The Ultimate Guide To
React / Firebase / TypeScript

Eric Morgan

Version v0.0.001, Dec. 30, 2021

Table of Contents

TODO: * [ ] (10) Other Services * Simple components (StyledInput) * Export from index file * How to share data across screens

Chapter 1. Introduction

This guide will walk you through the complete process for building a web application using React, Firebase, and TypeScript. Among the many things this guide has to offer, it will show you how to

- Setup your project and organize your code
- Model your data
- Build out UI components
- Manage user authentication
- Read and write from the database securely
- Setup services on the backend
- Deploy your web application to production

This guide not only provides the technical information for accomplishing these tasks, but it also provides a pattern you can use to build solid, sustainable web applications over and over.

To prove the efficacy of this guide, in the second half we will use all the steps in the guide to build out a full web application.

1.1. Technology

The tech stack chosen for this guide was selected for its capabilities, efficiency, and robustness. This tech stack is both minimal/compact and extremely powerful. It will enable you to build your web application quickly and to iterate on it as the needs of your application change.

1.1.1. React

React is one of the most popular JavaScript frameworks in existence at this time. The React framework makes building out user interfaces extremely simple. There is a lot of support in the community for React so if you get stuck on something, there’s a good chance you can find resources to help you out. React is also used extensively throughout the industry, so knowledge and expertise in this framework could help you professionally.

1.1.2. Firebase

Firebase is an all-in-one platform that provides a NoSQL database, user authentication support, the ability to run backend services, and much more. The Firebase platform is extremely powerful and having all these services wrapped up in one package means we have less frameworks to setup and configure.

1.1.3. TypeScript

TypeScript is essentially a typed version of JavaScript. Using TypeScript will make your code less prone to errors by allowing the TypeScript compiler to catch errors before they reach production.

Chapter 2. Setup

2.1. Project Planning

This guide is intended to get you up and running quickly and to set up your project for fast iteration. With that said, it is a wise idea to do some amount of planning before you actually start coding. This will ultimately improve your development efficiency over the course of your project.

Take a moment to write out some bullet points about what your application is required to do in order to be complete. Jot down some of the data that you'll need to save and think about how you might structure it. Maybe even sketch out a few screens for how users will interact with your application.

This doesn't have to be anything formal. Just spend a little time to make a few notes that you can refer back to as you develop your project.

2.2. Setup and Installation

To get started, let's first install React with TypeScript. You can find the instructions for this on the React documentation, but the command should be something like this:

```
yarn create react-app my-app --template typescript
```

Whenever I create a new React application, I always run the default project just to make sure everything is working properly and give myself a good baseline to work with. Once you've verified that your React app is working, add Firebase to your project.

```
yarn add firebase
```

Finally, we are going to add `react-router-dom` to help us with managing the screen navigation.

```
yarn add react-router-dom
```

Run the application again to ensure that everything is still working as expected.

Next, go to the Firebase website and set up a new project for your application. You should be given a JSON object with your API Key and some other information. Make a note of this because we will be saving this to a file soon.

2.3. Project Structure

The following structure is how I like to organize my code files for most projects. This is simply a suggestion, and you are free to make whatever adjustments you think will work best for you. Please note that some files have been omitted from this diagram for simplicity. In reality, you will have additional files like `package.json` and folders like `/node_modules`.

```
my-app/
 ├── server/
 ├── website/
 │    ├── build/
 │    ├── src/
 │    │    ├── components/
 │    │    ├── constants/
 │    │    │    ├── ApiKeys.ts
 │    │    ├── model/
 │    │    ├── navigation/
 │    │    ├── screens/
 │    │    │    ├── TestScreen.tsx
 │    │    ├── App.tsx
 │    │    ├── App.css
 │    │    ├── index.js
 ├── README.md
```

2.3.1. /server

You can ignore this for now, but this is where we will store our backend code eventually. Since we will be sharing some of the same modules between our client (website) and server, I like to keep everything under the same `my-app` directory.

2.3.2. /website

This is where all of the code for our UI (which will be a website) will be stored. The actual source code will be stored one layer deeper in `/src` and after we build the code, the `/build` directory will be generated for us with the compiled code.

2.3.3. /website/src/components

We will store our React components in this folder. If we have a lot of components, we may need to add additional subfolders to keep things organized.

2.3.4. /website/src/constants

We will store any application-wide constants in this folder. In the `ApiKeys.ts` file, we will paste our Firebase configuration object, so the file should look something like this, only with the data from your Firebase project:

/src/constants/ApiKeys.ts

```
const ApiKeys = {
    FirebaseConfig: {
        apiKey:             "",
        authDomain:         "",
        databaseURL:        "",
        projectId:          "",
        storageBucket:      "",
        messagingSenderId:  ""
    }
}

export default ApiKeys;
```

2.3.5. /website/src/model

This folder will store our interfaces, classes, data structures, etc. as well as functions that we want to use throughout our application. Be careful to keep the code in this folder as independent from your UI as possible because we will be sharing some of the code here with our backend code later on. This means that you should not put any code handling actual UI interaction in this folder.

2.4. The Importance of the "Test Screen"

You'll notice that I added a `TestScreen.ts` in the project structure. This `TestScreen` won't make it to the final production build of your application, but when first starting out, this is a great way to experiment with code. Think of it as a canvas that you can use to try out whatever you want.

I typically use it to design new components. I will build out my component on the `TestScreen` to get it close to how I want it to look, then store it away in my `/components` directory to use later.

Another advantage of this screen is that we can easily configure it to be the first screen that appears when we launch our application or refresh it. That way, if we make a change and end up needing to refresh, we don't have to click through multiple pages to get to the screen we want.

2.5. Miscellaneous Tips

2.5.1. Use Source Control Early

Add your code to source control as early as possible. My recommendation is to create a source code repository before you write a single line of code. Github is a great option.

2.5.2. Separate Generics from Application-Specific Code

If possible, try to separate any generic code that you write that you think could be reusable in other applications. For example, suppose your application requires you to write a bunch of functions that manipulate `Date` objects. Try to keep those in a separate module that you can easily copy into the next project that deals with `Dates`. Eventually, you will build up a library for yourself that will make every subsequent project easier and faster.

2.5.3. Avoid Too Many Frameworks

As you come across problems that you need to solve, you’ll be faced with the decision of implementing it yourself, or finding a framework that can do it for you. Think carefully before adding a new framework. Having too many of them can soon become unwieldly and difficult to work with. It can disturb the general style of your project. In addition to `Firebase` and `React` itself, we’ve already added one framework (`react-router-dom`). Whenever you add a new framework, just be certain that the benefits it provides you are worth it.

Chapter 3. Data Model

You don’t need to come up with the complete data model that your app will be using right away, but it’s a good idea to spend a little time thinking about some of the data elements that you’ll need to interact with and store in your database. You can even write this out on some notebook paper to start off with.

Once you have an idea of some of the data elements that you might need, create a `models.ts` file in your `/model` directory. Depending on the complexity of your application, you can break out your data model into whatever organizational structure you’d like, but for now, we’re just going to store things inside of `model/models.ts`.

Creating basic types up front and starting to use them as you develop your application will prevent you from doing a bunch of development and then coming back at some point "in the future" to clean everything up and add types. At times, it may feel like you’re constantly jumping back and forth between your `models.ts` file and your other modules to make adjustments. This may feel like it’s slowing you down. But this small extra effort to add type information now will ultimately make your application more reliable as you’ll be able to avoid certain types of bugs.

