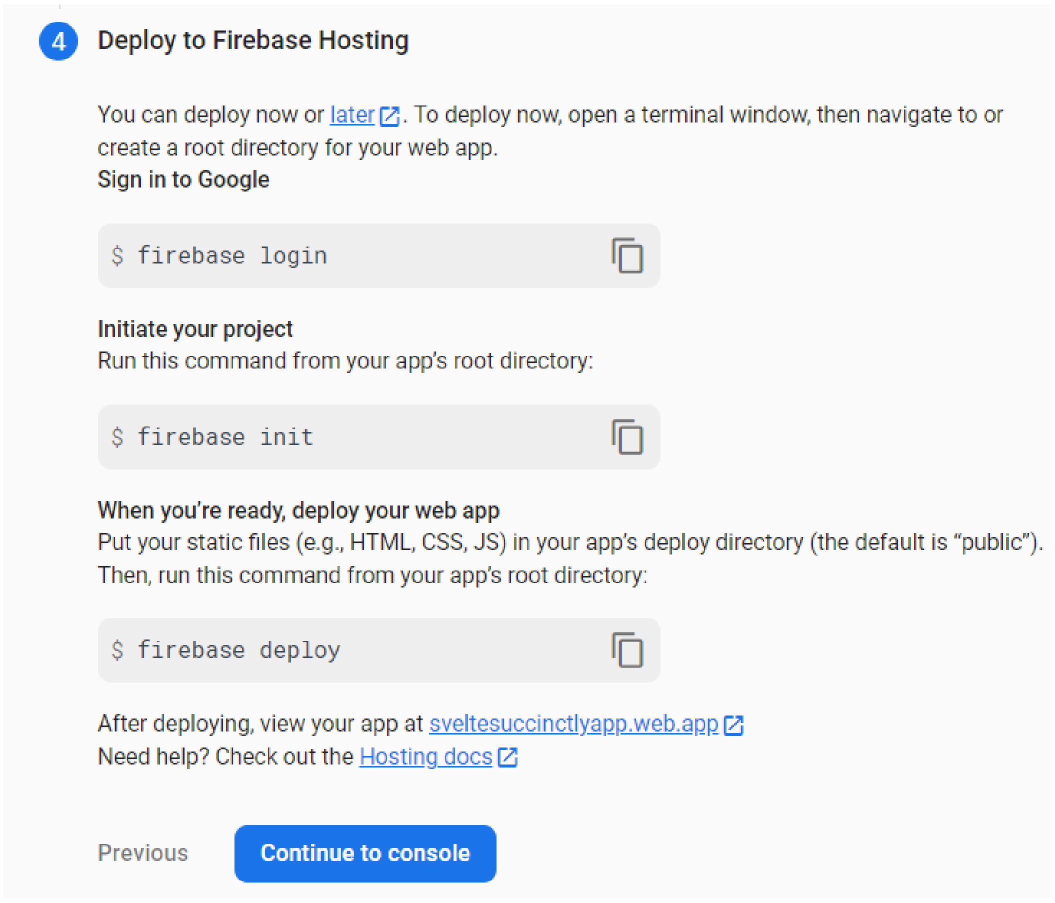


Figure 3-o: Installing the Firebase CLI (Final Step)

We'll now be able to see the following.


4 Deploy to Firebase Hosting

You can deploy now or later. To deploy now, open a terminal window, then navigate to or create a root directory for your web app.

Sign in to Google

```
$ firebase login
```

Initiate your project

Run this command from your app's root directory:

```
$ firebase init
```

When you're ready, deploy your web app

Put your static files (e.g., HTML, CSS, JS) in your app's deploy directory (the default is "public"). Then, run this command from your app's root directory:

```
$ firebase deploy
```

After deploying, view your app at sveltesuccinctlyapp.web.app

Need help? Check out the Hosting docs

Figure 3-p: How to Deploy to Firebase Hosting

Within this step, you'll see the commands you can later use to deploy the Svelte application to Firebase Hosting (if you wish to do so).

Note: Copy these commands individually and save them, so they can be used later to deploy to Firebase Hosting the Svelte application you will build.

To finalize this phase, let's click **Continue to console**. Notice that the application appears on the console's main page (highlighted in **red**).

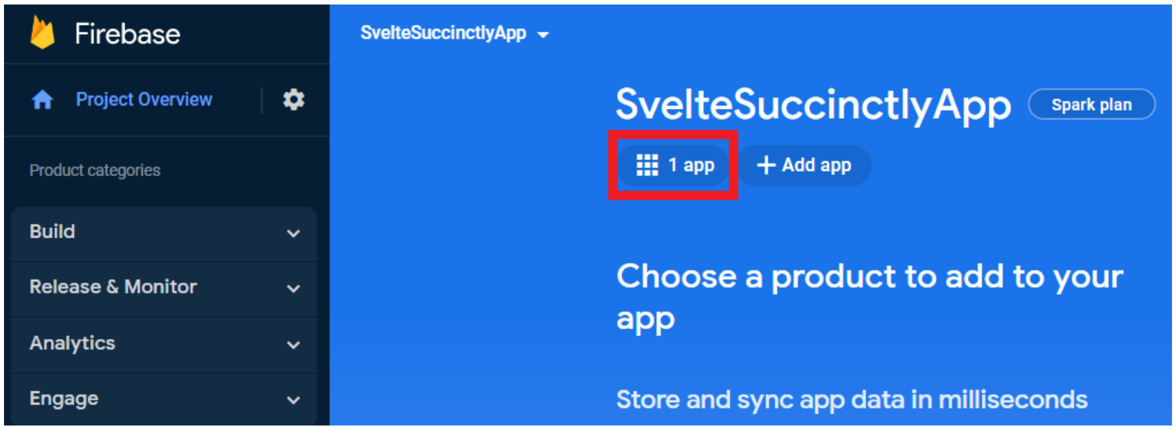


Figure 3-q: Firebase Console (With the App Created)

Creating a datastore

Now that we have created the Firebase application, we need a Firestore database, where the application's data will be kept. So, click this option and then click **Create database**.

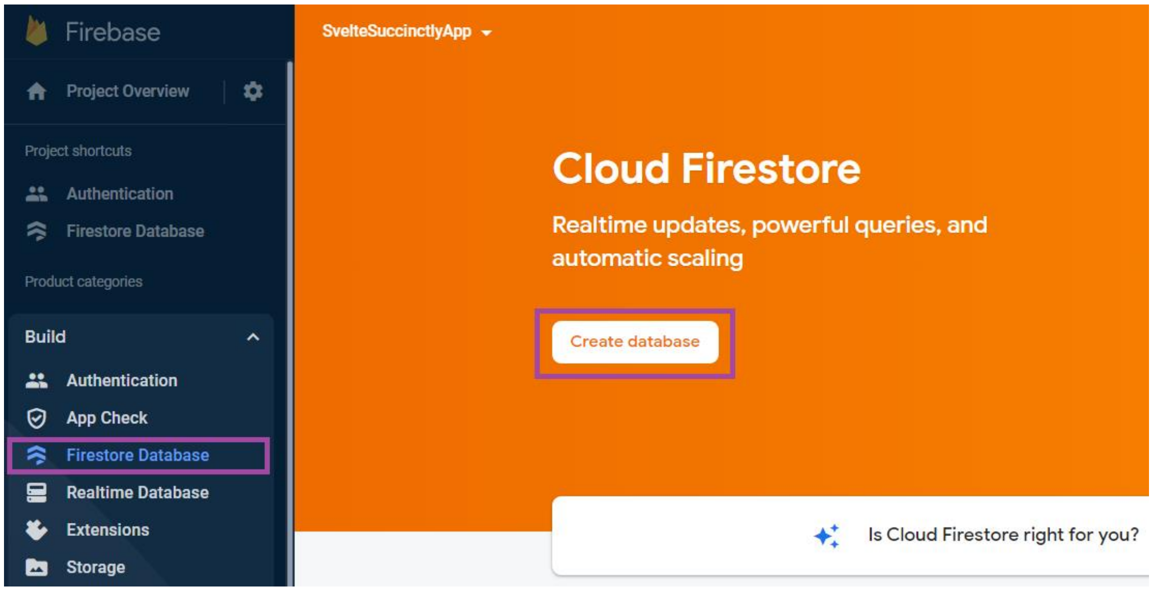
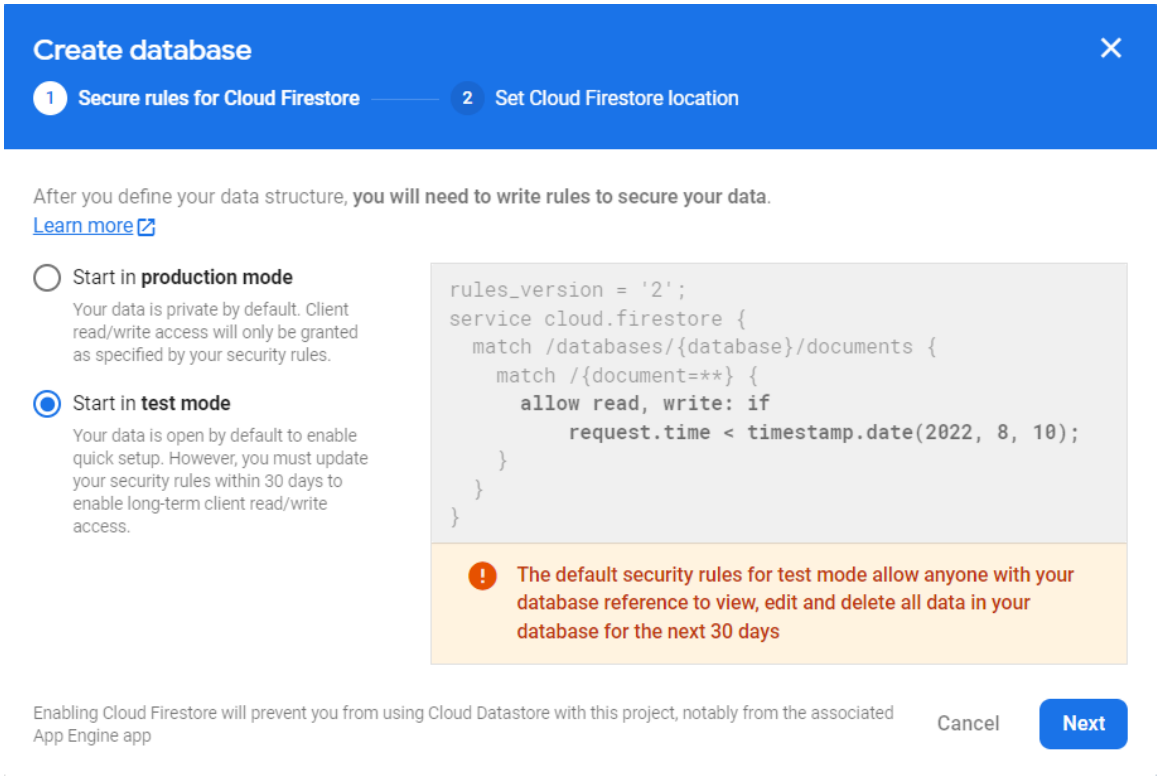


Figure 3-r: Firebase Console (The Firebase Database Option—Cloud Firestore)

We'll be presented with the following screen.


Create database
1 Secure rules for Cloud Firestore
2 Set Cloud Firestore location

After you define your data structure, **you will need to write rules to secure your data**.
Learn more

( ) Start in **production mode**
Your data is private by default. Client read/write access will only be granted as specified by your security rules.

[✓ Start in **test mode**]
Your data is open by default to enable quick setup. However, you must update your security rules within 30 days to enable long-term client read/write access.

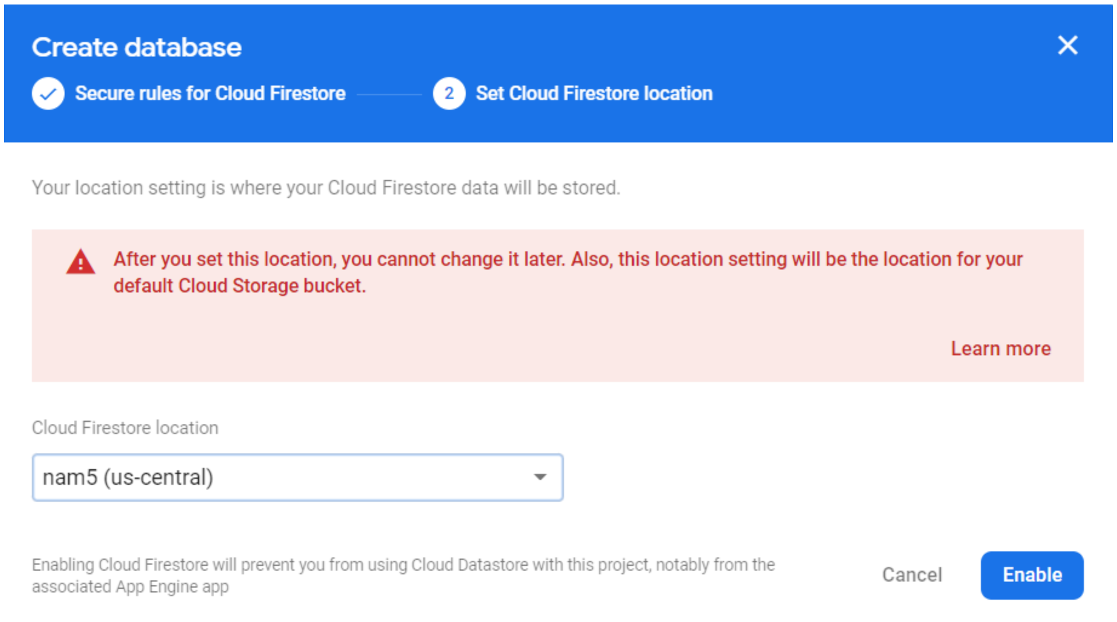
```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if
          request.time < timestamp.date(2022, 8, 10);
    }
  }
}
```

The default security rules for test mode allow anyone with your database reference to view, edit and delete all data in your database for the next 30 days

Enabling Cloud Firestore will prevent you from using Cloud Datastore with this project, notably from the associated App Engine app

Figure 3-s: Firebase Console (Create database—Step 1)

Make sure you select the option **Start in test mode** and click **Next**. You'll see the following screen.


Create database
Secure rules for Cloud Firestore
2 Set Cloud Firestore location

Your location setting is where your Cloud Firestore data will be stored.

After you set this location, you cannot change it later. Also, this location setting will be the location for your default Cloud Storage bucket.

Learn more

Cloud Firestore location: nam5 (us-central)

Enabling Cloud Firestore will prevent you from using Cloud Datastore with this project, notably from the associated App Engine app

Figure 3-t: Firebase Console (Create database—Step 2)

At this point, you can choose a different option for the Cloud Firestore location. In my case, I will go with the default option with which I've been presented.

If you would like your database hosted in a different region, please select another location now, since you won't be able to change it later.

Click **Enable**, and you'll be directed to the following screen.

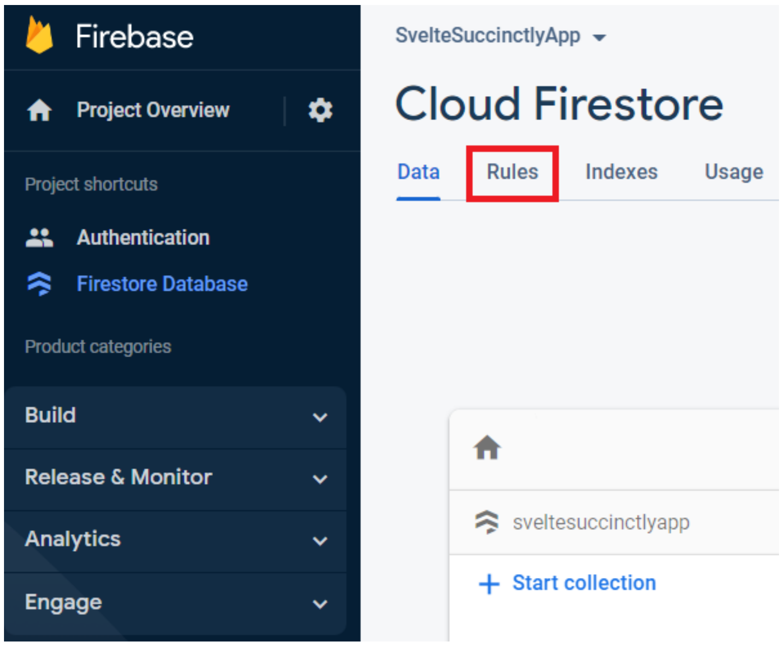


Figure 3-u: Firebase Console—Cloud Firestore Dashboard

Setting permissions

Our application must handle multiple users, so we must ensure that the correct permissions are in place. To do that, click the **Rules** tab (as highlighted in **red** in Figure 3-u).

Now you'll be able to change the default rule. We'll set the rule to **Allow authenticated access on all collections**, which is nicely explained on this Stack Overflow thread.

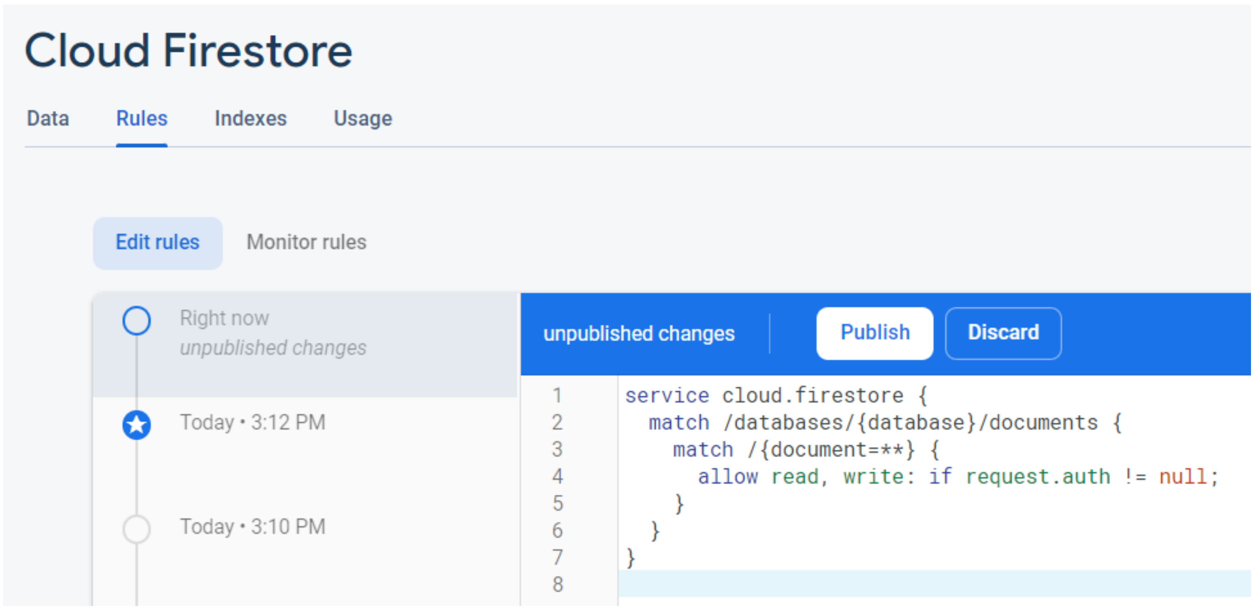
Changing the rule is easy, so copy the snippet from the following code listing (the same one I used on my Firebase environment) and paste it into the editor on the Firebase console.

Code Listing 3-c: New Rule—Allow Authenticated Access on All Collections

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth != null;
    }
```

```
   }
}
```

The editor will detect unpublished changes, and you must click **Publish**.



```
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Figure 3-v: Firebase Console—Cloud Firestore (Edit Rules)

Enabling authentication

With the correct permissions in place, the next thing to do is enable user authentication. To do that, click the **Authentication** option, then click on the **Sign-in method** tab > **Add new provider** > **Email/Password**, as highlighted in **red** in the following figure.

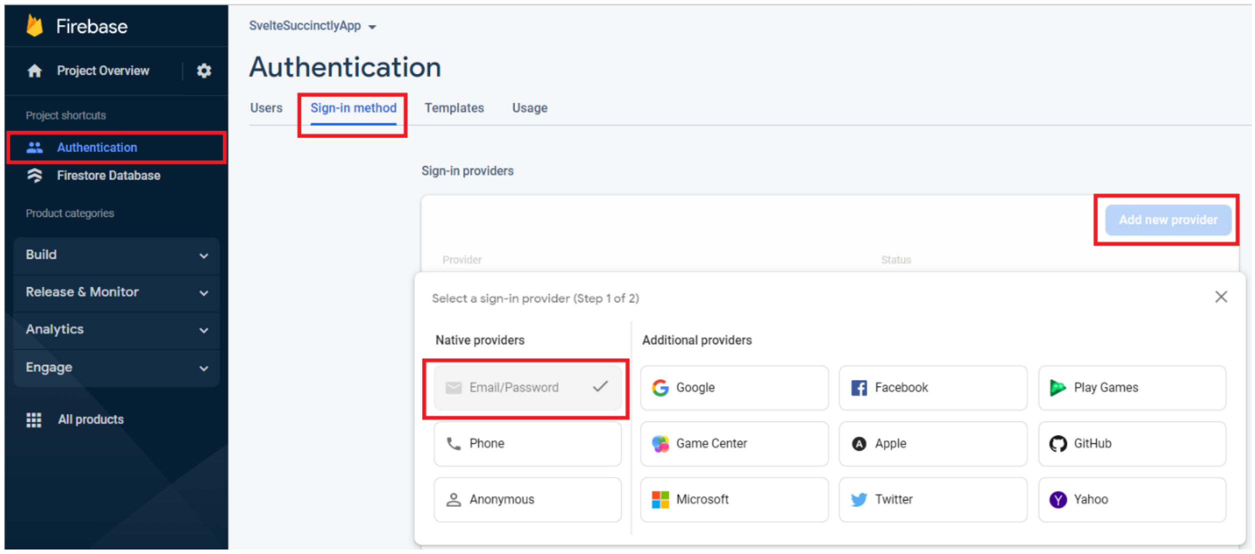
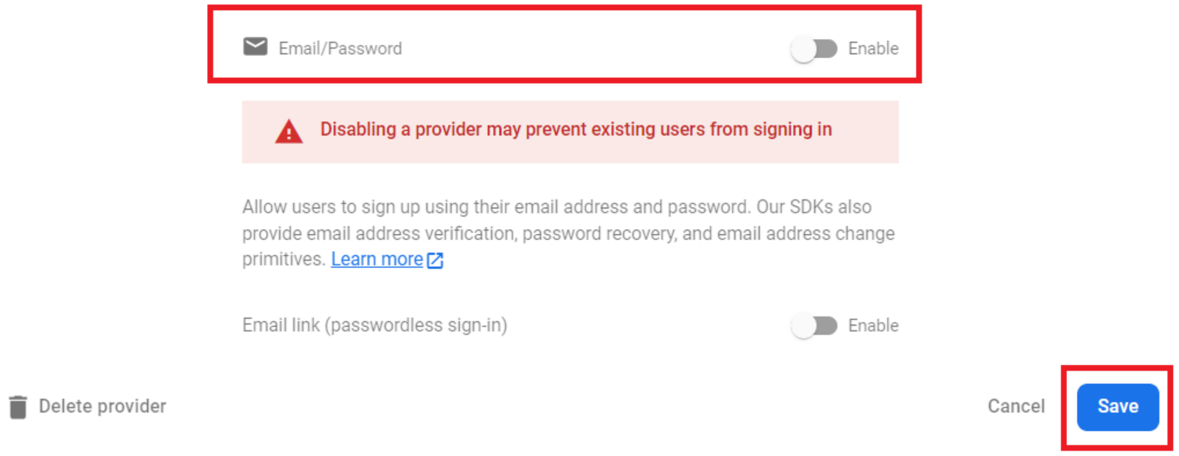


Figure 3-w: Firebase Console—Enabling Email/Password Authentication (Step 1)

Click **Enable** and **Save**, as highlighted in **red** in the following figure. There's no need to enable the Email link (passwordless sign-in) option, since we won't use this feature.


