Olga Filipova

Vue.js 2 and Bootstrap 4 Web Development

Build Responsive SPAs with Bootstrap 4, Vue.js 2, and Firebase

Packt

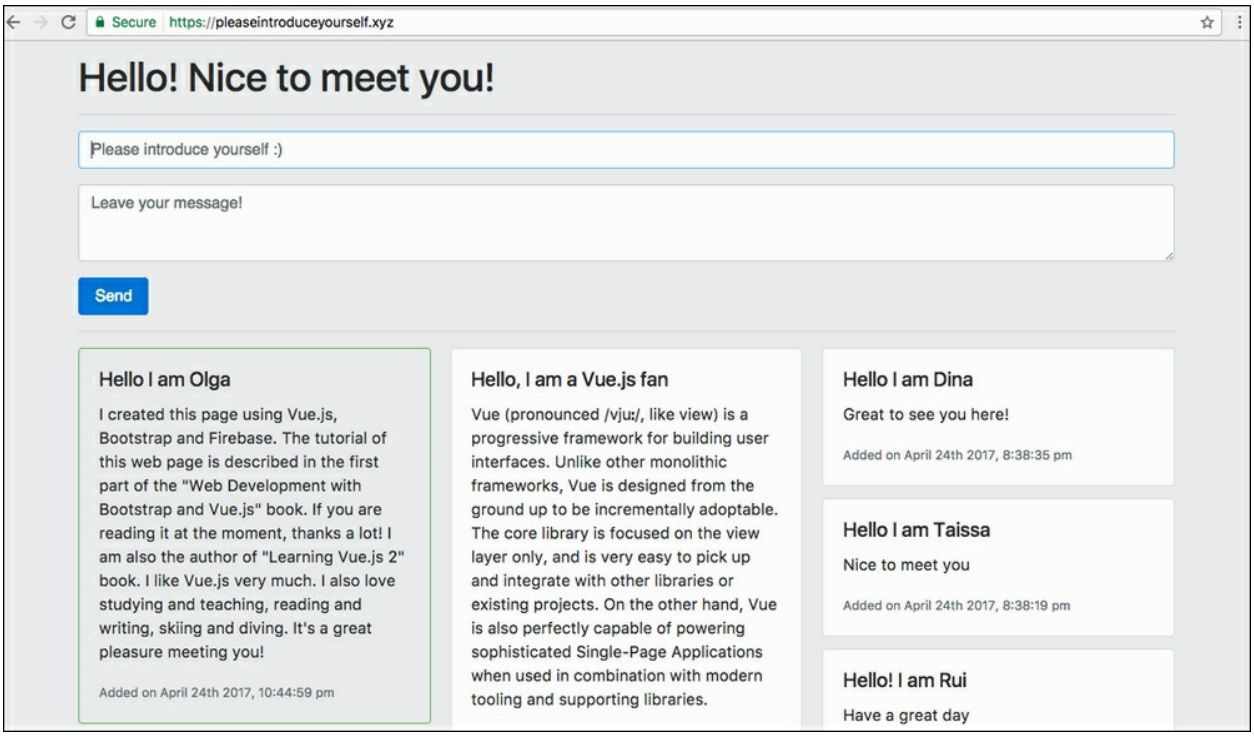
Contents

Chapter 1. Please Introduce Yourself – Tutorial

Hello, user

Hello dear reader, my name is Olga. Would you like to introduce yourself as well? Open https://pleaseintroduceyourself.xyz/ and leave a message for me and the other readers.

The page itself doesn't look like anything special. It's just a web page that allows users to write a message, and then, this message is immediately displayed along with the other users' messages in a reverse chronological order:



The please introduce yourself page

Do you want to know how long it took me to create this page? It took me around half an hour, and I am not only talking about writing the HTML markup or reversing the order of the messages but also about the database setup, deployment, and hosting.

You probably noticed that the very first message never changes, and it's actually my message where I wrote that I love to learn and teach. This is indeed true. That's why I will devote this chapter to teaching you how to create the exact same page in just 15 minutes. Are you ready? Let's go!

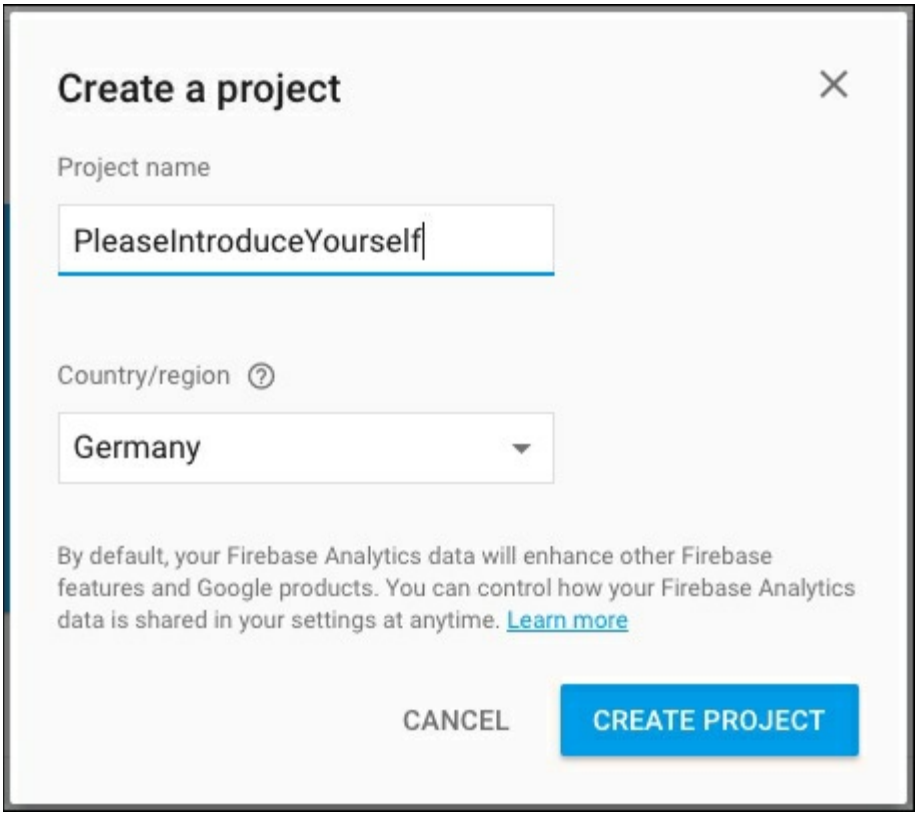
Creating a project in the Firebase console

If you still don't have a Google account but you really want to continue with this tutorial, then well, I am really sorry, but you will have to create one this time. Firebase is a service powered by Google, so a Google account is more than required.

If you already have your account, log in to the Firebase console:

https://console.firebase.google.com/.

Let's start by creating your new Firebase project. Click on the **Add project** button. Give it a meaningful name and select your country from the list. Once you are done, click on **CREATE PROJECT**:

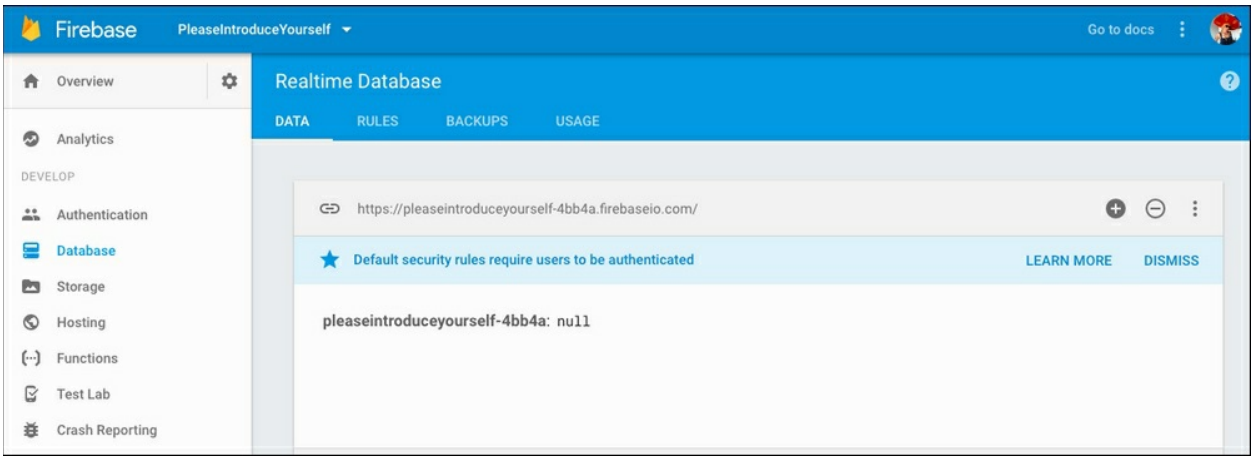


Create a project using the Firebase console

You're done! Now, you can use the Firebase-powered backend for your application, including a real-time database, authentication mechanism, hosting, and analytics.

Adding a first entry to the Firebase application database

Let's add the first database entry. Click on the **Database** tab on the left-hand side. You should see a dashboard similar to this one:



Real-time database on the Firebase project dashboard

Let's add an entry called `messages` and the very first message as a key-value object containing `title`, `text`, and

Scaffolding a Vue.js application

In this section, we will create a *Vue.js* application and connect it to the Firebase project that we created in the previous step. Make sure you have *Node.js* installed on your system.

You must also install Vue.js. Check out the instructions page from the official Vue documentation at https://vuejs.org/v2/guide/installation.html. Alternatively, simply run the `npm install` command:

```
$ npm install -g vue-cli
```

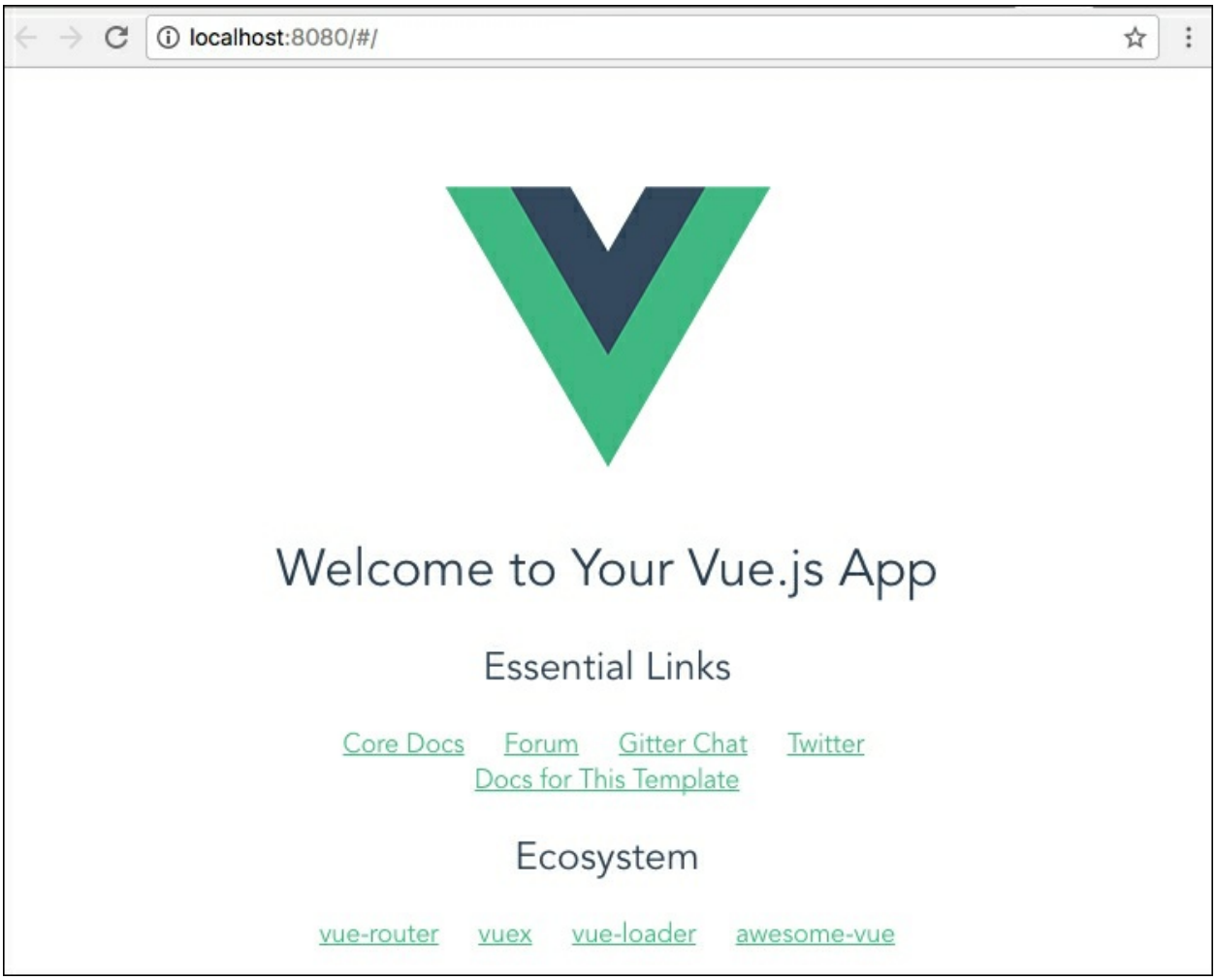
Now, everything is ready to start scaffolding our application. Go to the folder where you want your application to reside and type the following line of code:

```
vue init webpack please-introduce-yourself
```

It will ask you several questions. Just choose the default answer and hit *Enter* for each of them. After the initialization, you are ready to install and run your application:

```
cd please-introduce-yourself
npm install
npm run dev
```

If everything is fine, the following page will automatically open in your default browser:



Default Vue.js application after installing and running

If not, check the Vue.js official installation page again.

Connecting the Vue.js application to the Firebase project

To be able to

Adding a Bootstrap-powered markup

Let's add basic styling to our application by adding Bootstrap and using its classes.

First of all, let's include Bootstrap's `css` and `JS` files from Bootstrap's `CDN`. We will use the upcoming version 4, which is still in alpha. Open the `index.html` file and add the necessary `link` and `script` tags inside the `<head>` section:

```
//index.html
<link
rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.6/css/bootstrap.min.css"crossorigin="anonymous">
<script src="https://code.jquery.com/jquery-3.2.1.min.js"crossorigin="anonymous"></script>
<script src="https://npmcdn.com/tether@1.2.4/dist/js/tether.min.js">
</script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.6/js/bootstrap.min.js"crossorigin="anonymous">
</script>
```

You've probably noticed that I added *jQuery* and *Tether* dependencies as well; this is because Bootstrap depends on them.

Now, we can use Bootstrap classes and components in our application. Let's start by adding a bit of styling using Bootstrap's classes.

I will wrap the whole app `div` tag into the `jumbotron` class, and then, I will wrap the content of it...

Making things functional with Vue.js

So, what do we want to achieve with our form? We want the new message to be created. This message has to be composed of title, text, and the timestamp. We also want to add this message to our messages reference array.

Let's call this new message `newMessage` and add it to the `data` attributes of `App.vue`:

```
//App.vue
<script>
  <...>
  export default {
    data () {
      return {
        newMessage: {
          title: '',
          text: '',
          timestamp: null
        }
      }
    },
  <...>
  }
</script>
```

Now, let's bind the title and the text of this `newMessage` object to `input` and `textarea` of our form. Let's also bind a method called `addMessage` to the submit handler of our form so that the whole form's markup looks like this:

```
<template>
<...>
  <form @submit="addMessage">
    <div class="form-group">
      <input class="form-control"v-model="newMessage.title"maxlength="40"autofocus  placeholder="Please
introduce yourself :)" />
    </div>
    <div class="form-group">
      <textarea class="form-control"v-model="newMessage.text" placeholder="Leave your message!"
rows="3"></textarea>
   ...
```

Deploying your application

Well, now that we have a fully working application in our hands, it's time to make it public. In order to do this, we will deploy it to Firebase.

Start by installing Firebase tools:

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

Now, you have to tell your Firebase tools that you are actually a Firebase user who has an account. For this, you have to log in using Firebase tools. Run the following command:

```
firebase login
```

Follow the instructions to log in.

Now, you must initialize Firebase in your application. From the application root, call the following:

```
firebaseinit
```

You will be asked some questions. Select the third option for the first question:

Select the Hosting option for the first question

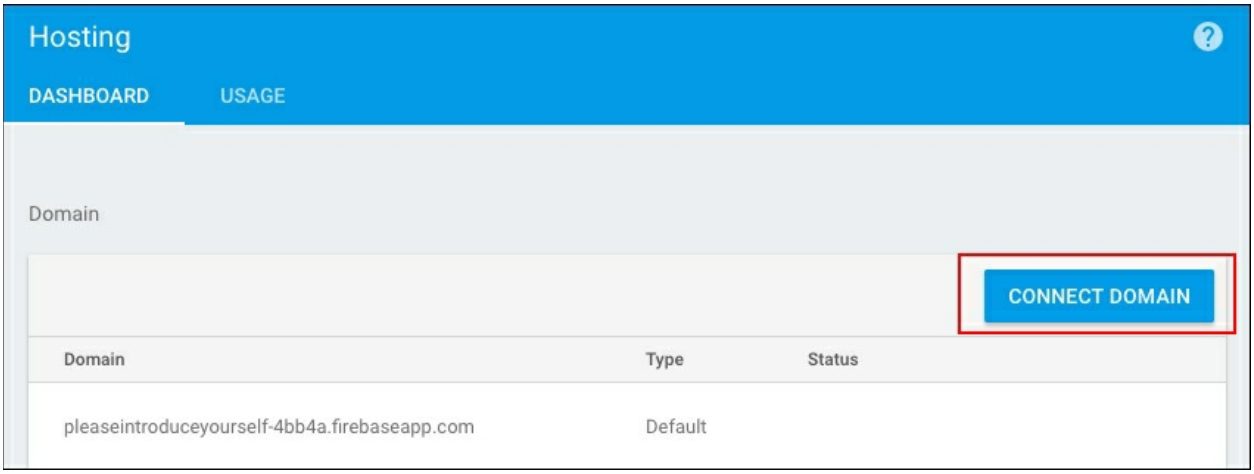
Select the `PleaseIntroduceYourself` project from the list of projects to associate to the application.