3.1. User Data

Your application will very likely need to save some information about the user. You’ll likely want to store their email address in case you need to send emails to certain users at some point. Go ahead and create a new interface called `IUser` to store some data you think you’re going to need for each user.

/model/models.ts

```
interface IUser {
    email: string;
    username: string;
    firstName: string;
    lastName: string;
}
```

3.2. Application Data

You will probably need additional data types depending on what your application does, so go ahead and add some additional interfaces in your `models.ts` file to cover whatever data you think you’re going to need. Remember, this is just an initial attempt and you will come back and make adjustments to this as you build out your application.

3.3. Flattening Data

For efficiency, you may want to "flatten" your data structure. This means rather than nesting things under one node, you spread it out (or "flatten") it across several nodes.

For example, if you are building a note-taking app, rather than having a structure like this:

```
{
    "users": {
        "email": "",
        "firstName": "",
        "lastName": "",
        "notes": {
            "1": {
                "title": "My Note 1 Title",
                "text": "My note 1 text",
                "date": 1459361875337,
            },

            "2": {
                "title": "My Note 2 Title",
                "text": "My note 2 text",
                "date": 157366585774,
            },

            "3": {
                "title": "My Note 3 Title",
                "text": "My note 3 text",
                "date": 1641679006603,
            }
        }
    }
}
```

Consider putting the note contents in a separate node and only referencing the IDs from your `/users` node. This way, if you need to pull data about a user, you aren't forced to download the contents of every single note. This can make loading data faster.

```
{
    "users": {
        "email": "",
        "firstName": "",
        "lastName": "",
        "notes": {
            "1": true,
            "2": true,
            "3": true
        }
    },

    "notes": {
        "1": {
            "title": "My Note 1 Title",
            "text": "My note 1 text",
            "date": 1459361875337,
        },

        "2": {
            "title": "My Note 2 Title",
            "text": "My note 2 text",
            "date": 157366585774,
        },

        "3": {
            "title": "My Note 3 Title",
            "text": "My note 3 text",
            "date": 1641679006603,
        }
    }
}
```

Now, when you grab a user's node, you get the IDs of all the notes belonging to that user and can perform another query specifically for that node within `/notes`. If a user creates a new note, you add the data to `/notes` and the ID to `/users/notes/{id}`. If a user deletes a note, you do something similar.

Chapter 4. Components

4.1. TypeScript Components

In this chapter we will define a pattern for creating React components using TypeScript. Our component will provide standard TypeScript type-checking. We will create two components: One to view an instance of a TypeScript class and one to edit an instance of that class.

To demonstrate, suppose we have the following TypeScript interface:

```
// User Interface
export interface IUser {
    username:   string;
    email:      string;
}
```

4.2. View Component

Let’s start with the View component and create the following skeleton component:

UserView.tsx

```
import React from 'react';
import { IUser } from './../model/models';
import './UserView.css';

const UserView: React.FC = (props) => {
    return (
        <div className="UserView-container">
            <p>UserView</p>
        </div>
    );
};

export default UserView;
```

Here we have a simple ReactJS component. We import the `IUser` interface from the `/models` folder and we also import the styles for this component. For this component, we’d like to pass in an `IUser` object in the following way:

```
<UserView user={myUser} />
```

To arrange this in our component, let’s define a new type:

```
interface Props { user: IUser };
```

We can now template our component so that it expects a property named `user` of type `IUser`. This allows us to get compile-time type checking on our component. We'll also display some of the `IUser` data. Our updated component looks like this:

UserView.tsx

```
import React from 'react';
import { IUser } from './../model/models';
import './UserView.css';

interface Props { user: IUser };
const UserView: React.FC<Props> = (props) => {
    return (
        <div className="UserView-container">
            <p>Username: {props.user.username}</p>
            <p>Email: {props.user.email}</p>
        </div>
    );
};

export default UserView;
```

We can also include a CSS file, which is optional if you'd like to use CSS to manage your styles:

UserView.css

```
.UserView-container {
    text-align: center;
    padding: 10px;
}
```

4.3. Edit Component

For the edit component, we will start with the `UserView` as a template, but add a few things to make it editable.

We want users of the component to receive an event whenever the user's email changes and provide them with the updated email address. Users will pass their event handler via a property so we will update our `Props` interface to include this event handler:

```
interface Props {
   user: IUser;
   onEmailChange(email: string): void;
}
```

Next, we will swap out the `<p>` element that we used to display the email with an `<input>` element that we can use to edit the property. Our `UserEdit` component looks like this:

UserEdit.tsx

```
import React, { FormEvent } from 'react';
import { IUser } from './../model/models';
import './UserEdit.css';

interface Props {
    user: IUser,
    onEmailChange(email: string): void;
};

const UserEdit: React.FC<Props> = (props) => {
    // Occurs when email is changed...
    const onEmailChange = (event: FormEvent) => {
        const email = (event.currentTarget as HTMLInputElement).value;
        props.onEmailChange(email);
    }

    return (
        <div className="UserEdit-container">
            <p>Username: {props.user.username}</p>
            <input type="email" value={props.user.email}
                onChange={(event) => { onEmailChange(event); }}
            />
        </div>
    );
};

export default UserEdit;
```

We can then update our `user` object like this:

```
<UserEdit user={myUser} onEmailChange={(email: string) => {
    setUser({ ...user, email: email });
}} />
```

4.4. Testing

Now, test your components. Open your Test Screen and import the `IUser` interface as well as the two components you've created:

```
import { IUser } from './model/models';
import UserView from './components/UserView';
import UserEdit from './components/UserEdit';
```

Create an `IUser` object and assign it to your application's state (here, we have created a concrete `User` class that implements `IUser`:

```
let user = new User("Paul", "paul@gmail.com");
const [user, setUser] = useState<IUser>(user);
```

Add the two components to your UI:

```
<UserView user={user} />
<UserEdit user={user} onEmailChange={(email: string) => {
    setUser({ ...user, email: email });
}} />
```

The first component should show you the values in your `User` object. The second component will allow you to update them in real-time. You should notice that as you change the email in `UserEdit`, that `UserView` will also update, confirming that your components are working properly.

Chapter 5. Client API

Once you have an idea of what functionality your app will require, it’s a good idea to start building up a library of functions to implement this functionality on the client. Even if you don’t fill in all the code right away, simply creating stub functions to be filled in later will be useful. This will allow you to keep your UI components simple, "dumb", and compact - as they should be. Your UI components should be as separated from your logic as possible

There’s a high likelihood that you’ll want to use some of this logic on the server as well so we will provide some tips on how to manage this. But, as you’re building out your client functionality, try to keep this in the back of your mind.

5.1. Stubbing Functionality

When you start a new project, you have an idea of what functionality it needs because that functionality is what defines your project. It’s useful to try to break down all of the functional pieces into methods and try to define the inputs and the outputs (also known as the interface). We can define these in "stub functions", which are just empty functions that simply define the interface of the functionality we need.

Suppose we’re developing an application that tracks the gas mileage for vehicles. At a minimum, we’ll need something to calculate the gas mileage, so let’s start with that. As our application becomes more defined, we can continue adding functions to this module. For example, as we add additional pages, we might discover functionality needed by those pages and we can add that here.