Email/Password
Enable
Disabling a provider may prevent existing users from signing in

Allow users to sign up using their email address and password. Our SDKs also provide email address verification, password recovery, and email address change primitives. Learn more

Email link (passwordless sign-in)
Enable
Delete provider
Cancel
Save

Figure 3-x: Firebase Console—Enabling Email/Password Authentication (Step 2)

Now we have all our back end configured, and there's nothing else we need to do with Firebase except add code for our app to work with it (which we'll see later).

Recap

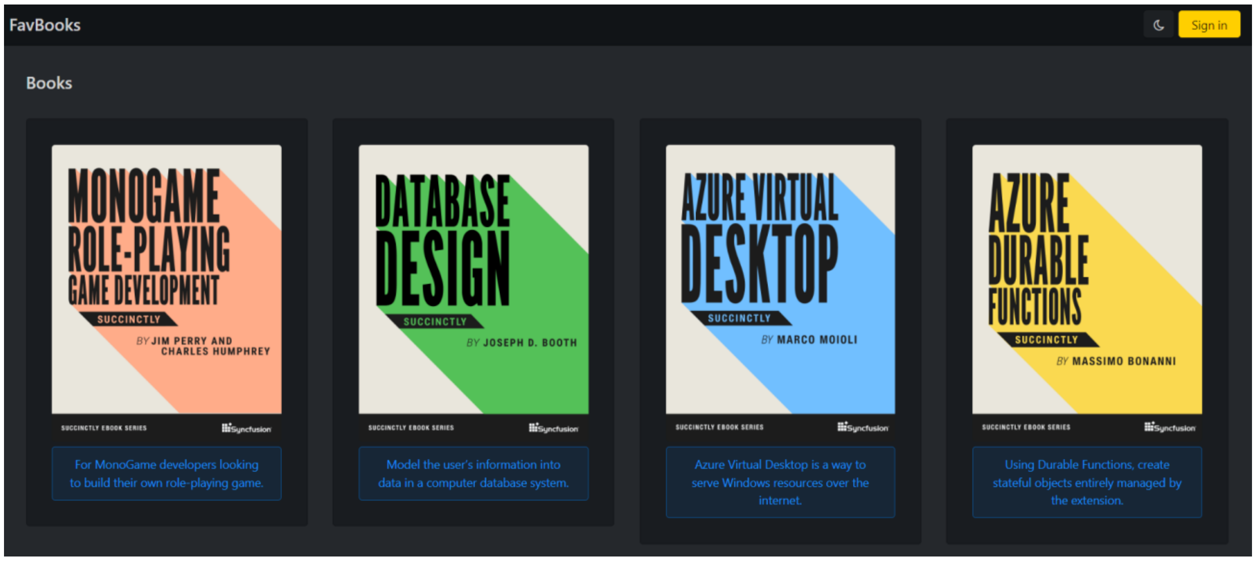
We've created the back end that our Svelte application will use—well done! Next, we'll start exploring the features of our application.

Chapter 4 Finished App Features

The finished app

We are going to create an application that keeps a list of our favorite *Succinctly* books.

You can find this application's finished and complete source code in this GitHub repository. Before jumping into the code and what it does, let's explore the finished application.


FavBooks
Sign in
Books
MONOGAME ROLE-PLAYING GAME DEVELOPMENT SUCCINCTLY BY JIM PERRY AND CHARLES HUMPHREY
DATABASE DESIGN SUCCINCTLY BY JOSEPH D. BOOTH
AZURE VIRTUAL DESKTOP SUCCINCTLY BY MARCO MOIOLI
AZURE DURABLE FUNCTIONS SUCCINCTLY BY MASSIMO BONANNI

For MonoGame developers looking to build their own role-playing game.

Model the user's information into data in a computer database system.

Azure Virtual Desktop is a way to serve Windows resources over the internet.

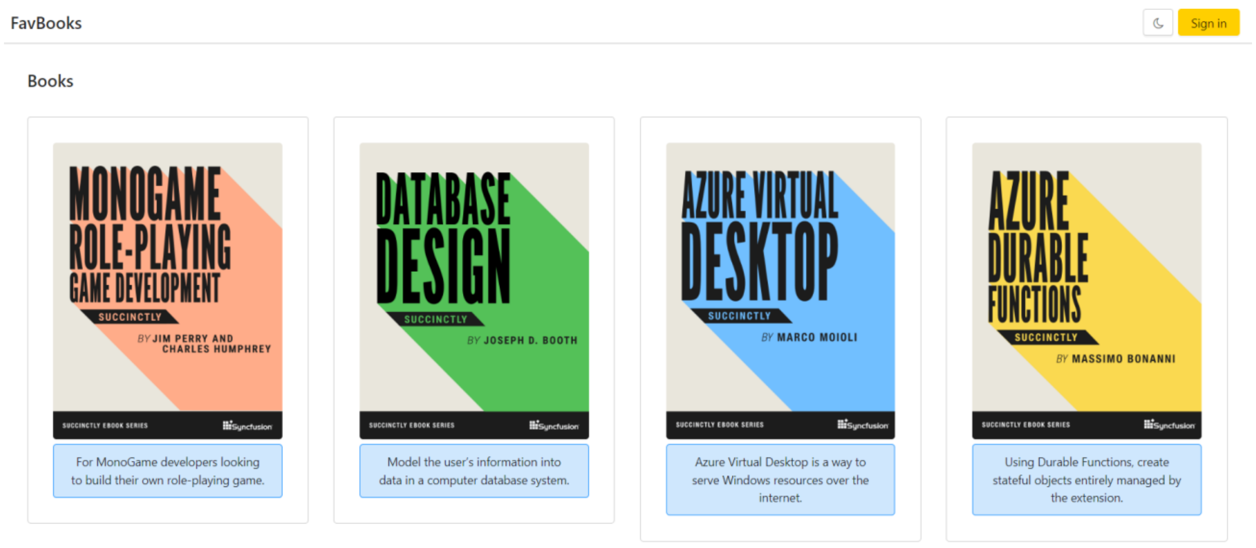
Using Durable Functions, create stateful objects entirely managed by the extension.

Figure 4-a: The Finished App (Main Page Using the Default Dark Mode)

The application has a main page that contains a static list of the most recent *Succinctly* books.

The list of the most recent books is obtained through an API endpoint that the application serves and returns as JSON.

You can also easily toggle between dark and light mode by clicking on the “moon” icon next to the Sign in button.


FavBooks

Books

For MonoGame developers looking to build their own role-playing game.

Model the user's information into data in a computer database system.

Azure Virtual Desktop is a way to serve Windows resources over the internet.

Using Durable Functions, create stateful objects entirely managed by the extension.

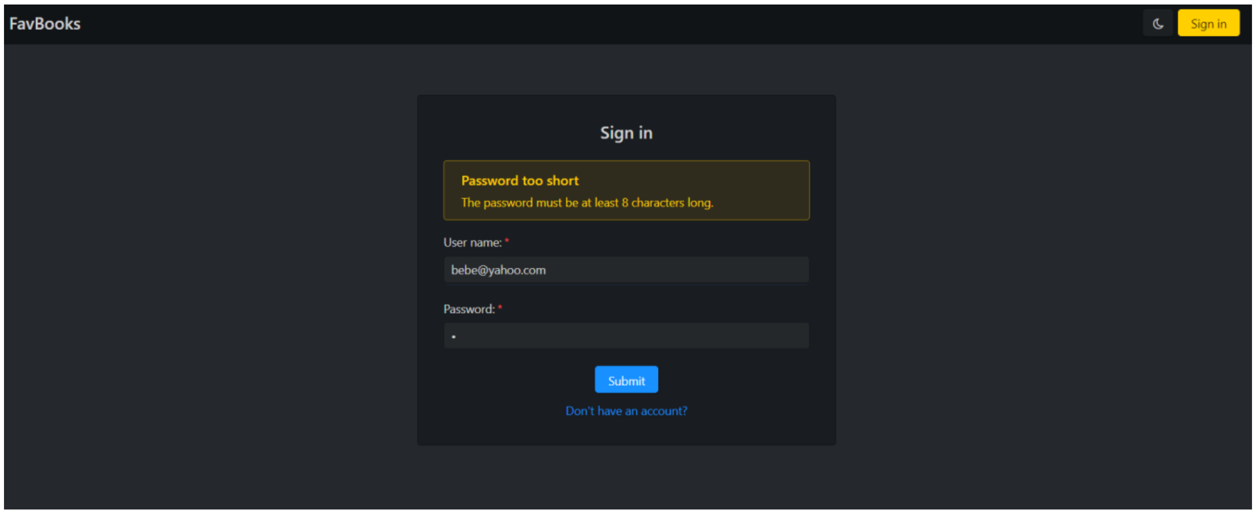
Figure 4-b: The Finished App (Main Page Using Light Mode)

The application comes with built-in authentication powered by Firebase, which means that if you sign up, you'll then become a user of the app and be able to sign in and add any of those books to your list of favorites.

The finished Sign in page

So, the finished Sign in page comes with built-in validation. Validation is executed to ensure that the user name (email address) is properly formatted (is an actual email address).

Validation also ensures that the database's user name and password combination exist, and the user can be retrieved from Firebase and authenticated.


FavBooks

Sign in

Password too short
The password must be at least 8 characters long.

User name: *
bebe@yahoo.com

Password: *
•

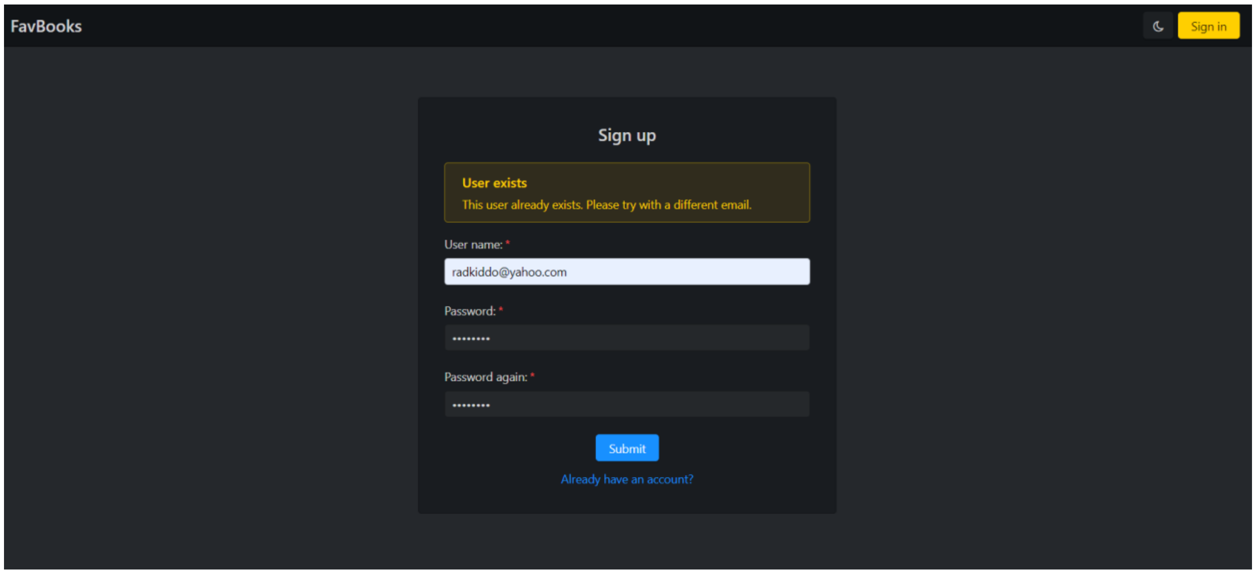
Figure 4-c: The Sign in Page

Note: I refer to this page as “Sign in” rather than “Sign-in” (because of the caption on the button on the top-right navigation bar), which is grammatically the correct way of writing it. The page is only accessible if you have not signed in. If you attempt to manually type in the URL of this page in the browser and you’re signed in, you’ll be redirected to the Favorites page.

The finished Sign up page

The Sign up page is almost identical to the Sign in page, except an additional password field is used to re-enter the password (to ensure they match).

The Sign up page also comes with full, built-in validation functionality and ensures that two users with the same email address cannot sign up. This functionality is visible in the following figure.



User exists
This user already exists. Please try with a different email.

User name: radkiddo@yahoo.com

Password: ••••••••

Password again: ••••••••

Figure 4-d: The Sign up Page

The finished main page (signed in)

Once we have an account and are signed in to the application, you’ll notice an Add to favorites button below each book, which allows you to add the book to your favorites list.

The Add to favorites button for each book is not visible if you are not signed in.

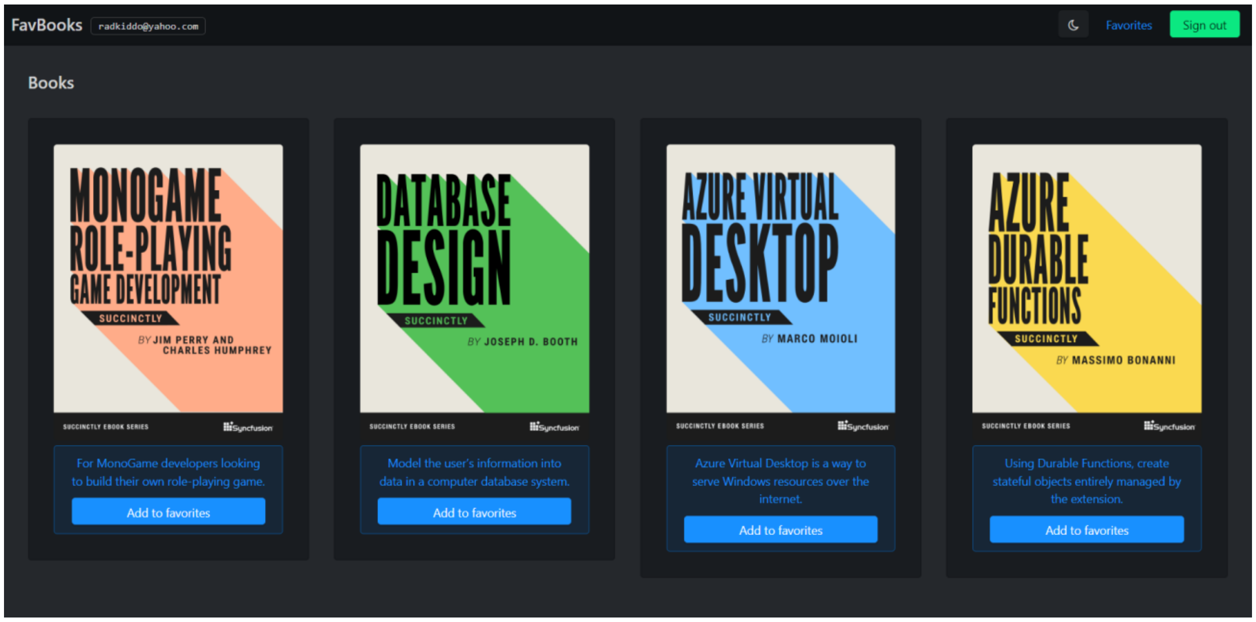

FavBooks
radkiddo@yahoo.com
Favorites
Sign out
Books
For MonoGame developers looking to build their own role-playing game.
Add to favorites
Model the user's information into data in a computer database system.
Add to favorites
Azure Virtual Desktop is a way to serve Windows resources over the internet.
Add to favorites
Using Durable Functions, create stateful objects entirely managed by the extension.
Add to favorites

Figure 4-e: The App's Main Page (Signed In)

When you click the **Add to favorites** button for a particular book, you'll add that book to your favorites list.

Notice that on this page, you'll also see the user name of the signed in user on the top-left side of the navigation bar, next to the application's name.

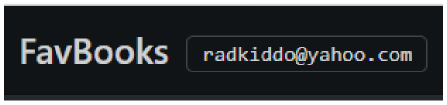


Figure 4-f: The Signed-In User Name (Top-Left Navigation Bar—Main Page Signed In)

Notice that on the top-right part of the navigation bar, the Sign in button (which used to have a yellow background color) now reads as Sign out (with a green background color).

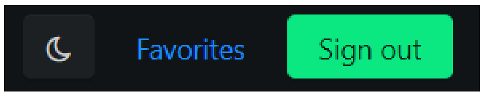


Figure 4-g: Favorites Link and Sign out Button (Top-right Navigation Bar—Main Page Signed In)

If you click **Sign out**, you'll be signed out of the application and routed back to the main page (and the Add to favorites buttons will no longer be visible).

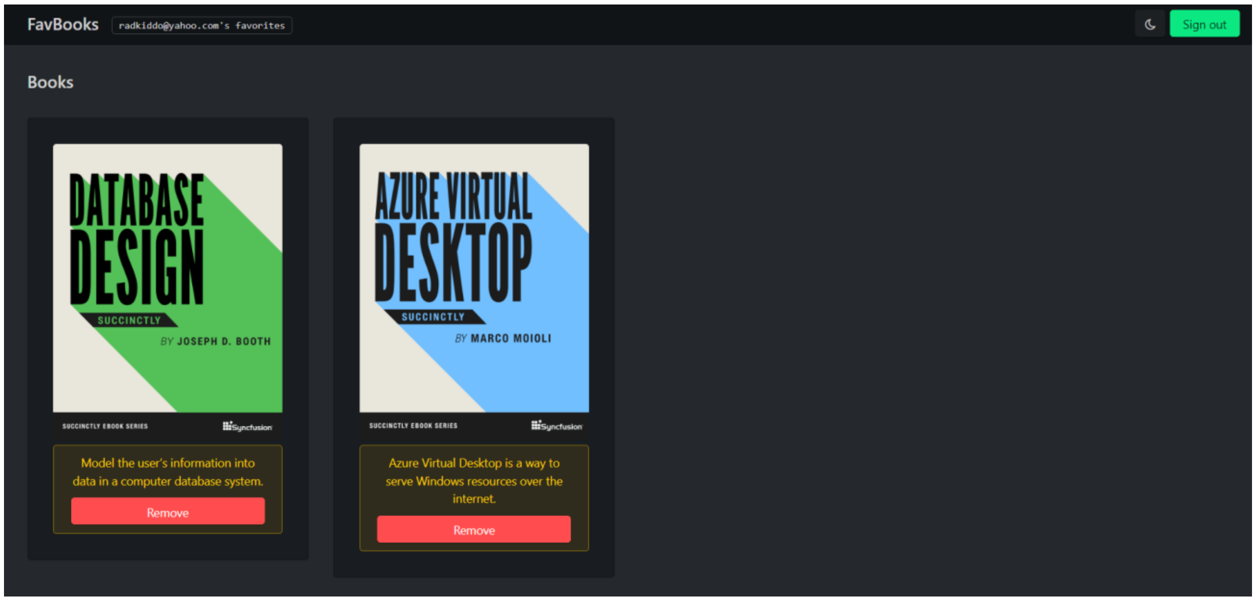
Notice that on the top-right of the main page's navigation bar, there's a Favorites link, which, as you might have guessed, will redirect you to the Favorites page.

The finished Favorites page (signed in)

The Favorites page is where the signed-in user has their list of favorite books, and it is only accessible if a user has signed in.

Note: If you attempt to manually type in the URL of the Favorites page in the browser and you're not signed in, you'll be redirected to the app's Sign in page.

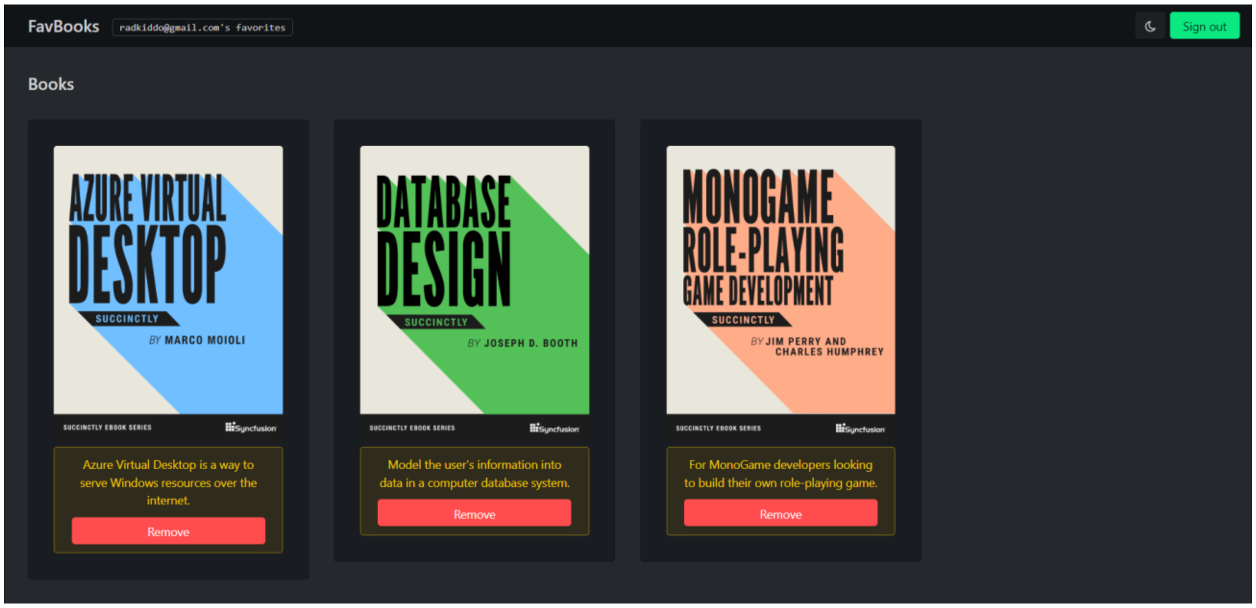
Here, the user can remove books from the favorites list by clicking the Remove button of a specific book. The list of favorites will differ from user to user.