Initialization is over. Check whether the file called `firebase.json` has been created in the project's folder. This file can contain an innumerous number of configurations. Check out the official Firebase documentation in this regard at https://firebase.google.com/docs/hosting/full-config. For us, the very basic indication of the public...

Extra mile – connecting your Firebase project to a custom domain

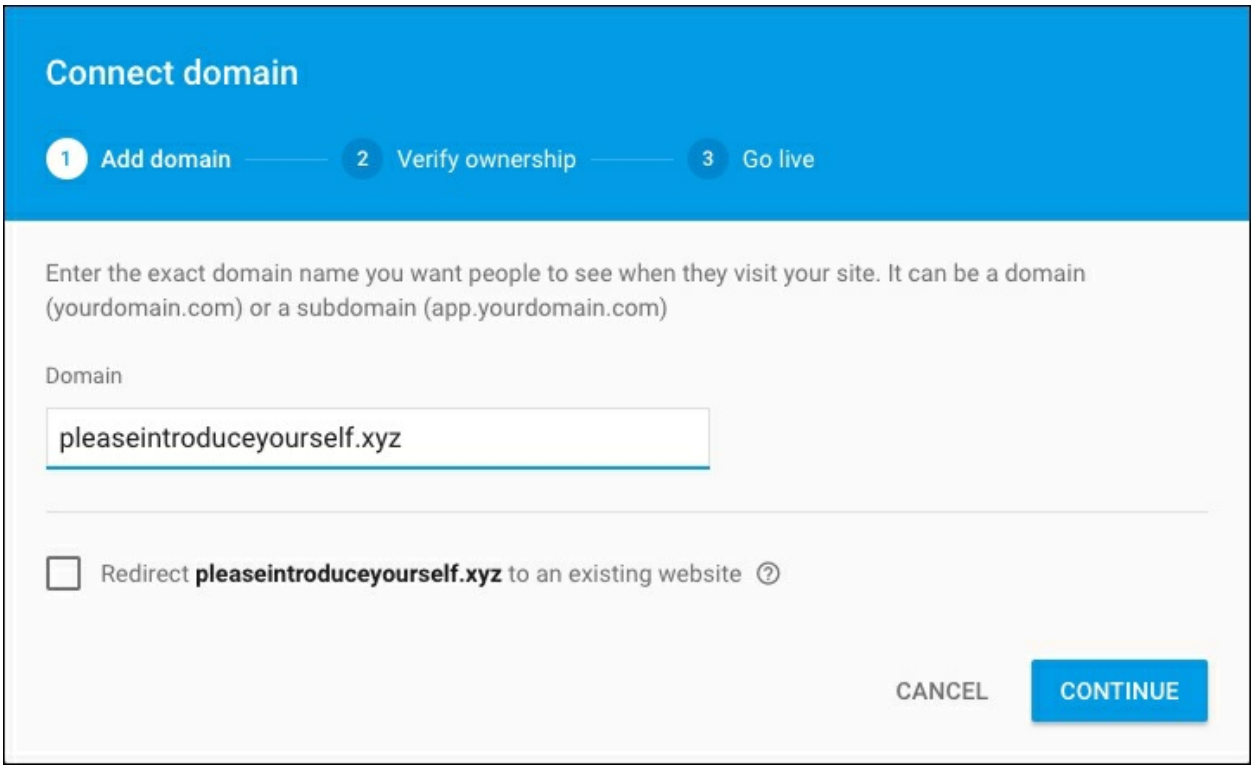
---

It's fairly easy to connect the Firebase project to a custom domain. First of all, of course, you need to buy this domain. For this application, I bought the *pleaseintroduceyourself* domain with the cheapest top-level domain, `.xyz`. It cost me a bit more than a dollar per year on GoDaddy (https://godaddy.com). After you have your domain, it's really easy. Go to the Firebase web console of the project. Click on the **Hosting** tab on the left-hand side. Then, click on the **CONNECT DOMAIN** button:



Click on the CONNECT DOMAIN button

In the popup, input your domain name:



Input your domain name

It will suggest that you add a TXT DNS record to your domain. Just open your DNS provider page, select your domain, find out how to add DNS records, and add the record with the TXT type. In my case, with GoDaddy, the record adding section looks like this:

Adding the DNS TXT record to our domain

After the handshake is established (mind, it might take some time), Firebase will propose you the final step—adding the **A** record to your domain. Follow the exact same procedure as in the previous...

Summary

In this chapter, we followed a tutorial where we have developed a single-page application from scratch. We used the Vue.js framework to structure our application, the Bootstrap framework to apply style to it, and the Firebase platform to manage the application's persistence layer and hosting.

In spite of being able to achieve a considerable result (a fully functional deployed application), we did everything without a deep understanding of what is going on behind the scenes. The tutorial didn't explain what Vue.js, Bootstrap, or Firebase was. We just took it for granted.

In the next chapter, we will understand the underlying technologies in detail. We will do the following:

- Take a closer look at the Vue.js framework, starting from a basic understanding and then covering topics such as directives, data binding, components, routing, and so on
- Have a deeper look at the Bootstrap framework, and check what is possible to achieve using it and how to do it
- Get to know the Firebase platform better; we'll gain some basic understanding about it and go through more complex topics such as data storage or functions
- Check out different...

Chapter 2. Under the Hood – Tutorial Explained

In the previous chapter, we built a simple single-page application from scratch. We used Vue.js to implement the application's functionality, Bootstrap to make it beautiful, and Firebase to manage the backend part of the application.

In this chapter, we will get to know all these technologies in depth and see how and why they can work nicely together. We will mostly discuss Vue.js since this will be our number one framework to build our application. Then, we will touch on Bootstrap and Firebase to get a basic understanding of how powerful these technologies are. Having said that, in this chapter we will:

- Discuss the Vue.js framework, reactivity, and data binding. Not only will we cover Vue.js' basics, but we will also dig into topics such as directives, components, routing, and so on.
- Discuss the Bootstrap framework. We will see what is possible to achieve with it, discuss how it can be useful to lay out an application, and discuss how its components can enrich your application with useful self-contained functionality.
- Discuss the Firebase platform. We will see what it is, what...

Vue.js

The official Vue.js website suggests that Vue is a progressive JavaScript framework:

Screenshot from the official Vue.js website

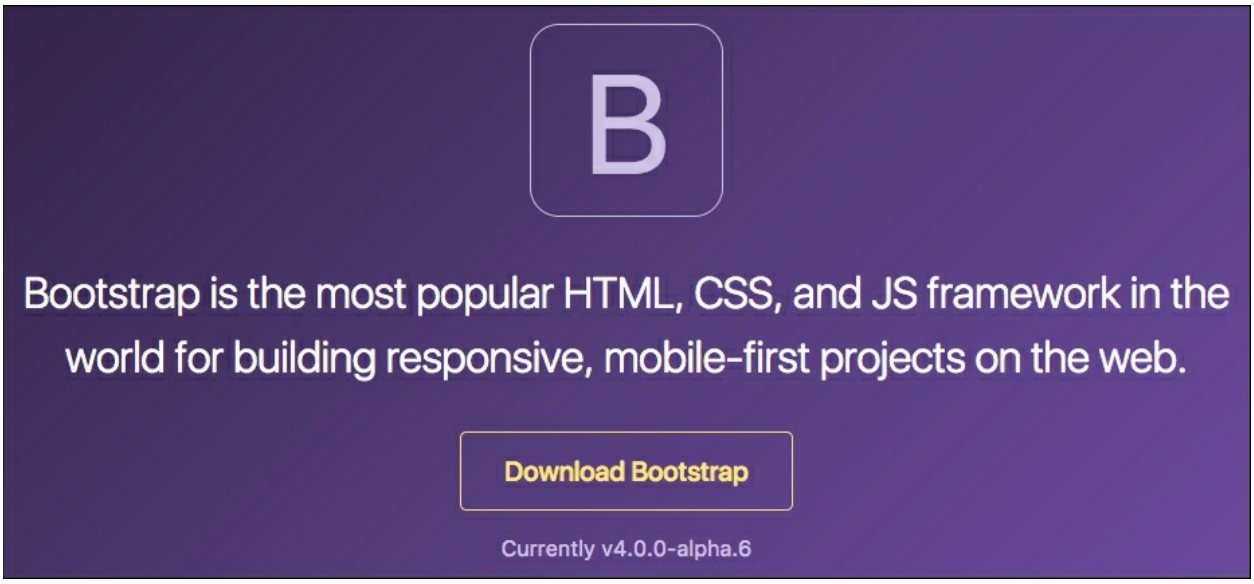
What does that mean? In a very simplified way, I can describe Vue.js as a JavaScript framework that brings reactivity to web applications.

It's undeniable that each and every application has some data and some interface. Somehow, the interface is responsible for displaying data. Data might or might not change during runtime. The interface usually has to react somehow to those changes. The interface might or might not have some interactive elements that might or might not be used by the application's users. Data usually has to react to those interactions, and consequently, other interface elements have to react to the changes that have been done to the data. All of this sounds complex. Part of this complex architecture can be implemented on the backend side, closer to where data resides; the other part of it might be implemented on the frontend side, closer to the interface.

Vue.js allows us to simply bind data to the interface and relax. All the reactions that must happen between data and the...

Bootstrap

Now that we know almost everything about Vue.js, let's talk about Bootstrap. Check out the official Bootstrap page at https://v4-alpha.getbootstrap.com/.



Bootstrap—framework for responsive projects

In a nutshell, Bootstrap gives you a broad set of classes that allow building nearly everything with any layout in an easy and effortless way.

Bootstrap provides with you four most important things:

- Easy layouts building at https://v4-alpha.getbootstrap.com/layout/overview/
- Broad range of classes to style nearly any web element at https://v4-alpha.getbootstrap.com/content/
- Self-contained components such as alerts, budges, modals, and so on at https://v4-alpha.getbootstrap.com/components/
- Some utilities for styling images, figures, for positioning, styling, and adding borders at https://v4-alpha.getbootstrap.com/utilities/