VehicleApi.ts

```
/** Computes the gas mileage */
export function getGasMileage(miles: number, gallons: number): number {
    //
    // TODO: Implement this later
    //
    return -1;
}

/** Computes the average gas mileage given a list of data */
export function getAverageGasMileage(miles: number[], gallons: number[]): number {
    //
    // TODO: Implement this later
    //
    return -1;
}
```

We can imagine that elsewhere in our application, we will receive data about how many miles were driven and how many gallons of gas were used (perhaps from a database), and rather than performing these calculations from within our screens or components, we can call this API and keep that logic seperate from our UI.

We use the marker `TODO:` to mark that the functionality is incomplete. Many IDEs have a way to search for this marker, but if not, it's easy and unique enough to search for yourself.

5.2. Procedural vs. Object-Oriented

In the example above, we exported functions. Consuming code can then import those functions and use them as needed. Another option would be to export a class containing static functions. For example:

VehicleApi.ts

```
export class VehicleApi {
    /** Computes the gas mileage */
    public static getGasMileage(miles: number, gallons: number): number {
        //
        // TODO: Implement this later
        //
        return -1;
    }

    /** Computes the average gas mileage given a list of data */
    public static getAverageGasMileage(miles: number[], gallons: number[]): number {
        //
        // TODO: Implement this later
        //
        return -1;
    }
}
```

Although it's very likely we will end up with a `Vehicle` class in our code, and we may wish to call something like `vehicle.getAverageGasMileage()`, I recommend sticking to these kind of non-object-based functions because in many cases, we will be loading objects from our database and when loaded, they will contain only the data and not the associated functions.

For instance, our code may end up looking something like this:

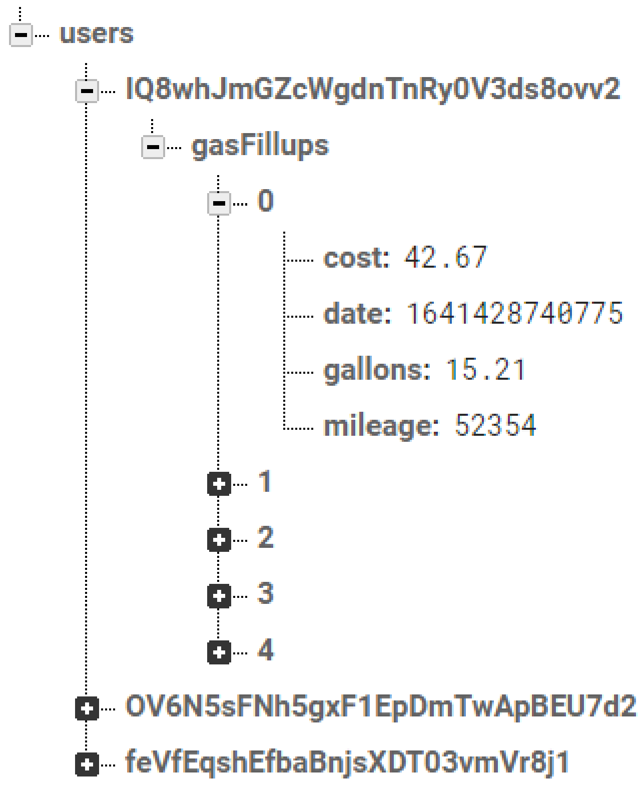
```
const vehicle = await Firebase.loadUserVehicle(userId);
const avgMileage = VehicleApi.getAverageGasMileage(vehicle.miles, vehicle.gallons);
```

When we load the `vehicle` from our database, all we have is a JSON object contianing data, not an actual instance of a `Vehicle` class.

5.3. Database API

Firebase allows us to easily and conveniently interact with the database from the client. Although Firebase has a great API, I find it useful to add an additional layer on top of that. This additional layer allows us to add typings to our inputs and outputs, and can help to abstract the data model a bit and prevent us from having to remember its exact structure.

For example, suppose our data structure consists of an array of times that the user has filled up the gas in their vehicle. Let's say it stores the date, the number of gallons they put into the vehicle, how much it cost, and what their vehicle's current mileage is.



```
users
    IQ8whJmGZcWgdnTnRy0V3ds8ovv2
        gasFillups
            0
                cost: 42.67
                date: 1641428740775
                gallons: 15.21
                mileage: 52354
            1
            2
            3
            4
    OV6N5sFNh5gxF1EpDmTwApBEU7d2
    feVfEqshEfbaBnjsXDT03vmVr8j1
```

Using the default Firebase API, we can grab the `gasFillups` data by calling:

```
firebase.database()
    .ref("users").child(userID)
    .child("gasFillups").once("value");
```

Every screen that needs to get gas mileage can simply call that chain of functions to get the data. The problem is that this is extremely difficult to remember. It's also really long, at risk of having typos, and would require each screen to import `firebase`.

Instead, let's wrap this in a more compact function that our screens can import. This will give us the benefit of type-checking.

```
export interface IGasFillup {
    date: number;
    gallons: number;
    mileage: number;
    cost: number;
}

// Get all IGasFillup instances for the given user...
export async function getGasFillups(userID: string): Promise<IGasFillup[]> {
    const snapshot = await firebase.database()
        .ref("users").child(userID)
        .child("gasFillups").once("value");
    return snapshot.val() as IGasFillup[];
}
```

Now our screens can simply `import` the `getGasFillups` method. Think of the benefits we are getting from this:

- Our output has type information, so we can get autocomplete and type-checking on all of the properties (`gallons`, `mileage`, etc).
- We can change the names of our data model (for example, if we want to rename the `gasFillups` node) without having to update our UI modules.
- The function is much more compact and makes the code look cleaner.

5.4. API Organization

I personally like to store functions like the ones in the previous section in a `Firebase` folder. I also like to separate functions that `read` data from functions that `write` data. I find this helps me to organize my code a little better, but you can stick with whatever style works for you. I would probably put this in a file called `/model/Firebase/Read.ts`.

/model/Firebase/Read.ts

```
export async function getGasFillups(userID: string): Promise<IGasFillup[]> {
    const snapshot = await firebase.database()
        .ref("users").child(userID)
        .child("gasFillups").once("value");
    return snapshot.val() as IGasFillup[];
}
```

In my `/model/Firebase` folder, I will also have a file called `index.ts` that will handle exporting all the modules in that folder:

/model/Firebase/index.ts

```
import * as Read from "./Read";
import * as Write from "./Write";
export { Read, Write };
```

And I would do something similar in the `/models` folder:

/model/Firebase/index.ts

```
import * as Firebase from "./Firebase";
export { Firebase };
```

In the end, your folder structure would look something like this, with each `index.ts` file exporting whatever folders or files are in the same directory as itself:

```
MyApp/
 ├── model/
 │    ├── Firebase/
 │    │    ├── Read.ts
 │    │    ├── Write.ts
 │    │    ├── index.ts
 │    ├── index.ts
```

The reason I like this organization is because then in my client modules (screens, components, etc.), I can use it like this:

MyScreen.tsx

```
import * as MyApp from "./../models"

//
// ...
//

// Load gas fillup data...
const loadData = async () => {
    const gasFillups = await MyApp.Firebase.Read.getGasFillups(userID);
    // Use data...
}
```

I personally like accessing the functions in this way, but like I said, feel free to organize the code in whatever style is best for you.