Model the user's information into data in a computer database system.

Azure Virtual Desktop is a way to serve Windows resources over the internet.

Figure 4-h: The Favorites Page (As Seen by the User with the Yahoo Email Address)



Azure Virtual Desktop is a way to serve Windows resources over the internet.

Remove

Model the user's information into data in a computer database system.

Remove

For MonoGame developers looking to build their own role-playing game.

Remove

Figure 4-i: The Favorites Page (As Seen by the User with the Gmail Email Address)

Notice that the navigation bar of the Favorites page includes the user's email address on the top-left side, as shown in Figure 4-j.

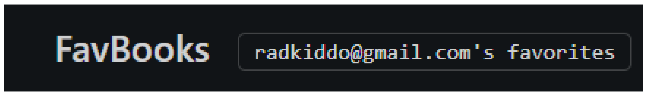


Figure 4-j: The Signed-In User Name (Top-Left Navigation Bar—Favorites Page Signed In)

The navigation bar of the Favorites page includes the Sign out button on the top-right side, as shown in Figure 4-k.

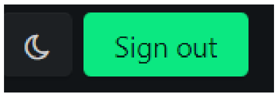


Figure 4-k: The Sign out Button (Top-Right Navigation Bar—Favorites Page Signed In)

Unlike the top-right navigation bar of the main page, the Favorites link is no longer visible; this is because we are already on the Favorites page.

If the signed-in user wants to add another book to their favorites list, the user must navigate to the main page (by clicking the **FavBooks** link on the top-left side) and then click **Add to favorites** for the relevant book.

Note: For code reusability, the list of books displayed on the main page and the list of favorite books shown on the Favorites page are rendered by the same Svelte component. The difference is that the books on the main page are retrieved via an API endpoint as JSON, and the favorites are fetched and stored in the Cloud Firestore Database (Firebase). We'll explore this topic in depth later.

Recap

Now that we know what the finished app looks like, what it does, and what its features do, we have a solid foundation to delve into the code and put things together. This is what we'll do next with the main user interface functionality.

Chapter 5 Main User Interface

Quick intro

In this chapter, we'll get our hands dirty and create the application's main user interface (the main page, log in, and register functionality). Let's focus on the HTML markup and its related functionality.

Note: As a reminder, you can find the code repository of the finished application on Github.

The app.html file

The application's entry point is the **app.html** file that resides within the project's **src** folder. It's the equivalent of an index.html file for a website (in other words, the application's entry point).

Code Listing 5-a: The Finished app.html File

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
        <link rel="icon" href="%sveltekit.assets%/favicon.png" />
        <!-- Font awesome -->
        <link rel="stylesheet"
          href="https://cdnjs.cloudflare.com/ajax/libs
          /font-awesome/4.7.0/css/font-awesome.min.css"
          integrity=
          "sha256-eZrrJcwDc/3uDhsdt61sL2oOBY362qM3lon1gyExkL0="
          crossorigin="anonymous" />
        <link href="https://fonts.googleapis.com/
          css2?family=Roboto:wght@400;
          500;700&display=swap"
          rel="stylesheet">

        <!-- Halfmoon CSS -->
        <link href="%sveltekit.assets%/css/halfmoon.min.css"
           rel="stylesheet" />
        <meta name="viewport"
           content="width=device-width, initial-scale=1" />
        %sveltekit.head%
    </head>
    <body class="dark-mode with-custom-webkit-scrollbars
        with-custom-css-scrollbars" data-dm-shortcut-enabled="true">
        <div>%sveltekit.body%</div>
        <script src="%sveltekit.assets%/js/halfmoon.min.js"></script>
        <script>
            var exports = {};
            halfmoon.onDOMContentLoaded();
        </script>
    </body>
</html>
```

Note: I split the two link tags highlighted in yellow in Code Listing 5-a into various readable lines so they could fit into the listing. However, each of those statements should be placed in one line.

Let's explore what we have here. The first thing we find within the **`head`** section of the HTML markup is the **`meta`** tag that sets the encoding the application will use, in this case **`utf-8`**.

Then, we find a **`link`** tag indicating where the application's favicon can be found. As you might have noticed, a relative path is used, which is **`%sveltekit.assets%/`**.

The shorthand **`%sveltekit.assets%`** indicates the resource will be found under the project's static folder.

Next, we can see two **`link`** tags highlighted in yellow that import Font Awesome. These should be placed in one line each.

Following that, we import the reference to the Halfmoon library, which I have downloaded and placed into the css subfolder within the project's static folder (src/static/css).

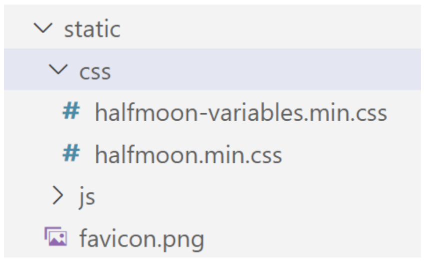

```
∨ static
  ∨ css
    # halfmoon-variables.min.css
    # halfmoon.min.css
  > js
  favicon.png
```

Figure 5-a: The Halfmoon Library CSS Files within the scr/static/css Project Folder (VS Code)

Note: Even though I'm using a Windows-based machine to build this Svelte application, since this is a web application, I will be using the forward-slash (/) convention when referring to folders and file paths, rather than the traditional

backslash (\) pattern commonly used in Windows. Thus, I'm referring to the folder path as scr/static/css, rather than src\static\css.

Halfmoon is a lightweight library that offers a wide range of CSS classes that will be familiar to Bootstrap developers and allow us to create modern and beautiful layouts.

Note: I'm not affiliated with Halfmoon. I just stumbled upon it and was captivated by its simplicity, beautiful design, and robust features.

After that, we can see the **`meta`** tag that defines the markup's viewport, determining where the web content will be displayed.

The following statement is **`%sveltekit.head%`**, which specifies the page's title displayed in the browser title bar.

Within the **`body`** section of the HTML markup, we can see the **`dark-mode with-custom-webkit-scrollbars with-custom-css-scrollbars`** classes, which provide the default dark mode that the application uses.

Next, we can see the **`%sveltekit.body%`** placeholder enclosed within a **`div`** tag. All this does is inject the **`body`** of each page into that **`div`**, which represents the content for each page.

Following that, we find the **`script`** tag that imports the JavaScript part of the Halfmoon library. The **`script`** tag that follows initializes Halfmoon.

The JavaScript part of the Halfmoon library also needs to be [downloaded]() from the Halfmoon website, and I've placed it into the project's src/static/js folder.

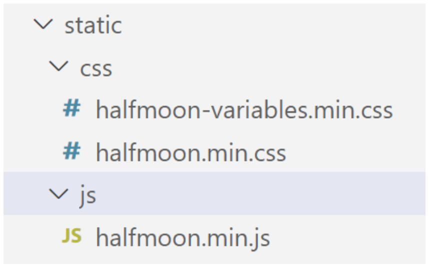

```
∨ static
  ∨ css
     # halfmoon-variables.min.css
     # halfmoon.min.css
  ∨ js
     JS halfmoon.min.js
```

Figure 5-b: The Halfmoon JavaScript File within the scr/static/js Project Folder (VS Code)

As you have seen, it's nothing complicated. Most of the markup was automatically generated when the Svelte project was scaffolded, and I just added the Font Awesome and Halfmoon tags and statements.

By exploring the app.html file, we also covered all the files found under the src/static folder and subfolders that our project includes.

The lib folder

By default, when a Svelte project is scaffolded using SvelteKit (as we've done), you won't find a lib folder within your project structure.

Creating a lib subfolder under the project's src folder is a non-official but widely adopted folder-naming convention used when creating a Svelte application.

A lib subfolder encourages the excellent practice of keeping any application-specific Svelte components separate from the application's pages (located in the src/routes subfolder).

Now, within the VS Code **EXPLORER**, select the **src** folder and click the **New Folder** icon, as shown in the following figure.

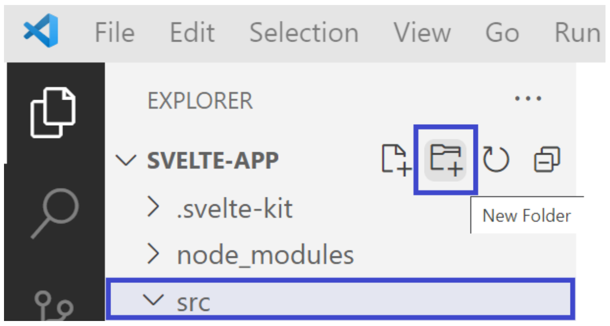


Figure 5-c: The New Folder Icon (VS Code EXPLORER—For Creating a lib Subfolder within src)

Name the new folder **lib** and press **Enter**.

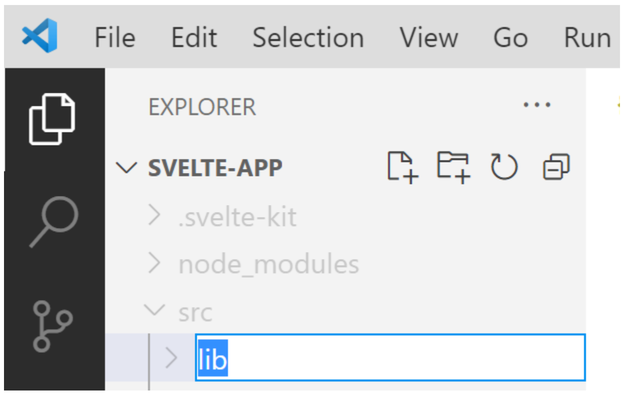


Figure 5-d: The New Folder Icon (VS Code EXPLORER—Naming the New Folder lib)

The lib folder will contain two files, a Svelte component called Books.svelte that's shared among different pages of the application, and a utility JavaScript file (utils.js) that will be used across several project files.

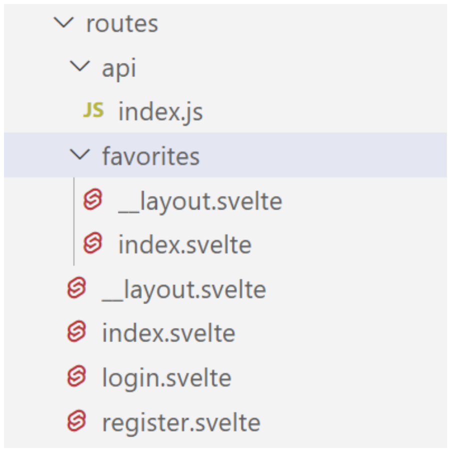
We'll get to those later. For now, we need to have this folder created, as this is not part of the default project structure.

The routes folder

A routes subfolder is added under the src folder when a SvelteKit project is created using the default process.

SvelteKit creates application routes based on the name of the folder structure and file names found under the routes folder.

Let’s look at the finished folder structure and files within the routes folder to understand how routing works with SvelteKit.



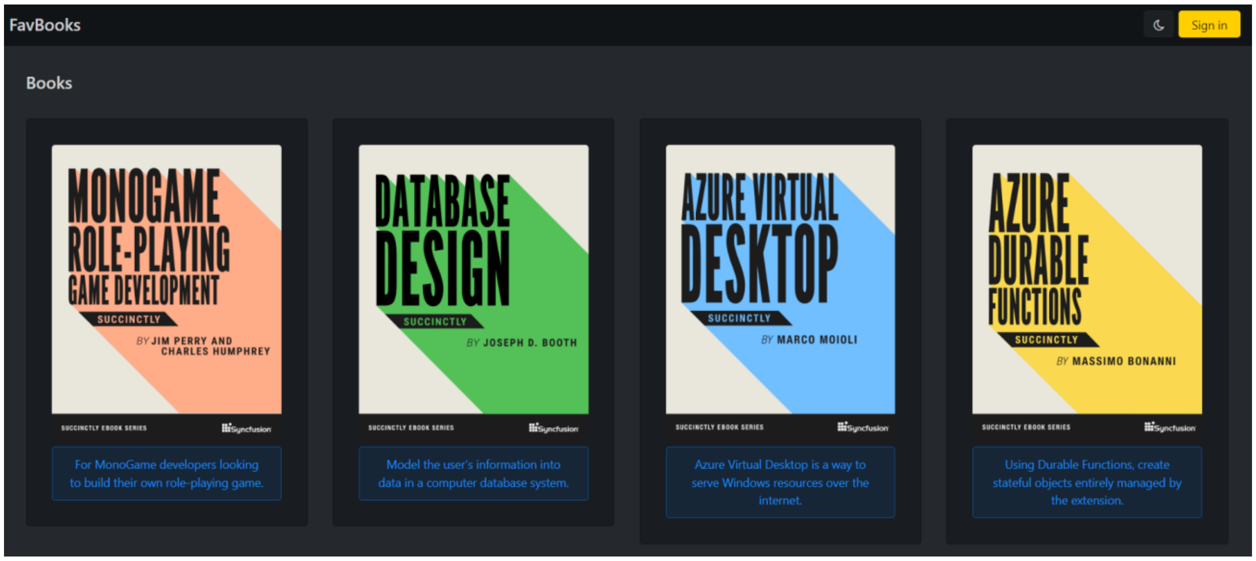
```
routes
  api
    index.js
  favorites
    __layout.svelte
    index.svelte
  __layout.svelte
  index.svelte
  login.svelte
  register.svelte
```

Figure 5-e: The routes Folder (Finished App—VS Code EXPLORER)

We can see various files within the root of the src/routes folder and two subfolders, api and favorites. Let’s first focus on the root folder.

index.svelte

The index.svelte file is the application’s main route, the app’s main page, which contains the list of books available to add as favorites. Visually, the application’s main page looks as follows.



For MonoGame developers looking to build their own role-playing game.

Model the user's information into data in a computer database system.

Azure Virtual Desktop is a way to serve Windows resources over the internet.

Using Durable Functions, create stateful objects entirely managed by the extension.

Figure 5-f: The App's Main Page

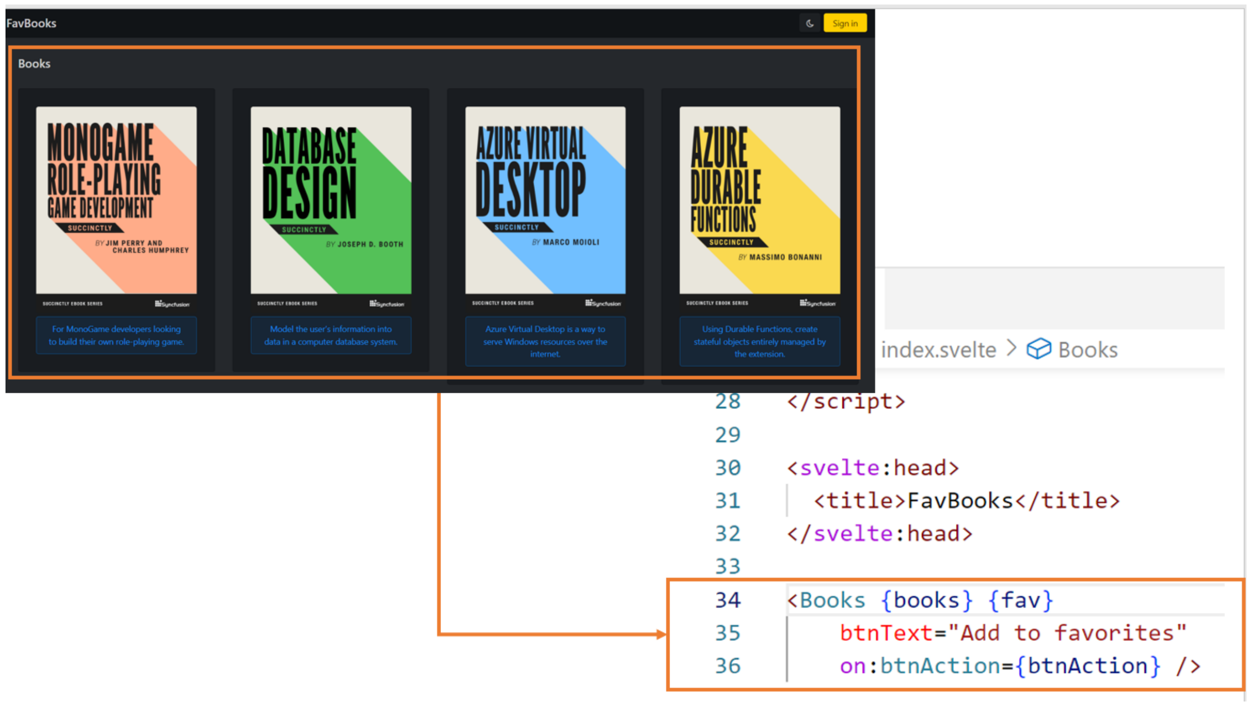
Most of the HTML markup required to produce the main's page visual appearance has been placed into the __layout.svelte file found under the root of the src/routes folder, and index.svelte is mainly limited to fetching the list of available books to display.

Here is the finished code for index.svelte found within the root of the src/routes folder.

Code Listing 5-b: The Finished index.svelte File (Root of the routes Folder)

```
<script context="module">
    import Books from "$lib/Books.svelte"

    export const load = async({fetch}) => {
        const res = await fetch("/api")
        const books = await res.json()
        return {
            props: {
                books,
            },
        }
    }
</script>

<script>
    import { addFav }
        from "../firebase.js";

    import { goto } from "$app/navigation"
    export let books
    const fav = ""

    const btnAction = async (e) => {
        await addFav(e.detail)
        await goto("/favorites")
    }
</script>

<svelte:head>
  <title>FavBooks</title>
</svelte:head>

<Books {books} {fav}
    btnText="Add to favorites"
    on:btnAction={btnAction} />
```

The following diagram shows how the markup code relates to the finished user interface for the main page.



For MonoGame developers looking to build their own role-playing game.

Model the user's information into data in a computer database system.

Azure Virtual Desktop is a way to serve Windows resources over the internet.

Using Durable Functions, create stateful objects entirely managed by the extension.

```
</script>

<svelte:head>
  <title>FavBooks</title>
</svelte:head>

<Books {books} {fav}
    btnText="Add to favorites"
    on:btnAction={btnAction} />
```

Figure 5-g: Main Page UI and Markup (index.svelte) Relationship (without Considering the Navigation Bar)

The preceding figure shows that most of the main page’s UI comes from the **Books** component.

So, to understand what is going on, let's explore each part of the code. First, we have the **`<script context="module">`** section.

In Svelte, a **`<script>`** code block includes code that runs when a component instance is initialized. For most components, that's all that's required.

Nevertheless, on some occasions, you might need to run some code outside of an individual component instance. You can do this by using a **`<script context="module">`** code block.

Within the **`<script context="module">`** code block, the first thing we do is **`import`** the **`Books`** component from the **Books.svelte** file, which we'll explore later. We do this with the following statement.

```
import Books from "$lib/Books.svelte"
```

Using the relative **`$lib/`** file path, we indicate that Svelte can find the Books.svelte file within the lib subfolder of the project's src folder.

The **`Books`** component displays a list of books, either obtained from the application's API (the list of latest books, which is static) or the list of favorites from Firebase.

The functionality was refactored into a **`Books`** component to avoid recreating the same logic twice. We'll have a look at how this component works later.

Next, we find the asynchronous (**`async`**) **`load`** function.

```
export const load = async({fetch}) => {
  const res = await fetch("/api")
  const books = await res.json()
  return { props: { books, }, }
}
```

This function uses the browser's **`fetch`** API (part of the browser context) to retrieve the static list of available books from the application's **`/api`** endpoint. The list of **`books`** is returned as a JSON response.

Then, we find the **`<script>`** section, which executes when the component instance initializes. Within this section, we **`import`** the **`addFav`** function from the **firebase.js** file—a utility file that includes the logic that allows the application to interact with Firebase. We'll explore this later.

```
import { addFav } from "../firebase.js";
```

The **`addFav`** function adds a book to the list of favorites. Next, we find the following **`import`** statement, which imports the **`goto`** method from Svelte's navigation module (**`app/navigation`**).

```
import { goto } from "$app/navigation"
```

The **`goto`** method will redirect the user to the Favorites page once a book has been added to the favorites list.

Then, we declare but do not initialize the **`books`** variable that will store the list of the available books obtained from the application's API.

Following that, we declare the constant **`fav`**, which will be used to indicate whether the list of available books will display the Add to favorites button for each book. If the value of **`fav`** is empty, then the Add to favorites buttons will not be shown.

Next, we find the **`btnAction`** function; this is the event that gets fired when clicking on the **Add to favorites** button.

```
const btnAction = async (e) => {
  await addFav(e.detail)
  await goto("/favorites")
}
```

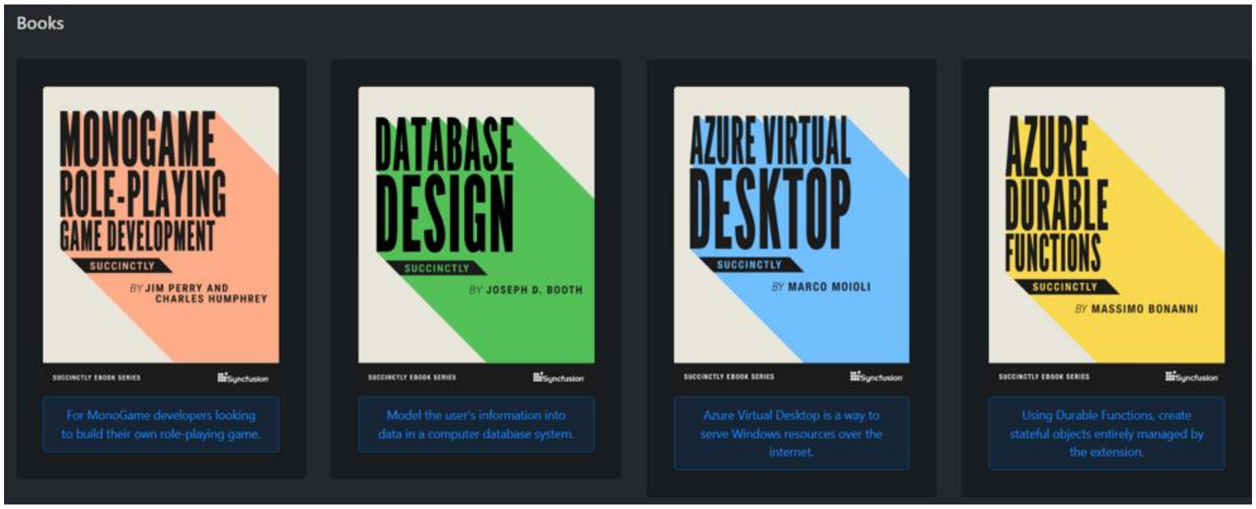
The **`e`** parameter passed to the **`btnAction`** function contains specific event information. The **`detail`** property (**`e.detail`**) includes the information about the book to be added to the favorites list, which occurs when the **`addFav`** function is invoked.

After the book is added to the favorites list, the **`goto`** method is invoked, redirecting the user to the **favorites** page (**`/favorites`**).

Following the script part of index.svelte, we have the HTML markup. The **`<svelte:head>`** tag is Svelte's way of adding or modifying the page header. In this case, the title of the page is changed as follows.