How to install Bootstrap? It can be installed from the CDN:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.6/css/bootstrap.min.css" integrity="sha384-
rwoIResjU2yc3z8GV/NPeZWAv56rSmLldC3R/AZzGRnGxQQKnKkoFVhFQhNUwEyJ" crossorigin="anonymous">
<script...
```

Combining Vue.js and Bootstrap

When we were talking about Vue, we devoted a big section to its components. When we talked about Bootstrap, we also talked about components. Doesn't it ring the same bell? Maybe we could create Vue components out of Bootstrap components? Maybe we can! Actually, we have already done it! Open the code of the first chapter's `PleaseIntroduceYourself` application. Check what we have inside the `components` folder. There's something that we called `MessageCard.vue`. Actually, this is an implemented Vue component for Card Bootstrap's component (https://v4-alpha.getbootstrap.com/components/card/)!

Open the `example13-vue-bootstrap-components-started/components` folder. Let's use this project as a playground to create the Vue component based on the Bootstrap alert component. Run `npm install` and `run`:

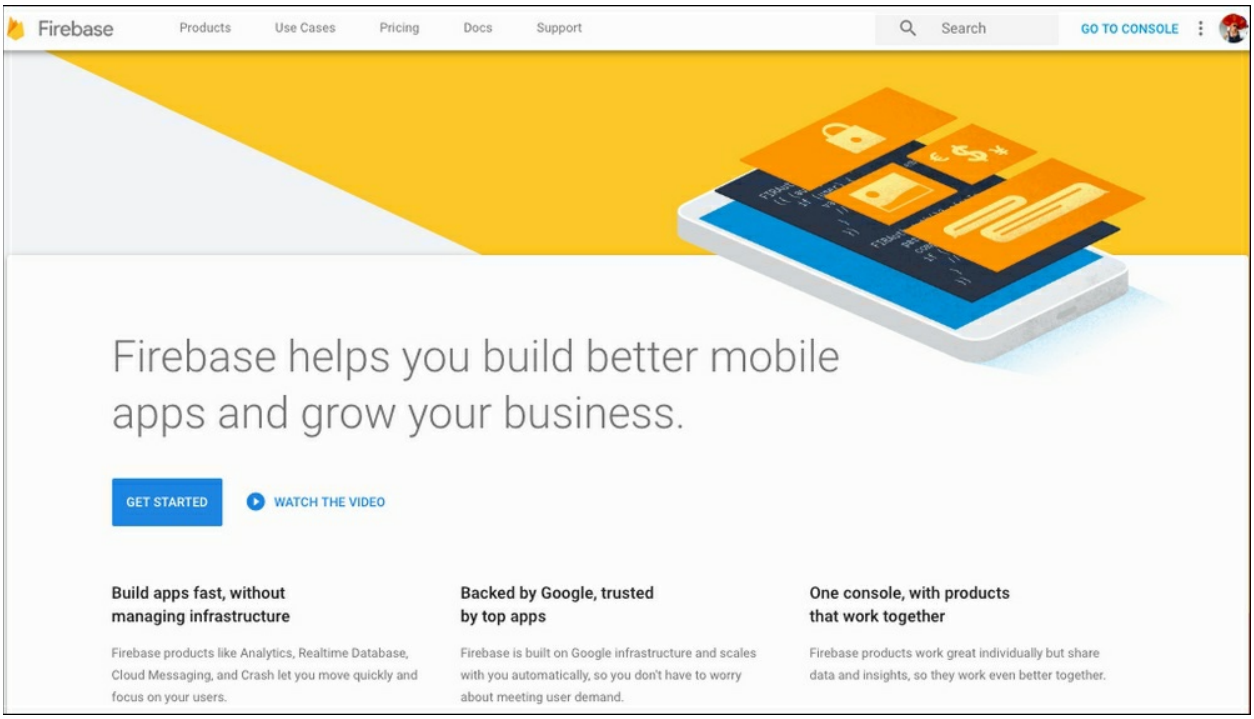
```
cd example13-vue-bootstrap-components-started/components
npm install
npm run dev
```

Let's create a Vue component called `Alert`. This component will contain the necessary code to simulate Bootstrap's alert component behavior.

Create a file named `Alert.vue` inside the `components` folder and add `template` tags. Our alert will...

What is Firebase?

To understand what is Firebase let's open its website https://firebase.google.com/. This is what we see:



Google Firebase landing page

Firebase for Google is yet another cloud service, like AWS for Amazon or Azure for Microsoft, a bit simpler though, because Google already has Google Cloud Platform, which is huge.

If you feel like you want to choose between Firebase and AWS, do not forget that you will most likely Google it. In any case, someone has already done this for you so here you have this question on Quora at https://www.quora.com/Which-is-better-cloud-server-Amazon-AWS-or-Firebase.

I would say that it's more similar to Heroku—it allows you to easily deploy your applications and integrate them with analytics tools. If you have read the Learning Vue.js 2 book (https://www.packtpub.com/web-development/learning-vuejs-2), then you already know how much I love Heroku. I even have Heroku socks!

My beautiful Heroku socks

However, I find Google Firebase console also quite nice and simple to use. It also provides a backend as a service. This backend is shared for your web and mobile applications, which comes as a huge...

Summary

In this chapter, we familiarized ourselves with Vue.js, Bootstrap and Firebase. We have also analyzed tools that integrate Vue.js with Bootstrap and Vue.js with Firebase.

Thus, now, we are familiar with Vue.js applications that are built using single-file components, Bootstrap's grid system, components, and CSS helpers to make our lives easier and to make Google Firebase console with its possibilities.

Also, we know how to initialize Vue.js project, and use Vue directives, components, store and routing.

You also learned how to leverage Bootstrap's grid system to achieve the responsibility of our application's layout.

And last but not least, you learned how to use the Firebase API within the Vue application using `vuefire` bindings.

With the end of this chapter, the first introduction part of our journey also comes to an end.

In the next chapter, we will actually dive deep inside the implementation. As a scuba diving tank, we will take everything that you have learned so far!

So, we will start developing the application that we will build during the whole book until it's ready for deployment. We will:

- Define what the application...

Chapter 3. Let's Get Started

In the previous chapter, we discussed the three main technologies that we will use throughout this book to build our application. We explored a lot about Vue.js; we introduced some of the functionalities of Bootstrap, and we checked what we can achieve using the Google Firebase console. We know how to start an application from scratch using Vue.js. We know how to make it beautiful with the help of Bootstrap, and we know how to use Google Firebase to deploy it to live! What does that mean? It means that we are 100 percent ready to start developing our application!

Coding an application is a fun, challenging, and exciting process... only if we know what we are going to code, right? In order to know what we will code, we have to define the concept of the application, its requirements, and its target users. In this book, we will not go through the whole process of design building as for this, you have plenty of other books, because it's a big science.

In this book, particularly in this chapter, and before diving into the implementation, we will at least define a set of personas and user stories. Thus, in...

Stating the problem

There are many time-management techniques in the world. Several gurus and professionals have given a great amount of talks on how to effectively manage your time so that you are efficient and all your KPI values are above any possible benchmarks of productivity. Some of these talks are really amazing. When it comes to time-management talks, I always suggest Randy Pausch's talk at https://youtu.be/oTugjssqOT0.

Speaking of time-management techniques, there is one popular technique I particularly like, which I find very simple to use. It's called Pomodoro (https://en.wikipedia.org/wiki/Pomodoro_Technique). This technique consists of the following principles:

- You work during a certain period without any interruptions. This period can be 20 to 25 minutes and it's called Pomodoro
- After the working Pomodoro, you have a 5 minute break. During this break, you can do whatever you want—check e-mails, social networks, and so on
- After working four Pomodoros with short breaks, you have the right to a longer break that can last from 10 to 15 minutes

There are numerous implementations of the Pomodoro timer. Some of them allow...

Gathering requirements

Now that we know what we are going to build, let's define a list of requirements for the application. The application is all about displaying a timer and displaying workouts. So, let's define what it must be able to do. Here's my list of functional requirements:

- The application should display a countdown timer.
- The countdown timer can be from 25 to 0 minutes, from 5 to 0 minutes, or from 10 to 0 minutes.
- It shall be possible to start, pause, and stop the timer at any moment of the application's execution.
- The application shall produce some sounds when the time reaches 0 and the next period of break or the working Pomodoro starts.
- The application shall display a workout during the short and long breaks. It shall be possible to skip the current workout and switch to the next one. It shall also be possible to skip workouts completely during a break and just stare at kittens. It shall also be possible to mark the given workout as done.
- The application must offer an authentication mechanism. Authenticated users can configure the Pomodoro timer, add new workouts to the system, and visualize their statistics.
- Statistics...

Personas

Usually, before developing an application we have to define its target users. For this, multiple questionnaires are conducted with the potential users of the application. The questionnaires usually include questions about the user's personal data, such as age, sex and so on. There should also be questions about the user's usage patterns—operating system, desktop or mobile, and so on. And of course, there should be questions about the application itself. For example, for the ProFitOro application, we could ask the following questions:

- How many hours per day do you spend in the office?
- For how long do you sit in the office during your working day?
- How often do you do sport activities such as jogging, fitness workouts, and so on?
- Do you work from the office or from home?
- Is there any area in your work place where you could do push-ups?
- Do you have problems with your back?

After all questionnaires are collected, the users are divided into categories by similar patterns and personal data. After that, each user's category forms one single persona. I will leave here four personas for the ProFitOro application.

Let's start with a...

User stories

After we've defined our users, let's write some user stories. When it comes to writing user stories, I just close my eyes and imagine that I am this person. Let's try out this mind exercise starting with Dwart Azevedo:

Dwart Azevedo

Dwart's working day consists of meetings, calls, video conferences, and paperwork. Today, he was really busy with interviews and meetings. Finally, he got a few hours for his paperwork that has been waiting for him for the whole week. Dwart wants to spend these hours in the most productive way. He opens the ProFitOro application, clicks on start, and starts working. After his paperwork is done, he clicks on stop, checks his statistics in ProFitOro, and feels happy. Even though his working time consisted of two hours only, he was able to finish everything he planned to finish.

Thus, we can come up with a formal user story like this:

As an authenticated user, I would like to check out my statistics page at ProFitOro in order to see the completeness of my working day.

Let's move on to our fitness instructor, Steve Wilson.

Steve Wilson

Steve is a fitness instructor. He knows everything about...

Retrieving nouns and verbs

Retrieving nouns and verbs from the user stories is a very fun task that helps you realize what parts your application consists of. For those who like **Unified Modeling Language** (**UML**), after you retrieve the nouns and verbs from your user stories, you'll have the classes and entity-relationship diagrams almost done! Do not underestimate the number of nouns and verbs to retrieve. Write them all down—literally! You can remove the words that don't make sense after. So, let's do it.

Nouns

The nouns that I was able to retrieve out of my user stories are the following:

- Working day
- Meeting
- Call
- Interview
- Hour
- Day
- Week
- Application
- Statistics
- Working time
- Plan
- Fitness
- Instructor
- Human body
- Nutrition
- Workout

- Section
- Exercise
- E-mail
- Data
- Page
- Registration

Verbs

The verbs that I was able to retrieve from the user stories are the following:

- Consist
- Be busy
- Open
- Spend time
- Start
- Pause
- Stop
- Check
- Finish
- Plan
- Add
- Create
- Register
- Authenticate
- Login
- Concentrate

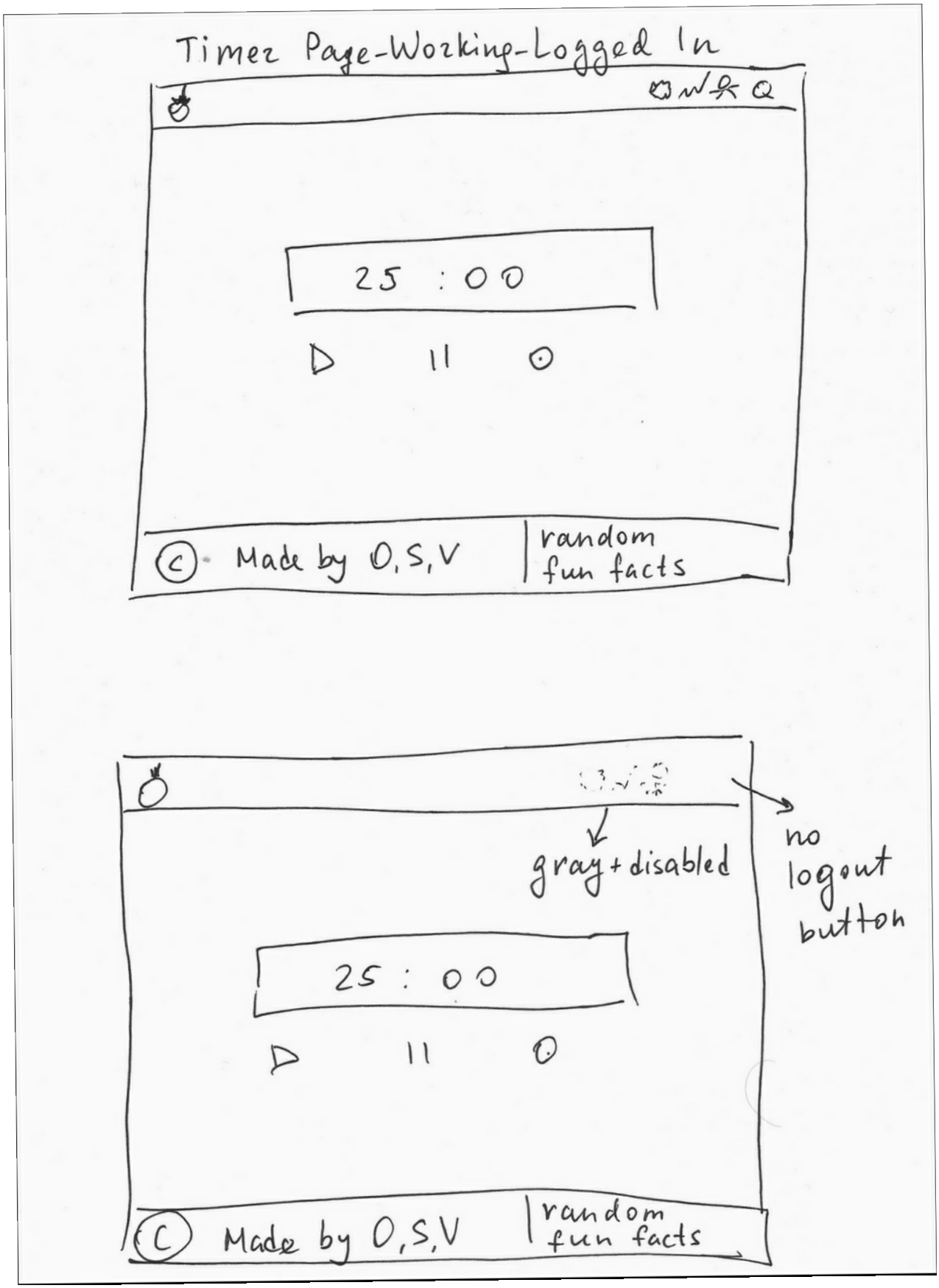
The fact that we have verbs such as **register**, **login,** and **authenticate** and nouns such as **e-mail,** and **registration** mean that the application will probably be used with and without registration. This means...

Mockups

Now that we have all our nouns and verbs, we can start making connections between all the sections of our application. We can actually start preparing some mockups. Sit down with someone, discuss, explain your idea, and collect feedback. Ask questions. Answer questions. Use a whiteboard, use post-its. Use paper: draw, discard, and redraw again.

I have a good friend called Safura. She is a working student currently studying computer science in Berlin, and we work together in the same team. She is interested in the UI/UX topic. Actually, she will write her master's thesis in the **Human-Computer Interaction** (**HCI**) area. So, we sat together, and I explained the idea of ProFitOro to her. You cannot imagine the number of questions she asked. Then, we started to draw. And to redraw. "And what if….?" redraw again.

This is how the first mockups on paper looked:



The first mockups on paper for the ProFitOro application

After all the brainstorming and drawing and redrawing, Safura prepared some nice mockups for me. She used *WireframeSketcher* for this purpose (http://wireframesketcher.com/).

The first page – login and register

The very...

Summary

In this chapter, we applied the very basic principles of designing an application's user interface. We brainstormed, defined our personas and wrote user stories, retrieved nouns and verbs from these stories, and ended up with some nice mockups for our application.

In the next chapter, we will start implementing our ProFitOro. We will use Vue.js to scaffold the application and split it into important components. Thus, in the next chapter we will do the following:

- Scaffold the ProFitOro application using vue-cli with the `webpack` template
- Split the application into the components and create all the necessary components for the application
- Implement a basic Pomodoro timer using Vue.js and Bootstrap

Chapter 4. Let It Pomodoro!

The previous chapter ended with a nice set of mockups for the *ProFitOro* application. We have previously defined what the application should do; we have also determined an average user profile, and we are ready to implement it. In this chapter, we will finally start coding. So, in this chapter, we will do the following:

- Scaffold *ProFitOro* using vue-cli with the `webpack` template
- Define all the needed application's components
- Create placeholders for all the components
- Implement a component that will be responsible for rendering the Pomodoro timer using Vue.js and Bootstrap
- Revisit the basics of trigonometric functions (you were not expecting that, right?)

Scaffolding the application

Before everything, let's make sure that we are on the same page, at least regarding the node version. The version of Node.js I'm using is *6.11.1*.

Let's start by creating a skeleton for our application. We will use vue-cli with the `webpack` template. If you don't remember what **vue-cli** is about and where it comes from, check the official Vue documentation in this regard at https://github.com/vuejs/vue-cli. If for some reason you still haven't installed it, proceed with its installation:

```
npm install -g vue-cli
```

Now, let's bootstrap our application. I'm sure you remember that, in order to initialize the application with `vue-cli`, you must run the `vue init` command followed by the name of the template to be used and the name of the project itself. We are going to use the `webpack` template, and our application's name is `profitoro`. So, let's initialize it:

```
vue init webpack profitoro
```

During the initialization process you will be asked some questions. Just keep hitting *Enter* to answer the default `Yes` to all of them; `Yes` because for this application we will need everything: linters, vue-router, unit testing,...

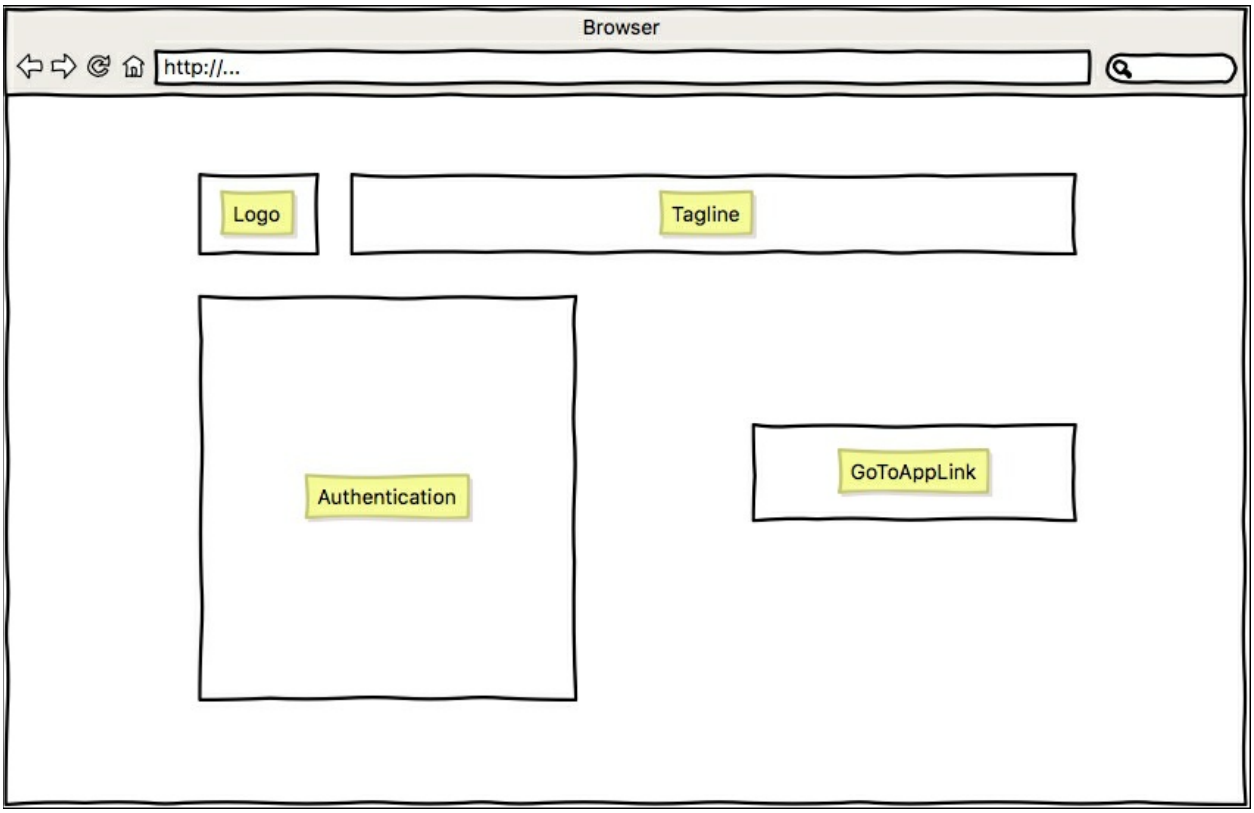
Defining ProFitOro components

Our application consists of two main screens.

One of the screens is the so-called *Landing page*; this page consists of the following parts:

- A logo
- A tagline
- An authentication section
- A link to the application to be used without being registered

Schematically, this is how our components are positioned on the screen:



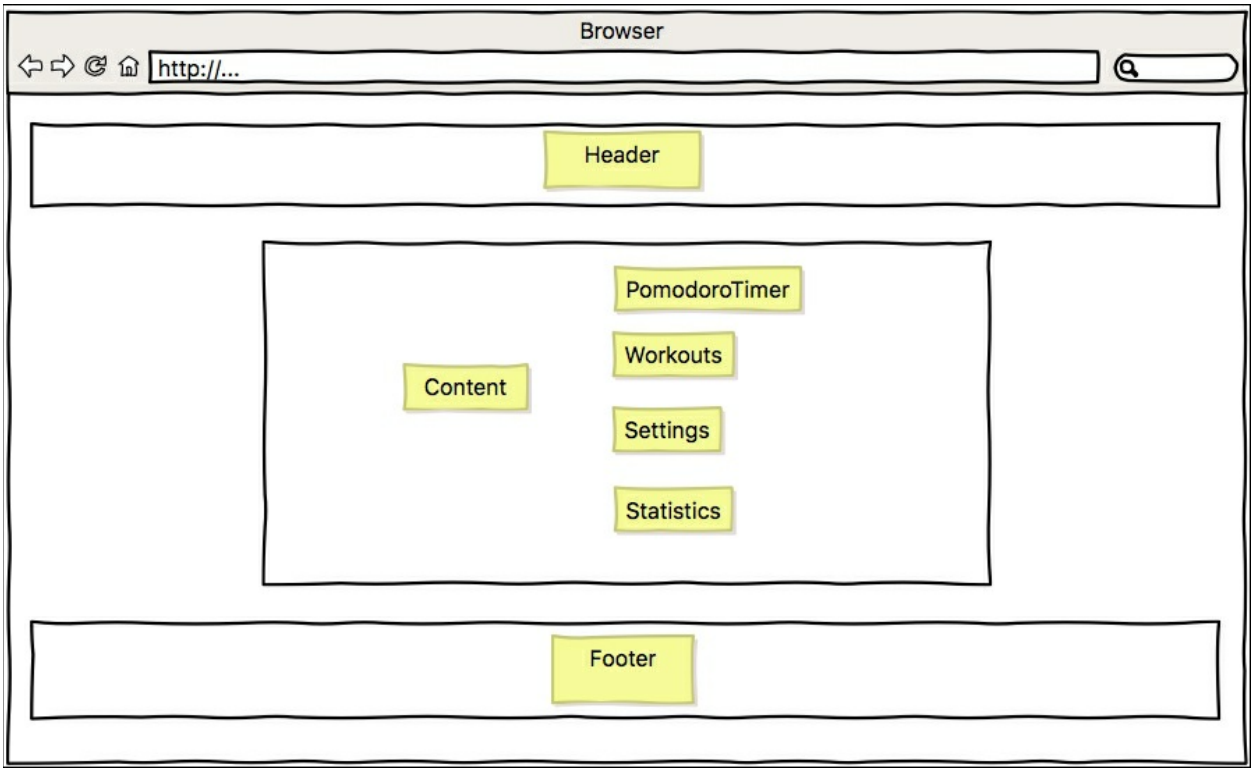
Landing page that contains logo, tagline, authentication section, and a link to the application

The second screen is the main application screen. This screen contains three parts:

- A header

- A footer
- The content

The content part contains the Pomodoro timer. If the user is authenticated, it will contain settings, workouts, and statistics as well:



Main application's screen that contains header, footer, and content

Let's create a folder called `components` and subfolders called `main`, `landing`, and `common` for the corresponding sub-components.

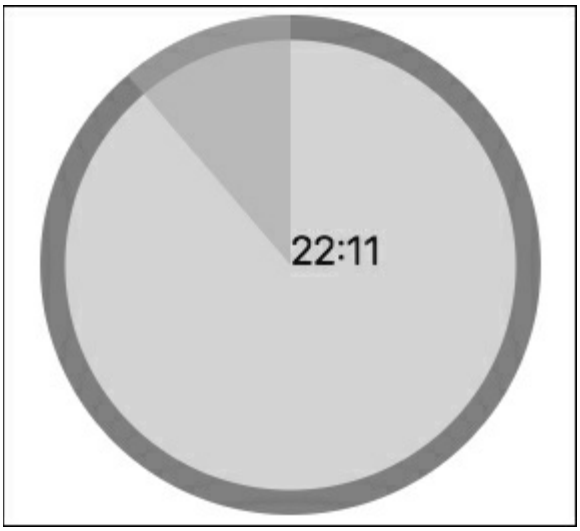
Components for the landing and main pages will reside in the `components` folder; the remaining 11 components will be distributed between the respective subfolders.

For each defined component file, add the `template`, `script`, and `style` sections. Add the `lang="sass"` attribute to...

Implementing the Pomodoro timer

One of the most important components of our application is, without any doubt, the Pomodoro timer. It performs the main functionality of the application. So, it might be a good idea to implement it in the first place.

I am thinking of some kind of a circular timer. Something like this:



Circular timer to be implemented as a Pomodoro timer

As time passes, the highlighted sector will move counterclockwise and the time will count down as well. To implement this kind of structure, I am thinking of three components:

- *SvgCircleSector*: This component will just receive an angle as a property and color the corresponding sector of the SVG circle.
- *CountDownTimer*: This component will receive the number of seconds to countdown, implement the timer and calculate the angle to pass to the `SvgCircularComponent` on each timer update.
- *PomodoroTimer*: We have already bootstrapped this component. This component will be responsible to call the `CountDownTimer` component with the initial time and update it to the corresponding number of seconds depending on the current working Pomodoro or break interval.

SVG and trigonometry

Let's...

Introducing workouts

I have been so enthusiastic writing this chapter, calculating sine, cosine, drawing SVG, implementing a timer, and taking care of the inactive tabs and stuff that I almost forgot to do my workout! I like planks and pushups, what about you? By the way, haven't you also forgotten that workouts are a part of our application? During the breaks, we are supposed to do simple exercises and not just check our social networks!

We will implement full-fledged workouts and their management in the next chapters; for now, let's just leave a nice placeholder for the workout and hard code one exercise in this placeholder (I vote for pushups since the book is mine, but you can add the workout or exercise of your own preference). Open the `PomodoroTimer.vue` component and wrap up a countdown component into a `div` with a class `row`. We will make this row contain two columns, one of which will be the countdown timer, and the other is a conditionally rendered element containing a workout. Why conditionally? Because we only need this element displayed during the Pomodoro breaks. We will use the `v-show` directive so that the containing...

Summary

In this chapter, we have done a lot of things. We have implemented the main functionality of our Pomodoro timer, and now it is fully functional, configurable, usable, and responsive. We bootstrapped our ProFitOro application, separated it into components, created a skeleton for each of the defined components, and fully implemented one of them. We even revisited some trigonometry, because math is everywhere. We implemented our timer and we made it work, even on the hidden and inactive tabs. We made the application responsive and adaptive to different device sizes using the powerful Bootstrap layout classes. Our application is functional, but it is far from beautiful. Don't mind these shades of gray though; let's stick to them for now. In the end of the book, you will get your beautiful ProFitOro styles, I promise you!

We are ready to continue our journey in the world of technology. In the next chapter, we will learn how to configure our Pomodoro and how to store the configuration and usage statistics using Firebase. Thus, in the next chapter we will:

- Get back to Vuex centralized state management architecture and combine it...

Chapter 5. Configuring Your Pomodoro

In the previous chapter, we implemented the main feature of our ProFitOro application – the Pomodoro timer. We even added a hardcoded workout, so we can exercise during our breaks. Actually, I already started using ProFitOro. While I'm writing these words, the Pomodoro clock counts down – *tick tick tick tick*.

In this chapter, we are going to explore the *Firebase Realtime Database's* possibilities and its API. We are going to manage storing, retrieving, and updating usage statistics and configuration of our application. We will use the Vuex store to bring the application's data from the database to the frontend application.

To bring this possibility to the UI, we will use Vue's reactivity combined with the power of Bootstrap. Thus, in this chapter we are going to implement the statistics and settings ProFitOro components using:

- Firebase Realtime Database
- Vue.js reactive data bindings and Vuex state management
- The power of Bootstrap to make things responsive

Setting up a Vuex store

Before starting with real data from the database, let's set up the Vuex store for our ProFitOro. We will use it to manage the Pomodoro timer configuration, user settings, such as the username, and a profile picture URL. We will also use it to store and retrieve the application's usage statistics.

From Chapter 2, *Hello User Explained*, you already know how the Vuex store works. We must define data that will represent the application's state and then we must provide all the needed getters to get the data and all the needed mutations to update the data. Once all this is set, we will be able to access this data from the components.

After the application's store is ready and set up, we can connect it to the real-time database and slightly adjust the getters and mutations to operate the real data.

First of all, we need to tell our application that it will use the Vuex store. To do that, let's add the `npm` dependency for `vuex`:

```
npm install vuex --save
```

Now, we need to define a basic structure of our store. Our Vuex store will contain the following:

- **State**: The initial state of the application's data.
- **Getters**: Methods

Defining actions and mutations

It's great that our components can now get data from the store, but it would be probably even more interesting if our components were also able to change the data in the store. On the other hand, we all know that we cannot modify the store's state directly.

The state should not be touched by any of the components. However, you also remember from our chapter about the Vuex store that there are special functions that can mutate the store. They are even called `mutations`. These functions can do whatever they/you want with the Vuex store data. These mutations can be called using the `commit` method applied to the store. Under the hood, they essentially receive two parameters – the state and the value.

I will define three mutations – one for each of the timer's definitions. These mutations will update the corresponding attribute of the `config` object with a new value. Thus, my mutations look as follows:

```
//store/mutations.js
export default {
  setWorkingPomodoro (state, workingPomodoro) {
    state.config.workingPomodoro = workingPomodoro
  },
  setShortBreak (state, shortBreak) {
    state.config.shortBreak...
```

Setting up a Firebase project

I hope that you still remember how to set up Firebase projects from the first chapters of this book. Open your Firebase console at https://console.firebase.google.com, click on the **Add project** button, name it, and choose your country. The Firebase project is ready. Wasn't that easy? Let's now prepare our database. The following data will be stored in it:

- **Configuration**: The configuration of our Pomodoro timer values
- **Statistics**: Statistical data of the Pomodoro usage

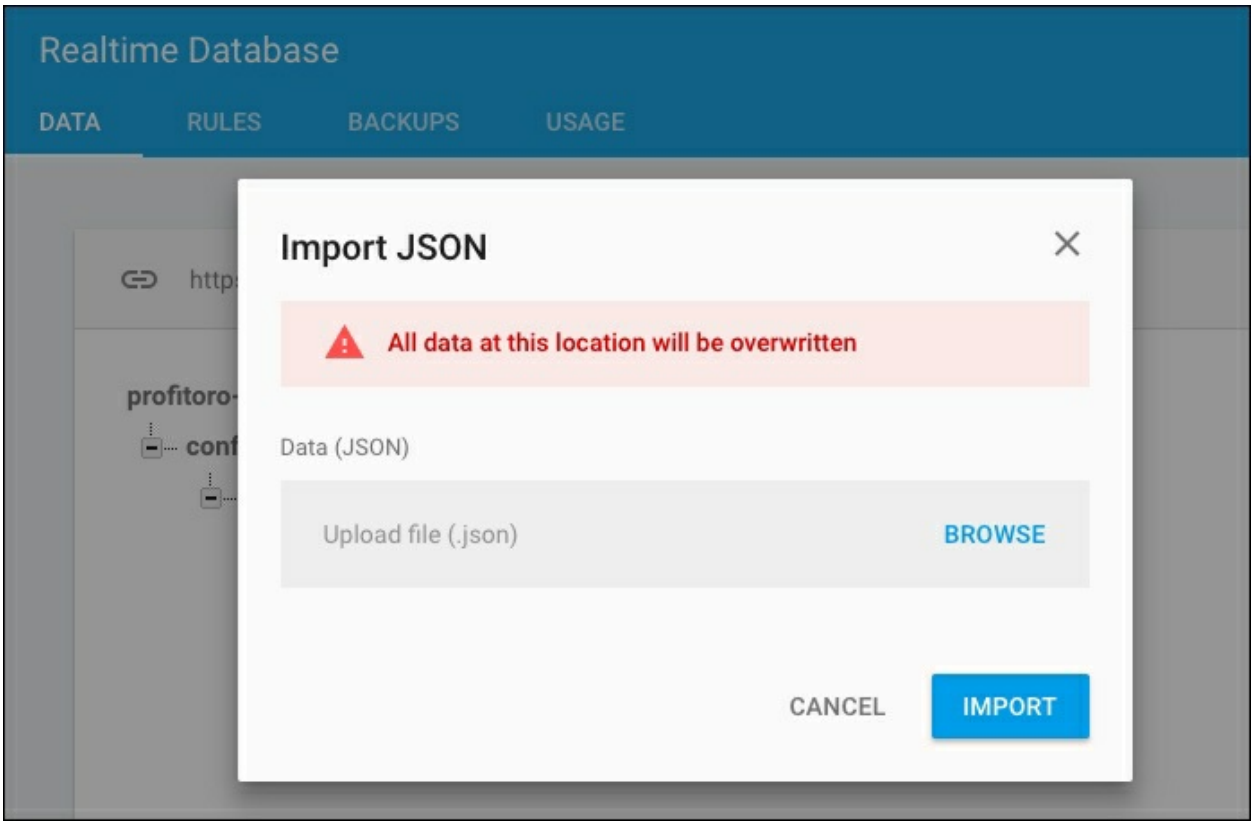
Each of these objects will be accessible via a special key that will correspond to a user's ID; this is because, in the next chapter, we are going to implement an authentication mechanism.

The configuration object will contain values – `workingPomodoro`, `longBreak` and `shortBreak` – that are already familiar to us.

Let's add a configuration object to our database with some fake data:

```
{
  "configuration": {
    "test": {
      "workingPomodoro": 25,
      "shortBreak": 5,
      "longBreak": 10
    }
  }
}
```

You can even create this as a simple JSON file and import it to your database:



Import JSON file to your real-time Firebase...

Connecting the Vuex store to the Firebase database

So, now we have to connect our Vuex store to the Firebase database. We could use the native Firebase API for binding the state data to the database data, but why would we deal with promises and stuff if someone already did that for us? This someone is called Eduardo and he has created Vuexfire – Firebase bindings for Vuex (https://github.com/posva/vuexfire). If you were at the *vueconf2017 conference* in *Wroclaw*, you probably remember this guy:

Eduardo talking about Vue and Firebase during the Vue conference

Vuexfire comes with Firebase mutations and actions that will do all the behind the scenes jobs for you, while you just export them within your mutations and actions objects. So, to start with, install both `firebase` and `vuexfire`:

```
npm install vue firebase vuexfire –save
```

Import `firebase` and `firebaseMutations` in your store's `index.js` entry point:

```
//store/index.js
import firebase from 'firebase'
import { firebaseMutations } from 'vuexfire'
```

Now, we need to obtain the reference to the Firebase application. Firebase comes with an initialization method, `initializeApp`, which receives an...

Exercise

You have learned how to connect the real-time Firebase database to your Vue application and used this knowledge to update the configurations for Pomodoro timers. Now, apply your knowledge to the statistics area. For the sake of simplicity, just display the total amount of Pomodoros executed since the user started using the application. For that you will need to do the following:

1. Add another object called `statistics` containing the `totalPomodoros` attribute that initially equals `0` in your Firebase database.
2. Create an entry in the store's `state` to hold the statistics data.
3. Map `totalPomodoros` of the statistics state's object to the Firebase reference using the `firebaseAction` enhancer and the `bindFirebaseRef` method.
4. Create an action that will update the `totalPomodoros` reference.
5. Call this action whenever it has to be called inside the `PomodoroTimer` component.
6. Display this value inside the `Statistics.vue` component.

Try to do it yourself. It shouldn't be difficult. Follow the same logic we applied in the `Settings.vue` component. If in doubt, check the `chapter5/4/profitoro` folder, particularly the store's files – `index.js`, `state.js`

Summary

In this chapter, you learned how to use the real-time Firebase database with the Vue application. You learned how to use Vuexfire and its methods to correctly bind our Vuex store state to the database reference. We were not only able to read and render the data from the database but we were also able to update it. So, in this chapter, we saw Vuex, Firebase, and Vuexfire in action. I guess we should be proud of ourselves.

However, let's not forget that we have used a hardcoded user ID in order to get the user's data. Also, we had to expose our database to the world by changing the security rules, which doesn't seem right either. It seems that it's time to enable the authentication mechanism!

And we will do it in the next chapter! In the next chapter, we are going to learn how to set up the authentication mechanism using the Firebase authentication framework. We will learn how to use it in our application using Vuefire (Firebase bindings for Vue: https://github.com/vuejs/vuefire). We will also implement the very initial view of our application responsible for providing a way of registering and performing the login. We will...

Chapter 6. Please Authenticate!

In the previous chapter, we connected our ProFitOro application to the real-time database. Whenever a user updates the Pomodoro timer settings, these are stored in the database and immediately propagated between the components that use them. Since we had no authentication mechanism, we had to use a fake user in order to be able to test our changes. In this chapter, we are going to have real users!

We will use the Firebase authentication API in this regard. So in this chapter, we are going to do the following:

- Discuss the meaning of AAA and the difference between authentication and authorization
- Explore the Firebase authentication API
- Create a page for sign-in and login, and connect it with the Firebase authentication API
- Connect the user's settings with the user's authentication

AAA explained

Triple-A, or **AAA**, stands for **Authentication, Authorization, and Accounting**. Initially, this term was invented as a term to describe the security network protocol; however, it can be easily applied to any system, web resource, or site.

So, what does AAA mean and why should we bother?

Authentication is the process of uniquely identifying the users of a system. An authenticated user is a user whose access to a system is granted. Usually, the authentication is done via some username and password. When you have to provide your username and password to open your Facebook page, you are authenticating yourself.

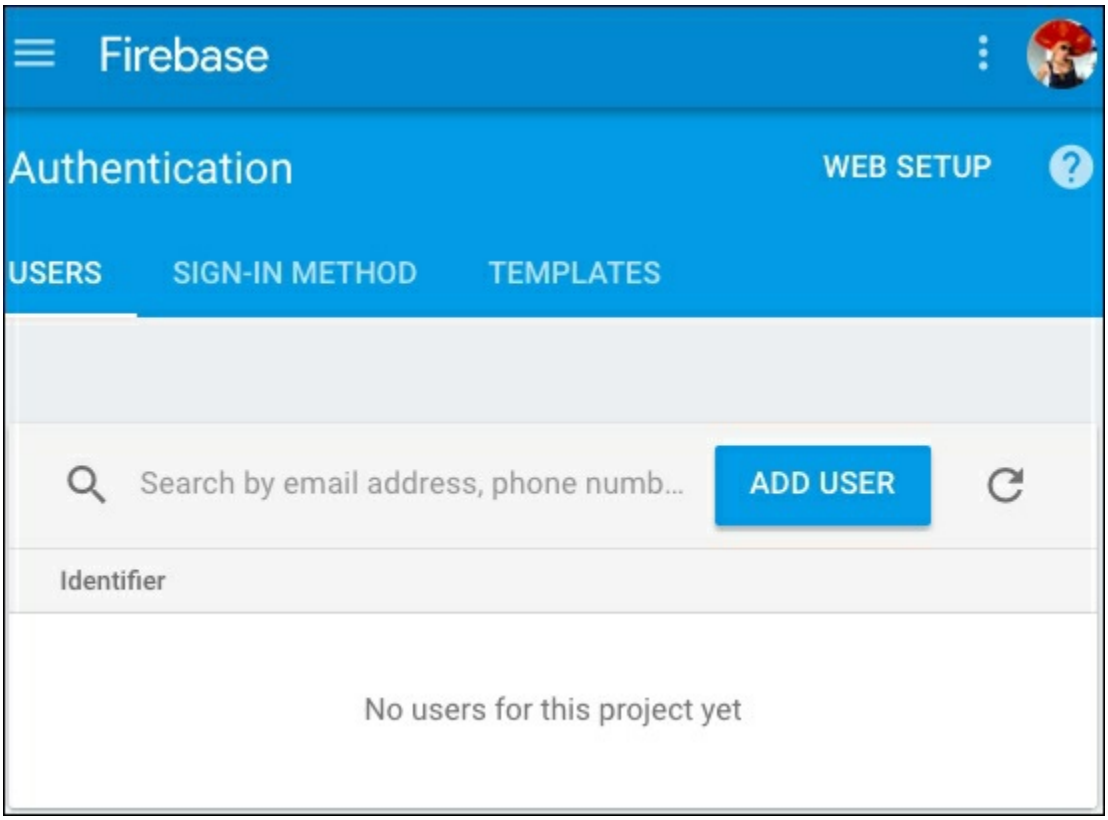
Your passport is a way of authenticating yourself at the airport. The passport control agent will look at your face and then check your passport. So anything that allows you to *pass* is a part of your authentication. It can be a special word (password) that is only known by you and the system or it can be something that you port (passport) with you that can help the system to uniquely identify you.

Authorization is a way to control what resources each user has rights (permissions) to access. If you are developing...

How does authentication work with Firebase?

In the previous chapter, you learned how to use the Firebase API to create a Firebase application instance and use it through your application. We were able to access the database, read it, and store data in it.

The way you work with the Firebase authentication API is very similar. You create a Firebase instance, providing a `config` object to it, and you use the `firebase.auth()` method to access different methods related with the authentication. Check your Firebase console's **Authentication** tab:



There are no users yet but we will fix it in a minute!

The Firebase SDK provides several ways for users to authenticate:

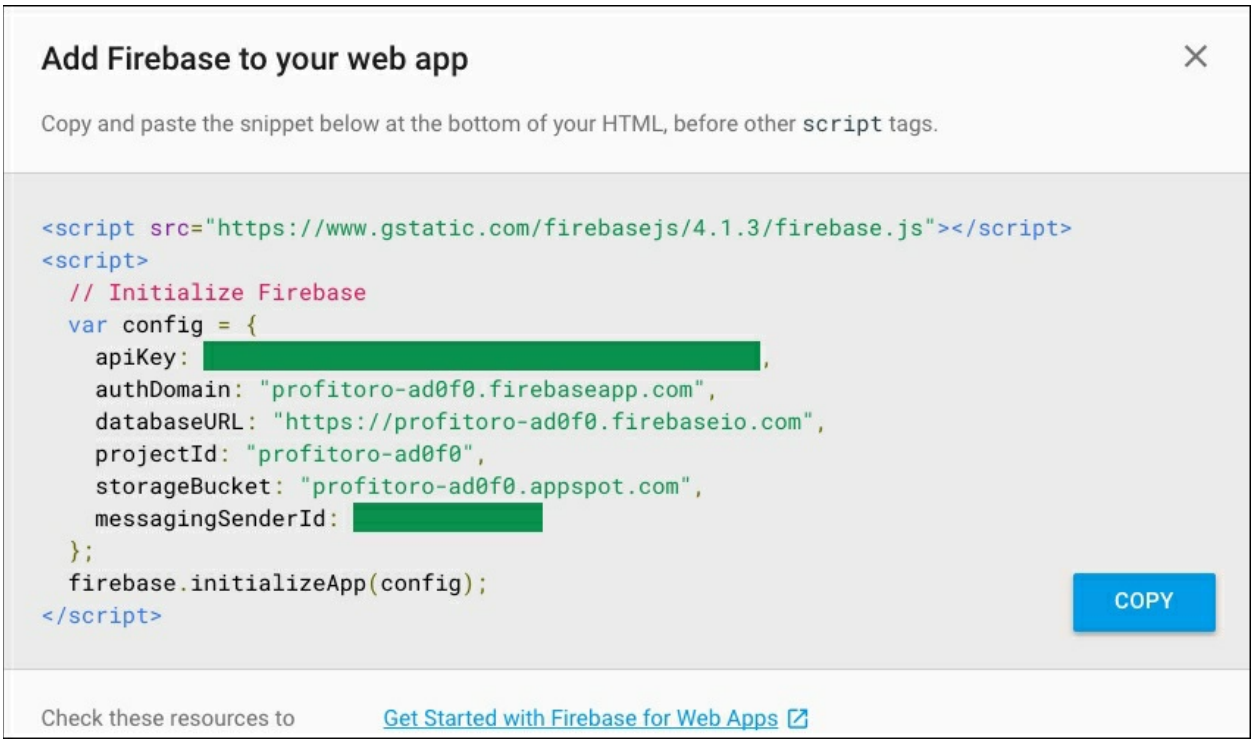
- **Email and password based authentication**: The classic way for authenticating users. Firebase provides a way to sign in users with email/password and log them in. It also provides methods to reset the user password.
- **Federated entity provider authentication**: The way of authenticating users with an external entity provider, such as Google, Facebook, Twitter, or GitHub.
- **Phone number authentication**: The way of authenticating users by sending them an SMS with a code that they will have to...

How to connect the Firebase authentication API to a web application

In order to connect your application to the Firebase authentication API, you should start by creating a Firebase application instance:

```
let config = {
  apiKey: 'YourAPIKey',
  databaseURL: 'YourDBURL',
  authDomain: 'YourAuthDomain'
}
let app = firebase.initializeApp(config)
```

You can find the necessary keys and URLs in the popup that opens if you click on the **Web Setup** button:



The setup config to use Firebase in a web application

Now you can use the app instance to access the `auth()` object and its methods. Check out the official Firebase documentation regarding the authentication API: https://firebase.google.com/docs/auth/users.

The most important part of the API for us is the methods to create and sign in a user, and the method that listens to the changes in the authentication state:

```
app.auth().createUserWithEmailAndPassword(email, password)
```

Or:

```
app.auth().signInWithEmailAndPassword(email, password)
```

The method that listens to the changes in the authentication state of the application is called `onAuthStateChanged`. You can set the important properties inside of this method...

Authenticating to the ProFitOro application

Let us now make signing in and logging in to our ProFitOro application possible! First, we have to set up the Firebase instance and figure out where we should put all the methods related to authentication. The Firebase application initialization has already been done inside the `store/index.js` file. Just add the `apiKey` and `authDomain` configuration entries if you still do not have them included in the `config`:

```
// store/index.js
let config = {
  apiKey: 'YourAPIKey',
  databaseURL: 'https://profitoro-ad0f0.firebaseio.com',
  authDomain: 'profitoro-ad0f0.firebaseapp.com'
}
let firebaseApp = firebase.initializeApp(config)
```

I will also export `firebaseApp` within the store's state property using the spread `...` operator:

```
//store/index.js
export default new Vuex.Store({
  state: {
    ...state,
    firebaseApp
  },
  <...>
})
```

I will also add a `user` property to our state so we can reset it on the `onAuthStateChanged` listener's handler:

```
// store/state.js
export default {
  config,
  statistics,
  user,
  isAnonymous: false
}
```

Let us also create a small mutation that will reset the value of the `user` object to...

Making the authentication UI great again

We have just implemented the authentication mechanism for our ProFitOro application. That's great, but the UI of our authentication page looks as if we've used a time machine and gone back 20 years to the early days of the internet. Let's fix it using our powerful friend – Bootstrap.

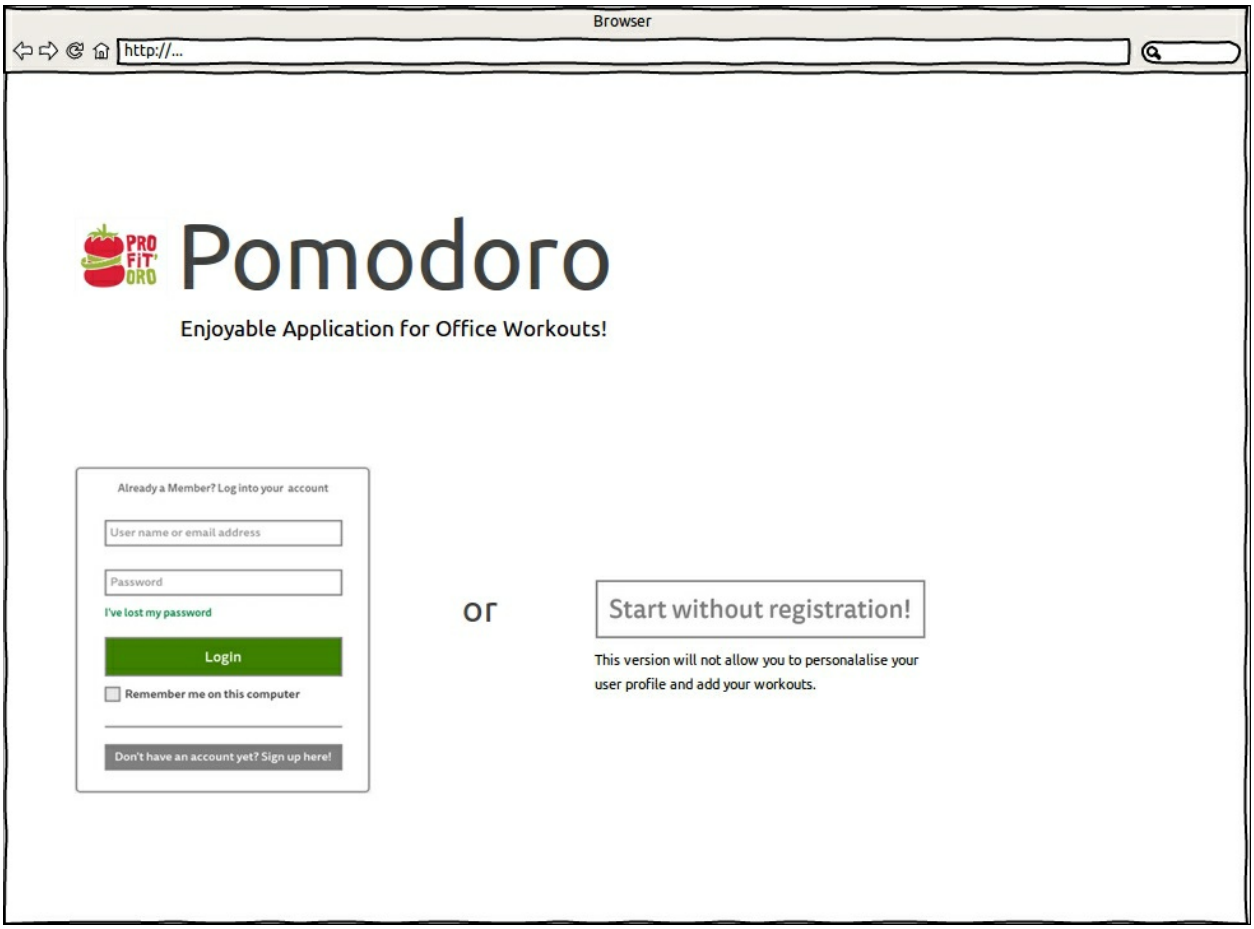
First of all, I would like to make my landing page layout a two-column grid layout, so the whole sign-in/login belongs to the left column and the button that leads the user to the application without being registered stays on the right side. However, I would like these two columns to be stacked on mobile devices.

This is nothing new for you; I suppose that you remember how to use Bootstrap's grid layout in order to achieve this behavior: https://v4-alpha.getbootstrap.com/layout/grid/. So, in our `LandingPage` component, I will just wrap the authentication and `go-to-app-link` components into the `div` with the `row` class and add the corresponding `col-*` classes to these components:

```
// LandingPage.vue
<template>
  <div>
    <...>
    <div class="container row justify-content-center">
      <div class="col-sm-12 col-md-6...
```

Managing the anonymous user

ProFitOro allows unregistered users to use the application as well. The only difference is that these unregistered users are not allowed to configure their settings as well, as they do not have access to their statistical data. They also cannot manage workouts. So, this is where we meet the second A of the triple-A definition – *authorization*. How can we manage these users? How can they actually enter the application if we only allow our users to sign up and log in? Well, for some reason, we have prepared the part that says **Go to App**. Let me remind you how it looks in the mockups:



Start without registration! button in the initial mockups

Luckily for us, the Firebase authentication API provides a method to sign in the anonymous user. The returned user object contains the `isAnonymous` attribute, which will allow us to manage the resources that can or can't be accessible to this anonymous user. So let's add the action called `authenticateAnonymous` and call the corresponding Firebase `auth` method within it:

```
// store/actions.js
authenticateAnonymous ({state}) {
```

```
state.firebaseApp.auth().
```

Personalizing the Pomodoro timer

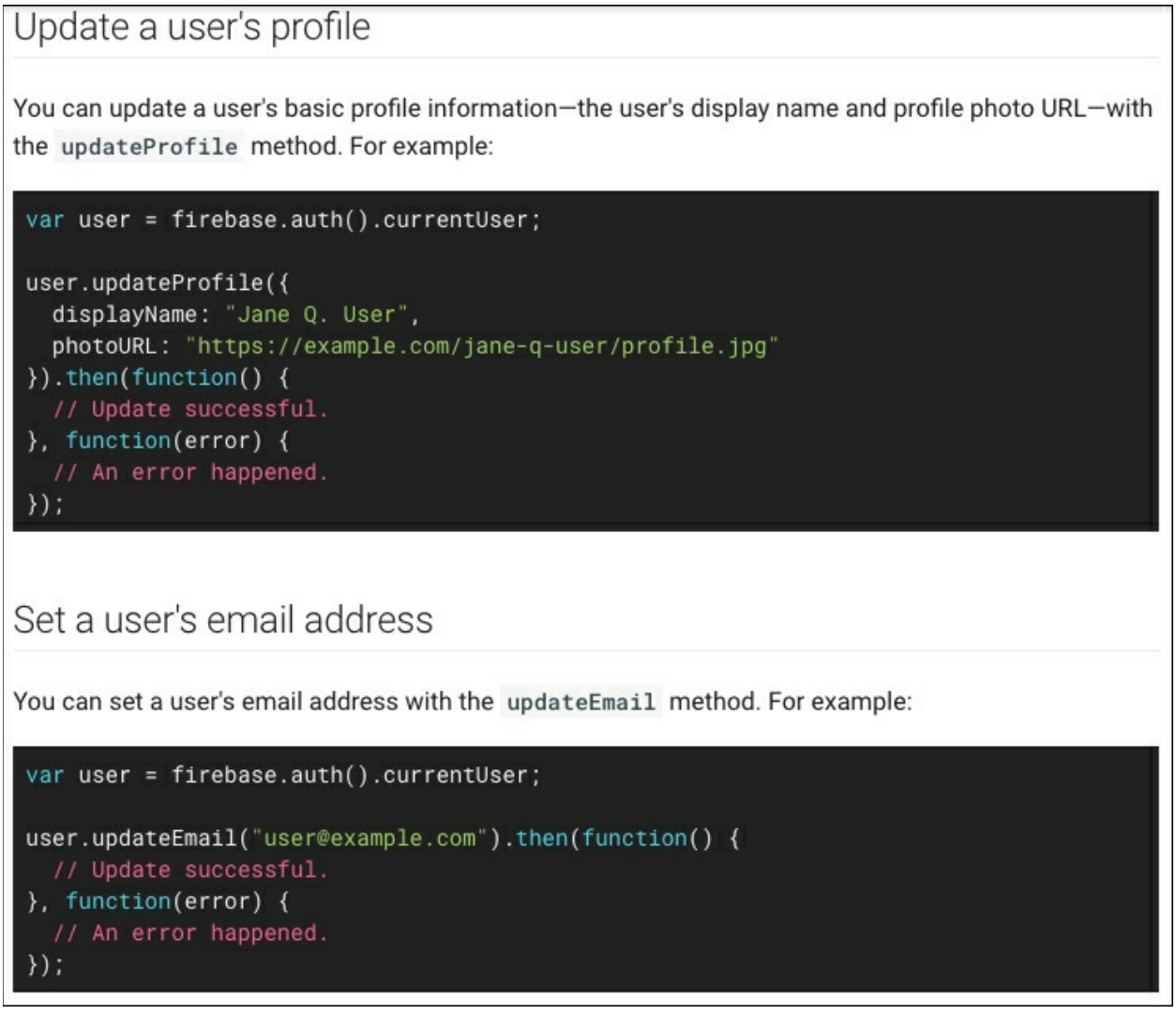
Well, now that we can already sign in new users and log in the existing ones, probably we should think about taking advantage of our authentication mechanism because right now we are actually not doing anything with it. We just sign up and we just log in. Yes, we also can hide or show some content based on the user's authentication, but this is not enough. The whole point of all this effort was to be able to store and retrieve the user's custom configuration for the Pomodoro timer and the user's statistical data.

Until now, we have been using a hardcoded database object with the key `test` in order to access the user's data, but now, since we already have our real users, it's time to populate the database with real users' data and use it in our application. Actually, the only thing we have to do is to replace this hardcoded value with the actual user's ID. So, for example, our code to bind the `config` reference was looking like this:

```
// store/actions.js
bindConfig: firebaseAction(({bindFirebaseRef, state}) => {
  if (state.user && !state.isAnonymous) {
    bindFirebaseRef('config', state.configRef)
 ...
```

Updating a user's profile

Wouldn't it be funny if we could welcome our user by displaying a welcome message saying something like **Welcome Olga**? But our users do not have names; they only have emails and passwords – two essential authentication components that are passed during the sign-up process. So, how can we do that? Well, if you have read with some attention the Firebase documentation regarding authentication (https://firebase.google.com/docs/auth/web/manage-users), you might have spotted these nice methods:



Update a user's profile

You can update a user's basic profile information—the user's display name and profile photo URL—with the `updateProfile` method. For example:

```
var user = firebase.auth().currentUser;

user.updateProfile({
  displayName: "Jane Q. User",
  photoURL: "https://example.com/jane-q-user/profile.jpg"
}).then(function() {
  // Update successful.
}, function(error) {
  // An error happened.
});
```

Set a user's email address

You can set a user's email address with the `updateEmail` method. For example:

```
var user = firebase.auth().currentUser;