5.5. Sharing With the Server

There's a good chance that you'll want to share some of this functionality with your server. For example, maybe you want to do some computations on a nightly schedule and you need to load the data from the database. If you keep your UI sufficiently separate from your logical functionality, you should be able to easily copy these functions into your server repository and reuse them.

There are other ways of sharing the functionality between your frontend and backend. For example, you could create an NPM package in a separate repository and install it in both your frontend and backend. Personally, I just stick to copy-and-pasting the code between the two. It's not ideal, but it's easy and it works.

Chapter 6. User Authentication

6.1. The Basics

If your application requires users to login and to save data specific to each user, then you will need some type of user authentication functionality. Fortunately, Firebase has excellent support for user authentication, so we won't need to spend a lot of time building out this kind of thing and we can spend more time on our actual application.

Below is a list of some of the basic features that any user-authenticated application should have:

- Create an account
- Login
- Logout
- Recover a forgotten password
- Change password

Firebase supports several types of user authentication, including Google, Twitter, and Github. However, for the purposes of this guide, we will focus on simple email/password authentication.

6.2. Authentication API

The Firebase library has great functions for managing user authentication workflows. I personally prefer to create my own simple wrapper for these APIs. The reasons for this are:

- Provide some abstraction from Firebase in case I want to do things differently in the future
- "Simplify" the API by limiting the application's access to only functions in my wrapper module
- Allow me to add additional functionality on top of these functions. For example, performing a task after a user's account has been created

To begin, let's create a file for all the basic functionality that was listed above. Since these actions will require going to the Firebase server, we want them all to return `Promises` so we can wait until they have completed.

Auth.ts

```
import firebase from "firebase/compat/app";
import "firebase/compat/auth";
type UserCredential = firebase.auth.UserCredential;

// Create a new user with the given email/password...
export async function createUserWithEmailAndPassword(email: string,
    password: string): Promise<void> {

    return firebase.auth().createUserWithEmailAndPassword(email, password)
        .then((userData: UserCredential) => {
            const user: firebase.User | null = userData.user;
            if (!user) { return; }

            // Send the email verification...
            return user.sendEmailVerification();
        });
}

// Sign-in user with given email/password...
export async function signInWithEmailAndPassword(email: string, password: string):
Promise<UserCredential> {
    return firebase.auth().signInWithEmailAndPassword(email, password);
}

// Sign-out current user...
export async function signOut(): Promise<void> {
    return firebase.auth().signOut();
}

// Send password reset email to given email...
export async function sendPasswordResetEmail(email: string): Promise<void> {
    return firebase.auth().sendPasswordResetEmail(email);
}

// Change current user's password...
export async function changeCurrentUserPassword(currentPassword: string,
    newPassword: string): Promise<void> {

    const currentUser = firebase.auth().currentUser;
    if (!currentUser) { return Promise.resolve(); }

    return reauthenticateCurrentUser(currentPassword).then(() => {
        return currentUser.updatePassword(newPassword);
    });
}

// Reauthenticates current user (only works if password-authenticated)...
export async function reauthenticateCurrentUser(password: string):
Promise<UserCredential> {
    const currentUser = firebase.auth().currentUser;
    if (!currentUser) { return Promise.reject("No current user"); }
    if (!currentUser.email) { return Promise.reject("Current user does not have an
email"); }

    const credential = firebase.auth.EmailAuthProvider.credential(currentUser.email,
password);
    return currentUser.reauthenticateWithCredential(credential);
}
```

We now have a class that we can `import` into our screens and access the set of functions to assist with our user authentication workflows. We will return to this module in later sections and make adjustments and additions to it.

You'll see that the functions pretty much mirror the Firebase API. In fact, some of the functions are a single line of code and simply call a Firebase API of the same name, so you may be wondering why we need this module at all. If you have this feeling, let me give you a few reasons why you might want to use this.

Simplicity and Code Reuse To start off with, maybe you have a `Create Account` page where users will create their accounts. You'll probably have a `button` that creates an account whenever the user clicks it. So maybe in that button's `onClick` event, you can call `firebase.auth().createUserWithEmailAndPassword(…)`. After all, it's only a few lines of code.

Although it's not much code, you still need to handle other tasks, such as cases where the user couldn't be created, sending a verification email, etc. So wouldn't it be much simpler to just call `Auth.signInWithEmailAndPassword(email, password)`? Then you don't need to clutter up your nice `Create Account` page with all the minutiae of creating an account.

Additionally, what if you allow a user to create an account from another page at some point in the future? You wouldn't want to have to copy all that stuff (null checks, verification emails) into multiple places and then change that code in multiple places if you need to adjust the functionality in the future.

Additional Functionality Suppose you want to add functionality on top of the basic stuff that we've implemented above. For example, suppose whenever a user creates an account, you want to store off some meta-data about that user (a task that we will cover in later sections). Having your logic contained within this module will make that easier to manage and will ensure that all places where a user gets created are handled the same way.

Portability If you want to use the same functionality in a new application that you're developing, it's easier to just copy over a single file (`Auth.ts`) rather than extracting all of that functionality from various pages/components that you've created.

6.3. Login Screen

Now that we've created a module (`Auth.ts`) for managing the basic user authentication tasks, let's utilize that to create a basic login page.

LoginScreen.tsx

```
import { useState } from "react";
import { Auth } from "./../../model/MyApp/Firebase";

function LoginScreen(props: any) {
    const [isLoading, setIsLoading] = useState<boolean>(false);
    const [email, setEmail] = useState<string>("");
    const [password, setPassword] = useState<string>("");

    // Occurs when "Login" is clicked...
    const onLoginClicked = () => {
        setIsLoading(true);
        Auth.signInWithEmailAndPassword(email, password)
            .then(() => {
                setIsLoading(false);
                // Redirect user
            }, (error) => {
                setIsLoading(false);
                // Show an error message to user
            });
    }

    return (
        <div>
            <h2>Login</h2>

            <input type="email" value={email} placeholder="Email"
                onChange={(event) => { setEmail(event.target.value); }}
            /><br />

            <input type="password" value={password} placeholder="Password"
                onChange={(event) => { setPassword(event.target.value); }}
                onKeyPress={(event) => {
                    if (event.key === "Enter" || event.charCode === 13) {
                        onLoginClicked();
                    }
                }}
            /><br />

            <button onClick={onLoginClicked}>Login</button><br />

            {isLoading ? <span>Loading...</span> : null}
        </div>
    );
}

export default LoginScreen;
```

Notes on this screen:

- We have two `<input>` elements for entering the email and password. And two `state` variables for holding those values.
- The `isLoading` state variable will show a "Loading…" message at the bottom while we are checking the user’s login credentials.
- When the `Login` button is pressed, we call our function `Auth.signInWithEmailAndPassword`.
- Depending on the result of our sign-in function, we should either redirect the user to another page, or we should show them an error (like `"Incorrect email/password"`).

6.4. Signup Screen

The Signup page will be very similar to the Login page, only we will be calling `createUserWithEmailAndPassword` instead. You may also want to implement some password strength requirements, or require the user to type their password twice to avoid typos.

6.5. Forgot Password Screen

The Forgot Password page is also very simple and you can copy the `LoginScreen` to get started and make adjustments. In this page, we will be calling `sendPasswordResetEmail`.