```
<svelte:head>
  <title>FavBooks</title>
</svelte:head>
```

Finally, the body of the HTML file includes the **`Books`** component, which displays the list of available books. Here is what the **`Books`** component looks like for the finished application.



For MonoGame developers looking to build their own role-playing game.

Model the user's information into data in a computer database system.

Azure Virtual Desktop is a way to serve Windows resources over the internet.

Using Durable Functions, create stateful objects entirely managed by the extension.

Figure 5-h: The Books Component

As you can see, the **Books** component is represented as a tag, which has four properties. The first property is the list of **books** to display, and the second property indicates whether the Add to favorites buttons (**fav**) will be shown or not.

```
<Books {books} {fav} btnText="Add to favorites" on:btnAction={btnAction} />
```

The third property is the caption of the Add to favorites or Remove button (**btnText**), and the fourth property is the event's name (**btnAction**), triggered when any Add to favorites or Remove buttons are clicked (**btnAction**).

Notice the syntax difference between **books**, **fav**, and **btnText**. The reason for this difference is that I'm using a Svelte shorthand.

When the name of the property and the name of the object assigned to the property are the same, then we can use this shorthand. So, instead of writing the following statement:

```
<Books books=books fav=fav btnText="Add to favorites"
on:btnAction={btnAction} />
```

We can change **books=books** to **{books}** and **fav=fav** to **{fav}**, and we end up with a more concise statement.

```
Books {books} {fav} btnText="Add to favorites" on:btnAction={btnAction} />
```

__layout.svelte

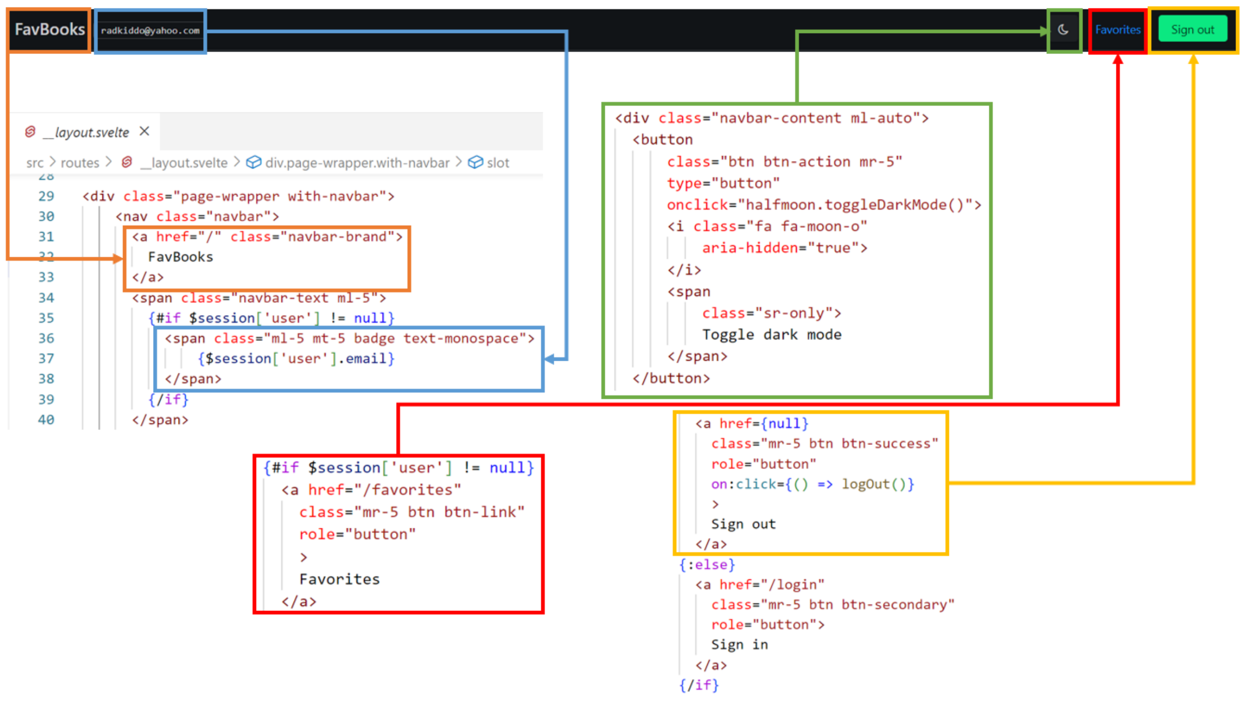
You might have noticed that the index.svelte file found within the root of the routes folder has barely any HTML markup. There are two reasons for this.

The first reason is that a significant part of the HTML markup shown, such as the list of books, is encapsulated in the **Books** component.

The second reason for the minimal amount of HTML in the index.svelte file concerns the nice navigation bar on the main page. Well, that is included within the __layout.svelte file contained within the root of the src/routes folder.

Now, let me warn you: there's more to the navigation bar than you expect. As you might have noticed from the figures in the previous chapter, the navigation bar knows if a user has signed in or not. The logic that regulates this behavior is part of the __layouts.svelte file.

The following diagram shows how the markup code relates to the finished user interface for the navigation bar of the main page, **__layout.svelte**.


FavBooks
radkiddo@yahoo.com
Favorites
Sign out
_layout.svelte
src > routes > _layout.svelte > div.page-wrapper.with-navbar > slot

```
<div class="page-wrapper with-navbar">
  <nav class="navbar">
    <a href="/" class="navbar-brand">
      FavBooks
    </a>
    <span class="navbar-text ml-5">
      {#if $session['user'] != null}
        <span class="ml-5 mt-5 badge text-monospace">
          {$session['user'].email}
        </span>
      {/if}
    </span>
```

```
<div class="navbar-content ml-auto">
  <button
      class="btn btn-action mr-5"
      type="button"
      onclick="halfmoon.toggleDarkMode()">
      <i class="fa fa-moon-o"
        aria-hidden="true">
      </i>
      <span
        class="sr-only">
        Toggle dark mode
      </span>
  </button>
```

```
{#if $session['user'] != null}
  <a href="/favorites"
    class="mr-5 btn btn-link"
    role="button"
    >
    Favorites
  </a>
```

```
  <a href={null}
    class="mr-5 btn btn-success"
    role="button"
    on:click={() => logOut()}
    >
    Sign out
  </a>
{:else}
  <a href="/login"
    class="mr-5 btn btn-secondary"
    role="button">
    Sign in
  </a>
{/if}
```

Figure 5-i: Main Page UI and Markup (__layout.svelte) Relationship (Just Considering the Navigation Bar)

As you can see, I've split the markup code of the __layout.svelte file into smaller chunks and colored each to match its corresponding UI element.

For example, the code highlighted in **yellow** corresponds to the Sign out button, and the code highlighted in **blue** corresponds to the signed-in user's email address.

The code in **green** corresponds to the dark mode button, the code in **orange** to the app title (**FavBooks**), and the code in **red** to the **favorites** link.

Here is finished code of the __layout.svelte file contained within the root of the src/routes folder.

Code Listing 5-c: The finished __layout.svelte File (Root of the routes Folder)

```
<script>
  import { onAuthStateChanged, signOut } from "Firebase/auth"
  import { onMount } from "svelte"
  import { auth } from "../firebase.js"
  import { getStores } from "$app/stores"
  import { goto } from "$app/navigation"

  let { session } = getStores()

  onMount(() => {
    onAuthStateChanged(
      auth,
      (user) => {
        session.set({ user })
      },
      (error) => {
        session.set({ user: null })
        console.log(error)
      }
    );
  });

  const logOut = async () => {
      await signOut(auth)
      await goto('/login')
  }
</script>

<div class="page-wrapper with-navbar">
    <nav class="navbar">
      <a href="/" class="navbar-brand">
        FavBooks
      </a>
      <span class="navbar-text ml-5">
        {#if $session['user'] != null}
          <span class="ml-5 mt-5 badge text-monospace">
              {$session['user'].email}
          </span>
        {/if}
      </span>

      <div class="navbar-content ml-auto">
        <button
            class="btn btn-action mr-5"
            type="button"
            onclick="halfmoon.toggleDarkMode()">
            <i class="fa fa-moon-o"
                aria-hidden="true">
            </i>
            <span
                class="sr-only">
                Toggle dark mode
            </span>
        </button>
        {#if $session['user'] != null}
          <a href="/favorites"
```

```
            class="mr-5 btn btn-link"
            role="button"
            >
            Favorites
          </a>
          <a href={null}
            class="mr-5 btn btn-success"
            role="button"
            on:click={() => logOut()}
            >
            Sign out
          </a>
        {:else}
          <a href="/login"
            class="mr-5 btn btn-secondary"
            role="button">
            Sign in
          </a>
        {/if}
      </div>
    </nav>
    <slot />
  </div>
```

As you have seen, there's much more code than in index.svelte. So, to properly understand this, let's break it down into smaller bits. Let's begin with the **`import`** statements.

```
import { onAuthStateChanged, signOut } from "Firebase/auth"
import { onMount } from "svelte"
import { auth } from "../firebase.js"
import { getStores } from "$app/stores"
import { goto } from "$app/navigation"
```

First, we begin by importing the **`onAuthStateChanged`** and **`signOut`** methods from the **`Firebase/auth`** module. We'll need these to know whether a user has signed in and is active and to sign out the active user.

Then, we **`import`** the onMount component lifecycle event from the core Svelte module (**`svelte`**), which executes following the component's rendering to the DOM.

After that, we **`import`** the **`auth`** object from the **firebase.js** utility file, which we'll explore later; this is used for retrieving the information about the authenticated user.

Next, we **`import`** the **`getStores`** method from the **`app/stores`** module responsible for state management in SvelteKit.

And finally, we **`import`** the **`goto`** method from the **`app/navigation`** module responsible for redirecting the navigation to another page.

Following the **`import`** statements, we retrieve **`session`** information for the signed-in user, which is done by invoking **`getStores`**.

```
let { session } = getStores()
```

Next, we have the **`onMount`** lifecycle event code, which subscribes to the onAuthStateChanged Firebase event that gets triggered when there's a change to the state of the signed-in user.

```
onMount(() => {
    onAuthStateChanged(
      auth,
      (user) => {
        session.set({ user })
      },
      (error) => {
        session.set({ user: null })
        console.log(error)
      }
    );
});
```

The **`onAuthStateChanged`** event uses three parameters. The first is **`auth`**, which indicates the current authentication state of the app from Firebase.

The second is a function that sets the **`session`** state for the current **`user`**. The third is a function that sets the **`session`** state for the current **`user`** to **`null`** if an error occurs.

In other words, whenever there's a change to a user's authentication state in Firebase and no errors are produced, the currently signed-in user (if any) will be saved with the **`session`** state. All this happens on the **`onMount`** lifecycle event of __layout.svelte.

```
session.set({ user })
```

Next, we have the **`logOut`** function. All this function does is sign out the currently signed-in (authenticated) user. The sign-out process is done by invoking the **`signOut`** method and passing the authentication details (**`auth`**).

```
const logOut = async () => {
  await signOut(auth)
  await goto('/login')
}
```

Once the sign-out process has taken place, we redirect the **`user`** to the **Sign in** page (**`/login`**) by invoking the **`goto`** method.

Next, we have the HTML markup that makes up the navigation bar. Now, I'd like to clarify that I'm not going to focus on the CSS classes used to style the navigation bar, as this is all nicely documented on the Halfmoon website. Instead, I'd like to focus on specific Svelte functionality.

Here is the first piece of Svelte functionality included within the navigation bar markup.

```
<span class="navbar-text ml-5">
  {#if $session['user'] != null}
    <span class="ml-5 mt-5 badge text-monospace">
      {$session['user'].email}
    </span>
  {/if}
</span>
```

What we have here is conditional rendering. We will only display the currently signed-in user's email address **`$session['user'].email`** if (**`#if`**) a user has signed in (**`$session['user'] != null`**).

Because we are including Svelte logic within the actual markup, we must wrap this logic using curly braces **`{}`**.

Notice that the opening if tag (**`#if`**) has to be closed with a corresponding end if tag (**`/if`**). With this short example, we've just learned how to perform conditional rendering, which means that this markup will only be displayed when that specific condition is met.

The next exciting part I'd like to discuss is displaying the Sign in button or Sign out button—which is also done with conditional rendering, as follows.

```
{#if $session['user'] != null}
  <a href="/favorites"
   class="mr-5 btn btn-link"
   role="button"
  >
   Favorites
  </a>
  <a href={null}
   class="mr-5 btn btn-success"
   role="button"
   on:click={() => logOut()}
  >
   Sign out
  </a>
{:else}
  <a href="/login"
   class="mr-5 btn btn-secondary"
   role="button">
   Sign in
  </a>
{/if}
```

By examining the previous code, we can see that the Favorites link and the Sign out button only show when a user has signed in—in other words, **`if $session['user'] != null`** is true.

On the other hand, if a user is not signed in, then the Sign in button shows—in other words, **`if $session['user'] != null`** is false (so, when **`$session['user'] == null`**).

Notice that the Sign out button includes a click event that triggers the execution of the **`logOut`** function, which is done as follows.

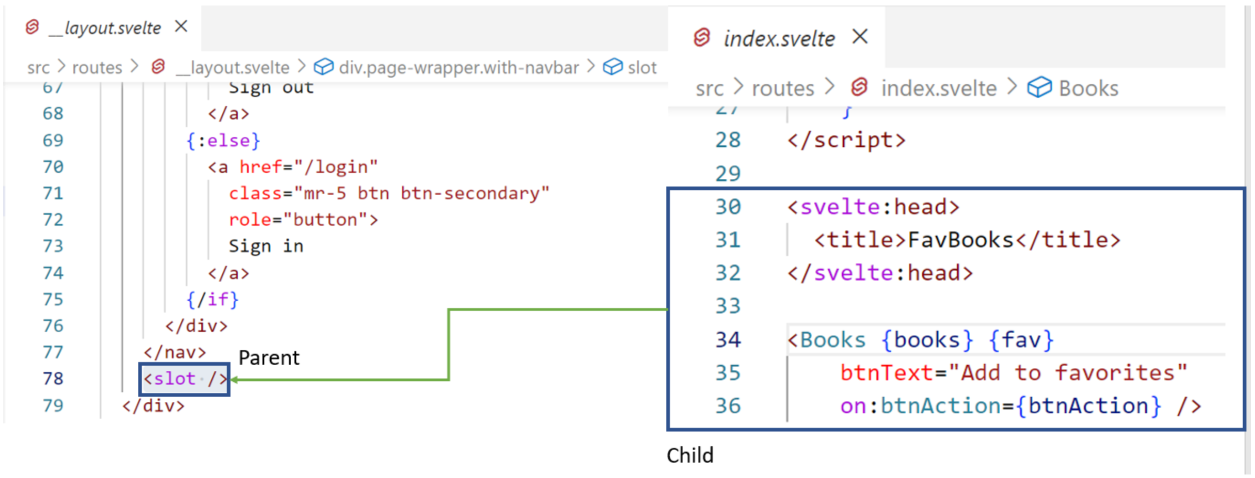
```
on:click={() => logOut()}
```

Notice that the syntax is not written as **`on:click={logOut}`** because doing so only references the function and doesn't invoke it.

The function's execution is done by creating an anonymous arrow function (**`() =>`**) that invokes the **`logOut`** function.

And finally, the slot tag is significant and can be easily missed. Svelte uses slots to indicate that a parent component can receive a child component, just like regular HTML elements have children.

In other words, the **`slot`** tag indicates where the child component will be placed within the parent component. To understand this better, let's look at the following diagram.



_layout.svelte
src > routes > _layout.svelte > div.page-wrapper.with-navbar > slot
index.svelte
src > routes > index.svelte > Books
Parent
Child

```
          Sign out
        </a>
      {:else}
        <a href="/login"
          class="mr-5 btn btn-secondary"
          role="button">
          Sign in
        </a>
      {/if}
    </div>
  </nav>
  <slot />
</div>
```

```
</script>

<svelte:head>
  <title>FavBooks</title>
</svelte:head>

<Books {books} {fav}
    btnText="Add to favorites"
    on:btnAction={btnAction} />
```

Figure 5-j: index.svelte Markup Injected into the __layout.svelte Slot (Code Approach)

In this figure, notice that the markup from index.svelte is placed by Svelte during compilation time into the slot available within __layout.svelte. So, __layout.svelte represents the parent component and index.svelte the child.

To understand this better, let's look at the following image.

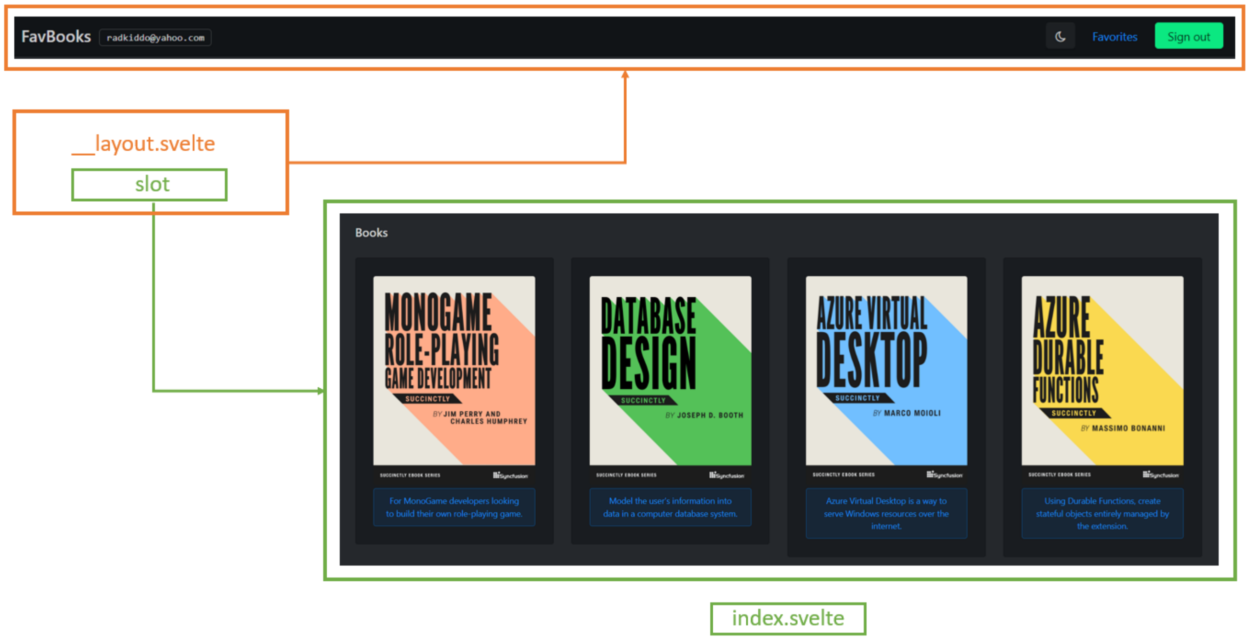

FavBooks
radkiddo@yahoo.com
Favorites
Sign out
__layout.svelte
slot
Books
For MonoGame developers looking to build their own role-playing game.
Model the user's information into data in a computer database system.
Azure Virtual Desktop is a way to serve Windows resources over the internet.
Using Durable Functions, create stateful objects entirely managed by the extension.
index.svelte

Figure 5-k: index.svelte Markup Injected into the __layout.svelte Slot (UI Approach)

This image shows that the **`Books`** component is placed into the **`slot`** tag within the **__layout.svelte** markup, which, as you have seen, includes the navigation bar markup.

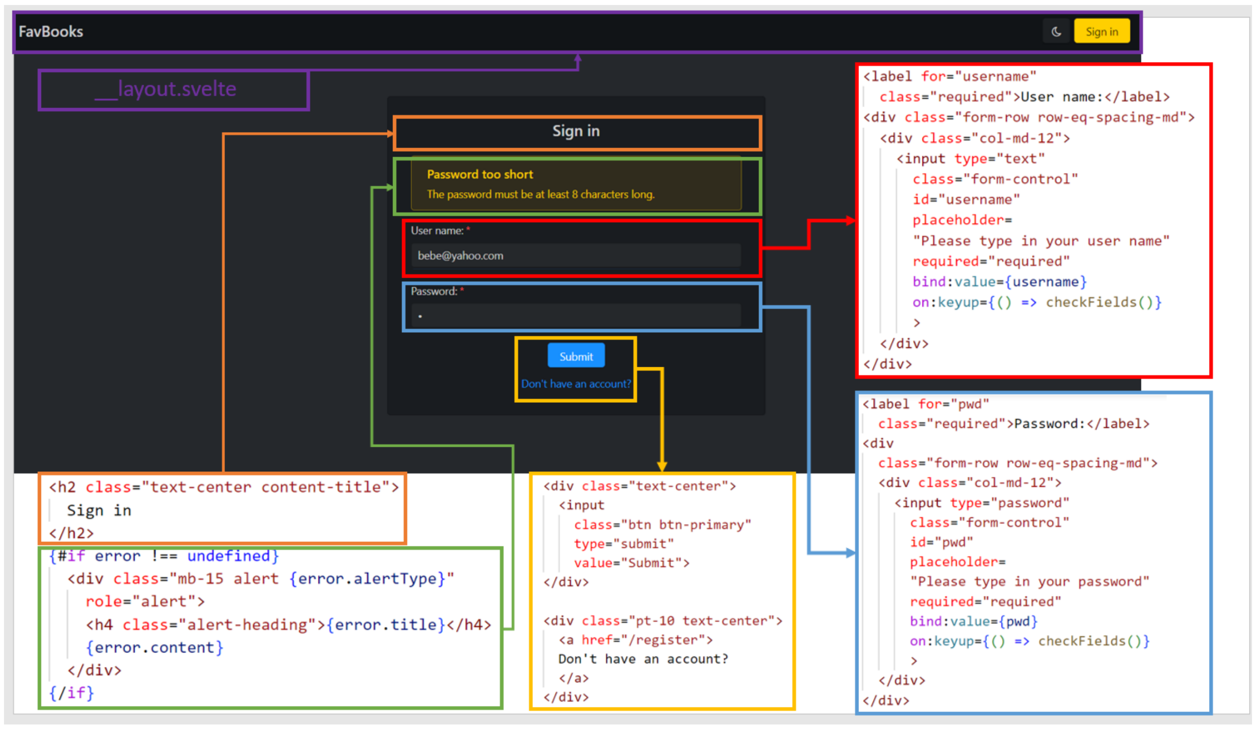
Sign in page (login.svelte)

Now that we have looked at the main page, let's explore the Sign in page. Before we look at the complete finished code, let's see how the markup corresponds to the UI elements of this page.

In the following figure, I've highlighted how each UI part of the Sign in page corresponds to its markup counterpart within login.svelte.

For example, the part highlighted in **purple** corresponds to the navigation bar (**__layout.svelte**). The **orange** part corresponds to the **Sign in** form caption. The **green** part is only shown if an error occurs.

The **red** part corresponds to the **User name** field, the **blue** corresponds to the **Password** field, and the **yellow** part corresponds to the **Submit** button and the **Don't have an account** link.