user.updateEmail("user@example.com").then(function() {
  // Update successful.
}, function(error) {
  // An error happened.
});
```

Firebase methods for updating a user's profile and email address

Let's use these methods to update our user's profile and user's profile picture!

We will define three new actions – one that will update the user's display name by calling the

Firebase `updateProfile` method, one that will update the user's profile picture's URL by calling the same method, and another one that will call the `updateEmail` method. Then we will create the necessary markup in the `Settings.vue` component that will bind those actions on the corresponding input's update. Sounds easy, right? Believe me, it's as easy to implement as it actually...

Summary

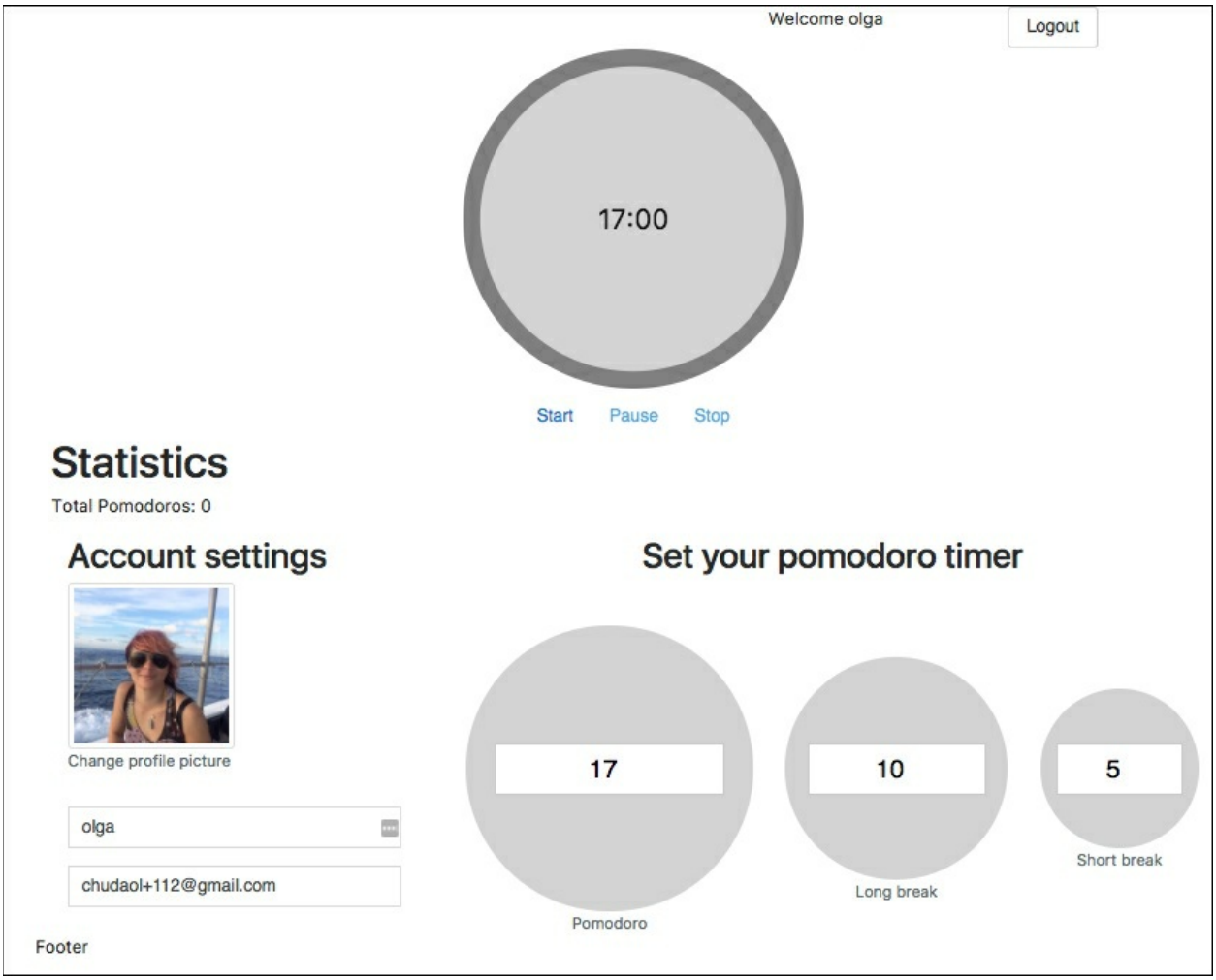
In this chapter, we have learned how to combine the Firebase real-time database and authentication API to update a user's settings. We have built a user interface that allows a user to update their profile settings. In just a few minutes, we have built the full authentication and authorization part of our application. I don't know about you, but I feel totally amazed about it.

In the next chapter, we will finally get rid of this huge page that contains all the parts of our application – the Pomodoro timer itself, statistics data, and the settings configuration view. We will explore one really nice and important feature of Vue – `vue-router`. We will combine it with Bootstrap's navigation system in order to achieve a nice and smooth navigation. We will also explore such a hot topic as code splitting in order to achieve lazy loading for our application. So, let's go!

Chapter 7. Adding a Menu and Routing Functionality Using vue-router and Nuxt.js

In the previous chapter, we added a very important feature to our application – *authentication*. Now, our users are able to register, log in to the application, and manage their resources once they are logged in. So, now they can manage the configuration of the Pomodoro timer and their account's settings. They also have access to their statistics data once they are logged in. We have learned how to use Firebase's authentication API and connect the Vue application to it. I must say, the previous chapter has been extensive in learning and a very backend oriented chapter. I enjoyed it a lot and I hope you enjoyed it as well.

Despite having this complex feature of authentication and authorization, our application still lacks navigation. For simplicity reasons, we are currently displaying all the application's parts on the main page. This is… ugly:



Admit it, this is ugly

In this chapter, we are not going to make things beautiful. What we are going to do is make things navigable so that all parts of the application are accessible through navigation. We...

Adding navigation using vue-router

I hope you still remember from the second chapter what `vue-router` is, what it does, and how it works. Just to remind you:

> Vue-router is the official router for Vue.js. It deeply integrates with Vue.js core to make building Single Page Applications with Vue.js a breeze.

-(From the official documentation of vue-router)

The `vue-router` is very easy to use, and we don't need to install anything – it already comes with the default scaffolding of Vue applications with a webpack template. In a nutshell, if we have Vue components that should represent the routes, this is what we have to do:

- Tell Vue to use `vue-router`
- Create a router instance and map each component to its path
- Pass this instance to the options of a Vue instance or component
- Render it using the `router-view` component

Note

Check the official `vue-router` documentation: https://router.vuejs.org

When you create your router, you should pass the array of routes to it. Each array item represents the mapping of a given component to some path:

```
{
  name: 'home',
  component: HomeComponent,
 ...
```

Using Bootstrap navbar for navigation links

Our current navigation bar is great – it's functional, but not responsive. Luckily for us, Bootstrap has a `navbar` component that implements responsiveness and adaptiveness for us. We just have to wrap our navigation elements with some Bootstrap classes and then sit back and check our beautiful navigation bar that collapses on mobile devices and expands on desktop devices. Check Bootstrap's documentation regarding the `navb ar` component: https://v4-alpha.getbootstrap.com/components/navbar/.

> ## Note
>
> Keep in mind that this URL is for the alpha version. The next stable version 4 will be available on the official website.

These are the classes we are going to use to transform our simple navigation bar into a Bootstrap-managed responsive navigation bar:

- `navbar`: This wraps the whole navigation bar element
- `navbar-toggleable-*`: This should also wrap the whole navigation bar element and will tell it when to toggle between expanded/collapsed state (for example, `navbar-toggleable-md` would make navigation bar collapse on medium-size devices)
- `navbar-toggler`: This is a class for the button that will be...

Code splitting or lazy loading

When we build our application to deploy for production, all the JavaScript is bundled into a unique JavaScript file. It's very handy, because once the browser loads this file, the whole application is already on the client side and no one is worried about loading more things. Of course, this is only valid for SPAs.

Our ProFitOro application (at least at this stage) benefits from such bundling behavior – it's small, it's a single request, everything is in place and we don't need to request anything from the server for any of the JavaScript files.

However, this kind of bundling might have some downsides. I am pretty sure that you have already built or have already seen huge JavaScript applications. There'll always be some point when loading huge bundles will become unbearably slow, especially when we want these apps to run on both desktop and mobile environments.

An obvious solution for this problem would be to split the code in such a way that different chunks of code are loaded only when they are needed. This is quite a challenge for single page applications and this is why we have a huge community...

Server-side rendering

Server-side rendering (**SSR**) recently became yet another popular abbreviation in the web development world. Used in addition to code splitting techniques, it helps you to boost the performance of your web application. It also positively affects your SEO, since all the content comes at once, and crawlers are able to see it immediately, contrary to cases where the content is being built in the browser after the initial request.

I found a great article about SSR that compares server and client side rendering (although it's from 2012). Check it out: http://openmymind.net/2012/5/30/Client-Side-vs-Server-Side-Rendering/.

It's fairly easy to bring server-side rendering to your Vue application – check the official documentation in this regard: https://ssr.vuejs.org.

It is important that our applications are performant; it is also important that SEO works. However, it is also important not to abuse the tools and not to introduce implementation overhead and overkill. Do we need SSR for the ProFitOro application? To answer this question let's think about our content. If there is a lot of content which is being brought...

Nuxt.js

While we were busy defining our router object, router links, code splitting and learning things about the server-side rendering, someone implemented a way of developing Vue.js applications without being worried about all these things at all. Just write your code. All the things like routing, code splitting and even server-side rendering will be handled behind the scenes for you! If you are wondering what the hell it is, let me introduce you to Nuxt.js: https://nuxtjs.org.

So, what is Nuxt.js?

Nuxt.js is a framework for creating Universal Vue.js Applications.

Its main scope is UI rendering while abstracting away the client/server distribution.

What's so great about it? Nuxt.js introduces the concept of pages – basically, pages are also Vue components, but each one of the pages represents a *route*. Once you define your components inside the `pages` folder they become routes without any additional configuration.

In this chapter, we will totally migrate our ProFitOro to the Nuxt architecture. So, brace yourself; we are going to make lots of changes! At the end of the chapter, our efforts will be rewarded with a piece of nice,...

Summary

In this chapter we have added basic routing to our application using different tools. First, we learned how to use vue-router to achieve routing functionality and then we used the Nuxt.js template to build a brand new application using old components and styles. We have used the concept of pages offered by Nuxt vue in order to achieve the same routing functionality as with `vue-router` and have transformed our ProFitOro application into a Nuxt application in an easy and unobtrusive way. We have significantly reduced the amount of code and learned something new. Total winners!

In this chapter we have also used Bootstrap's `navbar` to display our navigation routes in a nice and responsive way, and learned that even with the most drastic refactoring, the functionality and responsiveness stays with us when we use the Bootstrap approach. Once again – great success!

Our application is almost fully functional, however, it still lacks its main functionality – workouts. For now, during the Pomodoro intervals we are showing a hardcoded pushups workout.

Are you using the ProFitOro application while reading this book? If yes, I guess I...

Chapter 8. Let's Collaborate – Adding New Workouts Using Firebase Data Storage and Vue.js

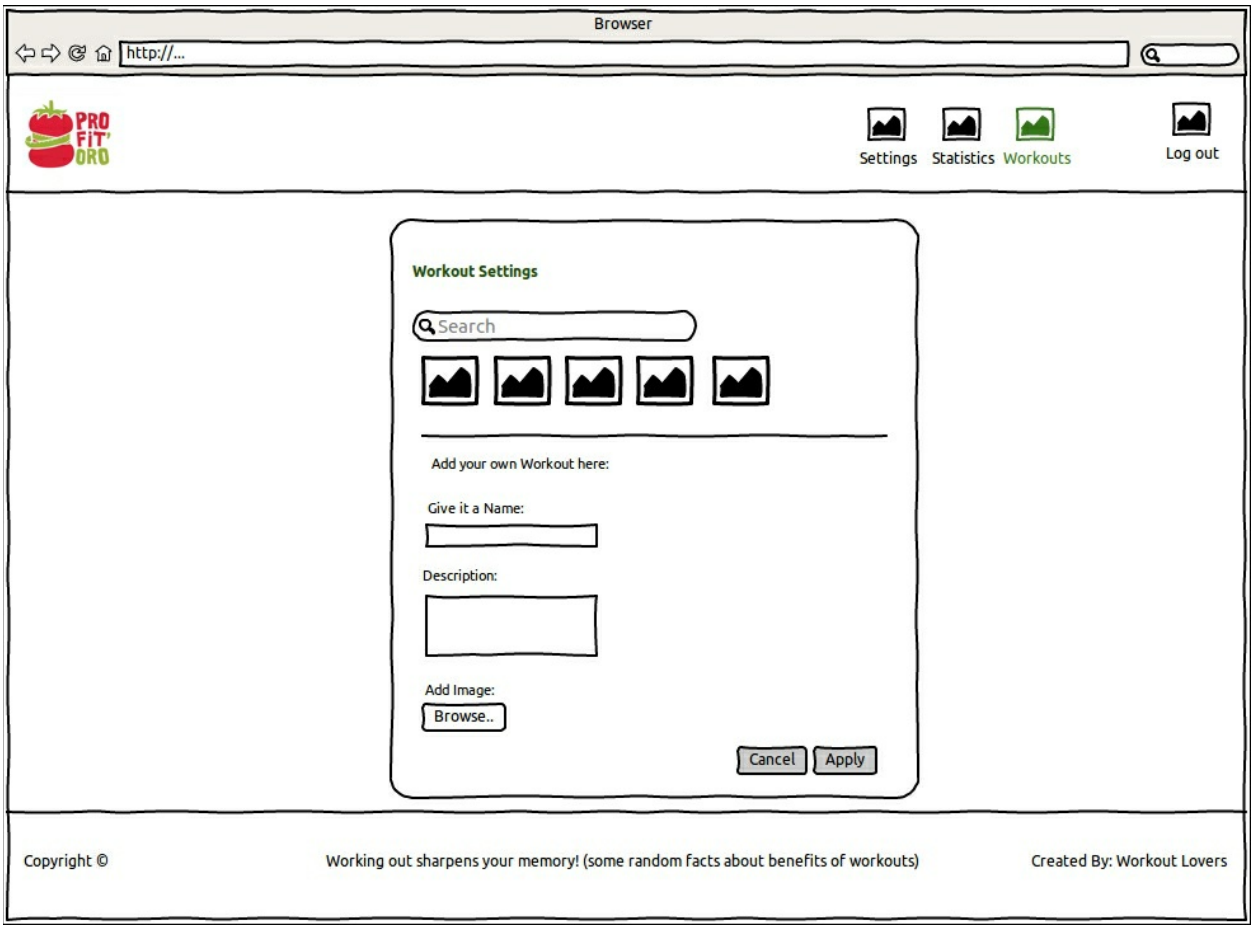
In the previous chapter, we learned how to add some basic navigation to the Vue application using both `vue-router` and `Nuxt.js`. We have redesigned our ProFitOro application, transforming it into a Nuxt-based application. Now our application is functional, it has an authentication mechanism, and it is navigable. However, it still lacks one of the most important features – workouts. In this chapter, we are going to implement the workout management page. Do you still remember its requirements from Chapter 2, *Under the Hood – Tutorial Explained*?

This page should allow users to see the existing workouts in the database, select or deselect them to be shown up during the Pomodoro breaks, rate them, and even add new workouts. We are not going to implement all these features. However, we are going to implement enough for you to continue this application and finish its implementation with great success! So, in this chapter we are going to do the following:

- Define a responsive layout for the workout management page, which will consist of two...

Creating layouts using Bootstrap classes

Before we start implementing a layout for our workouts page, let me remind you what the mockup looks like:



This is how we have defined things initially in our mockups

We will do some things slightly differently - something similar to what we have done in the settings page. Let's create the two-column layout that will stack on mobile devices. So, this mockup will be valid for mobile screens but it will display two columns on desktop devices.

Let's add two components – `WorkoutsComponent.vue` and `NewWorkoutComponent.vue` – inside the `components/workouts` folder. Add some dummy text to the templates of these new components and let's define our two-column layout in the `workouts.vue` page. You certainly remember that in order to have stack columns on small devices and different-sized columns on other devices, we have to use the `col-*-<number>` notation, where `*` represents the size of the device (`sm` for small, `md` for medium, `lg` for large, and so on) and the number represents the size of the column, which might ranges from `1` to `12`. Since we want our layout to stack on small devices (this means that...

Making the footer nice

Aren't you tired of this hardcoded word "**Footer**" always lying around beneath our content?

The ugly flying hardcoded Footer always glued to our content

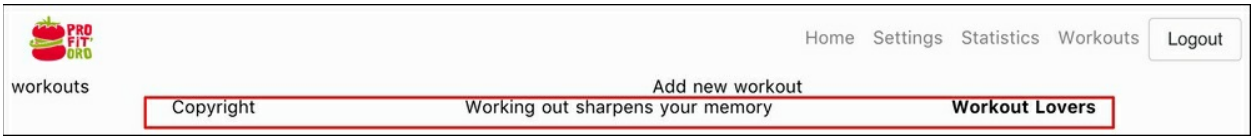
Let's do something with it! If you check our mockups, we have three columns there:

- One column for the copyright information
- Another one for the fact of the day
- And the last for the author information

You already know what to do, right? Again, we want these columns to be equally distributed on mediumand large-sized devices, and stack on mobile devices. Thus, our code will look like this:

```
// components/common/FooterComponent.vue
<template>
  <div class="footer">
    <div class="container row">
      <div class="copyright col-lg-4 col-md-4 col-sm-12">Copyright</div>
      <div class="fact col-lg-4 col-md-4 col-sm-12">Working out sharpens your memory</div>
      <div class="author col-lg-4 col-md-4 col-sm-12"><span class="bold">Workout Lovers</span></div>
    </div>
  </div>
</template>
```

Let's keep the fact of the day section hardcoded for now. Well, now our footer looks a bit nicer. At least it's not just the word "Footer" lying around:



Our footer is not just the word...

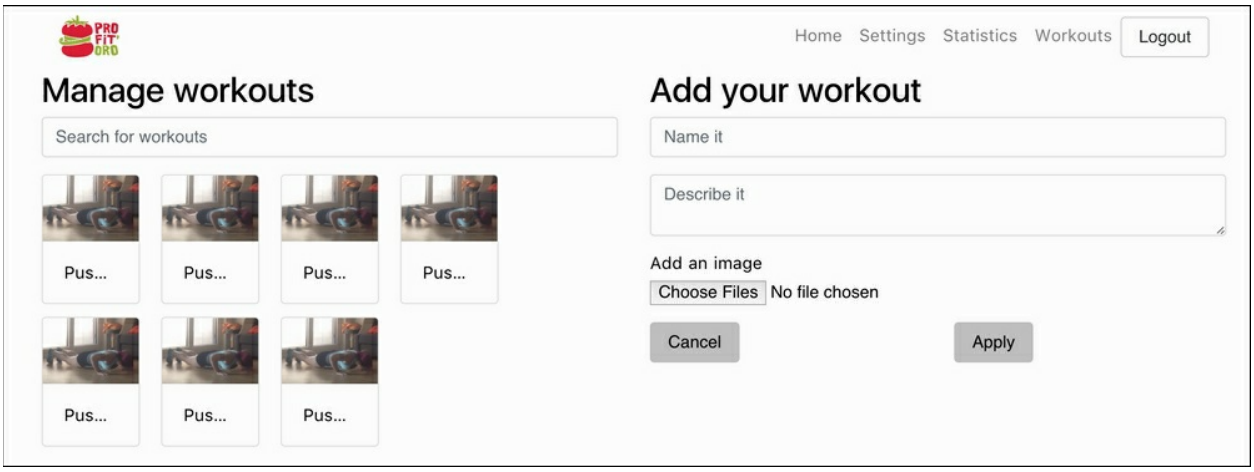
Storing new workouts using the Firebase real-time database

Before starting this section, check the code in the `chapter8/3/profitoro` folder. Both the `Workouts` and `NewWorkout` components are filled with a markup.

Note

Don't forget to run `npm install` and `npm run dev`!

It doesn't work yet, but it displays something:



Workout management page with some content

In this section, we are going to add workout objects to our workouts resource in the Firebase database. After that, we can finally learn how to store images using the Firebase data storage mechanism.

First, let's add Firebase bindings just like we've done for statistics and configuration objects. Open the `action.js` file and find the `bindFirebaseReferences` method. Here, we should add the binding for the `workouts` resource. So, this method now contains three bindings:

```
// state/actions.js
bindFirebaseReferences: firebaseAction(({state, commit, dispatch}, user) => {
  let db = firebaseApp.database()
  let configRef = db.ref(`/configuration/${user.uid}`)
  let statisticsRef = db.ref(`/statistics/${user.uid}`)
  let workoutsRef = db.ref('/workouts')

  dispatch('bindFirebaseReference',...
```

Storing images using the Firebase data storage

Firebase cloud storage allows you to upload and retrieve different content (files, videos, images, and so on). In a very similar way, Firebase provides a way of accessing and managing your database, where you can access and manage your storage buckets. You can upload Blobs, strings in Base64, file objects, and so on.

First of all, you should tell your Firebase application that you are going to use Google cloud storage. Thus, you need to add a `storageBucket` attribute to your application configuration object. Check your application's settings on the Google Firebase console and copy the `storageBucket` reference to the `firebase/index.js` file:

```
// Initialize Firebase
import firebase from 'firebase'
//...
let config = {
  apiKey: 'YOUR_API_KEY',
  databaseURL: 'https://profitoro-ad0f0.firebaseio.com',
  authDomain: 'profitoro-ad0f0.firebaseapp.com',
  storageBucket: 'gs://profitoro-ad0f0.appspot.com'
}
//...
```

Now your firebase application knows what storage bucket to use. Let's also open the data storage tab of the Firebase console and add a folder for our workout images. Let's call...

Using a Bootstrap modal to show each workout

Now we can see all the existing workouts on the page, which is great. However, our users would really like to have a look at each of the workouts in detail – see the workouts' descriptions, rate them, see who has created them and when, and so on. It's unthinkable to put all this information in the tiny `card` element, so we need to have a way of magnifying each element in order to be able to see its detailed information. A Bootstrap modal is a great tool that provides this functionality. Check the Bootstrap documentation regarding the modal API: https://v4-alpha.getbootstrap.com/components/modal/.

Note

Note that Bootstrap 4, at the time of writing, is in its alpha stage and that's why at some point this link might not work anymore, so just search for the relevant information on the official Bootstrap website.

Basically, we need to have an element that will trigger a modal and a modal markup itself. In our case, each of the small workout cards should be used as a modal trigger; `WorkoutComponent` will be our modal component. So, just add `data-toggle` and `data-target` attributes to the `card`...

It's time to apply some style

Our application is fully functional now; it can be used right away. Of course, it is still not perfect. It lacks validations and some functionality, several requirements have not been implemented yet, and the most important thing…it lacks beauty! It's all gray, it doesn't have style…we are humans, we love beautiful things, don't we? Everyone implements styles in their own way. I strongly recommend that if you want to use this application, please find your own style and theme for it, and please implement it and share with me. I would love to see it.

As for me, since I am not a designer, I asked my good friend Vanessa (https://www.behance.net/MeegsyWeegsy) to create a nice design for the ProFitOro application. She did a great job! Since I was busy writing this book, I had no time to implement Vanessa's design, therefore I asked my good friend, Filipe (https://github.com/fil090302), to help me with it. Filipe did a great job as well! Everything looks exactly how Vanessa implemented it. We have used `scss`, so it must be familiar to you since we've been using it already in this application as a...

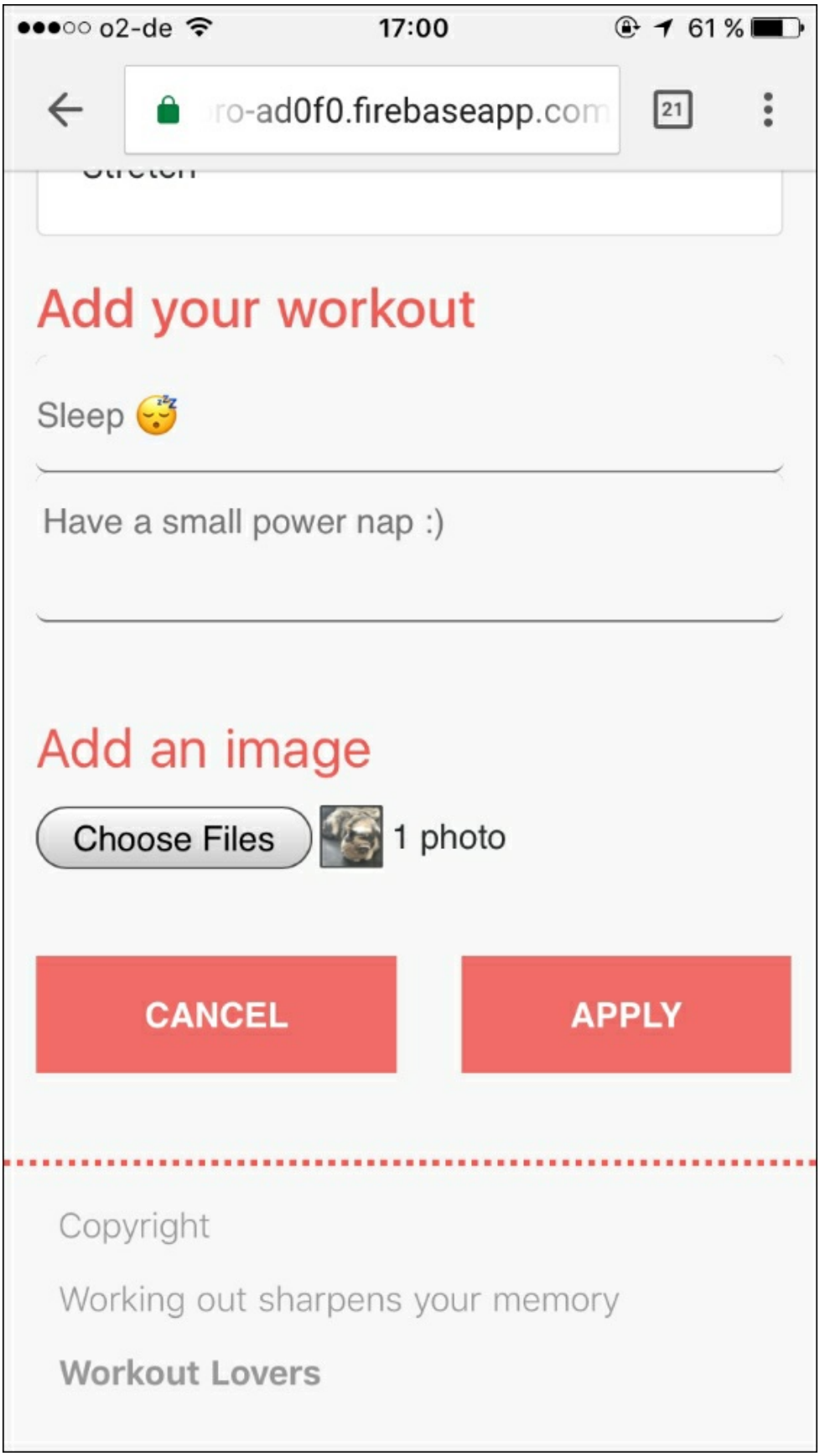
Summary

In this chapter, we have finally implemented the workout management page. Now we can see all the workouts stored in the database and create our own workouts. We have learned how to use the Google Firebase data storage system and API to store static files and we were able to store newly created workouts in the Firebase real-time database. We have also learned how to use a Bootstrap modal and used it to display each workout in a nice modal popup.

In the next chapter, we will do the most important job of every software implementation process – we will test what we have done so far. We will use Jest (https://facebook.github.io/jest/) to test our application. After that, we will finally deploy our application and define future work. Are you ready for testing your work? Then turn the page!

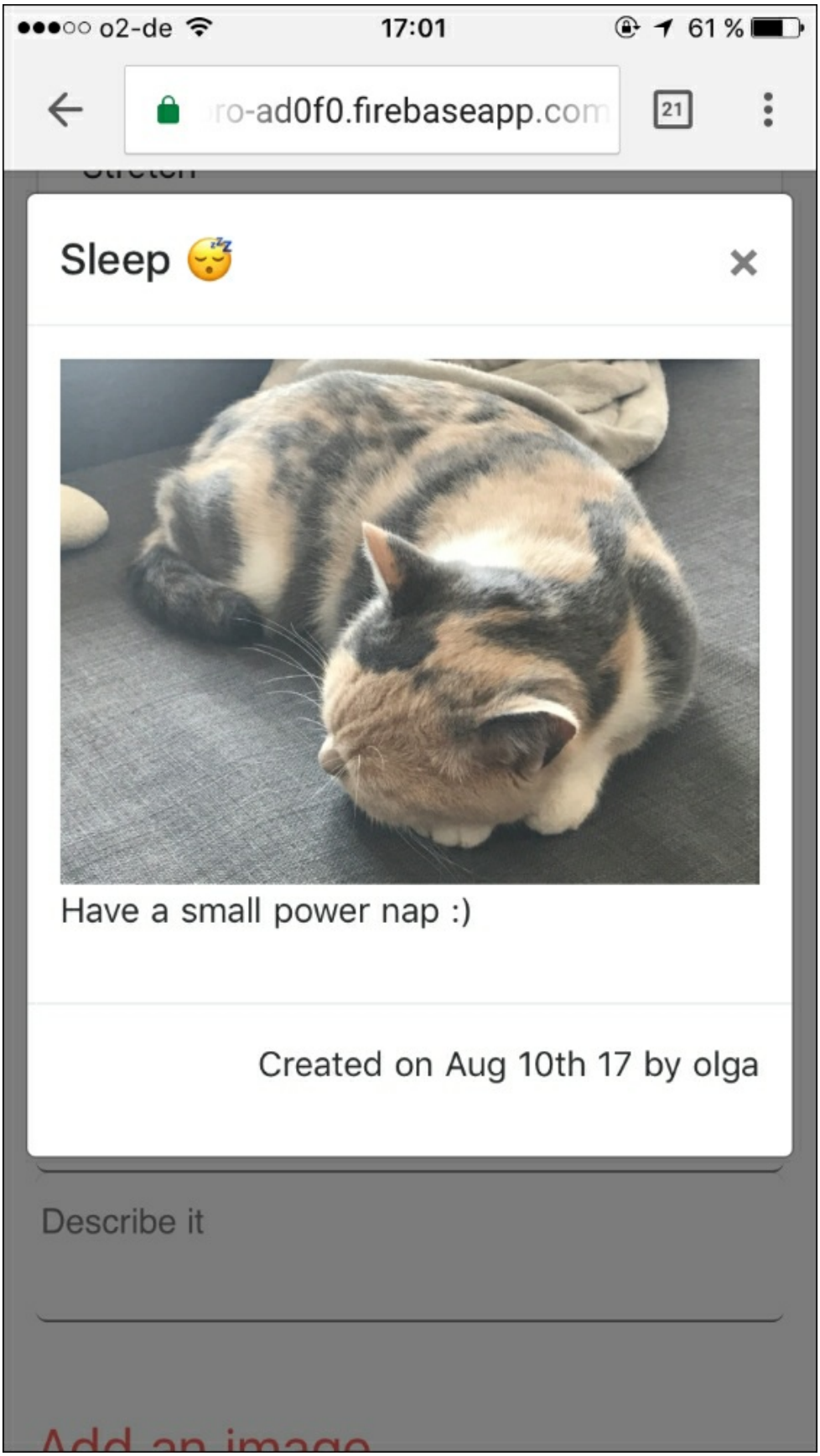
Chapter 9. Test Test and Test

In the previous chapter, we implemented the workout management page. We learned how to use the Google Firebase data storage mechanism to store static files and we again used the real-time database to store the workout objects. We used Bootstrap to build a responsive layout for the workout' management page and we learned how to use Bootstrap's modal component to display each individual workout in a nice popup. Now we have a totally responsible application. Thanks to Bootstrap, we had to implement nothing special to have a nice mobile representation. Here's what adding new workouts looks like on a mobile screen:



Adding a new workout on a mobile screen

And this is what our modal looks like on a mobile device:



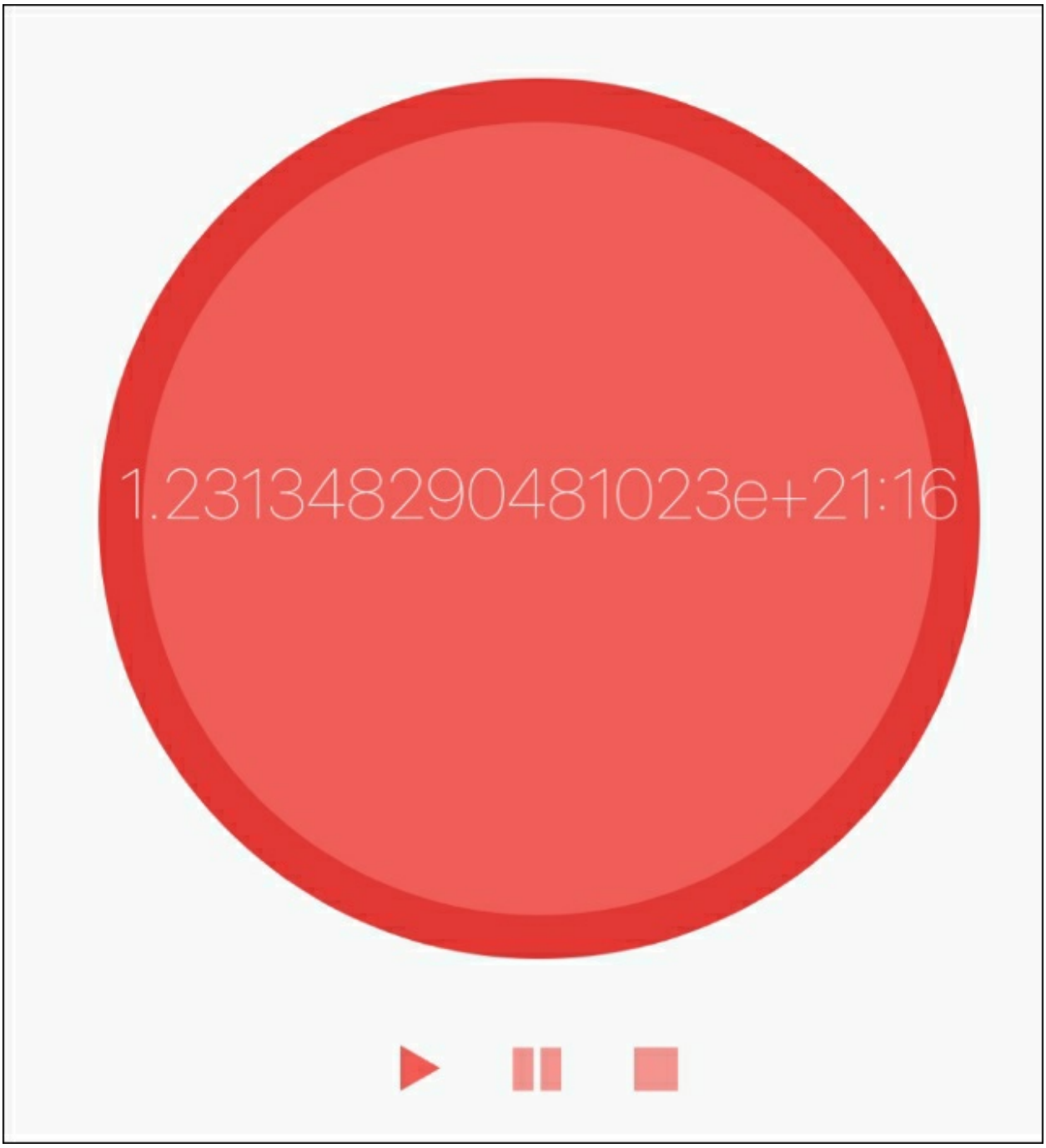
Workout modal displayed on a mobile device

Now it's time to test our application. We are going to use Jest (https://facebook.github.io/jest/) to build unit tests and run snapshot testing. In this chapter, we are going to do the following:

- Learn how to configure our Vue.js application to work with Jest
- Test Vuex stores using Jest assertions
- Learn how to mock complex objects with the `jest.mock` and `jest.fn`...

Why is testing important?

Our ProFitOro application works just fine, doesn't it? We have opened it so many times in the browser, we have checked all the implemented features, so it just works, right? Yes, that's true. Now go to your settings page and try to change the values of the timer to something strange. Try it with negative values, try it with huge values, try it with strings, and try it with empty values… do you think that can be called a nice user experience?



You wouldn't like to work during this number of minutes, would you?

Have you tried to create a strange workout? Have you tried to introduce a huge workout name at its creation and see how it displays? There are thousands of corner cases and all of them should be carefully tested. We want our application to be maintainable, reliable, and something that offers an amazing user experience.

What is Jest?

You know that Facebook guys are never tired of creating new tools. React, redux, react-native and all this reactive family was not enough for them and they created a really powerful, easy-to-use testing framework called Jest: https://facebook.github.io/jest/. Jest is pretty cool because it's self-contained enough for you to not to be distracted by extensive configuration or by looking for asynchronous testing plugins, mocking libraries, or fake timers to use along with your favorite framework. Jest is all in one, although pretty lightweight. Besides that, on every run, it only runs those tests that have been changed since the last test run, which is pretty elegant and nice because it's fast!

Initially created for testing React applications, Jest turned out to be suitable for other purposes, including Vue.js applications.

Check out the great talk given by Roman Kuba during the Vue.js conference in June 2017 in Poland (https://youtu.be/pqp0PsPBO_0), where he explains in a nutshell how to test Vue components with Jest.

Our application is not just a Vue application, it is a Nuxt application that uses Vuex stores and...

Getting started with Jest

Let's start by testing a small sum function and check that it correctly sums two numbers.

The first step would be, of course, to install Jest:

```
npm install jest
```

Create a directory `test` and add a file called `sum.js` with the following content:

```
// test/sum.js
export default function sum (a, b) {
  return a + b
}
```

Now add a test spec file for this function:

```
// sum.spec.js
import sum from './sum'

describe('sum', () => {
  it('create sum of 2 numbers', () => {
    expect(sum(15, 8)).toBe(23)
  })
})
```

We need a command to run tests. Add an entry `"test"` to the `package.json` file that will call a command `jest`:

```
// package.json
"scripts": {
  //...
  "test": "jest"
}
```

Now if you run `npm test`, you will see some errors:

Errors in the test output with when we run tests with Jest

This happens because our Jest is not aware we are using *ES6*! So, we need to add the `babel-jest` dependency:

```
npm install babel-jest --save-dev
```

After *babel-jest* is installed, we have to add a `.babelrc` file with the following content:

```
// .babelrc
{
  "presets": ["es2015"]
}
```

Aren't you annoyed about your IDE warnings regarding `describe`, `it`, and other globals that...

Testing utility functions

Let's test our code now! Let's start with utils. Create a file called `utils.spec.js` and import the `leftPad` function:

```
import { leftPad } from '~/utils/utils'
```

Have a look at this function again:

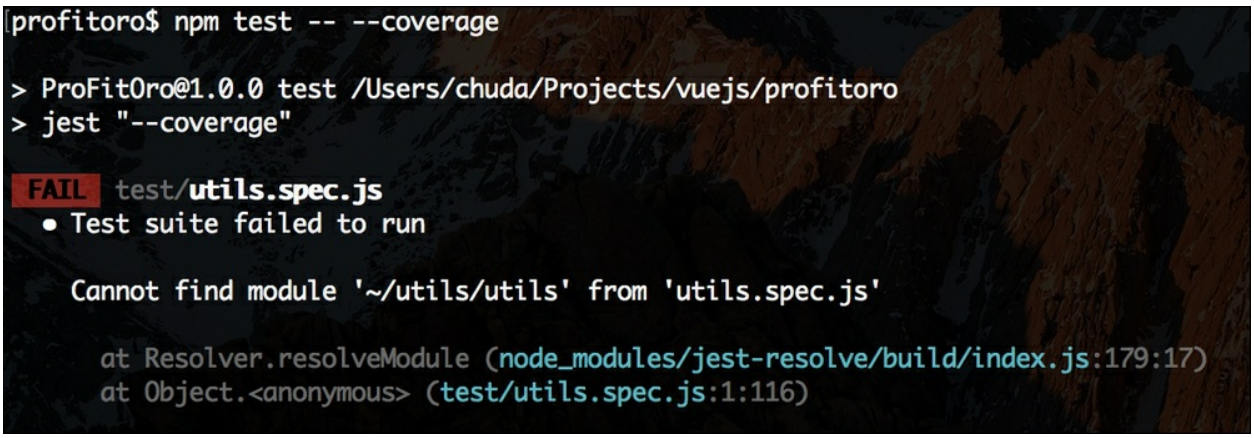
```
// utils/utils.js
export const leftPad = value => {
  if (('' + value).length > 1) {
    return value
  }

  return '0' + value
}
```

This function should return the input string if this string's length is greater than `1`. If the string's length is `1`, it should return the string with a preceding `0`.

Seems quite easy to test it, right? We would write two test cases:

```
// test/utils.spec.js
describe('utils', () => {
  describe('leftPad', () => {
    it('should return the string itself if its length is more than 1', () => {
      expect(leftPad('01')).toEqual('01')
    })
    it('should add a 0 from the left if the entry string is of the length of 1', () => {
      expect(leftPad('0')).toEqual('00')
    })
  })
})
```

Argh...if you run this test, you will get an error:



Of course, poor Jest, it is not aware of the aliases we've been using in our Nuxt application. The `~` notation for it equals nothing! Luckily for us, it is easy to fix....

Testing Vuex store with Jest

Let's now try to test our Vuex store. The most critical parts of our store to test are our actions and mutations because they can actually mutate the store's state. Let's start with the mutations. Create the `mutations.spec.js` file in the `test` folder and import `mutations.js`:

```
// test/mutations.spec.js
import mutations from '~/store/mutations'
```

We are ready to write unit tests for our mutation functions.

Testing mutations

Mutations are very simple functions that receive a state object and set some of its attribute to the given value. Thus, testing mutations is fairly easy—we have just to mock the state object and pass it to the mutation we want to test with a value we want to set. In the end, we have to check whether the value has been actually set. Let's, for example, test the mutation `setWorkingPomodoro`. This is what our mutation looks like:

```
// store/mutations.js
setWorkingPomodoro (state, workingPomodoro) {
  state.config.workingPomodoro = workingPomodoro
}
```

In our test, we need to create a mock for the state object. It doesn't need to represent the complete state; it needs to at least mock the

Making Jest work with Vuex, Nuxt.js, Firebase, and Vue components

---

It's not the easiest task to test Vue components that rely on the Vuex store and Nuxt.js. We have to prepare several things.

First of all, we must install `jest-vue-preprocessor` in order to tell Jest that Vue components files are valid. We must also install `babel-preset-stage-2`, otherwise Jest will complain about the ES6 *spread* operator. Run the following command:

```
npm install --save-dev jest-vue-preprocessor babel-preset-stage-2
```

Once the dependencies are installed, add the `stage-2` entry to the `.babelrc` file:

```
// .babelrc
{
  "presets": ["es2015", "stage-2"]
}
```

Now we need to tell Jest that it should use the `babel-jest` transformer for the regular JavaScript files and the `jest-vue-transformer` for the Vue files. In order to do so, add the following to the jest entry in the `package.json` file:

```
// package.json
"jest": {
    "transform": {
      "^.+\\.js$": "<rootDir>/node_modules/babel-jest",
      ".*\\.(vue)$": "<rootDir>/node_modules/jest-vue-preprocessor"
    }
  }
```

We use some images and styles in our components. This might result in some errors because Jest doesn't know...

Testing Vue components using Jest

Let's start by testing the `Header` component. Since it depends on the Vuex store which, in its turn, highly depends on Firebase, we must do the exact same thing we just did to test our Vuex actions—mock the Firebase application before injecting the store into the tested component. Start by creating a spec file `HeaderComponent.spec.js` and paste the following to its `import` section:

```
import Vue from 'vue'
import mockFirebaseApp from '~/__mocks__/firebaseAppMock'
jest.mock('~/firebase', () => mockFirebaseApp)
import store from '~/store'
import HeaderComponent from '~/components/common/HeaderComponent'
```

Note that we first mock the Firebase application and then import our store. Now, to be able to properly test our component with the mocked store, we need to inject the store into it. The best way to do that is to create a `Vue` instance with the `HeaderComponent` in it:

```
// HeaderComponent.spec.js
let $mounted

beforeEach(() => {
  $mounted = new Vue({
    template: '<header-component ref="headercomponent"></header-component>',
    store: store(),
    components: {
      'header-component': HeaderComponent

 ...
```

Snapshot testing with Jest

One of the coolest features of Jest is *snapshot testing*. What is snapshot testing? When our components are being rendered, they produce some HTML markup, right? It would be really important that once your application is stable, none of the newly added functionality breaks the already existing stable markup, don't you think? That's why snapshot testing exists. Once you generate a snapshot for some component, it will persist in the snapshot folder and on each test run, it will compare the output with the existing snapshot. Creating a snapshot is really easy. After you mount your component, you should just call the expectation `toMatchSnapshot` on this component's HTML:

```
let $html = $mounted.$el.outerHTML
expect($html).toMatchSnapshot()
```

I will run snapshot testing for all the pages inside one test suite file. Before doing that, I will mock the getters of our Vuex store because there are some pages that use the user object, which is not initialized, thus resulting in an error. So, create a file `gettersMock` inside our `__mocks__` folder and add the following content:

```
// __mocks__/gettersMock.js
export default {
```

Summary

In this chapter, we used very hot technology to test our Vue application. We used Jest and learned how to create mocks, test components, and run snapshot testing with it.

In the next chapter, we will finally see our application live! We will deploy it using Google Firebase Hosting and provide the necessary CI/CD tooling so our application is deployed and tested automatically each time it is pushed to the master branch. Are you ready to see your work live, up and running? Let's go!

Chapter 10. Deploying Using Firebase

In the previous chapter, we set up the testing framework for our application's code, which will allow us from now on to cover it with unit tests and snapshot tests. In this chapter, we are going to make our application live! We will also set up the **Continuous Integration** (**CI**) and **Continuous Deployment** (**CD**) environments. Hence, in this chapter we are going to learn how to do the following:

- Deploy to Firebase hosting using Firebase tools locally
- Set up the CI workflow using CircleCI
- Set up both staging and production environments using Firebase and CircleCI

Deploying from your local machine

In this section, we are going to deploy our application using the Firebase command-line tools. We have already done it. Check the Google Firebase documentation for a quick start: https://firebase.google.com/docs/hosting/quickstart.

Basically, if you haven't yet installed Firebase tools, do it now!

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

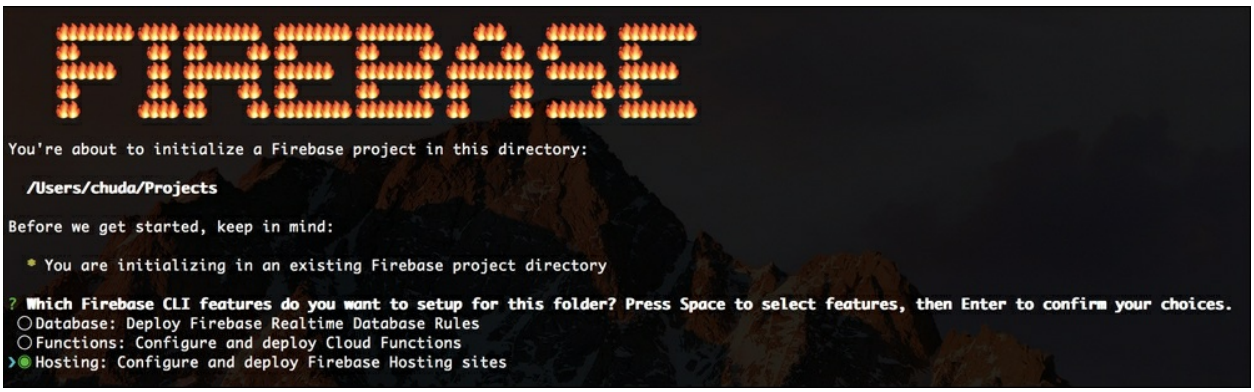
Now switch inside your project's directory and initialize a Firebase project:

```
firebase init
```

From the drop-down menu that appears, choose **hosting**.

Note

It's not really obvious, so keep in mind that to actually choose something from the list, you have to press *Space*.



Press Space to select the Hosting feature

After that, select your ProFitOro project from the list and after that, indicate the folder `dist` for the build's output directory:

Type dist for the public directory of your assets

Answer `No` to the next question and you are done! Make sure that Firebase creates both `firebase.json` and `.firebaserc` files in your project's folder.

This is what the `firebase.json` file looks like:

```
// firebase.json
{
  "hosting": {
    "public": "dist"
  }
}
```

And this is what your `.firebaserc` file will look...

Setting up CI/CD using CircleCI

Right now, if we want to deploy our application, we first have to run tests locally to ensure that everything is okay and nothing is broken and then deploy it using the `firebase deploy` command. Ideally, all of this should be automated. Ideally, if we push our code to the master branch, everything should just happen without our intervention. The process of automated deployment with automated test checks is called Continuous Deployment. This term means exactly what it sounds like – your code is being deployed continuously. There are lots of tools that allow you to automatically deploy your code to production once you hit the button or just push to the master branch. Starting with the good old but reliable Jenkins, going to Codeship, CloudFlare, CircleCI, Travis…the list is endless! We will use CircleCI, because it integrates nicely with GitHub. If you want to check how to deploy with Travis, check out my previous book on Vue.js:

https://www.packtpub.com/web-development/learning-vuejs-2

First of all, you should host your project on GitHub. Please follow the GitHub documentation to learn how to...

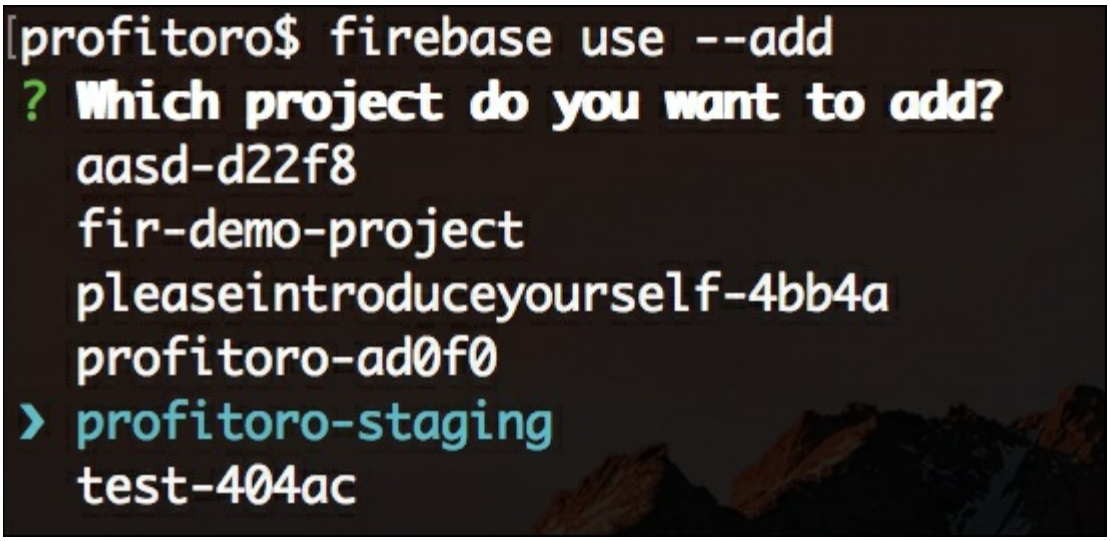
Setting up staging and production environments

You probably know that it's not very good practice to deploy to production right away. Even if the tests pass, we have to check whether everything is right first and that's why we need a *staging* environment.

Let's create a new project on the Firebase console and call it `profitoro-staging`. Let's now add a new environment to our project using the Firebase command-line tool. Just run this command in your console:

```
firebase use –add
```

Select the right project:



Select a newly created profitoro-staging project

Type the alias `staging` in the next step:

```
What alias do you want to use for this project? (e.g. staging) staging
```

Check that a new entry has been added to your `.firebaserc` file:

```
// .firebaserc
{
  "projects": {
    "default": "profitoro-ad0f0",
    "staging": "profitoro-staging"
  }
}
```

If you now locally run the command `firebase use staging` and `firebase deploy` after it, your project will be deployed to our newly created staging environment. If you want to switch and deploy to your production environment, just run the command `firebase use default` followed by the `firebase`

`deploy` command.

Now we...

What have we achieved?

Dear reader, we've been on a huge journey. We have built our responsive application from the very start until its deployment. We used nice technologies such as Vue.js, Bootstrap 4, and Google Firebase to build our application. Not only did we use all these technologies and learn how they play together, we actually followed the whole process of software development.

We started from the business idea, definition of requirements, definition of user stories, and creation of mockups. We continued with the actual implementation – both frontend and backend. We did thorough testing using Jest and we ended up with the deployment of our application into two different environments. Even more than just a deployment – we've implemented a CD strategy that will perform the deployment process for us automatically.

The most important thing – we've ended up with a fully functional application that will allow us to manage our time during work and stay fit!t live:

https://profitorolife.com/

I even created a Facebook page:

https://www.facebook.com/profitoro/

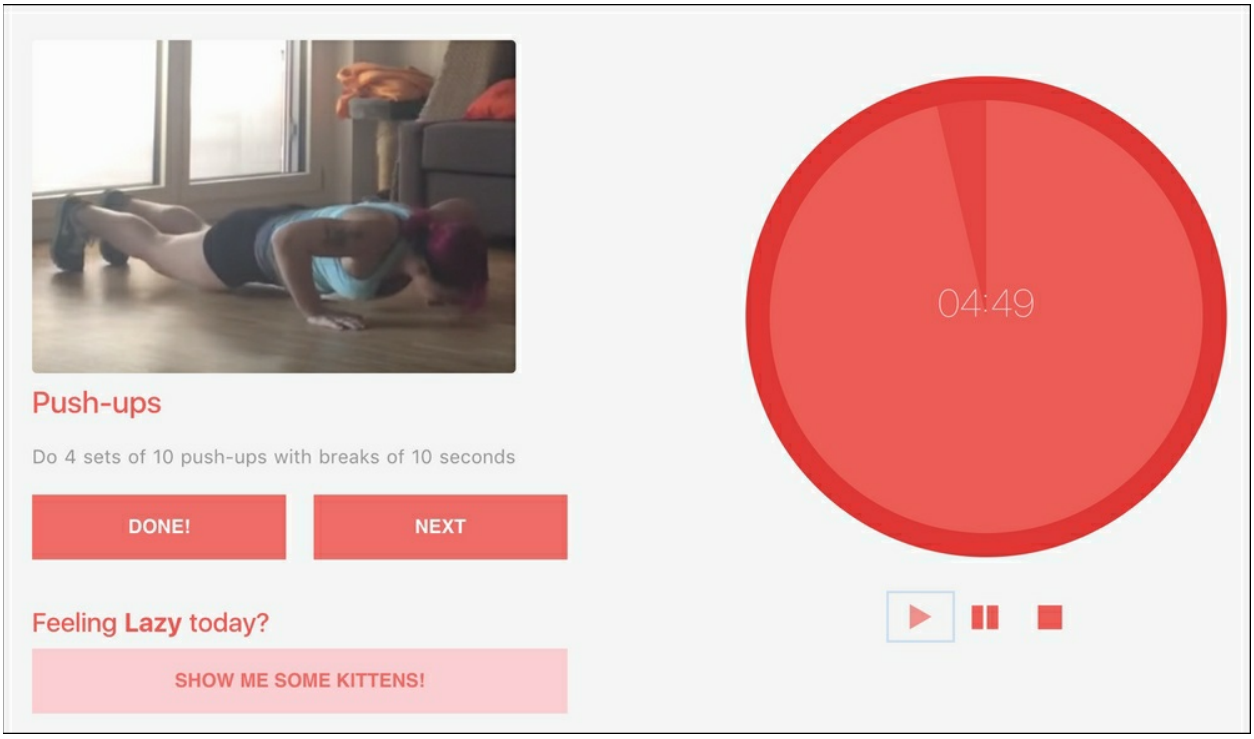
If you liked the ProFitOro logotype, send some love and thanks to...

Summary

In this chapter, we used CircleCI and Firebase to guarantee continuous quality of our continuously deployed software. As I already mentioned, it's so nice to see something that you've created from scratch up and running!

However, our work is not finished yet. There are so many improvements to make. We need validations. We need to write more tests to increase our code coverage! We need more workouts and we need them to look beautiful. We probably need some back office where someone responsible can check every added workout and approve it before it actually ends up in the list of workouts visible to everyone.

We need a proper statistics page with some beautiful graphics. We need to optimize the image rendering. We need to show more than one picture for each of the workouts. We probably need to add video support for the workouts. We also need to work a bit on the workout screen that appears once the Pomodoro working timer is over. Right now, it looks like this:



There are a lot of buttons here! None of them actually works :(

There are three buttons and none of them work.

So, as you can see, although we have finished the book and...

Index

A

B