Chapter 7. User Auth Navigation

For most user-authenticated applications, you will want to give users access to different pages based on whether they are logged in or not. When a user is not logged in, they are typically brought to a landing page where they have the option to login or create an account. Once they've logged in, they may be brought to a separate page showing the details of their account.

To assist with navigation, I recommend using a navigation library. For this guide, we will be using `react-router-dom`.

In our project, let's create a folder called `/navigation` and within that folder, create two files called `AuthNavigation.tsx` and `MainNavigation.tsx`.

`AuthNavigation.tsx` will handle navigation for users who are not authenticated yet. So it will handle all of the authentication pages, such as logging in and creating an account. `MainNavigation.tsx` will handle navigation for users who have logged in already.

/navigation/AuthNavigation.tsx

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import { LoginScreen, SignupScreen, ForgotPasswordScreen } from './../screens';

function AuthNavigation() {
    return (
        <Router>
            <Routes>
                <Route path="/" element={LoginScreen}></Route>
                <Route path="/login" element={LoginScreen}></Route>
                <Route path="/signup" element={SignupScreen}></Route>
                <Route path="/forgotPassword" element={ForgotPasswordScreen}></Route>
            </Routes>
        </Router>
    );
}

export default AuthNavigation;
```

- We were able to import all of our screens directly from the `/screens` folder by specifying an `index.ts` file that exports them all for us.
- We've created explicit paths for each screen and our default/unspecified path routes to the login page.

/navigation/MainNavigation.tsx

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import { DashboardScreen, SettingsScreen } from './../screens';

function MainNavigation() {
    return (
        <Router>
            <Routes>
                <Route path="/" element={DashboardScreen}></Route>
                <Route path="/dashboard" element={DashboardScreen}></Route>
                <Route path="/settings" element={SettingsScreen}></Route>
            </Routes>
        </Router>
    );
}

export default MainNavigation;
```

- Here, we have a `DashboardScreen` and `SettingsScreen` that we haven't created yet, but are there for demonstration purposes.

7.1. Switch Routers When Authentication State Changes

Now that we have our navigation routers, we need to switch them based on the user's authentication state. Firebase will provide us with an event that fires whenever the user's authentication state changes and we can use that to switch our navigation router.

In our `App.tsx` file, we will update it to look like this:

App.tsx

```
import { useEffect, useState } from 'react';
import firebase from 'firebase/compat/app';
import 'firebase/compat/auth';
import ApiKeys from './constants/ApiKeys';
import { AuthNavigation, MainNavigation } from './navigation';

function App() {
  // Track user authentication...
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  // Initialize Firebase app...
  if (!firebase.apps.length) { firebase.initializeApp(ApiKeys.FirebaseConfig); }

  // Listen for firebase auth changes and update authenticated status...
  useEffect(() => {
      const unsubscribe = firebase.auth().onAuthStateChanged((user) => {
          setIsAuthenticated(!!user);
      });
      return () => unsubscribe();
  }, []);

  // Render navigation based on authentication state...
  return (isAuthenticated)
      ? <MainNavigation />
      : <AuthNavigation />
}

export default App;
```

Now, whenver the user's authentication state changes (if they log in or log out), we re-render and show the corresponding navigator.

7.2. Link To Pages

To link to other pages, we can use `props.history` such as in this link to the Login page:

```
<button onClick={() => {
    this.props.history.push("/login");
}}>Go To Login Page</button><br />
```

Chapter 8. Server: Firebase Functions

Firebase Functions is a framework that allows us to write functionality in TypeScript that we can run on a backend server. We can use this framework to do many things including create an API for our client code to call into, performing some processing whenever data is changed, or to run functions on a regular schedule.

8.1. Getting Started

Our Firebase backend will be a separate project, so all of the code will be stored in the `/server` directory that we created earlier (no longer in the `/website` directory).

Follow the instructions on the Firebase website to install and configure a new project.

You'll first need to install the CLI using `npm install -g firebase-tools`.

Then, navigate to your `/server` directory, use `firebase login` to login, and then initialize the project using `firebase init` and selecting the features that you want.

8.2. Server API

A Firebase Function might look something like this

```
export const helloWorld = functions.https.onRequest((request, response) => {
  functions.logger.info("Hello logs!", {structuredData: true});
  response.set("Access-Control-Allow-Origin", "*");
  response.send({ text: "Hello from Firebase!" });
});
```

We can then call this function from our client code like this

```
async function fnHelloWorld(data: string): Promise<{ data: string }> {
    const fnData = { data: data };
    const helloWorld = firebase.functions().httpsCallable("helloWorld");
    const result: { data: string } = await helloWorld(fnData);
    return result;
}
```

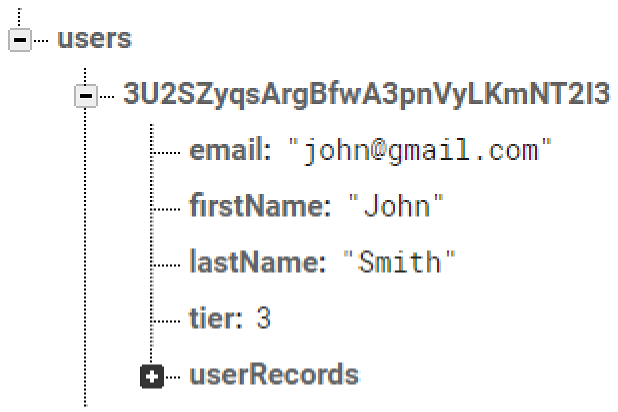
We can pass data into these functions and return data back from them. And just like that, we have a backend API that we can call from our client code!

8.3. Authentication Triggers

When users are created or deleted, you may want to perform certain tasks, such as initializing or clearing out nodes in the database.

For example, when a user is deleted, we'd like to clear out all the information about that user from the database. We can accomplish this using Firebase Authentication Triggers.

Suppose we have a `/users/[uid]` node in our database that stores information about each of our users. When a user's account has been deleted, we want to clear out this entire node.



```
users
    3U2SZyqsArgBfwA3pnVyLKmNT2I3
        email: "john@gmail.com"
        firstName: "John"
        lastName: "Smith"
        tier: 3
        userRecords
```

In your Firebase Functions file, create a function for the `onDelete` event:

```
const admin = require('firebase-admin');

// Clear /users/[uid] node when user is deleted...
export const onUserDelete = functions.auth.user().onDelete((user:
functions.auth.UserRecord) => {
    admin.database().ref("users").child(user.uid).remove();
});
```

If you have user data stored in other locations (for example, in the image above, maybe `userRecords` points to other nodes in the database), then you'll want to update your function to clear out everything.

Note: You can also create a trigger for the creation of a user. This could be useful if you want to initialize the user node with default values. However, I typically do my initialization from the client because when users are signing up to create a new account, I like to ask them for additional information (such as their first and last name) and that information would not be available to an authentication trigger.

8.4. Scheduled Tasks

Sometimes, you may want to run code on a specified schedule.

For example, you may want to perform cleanup tasks each night at midnight. Or, maybe you want to grab data from an external source on a periodic interval. This is sometimes referred to as a `Scheduled Task` or a `Cron Job`.

Suppose we want to update the `myData` node in our database every night at midnight. We also want to store the time at which it was updated, just in case there was a failure.