```
<label for="username"
  class="required">User name:</label>
<div class="form-row row-eq-spacing-md">
  <div class="col-md-12">
    <input type="text"
      class="form-control"
      id="username"
      placeholder=
      "Please type in your user name"
      required="required"
      bind:value={username}
      on:keyup={() => checkFields()}
      >
  </div>
</div>
```

```
<label for="pwd"
  class="required">Password:</label>
<div
  class="form-row row-eq-spacing-md">
  <div class="col-md-12">
    <input type="password"
      class="form-control"
      id="pwd"
      placeholder=
      "Please type in your password"
      required="required"
      bind:value={pwd}
      on:keyup={() => checkFields()}
      >
  </div>
</div>
```

```
<h2 class="text-center content-title">
  Sign in
</h2>
```

```
{#if error !== undefined}
  <div class="mb-15 alert {error.alertType}"
    role="alert">
    <h4 class="alert-heading">{error.title}</h4>
    {error.content}
  </div>
{/if}
```

```
<div class="text-center">
  <input
    class="btn btn-primary"
    type="submit"
    value="Submit">
</div>

<div class="pt-10 text-center">
  <a href="/register">
  Don't have an account?
  </a>
</div>
```

Figure 5-l: Sign in Page UI and Markup (login.svelte) Relationship

Here is the finished code for login.svelte.

Code Listing 5-d: The Finished login.svelte File (Root of the routes Folder)

```
<script context="module">
  export const load = async ({ session }) => {
        let { user } = session
        if (user != null) {
            return {
                status: 302,
                redirect: "/favorites",
            }
        }
        return {
            status: 200,
        }
  }
</script>

<script>
    import { loginChecks, fbLoginChecks }
      from "$lib/utils.js"
    import { signInWithEmailAndPassword } from "Firebase/auth"
    import { auth, userDoc } from "../firebase.js"
    import { goto } from "$app/navigation"
    import { setDoc } from "Firebase/firestore/lite"

    export let error = undefined
    export let username = ""
    export let pwd = ""

    const checkFields = () => {
      if (username === "" &&
          pwd === "") {
          error = undefined
      }
    }

    const signIn = async (email, pwd) => {
      if (email !== undefined &&
          pwd !== undefined) {

        error = loginChecks(email, pwd)
        if (error === undefined) {
          try {
            let user = await
              signInWithEmailAndPassword(auth, email, pwd)
            await setDoc(
              userDoc(auth.currentUser.uid),
                { username: user.user.displayName,
                  email: user.user.email })
            await goto("/favorites")
          }
          catch (e) {
            error = fbLoginChecks(e)
          }
        }
      }
    }
</script>

<svelte:head>
  <title>Sign in</title>
</svelte:head>

<div class="content-wrapper">
  <div class="container-fluid">
    <div class="content">
```

```
      <div class="row">
        <div class="col-sm"></div>
        <div class="card col-sm">
          <h2 class="text-center content-title">
            Sign in
          </h2>
          {#if error !== undefined}
            <div class="mb-15 alert {error.alertType}"
              role="alert">
              <h4 class="alert-heading">{error.title}</h4>
              {error.content}
            </div>
          {/if}
          <form on:submit|preventDefault=
          {() => signIn(username, pwd)}>
            <div>
              <label for="username"
                class="required">User name:</label>
              <div class="form-row row-eq-spacing-md">
                <div class="col-md-12">
                  <input type="text"
                    class="form-control"
                    id="username"
                    placeholder=
                    "Please type in your user name"
                    required="required"
                    bind:value={username}
                    on:keyup={() => checkFields()}
                  >
                </div>
              </div>
            </div>

            <div>
              <label for="pwd"
                class="required">Password:</label>
              <div
                class="form-row row-eq-spacing-md">
                <div class="col-md-12">
                  <input type="password"
                    class="form-control"
                    id="pwd"
                    placeholder=
                    "Please type in your password"
                    required="required"
                    bind:value={pwd}
                    on:keyup={() => checkFields()}
                    >
                </div>
              </div>
            </div>

            <div class="text-center">
              <input
                class="btn btn-primary"
                type="submit"
                value="Submit">
            </div>

            <div class="pt-10 text-center">
              <a href="/register">
              Don't have an account?
              </a>
            </div>
          </form>
        </div>
        <div class="col-sm"></div>
      </div>
    </div>
  </div>
</div>
```

As you can see, we have a part that runs outside of an individual component instance, which is done using the **`<script context="module">`** code block.

This code block only includes the **`load`** function, which checks whether a **`user`** is signed in (**`user != null`**). If so, the user is redirected to the Favorites page (**`/favorites`**), and the Sign in page is not shown.

The reason for doing that (redirecting the user to the Favorites page when a user has already signed in) is that there is no need to execute another sign-in until the currently signed-in user signs out.

```
export const load = async ({ session }) => {
  let { user } = session
  if (user != null) {
    return {
      status: 302,
      redirect: "/favorites",
    }
  }
  return {
    status: 200,
  }
}
```

If a **`user`** has not signed in (**`user == null`**), then the Sign in page is shown so a user can sign in. The following diagram explains how this flow works.

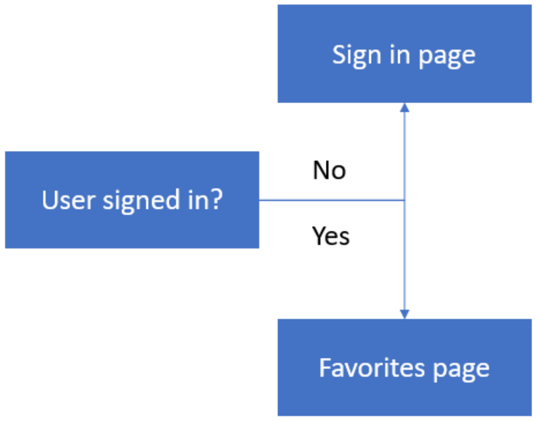


Figure 5-m: Login Flow

Next, we find the **`script`** section that executes within the individual component instance. Let's start by looking at the **`import`** statements.

```
import { signInWithEmailAndPassword } from "Firebase/auth"
import { auth, userDoc } from "../firebase.js"
import { goto } from "$app/navigation"
import { setDoc } from "Firebase/firestore/lite"
```

First, we can see that from the Firebase authentication module (**`Firebase/auth`**), we **`import`** the **`signInWithEmailAndPassword`** method. As the method name implies, we'll use it to authenticate any application user with Firebase.

Then, from the utility **firebase.js** file, we **`import`** the **`auth`** object and **`userDoc`** method. These are used to check if authentication has occurred and access the signed-in user.

Next, we **`import`** the **`goto`** method, which we use to redirect the user to the Favorites page if the user has signed in.

Then, we have the **`setDoc`** method, used for adding data to the Cloud Firestore database, which is why we **`import`** the Firebase Firestore Lite module (**`Firebase/firestore/lite`**).

Next, we declare the three variables for the **Sign in** page, **`error`**, **`username`**, and **`pwd`**. By adding the **`export`** keyword, we specify that these variables can be used outside of login.svelte.

```
export let error = undefined
export let username = ""
export let pwd = ""
```

Although we won't be using these variables outside of login.svelte, it's a good idea to export these variables if in the future we want to refactor login.svelte and register.svelte into a single component.

After that, we find the **`checkFields`** function, which checks that the **`username`**, **`pwd`**, and **`error`** variables have their initial values set as an empty string, or **`undefined`** in the case of the **`error`** variable.

```
const checkFields = () => {
  if (username === "" &&
    pwd === "") {
    error = undefined
  }
}
```

This function is used when the user deletes the content of either the **`username`** or **`pwd`** fields, and if a login **`error`** has occurred (and is still shown on the Sign in form), so that the **`error`** can be cleared (removed from the screen).

The **`checkFields`** function executes when the **`keyup`** event for the **`username`** or **`pwd`** field gets triggered. The **`keyup`** event occurs when the user presses and releases a key on either the **`username`** or **`pwd`** field.

Next, we find the **`signIn`** function. As its name implies, this function is responsible for authenticating the user against Firebase.

```
const signIn = async (email, pwd) => {
  if (email !== undefined &&
    pwd !== undefined) {

    error = loginChecks(email, pwd)
    if (error === undefined) {
      try {
        let user = await
          signInWithEmailAndPassword(auth, email, pwd)
        await setDoc(
          userDoc(auth.currentUser.uid),
            { username: user.user.displayName,
             email: user.user.email })
        await goto("/favorites")
      }
      catch (e) {
        error = fbLoginChecks(e)
      }
    }
  }
}
```

As you can see, the function is asynchronous (**`async`**), and the user's **`email`** (which represents the value of the **`username`** field) and **`pwd`** are passed as parameters.

The function first checks both the values of the **`email`** and **`pwd`** to ensure they are not **`undefined`**, which is performed by the **`if (email !== undefined && pwd !== undefined)`** condition.

Then, both **`email`** and **`pwd`** are checked to ensure that they are valid for login, which is done by the **`loginChecks`** function. As we'll see later, this function verifies that the **`email`** variable's value is a valid email address, and the **`pwd`** field's value is good enough for a password (minimum length check).

If the **`loginChecks`** validation goes well and there are no errors (**`error === undefined`**), then we can authenticate the user to Firebase by calling the **`signInWithEmailAndPassword`** method.

If the authentication process succeeds, the currently active user can be set up by invoking the **`setDoc`** function, in which **`userDoc`** is passed as a first parameter, and an object containing the user information is passed as a second parameter.

```
setDoc(userDoc(auth.currentUser.uid), { username: user.user.displayName,
  email: user.user.email })
```

Once that succeeds, the user is redirected to the Favorites page, which is done by executing the **`goto("/favorites")`** instruction. At this stage, the user is authenticated.

Suppose the authentication process fails at any given point (thus why it is wrapped around a **`try-catch`** statement). In that case, an error (**`e`**) is reported by Firebase, and the **`fbLoginChecks`** function then processes that error.

In that case, the **`fbLoginChecks`** converts the error to a user-friendly message displayed above the **`username`** field.

Moving on, we find the **`<svelte:head>`** section that sets the page's title, and we can see the rest of the markup.

As for the remaining markup, I won't go over the specifics of the Halfmoon CSS classes used (this is covered extensively within their excellent documentation), but instead focus solely on Svelte-specific instructions.

Notice the following conditional rendering, which is responsible for displaying an error in case there is one.

```
{#if error !== undefined}
  <div class="mb-15 alert {error.alertType}"
    role="alert">
    <h4 class="alert-heading">{error.title}</h4>
    {error.content}
  </div>
{/if}
```

An error will display if the **error** variable's value is different than **undefined**. Depending on the type of error (**error.alertType**), a specific CSS class will be applied. The **error.title** and **error.content** are shown.

The following interesting Svelte logic relates to the form submission.

```
<form on:submit|preventDefault={() => signIn(username, pwd)}>
```

This binds the form to the **submit** event and prevents the default browser form behavior (**preventDefault**), which would be to submit the form and redirect the user to another page.

So, what happens instead is that a **submit** event occurs, and the **signIn** function is executed, passing the value of the submitted **username** and **pwd** fields.

Notice that the function is executed because it is called through an anonymous function: **() => signIn(username, pwd)**. Otherwise, it wouldn't run and only be referenced.

Then, we have the **input** element used for entering the **username** variable, for which we can see the code as follows.

```
<input type="text"
 class="form-control" id="username"
 placeholder="Please type in your user name"
 required="required"
 bind:value={username}
 on:keyup={() => checkFields()}
>
```

Notice that this **input** element (**id="username"**) binds to the **username** variable. Whatever value is entered into the **input** element is automatically assigned to **username**.

Furthermore, the **input** element has a **keyup** event, which executes the **checkFields** function we covered before.

We can see the **input** element for entering the password (**id="pwd"**). Notice that the element binds to the **pwd** field; whatever value is entered into the **input** element is automatically assigned to **pwd**.

```
<input type="password"
 class="form-control" id="pwd"
 placeholder="Please type in your password"
 required="required"
 bind:value={pwd}
 on:keyup={() => checkFields()}
>
```

This **input** element also has a **keyup** event, which executes the **checkFields** function we covered before.

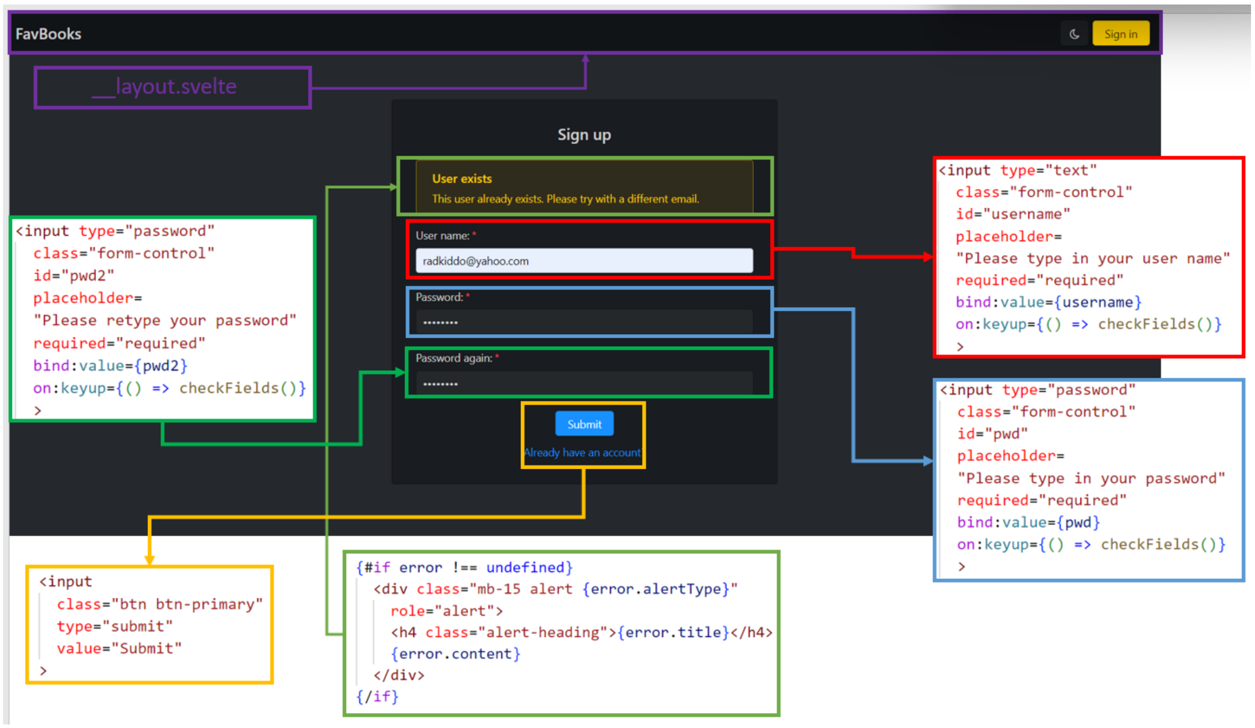
Finally, we can see the Submit button. When you click this button, the **`submit`** event of the **`form`** is triggered.

```
<input class="btn btn-primary" type="submit" value="Submit">
```

That's all there is to the Sign in page—as you have seen, it's relatively straightforward.

Sign up page (register.svelte)

With the Sign in page covered, let's now focus our attention on the Sign up page. The functionality is almost identical to the Sign in page. First, let's look at how the UI relates to the markup.



User exists

This user already exists. Please try with a different email.

radkiddo@yahoo.com

```
<input type="text"
  class="form-control"
  id="username"
  placeholder=
  "Please type in your user name"
  required="required"
  bind:value={username}
  on:keyup={() => checkFields()}
>
```

```
<input type="password"
  class="form-control"
  id="pwd"
  placeholder=
  "Please type in your password"
  required="required"
  bind:value={pwd}
  on:keyup={() => checkFields()}
>
```

```
<input type="password"
  class="form-control"
  id="pwd2"
  placeholder=
  "Please retype your password"
  required="required"
  bind:value={pwd2}
  on:keyup={() => checkFields()}
>
```

```
<input
  class="btn btn-primary"
  type="submit"
  value="Submit"
>
```

```
{#if error !== undefined}
  <div class="mb-15 alert {error.alertType}"
    role="alert">
    <h4 class="alert-heading">{error.title}</h4>
    {error.content}
  </div>
{/if}
```

Figure 5-n: Sign up Page UI and Markup (register.svelte) Relationship

As you can see, the part highlighted in **purple** corresponds to the navigation bar (**__layout.svelte**).

The **light green** part is only shown if an error occurs. The **red** part corresponds to the User name field, the **blue** to the Password field, the **bright green** to the Password again field, and the **yellow** part corresponds to the Submit button and the **Already have an account?** link.

The only significant difference between Sign in and this page is that a different password-related field is used for repeating the password.

Now, let's look at the finished code for the Sign up page.

Code Listing 5-e: The Finished register.svelte File (Root of the routes Folder)

```
<script context="module">
  export const load = async ({ session }) =>{
        let { user } = session
        if (user != null) {
            return {
                status: 302,
                redirect: "/favorites",
            }
        }
        return {
            status: 200,
        }
  }
</script>

<script>
    import { registerChecks, fbRegisterChecks }
      from "$lib/utils.js"

    import { createUserWithEmailAndPassword,
      updateProfile } from "Firebase/auth"
    import { goto } from "$app/navigation"
    import { auth, userDoc } from "../firebase.js"
    import { setDoc } from "Firebase/firestore/lite"

    export let
      error = undefined,
      username = "", pwd = "", pwd2 = ""

    const checkFields = () => {
      if (username === "" &&
          pwd === "" &&
          pwd2 === "") {
          error = undefined
      }
    }

    const signUp = async (email, pwd, pwd2) => {
      if (email !== undefined &&
          pwd !== undefined &&
          pwd2 !== undefined) {

        error = registerChecks(email, pwd, pwd2)
        if (error === undefined) {
          try {
            let user = await
              createUserWithEmailAndPassword(
                auth,
                email,
                pwd
            );
            await updateProfile(user.user,
              { displayName: username });
            await setDoc(
              userDoc(auth.currentUser.uid), {
                username: user.user.displayName,
                email: user.user.email
            });
            await goto("/favorites")
          }
          catch (e) {
            error = fbRegisterChecks(e)
          }
        }
      }
    }
</script>

<svelte:head>
  <title>Sign up</title>
</svelte:head>

<div class="content-wrapper">
  <div class="container-fluid">
    <div class="content">
      <div class="row">
        <div class="col-sm"></div>
        <div class="card col-sm">
          <h2 class="text-center content-title">
            Sign up
          </h2>
          {#if error !== undefined}
            <div class="mb-15 alert {error.alertType}"
              role="alert">
              <h4 class="alert-heading">{error.title}</h4>
              {error.content}
            </div>
          {/if}
          <form on:submit|preventDefault=
            {() => signUp(username, pwd, pwd2)}>
            <div>
              <label for="username"
                class="required">User name:</label>
              <div class="form-row row-eq-spacing-md">
                <div class="col-md-12">
                  <input type="text"
                    class="form-control"
                    id="username"
                    placeholder=
                    "Please type in your user name"
                    required="required"
                    bind:value={username}
                    on:keyup={() => checkFields()}
                    >
                </div>
              </div>
            </div>

            <div>
              <label for="pwd"
                class="required">Password:</label>
              <div
                class="form-row row-eq-spacing-md">
                <div class="col-md-12">
                  <input type="password"
                    class="form-control"
                    id="pwd"
                    placeholder=
                    "Please type in your password"
                    required="required"
                    bind:value={pwd}
                    on:keyup={() => checkFields()}
                    >
                </div>
              </div>
            </div>

            <div>
              <label for="pwd2"
                class="required">Password again:</label>
              <div
```

```
                 class="form-row row-eq-spacing-md">
                 <div class="col-md-12">
                   <input type="password"
                     class="form-control"
                     id="pwd2"
                     placeholder=
                     "Please retype your password"
                     required="required"
                     bind:value={pwd2}
                     on:keyup={() => checkFields()}
                     >
                 </div>
               </div>
             </div>

             <div class="text-center">
               <input
                 class="btn btn-primary"
                 type="submit"
                 value="Submit"
               >
             </div>

             <div class="pt-10 text-center">
               <a href="/login">
               Already have an account?
               </a>
             </div>
           </form>
         </div>
         <div class="col-sm"></div>
       </div>
     </div>
   </div>
</div>
```

As you can see in the preceding code, just like with the Sign in page, we have a part that runs outside of an individual component instance, which is done using the **`<script context="module">`** code block.

This code block only includes the **`load`** function, which checks whether a **`user`** is signed in (**`user != null`**). If so, the user is redirected to the Favorites page (**`/favorites`**), and the Sign up page is not shown.

The reason for doing that (redirecting the user to the Favorites page when a user has already signed up) is that there is no need to execute another sign up on the same browser session because the signed-up user will be signed in automatically.

```
export const load = async ({ session }) => {
  let { user } = session
  if (user != null) {
    return {
      status: 302,
      redirect: "/favorites",
    }
  }
  return {
    status: 200,
  }
}
```

Next, we find the **`script`** section that executes within the individual component instance. Let's start by looking at the **`import`** statements.

```
import { registerChecks, fbRegisterChecks } from "$lib/utils.js"
import { createUserWithEmailAndPassword, updateProfile } from "Firebase/auth"
import { goto } from "$app/navigation"
import { auth, userDoc } from "../firebase.js"
import { setDoc } from "Firebase/firestore/lite"
```

First, from the utility file (**util.js**) found within the **src/lib** folder, we **`import`** the **`registerChecks`** and **`fbRegisterChecks`** functions used for validating the entered data.

Next, we can see that from the Firebase authentication module (**`Firebase/auth`**), we **`import`** the **`createUserWithEmailAndPassword`** method. As the method name implies, we'll use it to register any application user with Firebase. Once the user is registered, the user profile is updated using the **`updateProfile`** method.

Then, we **`import`** the **`goto`** method, which we use to redirect the user to the Favorites page if the user has signed up (which automatically signs the user in).

Next, from the utility **firebase.js** file, we **`import`** the **`auth`** object and **`userDoc`** method. These are used to check if authentication has occurred and access the signed-in user details.

Then, we have the **`setDoc`** method, used for adding data to the Cloud Firestore, which is why we **`import`** the Firebase Firestore Lite module (**`Firebase/firestore/lite`**).

After that, just like we did for the Sign in page, we define the variables we will use for the sign-up process.

```
export let error = undefined, username = "", pwd = "", pwd2 = ""
```

In this case, the difference is that we have an additional variable (**`pwd2`**) that we did not need on the Sign in page.

Moving on, we find the **checkFields** function, which is almost identical to its counterpart within the Sign in page, except that the **pwd2** field is included.

```
const checkFields = () => {
  if (username === "" &&
    pwd === "" && pwd2 === "") {
      error = undefined
  }
}
```

This function is used when the user deletes the content of the **username**, **pwd**, or **pwd2** fields, and if a register **error** has occurred (and is still shown on the Sign up form), so that the **error** can be cleared (not shown anymore on the screen).

The last part of the user Sign up page logic is the user registration process, done with the **signUp** function.

```
const signUp = async (email, pwd, pwd2) => {
  if (email !== undefined &&
    pwd !== undefined &&
    pwd2 !== undefined) {

    error = registerChecks(email, pwd, pwd2)
    if (error === undefined) {
      try {
        let user = await createUserWithEmailAndPassword(auth, email, pwd)
        await updateProfile(user.user, { displayName: username })
        await setDoc(userDoc(auth.currentUser.uid), {
          username: user.user.displayName,
          email: user.user.email
        });
        await goto("/favorites")
      }
      catch (e) {
        error = fbRegisterChecks(e)
      }
    }
  }
}
```

Paying close attention to the **signUp** function, we can see that the sequence of instructions is almost identical to the **signIn** process from the Sign in page.

The main differences are that for input validation, we are now using the **registerChecks** function; for error validation, the **fbRegisterChecks** function is used, and the user registration is done when the **createUserWithEmailAndPassword** function is invoked.

Once the user registration has taken place, the user profile is updated, which is done with the **updateProfile** function.

If the authentication process succeeds, the currently active user can be set up by invoking the **setDoc** function, in which **userDoc** is passed as a first parameter, and an object containing the user information is passed as a second parameter.

```
setDoc(userDoc(auth.currentUser.uid), { username: user.user.displayName,
  email: user.user.email })
```

If the registration and subsequent automatic authentication are successful, the user is redirected to the Favorites page, which is done by executing the **goto("/favorites")** instruction. At this stage, the user is authenticated.

Suppose the authentication process fails at any given point (thus why it is wrapped around a **try-catch** statement). In that case, an error (**e**) is returned by Firebase, and the **fbRegisterChecks** function then processes that error.

Moving on, we find the **<svelte:head>** section that sets the page's title, and we can see the rest of the markup.

As for the remaining markup, as I mentioned previously, I won't go over the specifics of the Halfmoon CSS classes, but instead focus solely on Svelte-specific instructions.

Looking at the markup, we can see that the following code displays the **error** if its value is not **undefined**, which should look familiar to you.

```
{#if error !== undefined}
  <div class="mb-15 alert {error.alertType}" role="alert">
    <h4 class="alert-heading">{error.title}</h4>
    {error.content}
  </div>
{/if}
```

Depending on the type of error (**error.alertType**), a specific CSS class will be applied. The **error.title** and **error.content** are shown.

The next instruction relates to the form submission.

```
<form on:submit|preventDefault={() => signUp(username, pwd, pwd2)}>
```

This binds the form to the **submit** event and prevents the default browser form behavior (**preventDefault**), which would be to submit the form and redirect the user to another page.

What happens instead is that the **submit** event occurs and the **signUp** function runs, passing the value of the submitted **username**, **pwd**, and **pwd2** fields.

Notice that the function is executed because it is called through an anonymous function: **() => signUp(username, pwd, pwd2)**. Otherwise, it wouldn't run, and only be referenced.

Moving on, we have the **input** element for entering the **username** variable, for which we can see the code as follows.

```
<input type="text"
 class="form-control" id="username"
 placeholder="Please type in your user name"
 required="required"
 bind:value={username}
 on:keyup={() => checkFields()}
>
```

Notice that this **input** element (**id="username"**) binds to the **username** variable; whatever value is entered into the **input** element is automatically assigned to **username**.

Additionally, notice that the **input** element has a **keyup** event, which executes the **checkFields** function we covered before. It is specific for user registration, as it also evaluates the value of **pwd2**.

Next, we can see the **input** element for entering the password (**id="pwd"**). Notice that the element binds to the **pwd** field; whatever value is entered into the **input** element is automatically assigned to **pwd**. This is the same functionality we used for the Sign in page.

```
<input type="password"
 class="form-control" id="pwd"
 placeholder="Please type in your password"
 required="required"
 bind:value={pwd}
 on:keyup={() => checkFields()}
>
```

The difference, though, is that for the Sign up page, we have another password field, which needs to match the value of the first password field (the values of **pwd** and **pwd2** must be the same).

```
<input type="password" class="form-control" id="pwd2"
 placeholder="Please retype your password"
 required="required"
 bind:value={pwd2}
 on:keyup={() => checkFields()}
>
```

The main difference here is that this **input** element binds to the **pwd2** field instead of **pwd**. This **input** element also has a **keyup** event, which executes the **checkFields** function we previously explored.

Finally, we can see the following markup for the Submit button. When you click on this button, the **submit** event of the **form** is triggered.

```
<input class="btn btn-primary" type="submit" value="Submit">
```

That’s all there is to the Sign up page—which, as you have seen, is also straightforward.

Recap

We have made substantial progress in creating and gluing our application's main user interface functionality. However, we are not entirely done.

We still have to put together the UI of the Favorites page and delve into the **`Books`** component, which is used by the app’s main page, and the Favorites page. That’s what the next chapter is all about.

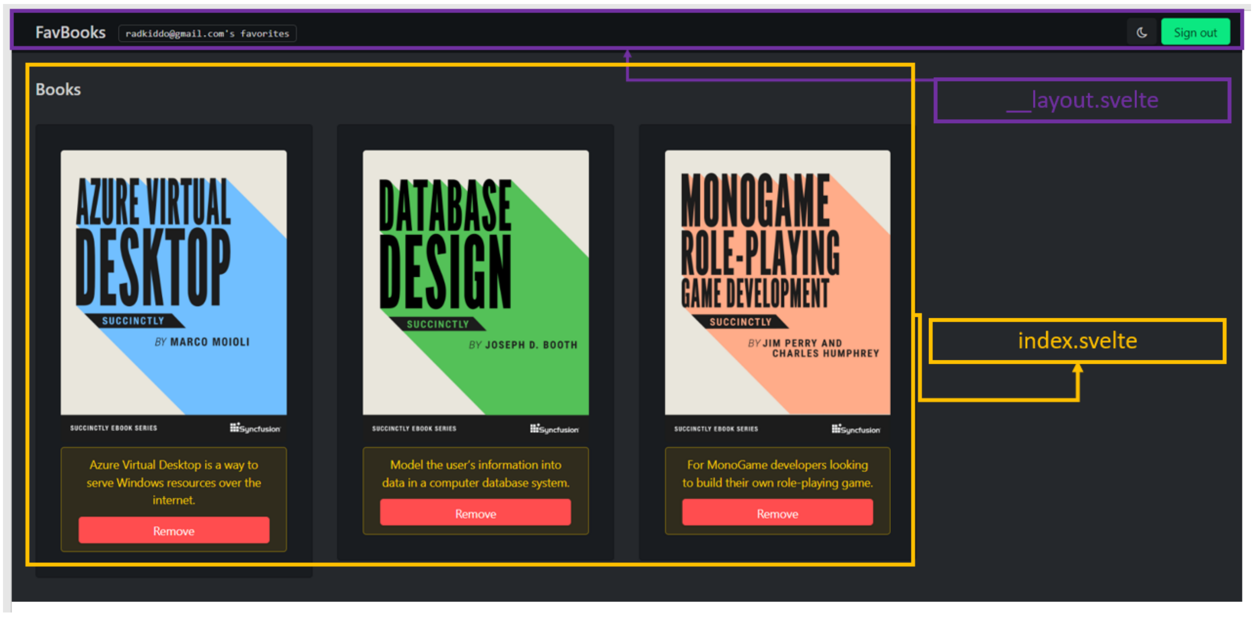
Chapter 6 Favorites UI and Books Component

Quick intro

We will focus on the Favorites page's user interface, which is only accessible once a user has signed in.

The Favorites page resides within the src/routes/favorites folder within the project's structure and has two parts, __layout.svelte and index.svelte.

To get a visual understanding of how the __layout.svelte and index.svelte files make up the Favorites page, let's look at the following figure.



FavBooks
radkiddo@gmail.com's favorites
Sign out
__layout.svelte
Books
index.svelte
Azure Virtual Desktop is a way to serve Windows resources over the internet.
Remove
Model the user's information into data in a computer database system.
Remove
For MonoGame developers looking to build their own role-playing game.
Remove

Figure 6-a: Favorites Page (__layout.svelte and index.svelte)

We can see that the Favorites page has a navigation bar and that markup is found within the __layout.svelte file (highlighted in **purple**).

The Favorites page also displays a list of favorite books, and that markup is found within the index.svelte file (highlighted in **yellow**).

As we'll see later, most of the index.svelte references Books.svelte, which is the component that encapsulates the markup for displaying the list of books.

Let's begin by exploring __layout.svelte.

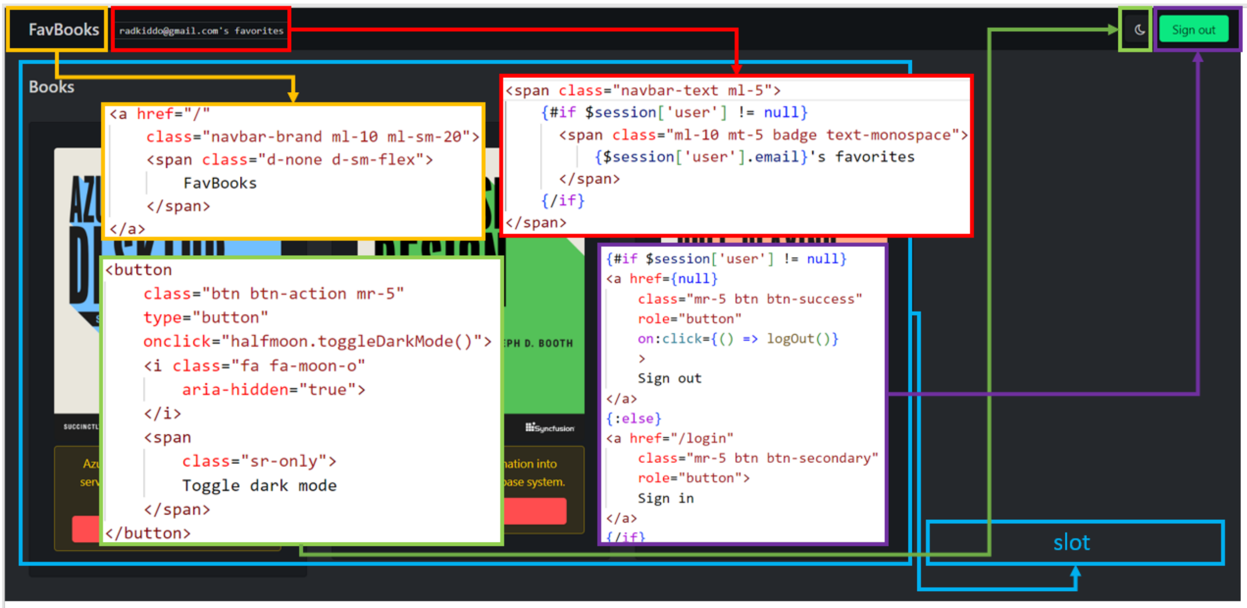
Favorites page (__layout.svelte)

The following listing shows the finished code for the __layout.svelte file found within the src/routes/favorites folder (not to be confused with the __layout.svelte file located under the root of the src/routes folder).

Code Listing 6-a: The finished __layout.svelte File (src/routes/favorites folder)

```
<script>
    import { onAuthStateChanged, signOut } from "Firebase/auth"
    import { onMount } from "svelte"
    import { auth } from "../../firebase.js"
    import { getStores } from "$app/stores"
    import { goto } from "$app/navigation"

    let { session } = getStores()

    onMount(() => {
          onAuthStateChanged(
              auth,
              (user) => {
                  session.set({ user })
              },
              (error) => {
                  session.set({ user: null })
                  console.log(error)
              }
          );
    });

    const logOut = async () => {
        await signOut(auth)
        await goto('/')
    }
</script>

<div id="page-wrapper"
    class="page-wrapper with-navbar"
    data-sidebar-type="overlayed-sm-and-down">

    <nav class="navbar">
        <div class="navbar-content">
        </div>
        <a href="/"
            class="navbar-brand ml-10 ml-sm-20">
            <span class="d-none d-sm-flex">
                FavBooks
            </span>
        </a>
        <span class="navbar-text ml-5">
            {#if $session['user'] != null}
              <span class="ml-10 mt-5 badge text-monospace">
                  {$session['user'].email}'s favorites
              </span>
            {/if}
        </span>
        <div class="navbar-content ml-auto">
            <button
                class="btn btn-action mr-5"
                type="button"
                onclick="halfmoon.toggleDarkMode()">
                <i class="fa fa-moon-o"
                    aria-hidden="true">
                </i>
                <span
                    class="sr-only">
                    Toggle dark mode
                </span>
            </button>
            {#if $session['user'] != null}
            <a href={null}
                class="mr-5 btn btn-success"
                role="button"
                on:click={() => logOut()}
                >
                Sign out
            </a>
            {:else}
            <a href="/login"
                class="mr-5 btn btn-secondary"
                role="button">
                Sign in
            </a>
            {/if}
        </div>
    </nav>
    <slot />
</div>
```

Before reviewing the details of the code, let's have a quick look at the following figure, to get a visual understanding of how __layout.svelte is structured.



```
<a href="/"
    class="navbar-brand ml-10 ml-sm-20">
    <span class="d-none d-sm-flex">
        FavBooks
    </span>
</a>
```

```
<span class="navbar-text ml-5">
    {#if $session['user'] != null}
      <span class="ml-10 mt-5 badge text-monospace">
          {$session['user'].email}'s favorites
      </span>
    {/if}
</span>
```

```
<button
    class="btn btn-action mr-5"
    type="button"
    onclick="halfmoon.toggleDarkMode()">
    <i class="fa fa-moon-o"
        aria-hidden="true">
    </i>
    <span
        class="sr-only">
        Toggle dark mode
    </span>
</button>
```

```
{#if $session['user'] != null}
<a href={null}
    class="mr-5 btn btn-success"
    role="button"
    on:click={() => logOut()}
    >
    Sign out
</a>
{:else}
<a href="/login"
    class="mr-5 btn btn-secondary"
    role="button">
    Sign in
</a>
{/if}
```

Figure 6-b: Favorites Page (Focusing on __layout.svelte)

Looking at the preceding image, we can see that the part highlighted in **yellow** corresponds to the page's title. When you click it, you'll be redirected to the app's main page.

The section in **red** corresponds to the user's email that is signed in. The section highlighted in **green** corresponds to the dark mode toggle button.

The section in **purple** corresponds to the Sign out button, and the section in **blue** corresponds to the **slot** tag where the list of favorite books is shown.

Now, let's focus on the code. Let's begin with the **import** statements.

```
import { onAuthStateChanged, signOut } from "Firebase/auth"
import { onMount } from "svelte"
import { auth } from "../../firebase.js"
import { getStores } from "$app/stores"
import { goto } from "$app/navigation"
```

From the **Firebase/auth** library, we **import** the **onAuthStateChanged** event and the **signOut** method.

From the Svelte core module (**svelte**), we **import** the **onMount** lifecycle event. Then, we **import** the **auth** object from the **firebase.js** utility file (which we'll explore later).

We also **import** the **getStores** method from the **app/stores** module and the **goto** method from **app/navigation**.

By invoking the **getStores** method, we can get the current **session** information by destructuring the result. The **session** information is used for accessing user details.

```
let { session } = getStores()
```

When the component mounts, we subscribe to the Firebase **onAuthStateChanged** event, which we can use to assign the current **user** details to the active **session**.

```
onMount(() => {
  onAuthStateChanged(
    auth,
    (user) => {
      session.set({ user })
    },
    (error) => {
      session.set({ user: null })
      console.log(error)
    }
  );
});
```

This way, we always know which **user** is active and signed in. Finally, the **logOut** function executes when the **Sign out** button is clicked.

```
const logOut = async () => {
  await signOut(auth)
  await goto('/')
}
```

This function invokes the Firebase **signOut** method, which logs out the signed-in user, and then redirects the user to the application's main page.

With regards to the markup, we find the following conditional rendering.

```
{#if $session['user'] != null}
  <span class="ml-10 mt-5 badge text-monospace">
    {$session['user'].email}'s favorites
  </span>
{/if}
```

For the currently signed-in user (**$session['user'] != null**), the user's email address will be shown **{$session['user'].email}**.

The Sign out button will be rendered if the user has signed in (**$session['user'] != null**). Otherwise, the Sign in button is displayed (as a fail-safe option).

```
{#if $session['user'] != null}
  <a href={null}
   class="mr-5 btn btn-success"
   role="button"
   on:click={() => logOut()}
  >
   Sign out
  </a>
{:else}
  <a href="/login"
   class="mr-5 btn btn-secondary"
   role="button">
   Sign in
  </a>
{/if}
```

You might have noticed that it's not in the __layout.svelte file that the redirection to login.svelte happens when there's no signed-in user. This occurs in the **`load`** function of index.svelte, as we'll see shortly.

Within **__layout.svelte**, we subscribe and retrieve the current **`session`** and **`user`** details, and do not take action based on the **`user`** status (signed-in or not).

Finally, we have the **`slot`** tag.

Favorites page (index.svelte)

The markup content of the index.svelte file will be inserted into the **`slot`** tag found within __layout.svelte.

Let's have a look at the finished code of index.svelte found within the src/routes/favorites folder.

Code Listing 6-b: The Finished index.svelte File (src/routes/favorites folder)

```
<script context="module">
    import Books from "$lib/Books.svelte"
    import { auth,
        delFav } from "../../firebase.js"

    export const load = async () => {
          if (auth?.currentUser == null) {
              return {
                  status: 302,
                  redirect: "/login",
              }
          }

          return {
              status: 200
          }
    }
```

```
</script>

<script>
    import { navigating } from '$app/stores'

    const fav = "fav"
    const btnAction = async (event) => {
        await delFav(event.detail.uuid)
    }
</script>

{#if $navigating}
    <p>Fetching favorites...</p>
{:else}
    <Books books=favorites {fav}
        btnText="Remove"
        on:btnAction={btnAction} />
{/if}
```

The code within the **`<script context="module">`** tag runs outside the component instance. First, we **`import`** what we need.

```
import Books from "$lib/Books.svelte"
import { auth, delFav } from "../../firebase.js"
```

In this case, we import the **`Books`** component from the **Books.svelte** file and import the **`auth`** object and **`delFav`** function from our **firebase.js** utility file.

Next, we find the **`load`** function, which checks whether the current user is authenticated, and if not, redirects the user to the **Sign in** page (**login.svelte**).

```
export const load = async () => {
  if (auth?.currentUser == null) {
    return {
      status: 302,
      redirect: "/login",
    }
  }

  return {
    status: 200
  }
}
```

If the current user is authenticated, then a **`status`** with a value of **`200`** is returned, and no redirect takes place, which means that the browser stays on the Favorites page.

Next, within the **`script`** tag that runs within the component instance, we import **`navigating`** from **`app/stores`**. We'll use **`navigating`** to know whether the browser has fully loaded the page or not.

Then, because we are within the **Favorites** page, we declare **`fav`** and give it a non-empty value: **`const fav = "fav"`**.

If we don't do this, when we pass an empty value to the **`Books`** component, the **`Books`** component will assume that we are working with the static list of available books, not the list of favorite books (obtained from Firebase).

Note: Assigning the value "fav" to fav is critical for the Books component and the Favorites page to function correctly.

Next, we find the **`btnAction`** function, responsible for removing a specific book from the list of favorite books (stored in Firebase).

```
const btnAction = async (event) => {
  await delFav(event.detail.uuid)
}
```

Notice that this function receives the **`event`** parameter dispatched from the **`Books`** component, which contains the book's details to be removed from the list of favorite books.

The book's details to be removed are accessible via the **`detail`** property (**`event.detail`**), and **`uuid`** (**`event.detail.uuid`**) represents the book's unique identifier within Firebase.

The book's removal from Firebase is done by invoking the **`delFav`** function contained within the **firebase.js** utility file, which we'll explore later.

Regarding the markup, all that happens is that we conditionally render the **`Books`** component (to show the list of favorite books) if the page has fully loaded (determined by checking the value of **`$navigating`**).