In your Firebase Functions file, create a function for the `pubsub.schedule.onRun` event:

```
const admin = require('firebase-admin');

// Runs every night at midnight...
export const nightlyRefresh = functions.pubsub.schedule("0 0 * * *").onRun((context:
functions.EventContext) => {
    const lastRefreshedTime = new Date().getTime();
    admin.database().ref("myData").set({ "lastRefreshedTime": lastRefreshedTime });
});
```

Note: The `0 0 * * *` portion is Unix Crontab syntax used to specify what time the function should run. You can read more about this yourself, but in the above example, the first two ``0`s refer to which minute and hour on which to run the function. The next three `*`s specify the `day of month``, `month`, and `day of week` on which to run the function, however because we have a `*`, that means it should run on all possible days.

8.5. CORS

Often, you'll need to get data from external domains. Perhaps you need to scrape some data from a public website. This is not usually possible on the client, but on the server, we can do it using the `cors` library.

First, you'll have to install `cors` and then import it into your functions module:

```
const cors = require("cors")({origin: true});
```

Then wrap your request inside of a `cors` call.

```
export const getCaseMultiStatusInfoChunks = functions.https.onRequest((request,
response) => {
   // Get input data...
   const requestData = request.body.data;
   const name = (requestData) ? requestData.name : "";

   // Get data...
   cors(request, response, () => {
       doSomething(name).then((result: any) => {
           response.send({ data: result });
       }).catch((error: any) => {
           response.send(500);
       });
    });
});
```

8.6. Testing Locally

Sometimes you don't want to deploy your Firebase Functions just to test them. It's time consuming to wait for them to be deployed only to find out that you made an error.

Testing them "locally" (meaning on `localhost`) involves these steps:

1. `cd server`
2. `firebase emulators:start --only functions`
3. On the client, add this code before calling a Firebase Function:
    - `firebase.functions().useEmulator("localhost", 5001);`

After making a change to a Firebase Function,

1. `cd server\functions`
2. `npm run-script build`
3. Refresh the website

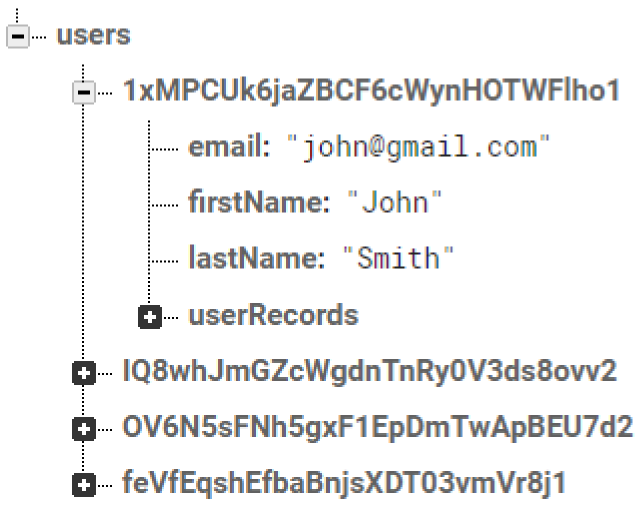
Chapter 9. Database Security

Securing your database code is important to avoid leaking private user data or allowing malicious hackers to destroy your database. Your aim should be to make the data as secure as necessary to do what needs to be done. Users should have access to their own data, but not to the data of other users. Users should be prevented from modifying internal data.

9.1. Database Rules

In Firebase, database security is setup via a `JSON` object that defines the database rules. You configure the rules by setting properties of nodes in your database to `true` or `false` depending on what kind of access they should have (read access, write access, or no access). By default, all nodes will be given no access.

Suppose our database looks like the following.



```
users
    1xMPCUk6jaZBCF6cWynHOTWFlho1
        email: "john@gmail.com"
        firstName: "John"
        lastName: "Smith"
        userRecords
    IQ8whJmGZcWgdnTnRy0V3ds8ovv2
    OV6N5sFNh5gxF1EpDmTwApBEU7d2
    feVfEqshEfbaBnjsXDT03vmVr8j1
```

We have a `users` node where each child is the `UID` of the user. Within that node, we have all of the user's data (name, email, user-generated content). We want to make sure that each user has access to read and modify their own node, but not the nodes of other users.

We can accomplish this by setting up our database rules like so:

Database Rules

```
{
    "rules": {
        "users": {
            "$uid": {
                ".read": "$uid === auth.uid",
                ".write": "$uid === auth.uid"
            }
        }
    }
}
```

This sets the `.read` and `.write` access to true only for the `users` node corresponding to the authenticated user’s UID (`auth.uid`).

Chapter 10. Other Services

10.1. Email

Firebase provides an extension (Trigger Email) that makes it really easy to send emails from a Firebase Function.

The way it works is that you add a document containing the email contents to a collection in the Firestore and then Firebase will trigger a 3rd-Party email provider to send our your email.

You can use this feature to send users emails when specific events occur, or you can combine it with the `pubsub` scheduler to send emails on a periodic schedule.

In this article, we will use SendGrid as our 3rd-Party email provider.

10.1.1. Setup

1. Create an account with SendGrid that you'll use to send emails from Firebase
2. Create an API key
   - Email API > Integration Guide > SMTP Relay
   - Make a note of the values SendGrid provides you

10.1.2. Installation

1. Install the Firebase Trigger Email extension from the Firebase Console
2. Fill in the values required for the extension.
   - For `SMTP connection URI`, enter: `smtps://apiKey:{your-sendgrid-api-key}@smtp.sendgrid.net:465`
   - Set your `Email documents collection` to something like `mail_collection` (This is the collection you will add documents to in order to trigger an email)

10.1.3. Send Emails (Manually)

You can test out this extension by manually writing a document to your `mail_collection`. The document must be formatted a specific way for Firebase to send an email.

```
to: 'someone@example.com',
message: {
  subject: 'Hello from Firebase!',
  html: 'This is an <code>HTML</code> email body.',
}
```

10.1.4. Send Emails

You can use the Firestore API to write documents to this collection and trigger emails to be sent.

10.1.5. Templates

The Trigger Email extension also supports templated emails. The extension documentation can provide more details on this.

10.2. Payments (with Stripe)

Stripe is a payment provider that makes it really easy to integrate payments into your application. A quick summary will be provided here, but you should consult the most up-to-date documentation on the Stripe website.

10.2.1. Setup

First, create your Stripe account. This should give you some API keys that we will use when building out our functionality.

> ⚠ One of your API keys is for the client and one is for the server. Make sure you do not expose your private server API key anywhere on the client.

Firebase provides an extension for working with Stripe, so we will use that. Please note that this is not the conventional way to work with Stripe in a React application. This is a convenient feature that we happen to have access to since we are using Firebase.

Follow the instructions for installing the Firebase extension: https://firebase.google.com/products/extensions/stripe-firestore-stripe-payments

10.2.2. Client

For the client,

10.2.3. Server

10.2.4. Recurring Payments

Chapter 11. Deployment

11.1. Build the Website

The first step to deploying your website is to build it. The build process will minify all of your code and prepare it to be placed on a web server for hosting.

Building is as simple as running a build script: `yarn build`.

11.2. Host the Website

Once you have built the website, the files will be generated in a `/build` directory. All you need to do is copy them into your web server and you're good to go!

INFO: Firebase does provide a hosting service, so you can also host your website directly from Firebase.

Chapter 12. Miscellaneous

Part 2

Sample App: Gas Mileage Tracker

We will now use the guidelines in Part 1 to create an application using React, Firebase, and TypeScript.