```
{#if $navigating}
    <p>Fetching favorites...</p>
{:else}
    <Books books=favorites {fav}
     btnText="Remove"
     on:btnAction={btnAction} />
{/if}
```

Notice that on this occasion, as part of the **`Books`** component parameters, we assign to the **`books`** property the value of **`favorites`**, and to the button's text (**`btnText`**), the text **`Remove`** is assigned.

Notice how the function **`btnAction`** binds to **`btnAction`** dispatched from within the **`Books`** component.

Great! We are now ready to see where the magic happens by exploring the **Books** component.

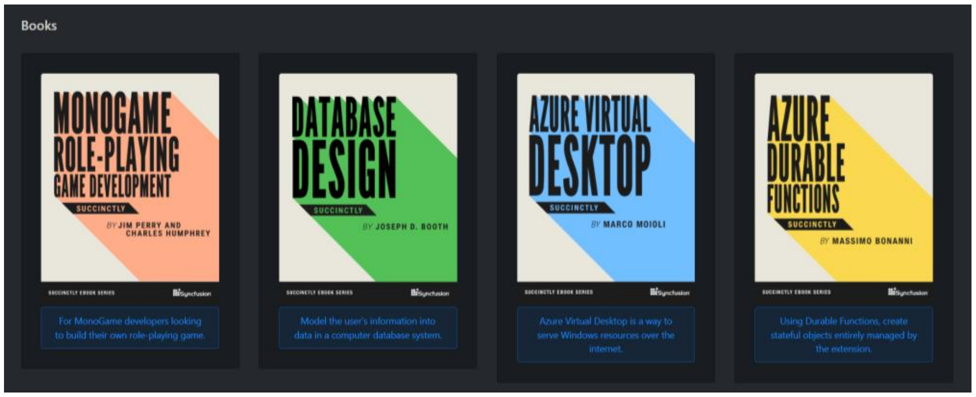
Books.svelte

One fundamental part of the UI is required by both the application's main page and the Favorites page, and that's the **Books** component.

The Books.svelte file is located under the src/lib folder, and it encapsulates all the logic of the reusable **Books** component.

Perhaps the most remarkable feature of the component is its ability to display the list of available books obtained through the execution of src/routes/api/index.js and the list of favorite books from Firebase.

We already know how the **Books** component renders its elements. For example, the **Books** component displays the list of available books on the application's main page (when a user is not signed in).


Books
MONOGAME ROLE-PLAYING GAME DEVELOPMENT SUCCINCTLY BY JIM PERRY AND CHARLES HUMPHREY
DATABASE DESIGN SUCCINCTLY BY JOSEPH D. BOOTH
AZURE VIRTUAL DESKTOP SUCCINCTLY BY MARCO MOIOLI
AZURE DURABLE FUNCTIONS SUCCINCTLY BY MASSIMO BONANNI

For MonoGame developers looking to build their own role-playing game.

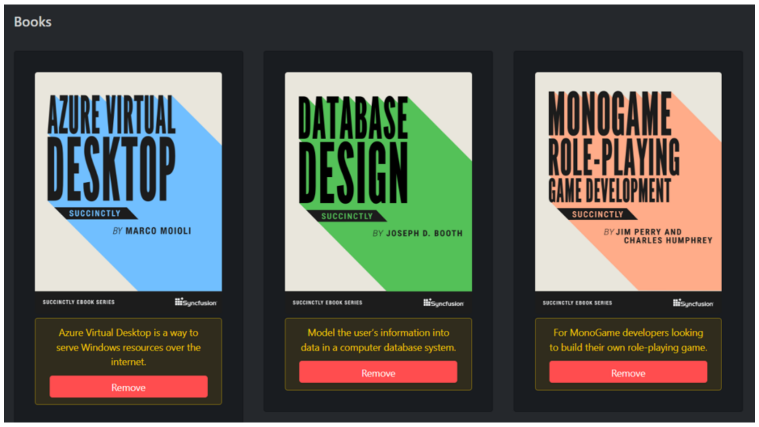
Model the user's information into data in a computer database system.

Azure Virtual Desktop is a way to serve Windows resources over the internet.

Using Durable Functions, create stateful objects entirely managed by the extension.

Figure 6-c: The Books Component (As Seen on the Main Page—No Signed-in User)

The following figure shows how the **Books** component displays the list of favorite books for a signed-in user.



Azure Virtual Desktop is a way to serve Windows resources over the internet.

Model the user's information into data in a computer database system.

For MonoGame developers looking to build their own role-playing game.

Figure 6-d: The Books Component (As Seen on the Favorites Page—The User Signed In)

The difference between both scenarios is the value passed to the **books** property and how the **Books** component reacts accordingly. To understand this better, let's explore this component's finished code.

Code Listing 6-c: The Finished Books.svelte File (src/lib folder)

```
<script>
    import { createEventDispatcher,
        onMount } from "svelte"
    import { getStores } from "$app/stores"
    import { getFavs } from "../firebase.js"

    const dispatch = createEventDispatcher()
    let { session } = getStores()

    export let books = []
    export let fav = ""
    export let btnText

    onMount(async () => {
      if (fav !== "") {
        books = []
        books = await getFavs()
      }
    })

    const emitBtnAction = (book) => {
        dispatch("btnAction", book)

        if (fav !== "") {
            books = books.filter(item => item.title != book.title)
```

```
         }
     }
  </script>

 {#if books?.length > 0}
     <div class="content-wrapper">
         <div class="container-fluid">
         <div class="content">
             <h2 class="content-title">
             Books
             </h2>
         </div>
         <div class="row row-eq-spacing">
             {#each books as book}
                 <div class="col-6 col-lg-3">
                     <div class="mb-20 card">
                         <a target="_blank" href={book.url}>
                         <img src={book.cover}
                             class="img-fluid rounded"
                             alt="book cover" />
                         </a>
                         <div
                             class=
                                 "{fav === 'fav' ?
                                 'alert-secondary ' :
                                 'alert-primary '}
                                 text-center alert"
                             role="alert">
                             {book.description}
                             {#if $session['user'] != null}
                                 <a href={null}
                                     class=
                                     "{fav === 'fav' ?
                                     'btn-danger ' :
                                     'btn-primary '}
                                     mt-10 btn
                                     btn-block"
                                     role="button"
                                     on:click={() => emitBtnAction(book)}>
                                     {btnText}
                                 </a>
                             {/if}
                         </div>
                     </div>
```

```
                </div>
            {/each}
        </div>
        </div>
    </div>
{:else}
    <div class="content-wrapper">
    <div class="row row-eq-spacing">
      <div class="col-sm"></div>
      <div class="col-sm">
        <div class="text-center alert alert-primary" role="alert">
            <h4 class="text-center alert-heading">
                No {fav !== "" ? 'favorites' : 'books'} found
            </h4>
            Click <a href="/" class="alert-link">here</a> to add one :).
        </div>
      </div>
      <div class="col-sm"></div>
    </div>
  </div>
{/if}
```

When the component instance runs, the code contained within the **`script`** tag executes. We begin, as usual, by importing what we need.

```
import { createEventDispatcher, onMount } from "svelte"
import { getStores } from "$app/stores"
import { getFavs } from "../firebase.js"
```

First, we **`import`** createEventDispatcher and the **`onMount`** lifecycle event from the Svelte core module (**`svelte`**).

Next, we **`import`** the **`getStores`** method from **`app/stores`**, which we'll use to get the **`session`** and **`user`** details.

And finally, we **`import`** the **`getFavs`** function from our **firebase.js** utility file, which will be responsible for retrieving from Firebase the list of favorite books for the signed-in user.

Following that, we create an event dispatcher instance, which, as its name implies, will be used to emit an event to the parent components that will implement the **`Books`** component. The parent components are src/routes/index.svelte and src/routes/favorites/index.svelte.

```
const dispatch = createEventDispatcher()
```

Next, we invoke the **`getStores`** method to get the current **`session`** information.

```
let { session } = getStores()
```

After we declare the component properties in Svelte (as exported variables), this is how Svelte component properties are created:

```
export let books = []
export let fav = ""
export let btnText
```

Then, on the component's **onMount** lifecycle event, if the value of **fav** is not an empty string (which means that we want to get the list of favorite books), the **getFavs** function is invoked, and the list of favorite books is retrieved from Firebase.

```
onMount(async () => {
  if (fav !== "") {
    books = []
    books = await getFavs()
 }})
```

Then, we have the **emitBtnAction** function, primarily responsible for dispatching the **btnAction** event to the parent components that implement the **Books** component.

```
const emitBtnAction = (book) => {
  dispatch("btnAction", book)

  if (fav !== "") {
    books = books.filter(item => item.title != book.title)
  }
}
```

That event dispatching is needed so that src/routes/index.svelte can implement the Add to favorites button functionality for each book, and src/routes/favorites/index.svelte can implement the Remove button functionality for each book.

Remember that the Add to favorites button functionality adds the book to Firebase as a favorite book, and the Remove button functionality deletes the book from Firebase.

Notice, however, that the actual deletion of a book from the **books** array (only when the Favorites page is displayed—**fav !== ""**) is not delegated to a parent component, but done directly within the **Books** component.

That's because the **books** array is not the responsibility of any parent component, but instead of the **Books** component. The **Books** component contains the local copy of the list of books on display.

Moving on to the markup, we find the following conditional rendering.

```
{#if books?.length > 0}
```

This means that the list of books (either the static list of available books or the list of favorite books obtained from Firebase) will only be rendered if the **books** array contains at least one book.

Otherwise (`{:else}`), a message is shown telling the user that there are no books to display, and how the user can add a book to the favorites list—redirecting the user to the application's main page.

```
<div class="text-center alert alert-primary" role="alert">
  <h4 class="text-center alert-heading">
    No {fav !== "" ? 'favorites' : 'books'} found
  </h4>
  Click <a href="/" class="alert-link">here</a> to add one :).
</div>
```

Notice that this message and scenario only apply to the Favorites page because the static list of available books is prefilled and never empty, as we'll see later.

Here comes the exciting part: how the **books** are rendered. This is possible thanks to Svelte's iterator: `{#each books as book}`.

In essence, for every **book** contained within the **books** array, the book's properties, including the book's cover (`book.cover`), will be rendered.

Notice that only if a user has signed in (`$session['user'] != null`) will the Add to favorites button or the Remove button be shown, depending on the value of `fav`.

The value of `fav` also determines the button's color by establishing the correct CSS class to apply to the button.

```
{#if $session['user'] != null}
  <a href={null}
   class=
   "{fav === 'fav' ?
   'btn-danger ' :
   'btn-primary '}
   mt-10 btn
   btn-block"
   role="button"
   on:click={() => emitBtnAction(book)}>
   {btnText}
  </a>
{/if}
```

When `fav === 'fav'`, it means that the book is already part of the favorites list, and as such, the `btn-danger` CSS class makes the button **red**, making it a **Remove** button.

When `fav !== 'fav'`, it means that the book is not part of the favorites list but instead part of the static list of available books, and as such, the `btn-primary` CSS class makes the button **blue**; making it an **Add to favorites** button.

When either the Add to favorites button or the Remove button is clicked, the `emitBtnAction` function is executed. Thus, the event is dispatched to the parent component to implement the respective functionality, with the currently selected `book` passed as a parameter.

Recap

Well done for following along until now! We finally have the application’s UI ready. Next, we’ll wrap up the application and book by exploring the code that makes the back-end part of the app work.

Chapter 7 Back-end App Functionality

Quick intro

We've covered quite a bit of ground and are now ready to wrap up this book with this final chapter, which will cover application-specific, back-end functionality. Let's begin by talking about the static list of available books.

api/index.js

The list of books on the application's main page is retrieved from the index.js file within the src/routes/api folder.

Code Listing 7-a: The Finished index.js File (src/routes/api Folder)

```
export const get = () => {
    return {
        body: [
            {   b_id: 1,
                title:
                  'MonoGame Role-Playing Game Development Succinctly',
                cover: 'https://cdn.syncfusion.com/
                  content/images/downloads/
                  ebook/ebook-cover/
                  monogame-role-playing-game-development-succinctly.png',
                description: 'For MonoGame developers looking to build
                  their own role-playing game.',
                url: 'https://www.syncfusion.com/succinctly-free-
                  ebooks/
                  monogame-role-playing-game-development-succinctly'
            },
            {
                b_id: 2,
                title: 'Database Design Succinctly',
                cover: 'https://cdn.syncfusion.com/
                  content/images/downloads/
                  ebook/ebook-cover/
                  database-design-succinctly.png',
                description: 'Model the user's information
                  into data in a computer database system.',
                url: 'https://www.syncfusion.com/succinctly-free-
                  ebooks/database-design-succinctly'
            },
            {
                b_id: 3,
                title: 'Azure Virtual Desktop Succinctly',
                cover: 'https://cdn.syncfusion.com/
                  content/images/downloads/
                  ebook/ebook-cover/
                  azure-virtual-desktop-succinctly.png',
                description: 'Azure Virtual Desktop is a way to serve
                  Windows resources over the internet.',
                url: 'https://www.syncfusion.com/succinctly-free-
                  ebooks/azure-virtual-desktop-succinctly'
            },
            {
                b_id: 4,
                title: 'Azure Durable Functions Succinctly',
                cover: 'https://cdn.syncfusion.com/
                  content/images/downloads/
                  ebook/ebook-cover/
                  azure-durable-functions-succinctly.png',
                description: 'Using Durable Functions,
                  create stateful objects
                  entirely managed by the extension.',
                url: 'https://www.syncfusion.com/succinctly-free-
                  ebooks/azure-durable-functions-succinctly'
            },
        ],
    };
};
```

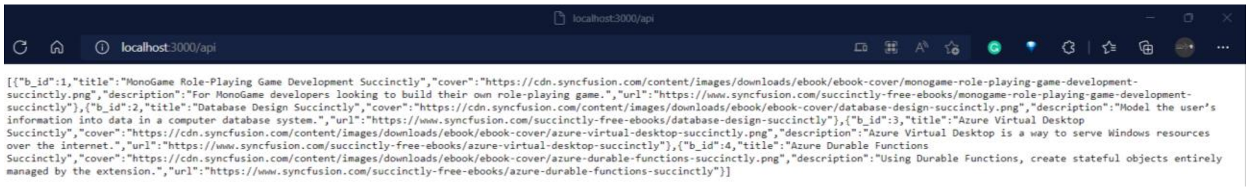
Note: The items and lines highlighted in yellow in the preceding listing are single-line strings, and I've split them into multiple lines to make the listing more readable. However, if you are going to copy and paste the content of this listing elsewhere, make sure those highlighted items aren't split into multiple lines.

Let's review what's going on. Essentially, we are exporting a single function called **`get`** that returns a hard-coded JSON object containing a **`body`** property with an array of objects (each representing a book).

Each book object has various properties, such as the book ID (**`b_id`**), the book title (**`title`**), the book cover (**`cover`**), the book description (**`description`**), and the book URL (**`url`**).

index.js is exposed as an API endpoint, which means that if you run the application from the built-in terminal in VS Code using the **`npm run dev`** command, you'll be able to access the endpoint as http://localhost:3000/api (if your app runs on port 3000, like mine).

Following is how it looks in my environment.



```
[{"b_id":1,"title":"MonoGame Role-Playing Game Development Succinctly","cover":"https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-cover/monogame-role-playing-game-development-
succinctly.png","description":"For MonoGame developers looking to build their own role-playing game.","url":"https://www.syncfusion.com/succinctly-free-ebooks/monogame-role-playing-game-development-
succinctly"},{"b_id":2,"title":"Database Design Succinctly","cover":"https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-cover/database-design-succinctly.png","description":"Model the user’s
information into data in a computer database system.","url":"https://www.syncfusion.com/succinctly-free-ebooks/database-design-succinctly"},{"b_id":3,"title":"Azure Virtual Desktop
Succinctly","cover":"https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-cover/azure-virtual-desktop-succinctly.png","description":"Azure Virtual Desktop is a way to serve Windows resources
over the internet.","url":"https://www.syncfusion.com/succinctly-free-ebooks/azure-virtual-desktop-succinctly"},{"b_id":4,"title":"Azure Durable Functions
Succinctly","cover":"https://cdn.syncfusion.com/content/images/downloads/ebook/ebook-cover/azure-durable-functions-succinctly.png","description":"Using Durable Functions, create stateful objects entirely
managed by the extension.","url":"https://www.syncfusion.com/succinctly-free-ebooks/azure-durable-functions-succinctly"}]
```

Figure 7-a: The Static List of Available Books (Exposed as an API Endpoint)

Note: I just included four books within that list. You can add others if you wish. You can also think for the future about how to make this list dynamic and store the available books in Firebase instead.

The list of static books is retrieved within index.svelte (found under the root of the src/routes folder) by the `load` function, which uses the browser’s built-in `fetch` API to access the application’s API endpoint (`/api`) and get the data.

```
export const load = async({fetch}) => {
  const res = await fetch("/api")
  const books = await res.json()
  return {
    props: {
      books,
    },
  }
}
```

That’s how the list of available books is retrieved. As you can see, it’s straightforward to create an API endpoint with SvelteKit.

utils.js

Another key aspect of our application is the set of utility functions available for validating data used by the Sign in and Sign up pages. Let’s have a look at these functions.

Code Listing 7-b: The Finished utils.js File (src/lib Folder)

```
export const checkEmail = (email) => {
    const pattern = new RegExp("([!#-'*+/-9=?A-Z^-~-]+(\.[!#-'*+/-9=?A-Z^-~-]+)*|\"\(\[\]!#-[^-~ \t]|(\\[\t -~]))+\")@([!#-'*+/-9=?A-Z^-~-]+(\.[!#-'*+/-9=?A-Z^-~-]+)*|\[[\t -Z^-~]*])")
    return pattern.test(email)
}

export const loginChecks = (email, pwd) => {
    let res = undefined

    if (!checkEmail(email)) {
        res = {
          content: "The email is not formatted correctly.",
          title: "Invalid email",
          alertType: "alert-primary",
        }
    }
    else if (pwd?.length < 8) {
        res = {
          content: "The password must be at least 8 characters long.",
          title: "Password too short",
          alertType: "alert-secondary",
        }
    }

    return res
}

export const registerChecks = (email, pwd, pwd2) => {
    let res = undefined

    if (!checkEmail(email)) {
        res = {
          content: "The email is not formatted correctly.",
          title: "Invalid email",
          alertType: "alert-primary",
        }
    }
    else if (pwd?.toLowerCase() !== pwd2?.toLowerCase()) {
        res = {
          content: "The passwords provided do not match.",
          title: "Passwords do not match",
          alertType: "alert-secondary",
        }
    }
    else if (pwd?.length < 8 || pwd2?.length < 8) {
        res = {
          content: "The password must be at least 8 characters long.",
          title: "Password too weak",
          alertType: "alert-secondary",
        }
    }
    return res
}

export const fbLoginChecks = (e) => {
    let res = undefined

    if (e.toString().indexOf("wrong-password") > 0) {
        res = {
          content:
            "The password is incorrect. Please try the correct one.",
          title: "Incorrect password",
          alertType: "alert-secondary",
        }
    }
    else if (e.toString().indexOf("user-not-found") > 0) {
        res = {
          content:
   "This user does not exist. Please use a different user or sign up.",
          title: "User does not exist",
          alertType: "alert-secondary",
        }
    }
    else {
        res = {
          content: "Internal database error. Please try again later.",
          title: "Internal error",
          alertType: "alert-danger",
        }
    }

    return res
}

export const fbRegisterChecks = (e) => {
    let res = undefined

    console.log(e.toString())
    if (e.toString().indexOf("in-use") > 0) {
        res = {
          content:
          "This user already exists. Please try with a different email.",
          title: "User exists",
          alertType: "alert-secondary",
        }
    }
    else {
        res = {
          content: "Internal database error. Please try again later.",
          title: "Internal error",
          alertType: "alert-danger",
        }
    }

    return res
}
```

First, we have the **checkEmail** function. As its name implies, this function validates (using a regular expression) that the **email** provided is valid (correctly formatted as a proper email address).

Then, we have the **loginChecks** function, which invokes the **checkEmail** function and checks the minimum password (**pwd**) length, returning a message to the user if validation fails. This function is used by the Sign in page.

```
export const loginChecks = (email, pwd) => {
  let res = undefined

  if (!checkEmail(email)) {
    res = {
      content: "The email is not formatted correctly.",
      title: "Invalid email",
      alertType: "alert-primary",
    }
  }
  else if (pwd?.length < 8) {
    res = {
      content: "The password must be at least 8 characters long.",
      title: "Password too short",
      alertType: "alert-secondary",
    }
  }

  return res
}
```

Then, we have the **registerChecks** function, which performs the same validations as the **loginChecks** function.

The only difference is that the **registerChecks** function also validates that both the password (**pwd**) and the repeated password (**pwd2**) are the same. The **registerChecks** function is used by the Sign up page.

```
export const registerChecks = (email, pwd, pwd2) => {
    let res = undefined

    if (!checkEmail(email)) {
        res = {
          content: "The email is not formatted correctly.",
          title: "Invalid email",
          alertType: "alert-primary",
        }
    }
    else if (pwd?.toLowerCase() !== pwd2?.toLowerCase()) {
        res = {
          content: "The passwords provided do not match.",
          title: "Passwords do not match",
          alertType: "alert-secondary",
        }
    }
    else if (pwd?.length < 8 || pwd2?.length < 8) {
        res = {
          content: "The password must be at least 8 characters long.",
          title: "Password too weak",
          alertType: "alert-secondary",
        }
    }

    return res
}
```

Next, we have the **fbLoginChecks** function, which executes if an error occurs when attempting to authenticate the user with Firebase.

This function is used by the Sign in page and interprets possible Firebase errors, and it outputs a user-friendly version of the error.

```
export const fbLoginChecks = (e) => {
    let res = undefined

    if (e.toString().indexOf("wrong-password") > 0) {
        res = {
          content:
            "The password is incorrect. Please try the correct one.",
          title: "Incorrect password",
          alertType: "alert-secondary",
        }
    }
    else if (e.toString().indexOf("user-not-found") > 0) {
        res = {
          content:
   "This user does not exist. Please use a different user or sign up.",
          title: "User does not exist",
          alertType: "alert-secondary",
        }
    }
    else {
        res = {
          content: "Internal database error. Please try again later.",
          title: "Internal error",
          alertType: "alert-danger",
        }
    }

    return res
}
```

Then, we have the **fbRegisterChecks** function, which has a similar purpose and functionality as the **fbLoginChecks** function but is used by the Sign up page.

```
export const fbRegisterChecks = (e) => {
    let res = undefined

    console.log(e.toString())
    if (e.toString().indexOf("in-use") > 0) {
        res = {
          content:
          "This user already exists. Please try with a different email.",
          title: "User exists",
          alertType: "alert-secondary",
        }
    }
    else {
        res = {
          content: "Internal database error. Please try again later.",
          title: "Internal error",
          alertType: "alert-danger",
        }
    }

    return res
}
```

These are the utility functions used by the Sign in and Sign up pages, primarily for data validation and handling any possible Firebase errors during authentication or user registration.

firebase.js

The firebase.js file, which resides in the root of the src folder of our project, as its name implies, handles everything to do with Firebase.

If you recall, we added the required Firebase parameters in one of the earlier chapters. Here is the finished file.

Code Listing 7-c: The Finished firebase.js File (src Folder)

```
import { initializeApp } from "firebase/app";
import { getAuth } from "Firebase/auth"
import { collection, query, where,
  doc, addDoc, getDocs, getFirestore,
  serverTimestamp, deleteDoc }
  from "Firebase/firestore/lite"

// Your web app's Firebase configuration
const firebaseConfig = {
  apiKey: "<< Your Firebase apiKey goes here >>",
  authDomain: "<< Your Firebase authDomain goes here >>",
  projectId: "<< Your Firebase projectId goes here >>",
  storageBucket: "<< Your Firebase storageBucket goes here >>",
  messagingSenderId: "<< Your Firebase messagingSenderId goes here >>",
  appId: "<< Your Firebase appId goes here >>"
}

let app = initializeApp(firebaseConfig);
const auth = getAuth(app);
let db = getFirestore(app);
const favCollection = collection(db, "favs");
const userDoc = (userId) => doc(db, "users", userId);
const favDoc = (id) => doc(db, "favs", id)

const delFav = async(id) => {
  await deleteDoc(favDoc(id))
}

const addFav = async(book) => {
  const q = query(collection(db, "favs"),
    where("title", "==", book.title),
    where("owner", "==", auth?.currentUser?.uid))
  const querySnapshot = await getDocs(q)

  if (querySnapshot?.docs?.length === 0) {
    await addDoc(favCollection, {...book,
       owner: auth.currentUser.uid,
       timestamp: serverTimestamp()})
  }
}

const getFavs = async() => {
  let books = []
  let fv = collection(db, "favs")

  if (auth?.currentUser?.uid !== undefined) {
    const q = query(fv, where("owner", "==",
      auth?.currentUser?.uid))

    const snapshot = await getDocs(q)
    books = snapshot.docs.map(doc => {
        return {
            ...doc.data(),
            uuid: doc.id
        }
    })
  }

  return books
}

export {
    auth,
    db,
    favCollection,
    userDoc,
    favDoc,
    addFav,
    getFavs,
    delFav
}
```

Let’s have a look at what’s going on. We **`import`** the required libraries and modules, such as the Firebase SDK and authentication library.

```
import { initializeApp } from "firebase/app";
import { getAuth } from "Firebase/auth"
```

Then, we **`import`** the objects and methods we need to use to interact with the Cloud Firestore database.

```
import { collection, query, where, doc, addDoc, getDocs, getFirestore,
  serverTimestamp, deleteDoc } from "Firebase/firestore/lite"
```

Then, we specify the Firebase configuration parameters. As explained earlier, please note that these will be specific to your Firebase project.

```
const firebaseConfig = {
  apiKey: "<< Your Firebase apiKey goes here >>",
  authDomain: "<< Your Firebase authDomain goes here >>",
  projectId: "<< Your Firebase projectId goes here >>",
  storageBucket: "<< Your Firebase storageBucket goes here >>",
  messagingSenderId: "<< Your Firebase messagingSenderId goes here >>",
  appId: "<< Your Firebase appId goes here >>"
}
```

After that, we declare the constants and variables we will use to interact with Firebase. I've added a **comment** to each so you can understand what each represents.

```
let app = initializeApp(firebaseConfig); // Firebase app reference

const auth = getAuth(app); // Firebase authentication object

let db = getFirestore(app); // Cloud Firestore database reference

const favCollection = collection(db, "favs"); // List of favorite books

const userDoc = (userId) => doc(db, "users", userId); // Specific user

const favDoc = (id) => doc(db, "favs", id) // Specific book within favorites
```

Then we have the **delFav** function, which invokes the **deleteDoc** method from Firebase that removes a document (a specific book—**favDoc(id)**) from the Cloud Firestore database.

```
const delFav = async(id) => {
  await deleteDoc(favDoc(id))
}
```

Next, we have the **addFav** function. As its name implies, this function is responsible for adding a specific book to the list of favorites for the signed-in user, if it's not been added before.

This function works because the **book** to be added to the list of favorites is passed as a parameter to the function.

Then, the **favs** Cloud Firestore **collection** that contains the list of favorite books is queried for a **book** with a **title** equal to **book.title** and whose **owner** is the user currently signed in (**auth?.currentUser?.uid**).

```
const addFav = async(book) => {
  const q = query(collection(db, "favs"),
    where("title", "==", book.title),
    where("owner", "==", auth?.currentUser?.uid))
  const querySnapshot = await getDocs(q)

  if (querySnapshot?.docs?.length === 0) {
    await addDoc(favCollection, {...book,
      owner: auth.currentUser.uid,
      timestamp: serverTimestamp()})
  }
}
```

Suppose that query yields no result (meaning that the **book** is not yet part of the list of favorite books). In that case, the **book** is added to the **collection** by invoking the Firebase **addDoc** method, indicating the **owner** (the currently signed-in user: **auth.currentUser.uid**) and the server **timestamp**.

Moving on, we find the **getFavs** function, which retrieves the list of all the favorite books for a specific user (the user currently signed in: **auth?.currentUser?.uid**).

```
const getFavs = async() => {
  let books = []
  let fv = collection(db, "favs")

  if (auth?.currentUser?.uid !== undefined) {
    const q = query(fv, where("owner", "==",
      auth?.currentUser?.uid))

    const snapshot = await getDocs(q)
    books = snapshot.docs.map(doc => {
        return {
            ...doc.data(),
            uuid: doc.id
        }
    })
  }

  return books
}
```

The list of favorite books for the user currently signed in is returned as **books**.

And finally, we **export** all the objects, variables, and functions that the rest of the application code will consume as follows.

```
export {
    auth,
    db,
    favCollection,
    userDoc,
    favDoc,
    addFav,
    getFavs,
    delFav

}
```

As you have seen, the Firebase-related functionality of our application was pretty straightforward.

hooks.js

There's one last item I'd like to cover before we finish the application: Svelte Hooks and server-side rendering (SSR) with the handle function.

Within the root of the src folder of our project, I've added a file called hooks.js. Let's have a look at the finished code for hooks.js.

Code Listing 7-d: The Finished hooks.js File (src folder)

```
// See: https://kit.svelte.dev/docs/hooks#handle

/** @type {import('@sveltejs/kit').Handle} */
export async function handle({ event, resolve }) {
    const response = await resolve(event, {
      ssr: event.url.pathname.startsWith('/favorites')
    });

    return response;
}
```

We can override how SvelteKit does server-side rendering by default for a specific route within our application, in this case, for the Favorites page (**`/favorites`**).

By default, SvelteKit prerenders the page on the server-side instead of delivering a blank HTML to the client (browser).

However, by using the **`handle`** function, we can indicate if we want to disable SSR for a specific route.

If you think about it, we don't need the Favorites page to be prerendered, as it's a protected page and can only be accessed once a user is authenticated.

Disabling SSR for that specific page is as simple as setting the value of the **`ssr`** property to false. So, if we want to disable SSR for the Favorites page, we must do the following.

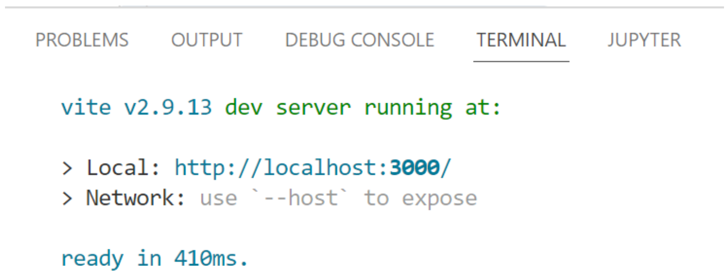
```
ssr: !event.url.pathname.startsWith('/favorites')
```

We would need to negate the value of **`event.url.pathname.startsWith('/favorites')`** as highlighted in yellow in the preceding code.

In other words, hooks.js doesn't alter the functionality of the application. Still, it's a nice feature that can give you more control over how you can interact with SvelteKit's default behavior.

Running the app

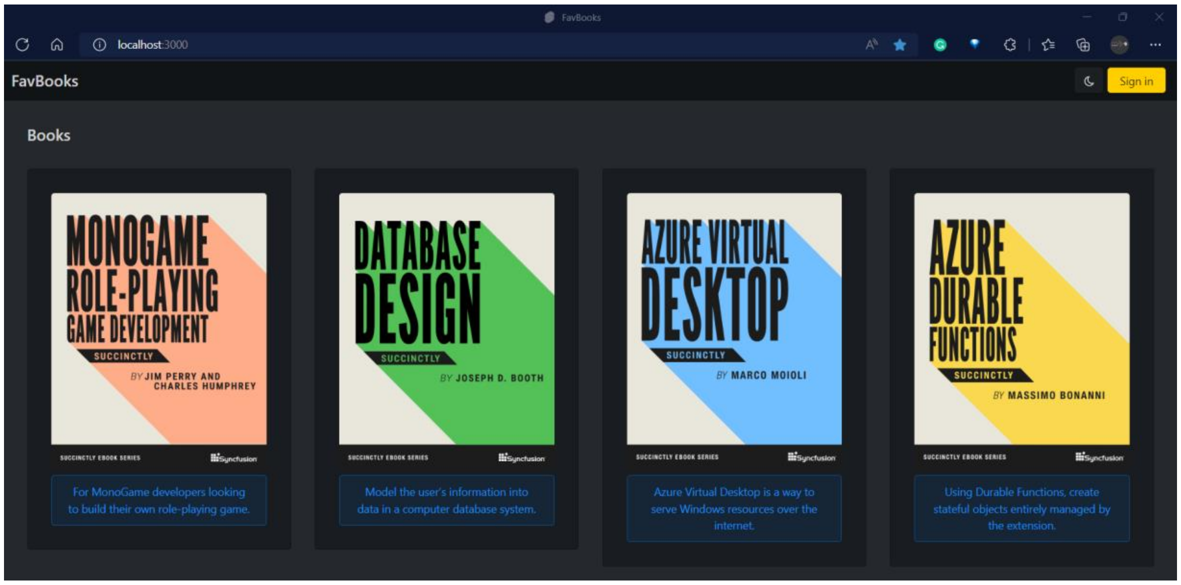
Running the application in development mode is easy, and you must execute the **`npm run dev`** command from the built-in terminal within VS Code.



```
  vite v2.9.13 dev server running at:

  > Local:    http://localhost:3000/
  > Network:  use `--host` to expose

  ready in 410ms.
```

Figure 7-b: Built-in Terminal in VS Code—Running the App

If you're on a Windows machine and press the **Ctrl** key and then click on the **Local** URL (in my case, **http://localhost:3000/**), you'll see that your default browser (in my case, Edge) will open and show the application.



For MonoGame developers looking to build their own role-playing game.

Model the user's information into data in a computer database system.

Azure Virtual Desktop is a way to serve Windows resources over the internet.

Using Durable Functions, create stateful objects entirely managed by the extension.

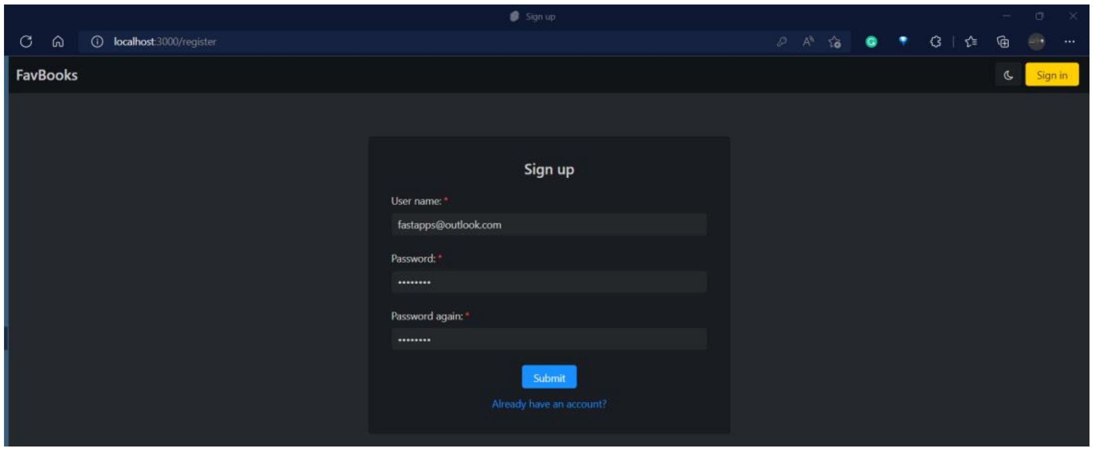
Figure 7-c: Running the App—Edge Browser

Note: If you're using a Mac instead of a Windows machine, press the Command key instead of Ctrl.

Note: As a reminder, you can find the code [repository]() of the finished application on Github.

Let's give the application a try. Perform these steps:

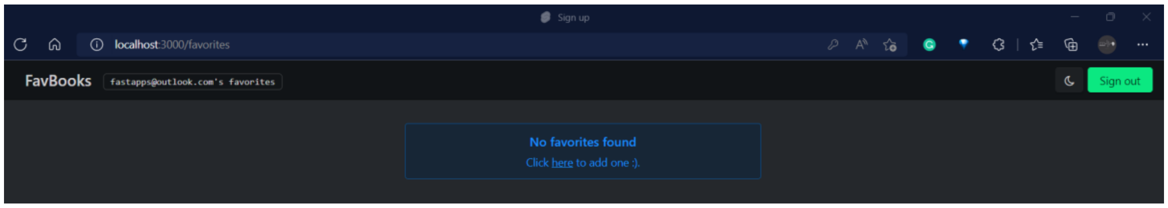
1. Click **Sign in** on the app's main page.
2. Click the **Don't have an account?** link on the bottom of the **Sign in** page.
3. On the **Sign up** page, enter the new account details using a valid email address you own, and click **Submit.**



User name: *
fastapps@outlook.com
Password: *
••••••••
Password again: *
••••••••

Figure 7-d: Testing the App (Signing up)

If this is a new user, you'll automatically register, authenticate, and see the following on the Favorites page.



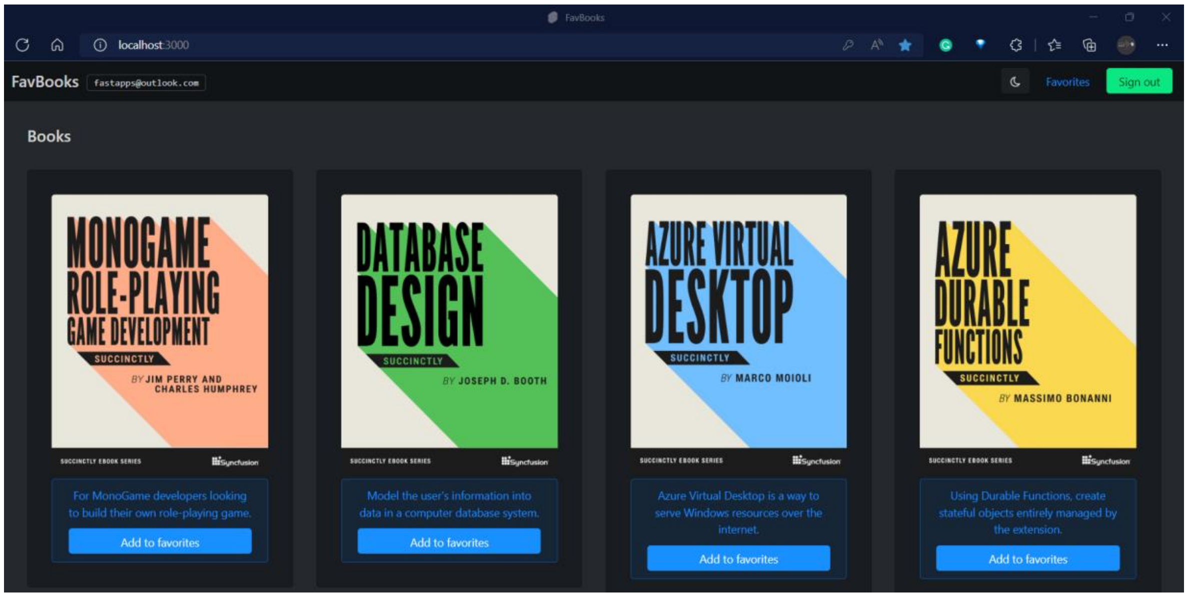
No favorites found

Click here to add one :).

Figure 7-e: Testing the App (Empty Favorites Page)

Notice that the Favorites page is empty, and the Sign out button is now visible.

You'll have to click the **Click here to add one :)** link to add a book to the favorites list, which will take you to the application's main page.


FavBooks
localhost:3000
FavBooks
fastapps@outlook.com
Favorites
Sign out
Books
MONOGAME ROLE-PLAYING GAME DEVELOPMENT SUCCINCTLY BY JIM PERRY AND CHARLES HUMPHREY
DATABASE DESIGN SUCCINCTLY BY JOSEPH D. BOOTH
AZURE VIRTUAL DESKTOP SUCCINCTLY BY MARCO MOIOLI
AZURE DURABLE FUNCTIONS SUCCINCTLY BY MASSIMO BONANNI
Add to favorites
Add to favorites
Add to favorites
Add to favorites

For MonoGame developers looking to build their own role-playing game.

Model the user's information into data in a computer database system.

Azure Virtual Desktop is a way to serve Windows resources over the internet.

Using Durable Functions, create stateful objects entirely managed by the extension.

Figure 7-f: Testing the App (Main Page—User Signed In)

On the application's main page, notice the **Sign out** button is also visible, and for each book available, there's an Add to favorites button.

So, click **Add to favorites** for the first book (as seen from left to right). Once you do that, you'll be redirected to the **Favorites** page, and the book should appear on your list of favorites.

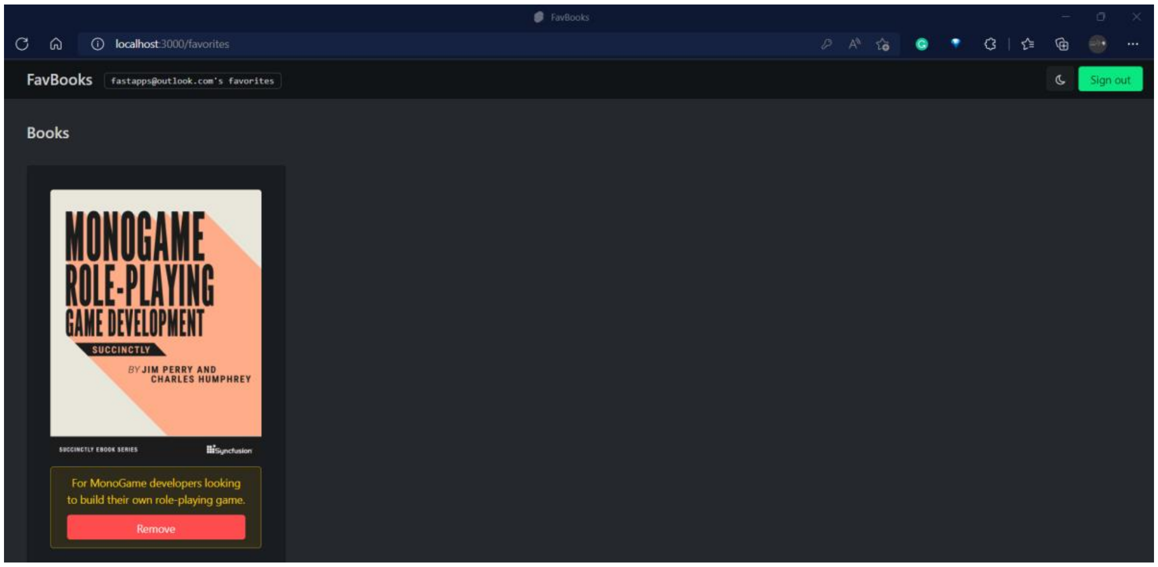
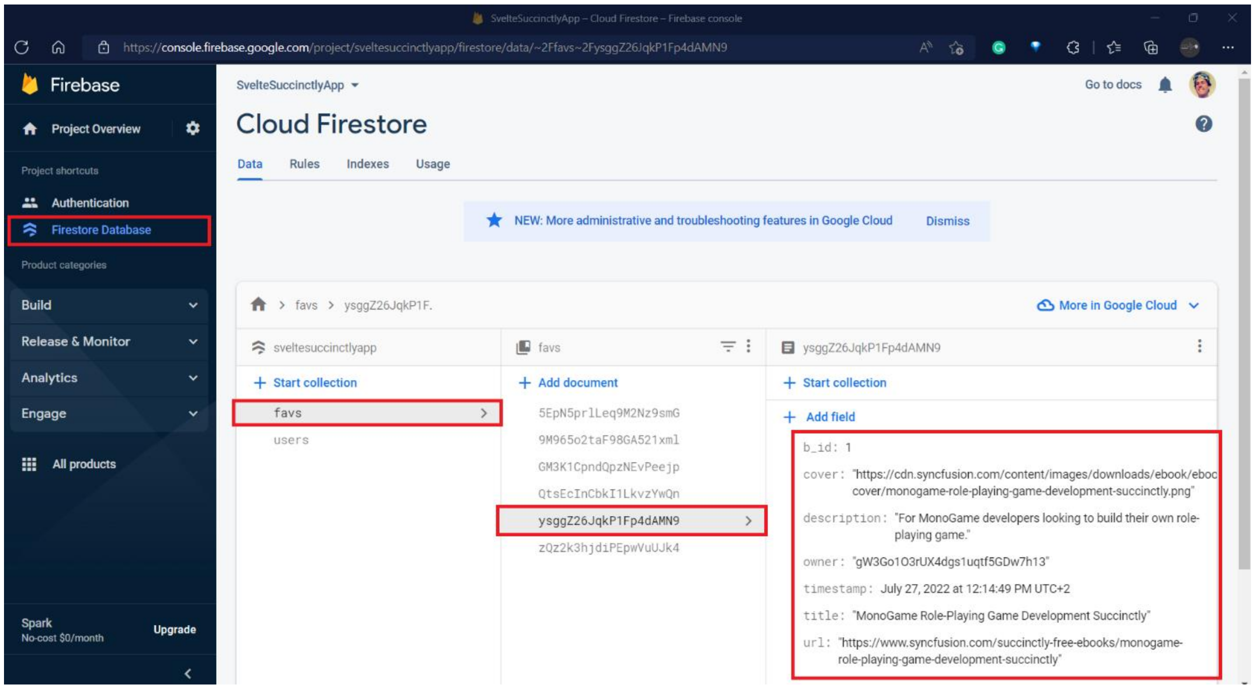

FavBooks
localhost:3000/favorites
FavBooks
fastapps@outlook.com's favorites
Sign out
Books
For MonoGame developers looking to build their own role-playing game.
Remove

Figure 7-g: Testing the App (Favorites Page with a Book Added—User Signed In)

Great, you have added a book to the list of favorites for this new user!

If you navigate to your [Firebase console](), click your project, then navigate to the **Firestore Database** section, you should see your new book within the **favs** collection, which in my case, looks as follows.



```
b_id: 1
cover: "https://cdn.syncfusion.com/content/images/downloads/ebook/eboc
       cover/monogame-role-playing-game-development-succinctly.png"
description: "For MonoGame developers looking to build their own role-
             playing game."
owner: "gW3Go1O3rUX4dgs1uqtf5GDw7h13"
timestamp: July 27, 2022 at 12:14:49 PM UTC+2
title: "MonoGame Role-Playing Game Development Succinctly"
url: "https://www.syncfusion.com/succinctly-free-ebooks/monogame-
     role-playing-game-development-succinctly"
```

Figure 7-h: Testing the App (Checking the Newly Added Favorite Book—Firebase Console)

You can continue testing the application further and playing with it by creating other user accounts and adding or removing more books from the favorites list for those other users.

Next steps and final thoughts

Well done! It’s exciting that you’re here and have followed along. Hopefully you’ve managed to get everything up and running.

Throughout this book, my goal was to give you a condensed overview of how to build something with Svelte using SvelteKit. I believe we managed to achieve that with the app that we created.

Although we couldn’t cover everything that Svelte and SvelteKit have to offer, we did manage to get acquainted with the essential aspects of Svelte in a short period, and in a succinct way.

Deployment is one aspect we couldn’t cover, given that it depends on the back-end platform of choice. However, we did enable Firebase Hosting as a deployment option, something you might want to explore in your spare time. Nevertheless, many other deployment options are available in the market to consider and explore.

Before closing, I’d like to leave you with some challenges and thoughts on possible next steps in your Svelte learning journey. Something you might want to explore is how to refactor the existing code even further.

For instance, there are quite a few similarities between login.svelte and register.svelte, so it’s perfectly possible to create a common component with shared functionality that both pages can use.

You could also think of making the list of books available non-static and creating an admin panel (accessible only for authenticated users) where you could add and remove those books directly from Firebase.

You could add other authentication features for Google or Microsoft accounts to the application's user authentication mechanism.

Those examples come to mind, but there could be many things to do with the application. If you pursue any of them, implement something extraordinary, and improve on what I’ve done, I’d love to hear from you.

Hopefully, this is just the start of your journey in learning Svelte. Thank you for reading. Until next time, all the best.