The app will track a user’s gas mileage. Users can create an account and each time they fill up their gas tank, they can enter how much gas they put in as well as their current mileage and the app will calculate their average gas mileage.

The app can also be extensible and perhaps allow the user to run reports and get gas mileage over a specified time period. Or maybe we can add to it and allow users to track how much money they are spending on gas.

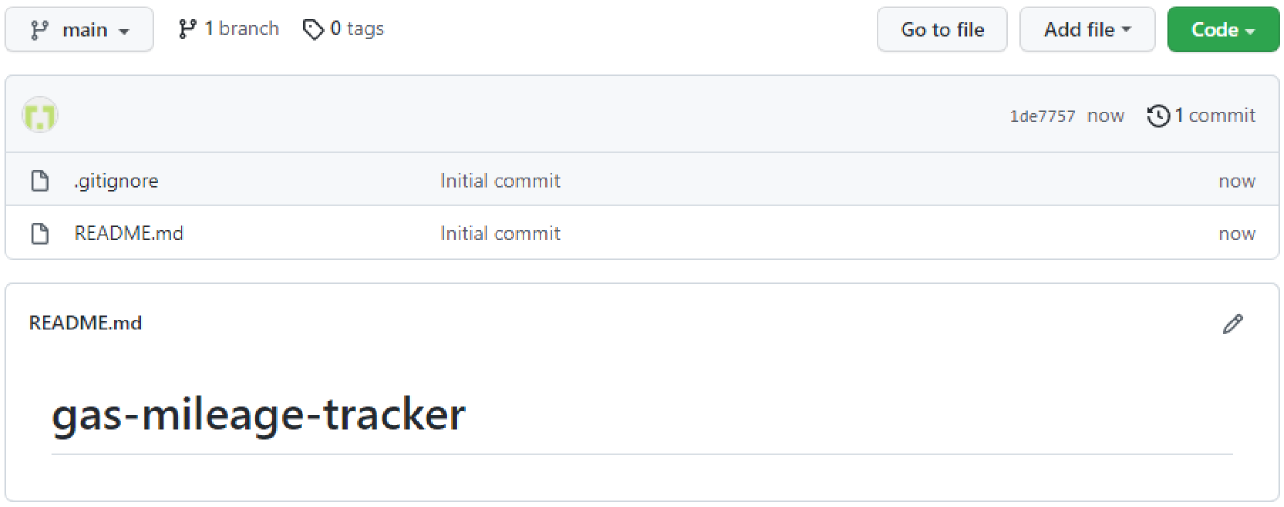
An app like this isn’t entirely practical. For one thing, this could be accomplished pretty easily with a basic spreadsheet; no programming required. And newer cars these days track the gas mileage themselves, making an app like this obsolete.

However, it was the best idea I could come up with that demonstrates all of the elements defined in this guide and I believe it is still a good exercise to build it out.

Chapter 1. Getting Started

1.1. Source Control

The first thing we're going to do is go to Github and create a new repository for the project. This is quick and easy to do and we're going to do it now so that we will always have it ready whenever we're ready to save our code. We'll use the default `README.md` to take notes of things as we're doing development.



.gitignore	Initial commit	now
README.md	Initial commit	now

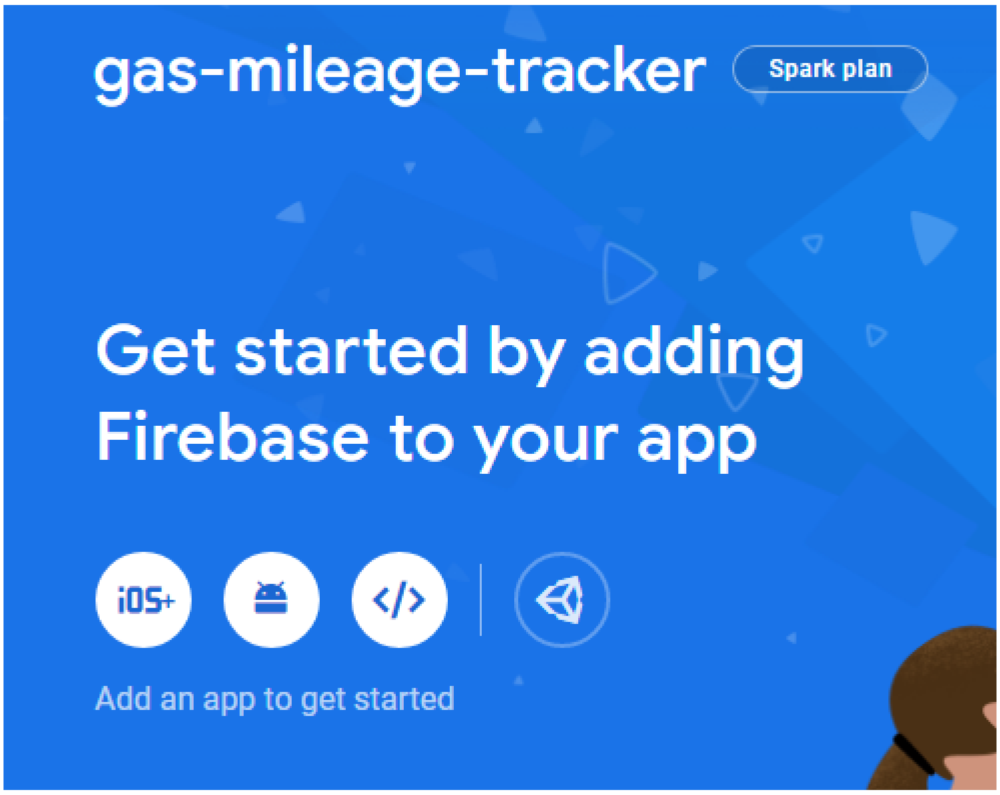
README.md

gas-mileage-tracker

Check out this branch locally on your computer by running `git clone <url>`.

1.2. Firebase Project Setup

Now head over to the Firebase website and create a new project.


gas-mileage-tracker
Spark plan
Get started by adding Firebase to your app
iOS+
Add an app to get started

Click the "Web" button and follow the steps and you should be given a JSON object with your Firebase configuration. Make a note of that, as we will be using it soon.

Chapter 2. Setup

2.1. Project Planning

Let’s take a moment to think about what our application requirements are. These don’t need to be final, but they will give us a concrete starting point to work towards on our first iteration. We’ve already got an idea in our heads, but let’s transcribe it to bullet points now.

Our app must…

- Allow users to login to keep their data secure and personalized
- Allow users to enter data about each time they fill their gas tank
- Calculate a user’s average gas mileage

2.2. Setup and Installation

2.2.1. Create React App

We can now create the default React app. Open a command line terminal inside the folder where you checked out the Git repository and run the following command.

`yarn create react-app website --template typescript`

We are calling the project `website` because we are already inside the `gas-mileage-tracker` directory and the React app will be our "website" view. (We will, in the future, add a "server" folder to hold our backend code).

Go ahead and install Firebase and `react-router-dom` as well, since we will be using those too.

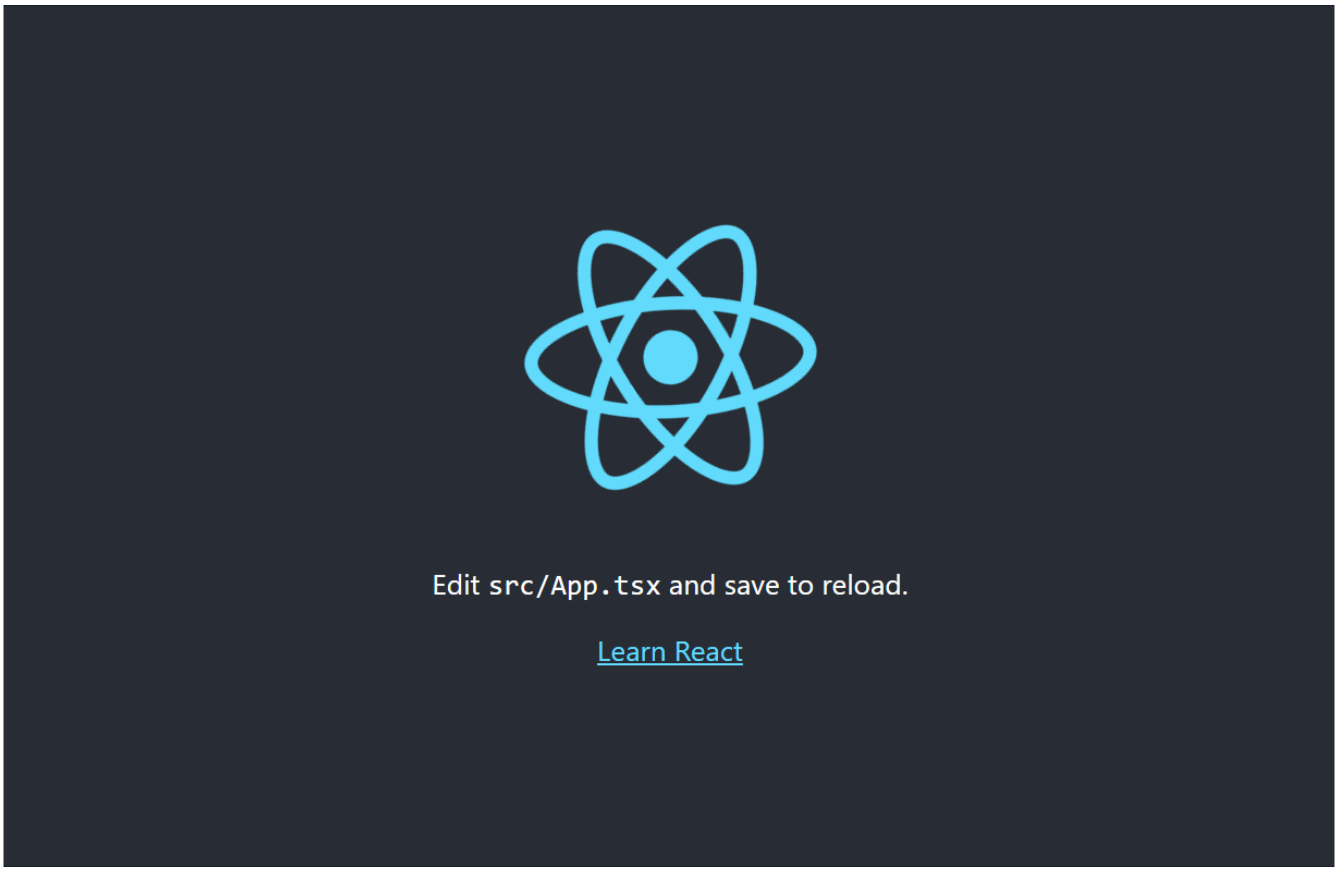
```
cd website
yarn add firebase
yarn add react-router-dom
```

2.2.2. Test Your App!

We haven’t written any code yet, but let’s make sure that everything is working properly. If it doesn’t work, we’ll need to do some investigation into why, but in that case, we can be confident it’s not because of any code that *we* wrote.

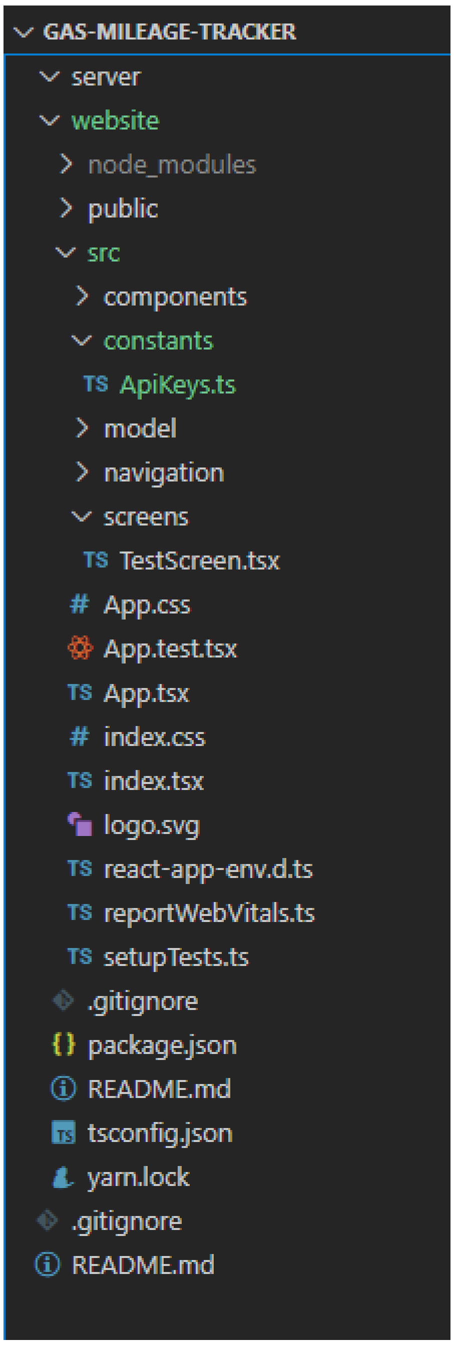
`yarn start`

If everything is working, you should see the default application running in your browser.



2.3. Project Structure

Go into your project folder and setup the folders as demonstrated in the Project Structure chapter in the guide.



```
GAS-MILEAGE-TRACKER
  server
  website
    node_modules
    public
    src
      components
      constants
        ApiKeys.ts
      model
      navigation
      screens
        TestScreen.tsx
      App.css
      App.test.tsx
      App.tsx
      index.css
      index.tsx
      logo.svg
      react-app-env.d.ts
      reportWebVitals.ts
      setupTests.ts
    .gitignore
    package.json
    README.md
    tsconfig.json
    yarn.lock
  .gitignore
  README.md
```

You'll see there are a few additional files that were added by the `create react-app` script that we ran. You can ignore these for now.

Paste the JSON object from the Firebase website into the `ApiKeys.js` file. Your file should be formatted like the following, but the data should be set to the values from your Firebase project.

/src/constants/ApiKeys.ts

```
const ApiKeys = {
    FirebaseConfig: {
        apiKey:             "",
        authDomain:         "",
        databaseURL:        "",
        projectId:          "",
        storageBucket:      "",
        messagingSenderId:  ""
    }
}

export default ApiKeys;
```

2.4. Test Screen

Create a `TestScreen.tsx` file that we will use as a sort of scratch pad to test things out during development.

/src/screens/TestScreen.tsx

```
import React from "react";

function TestScreen() {
    return (
        <div>
            TestScreen
        </div>
    );
}

export default TestScreen;
```

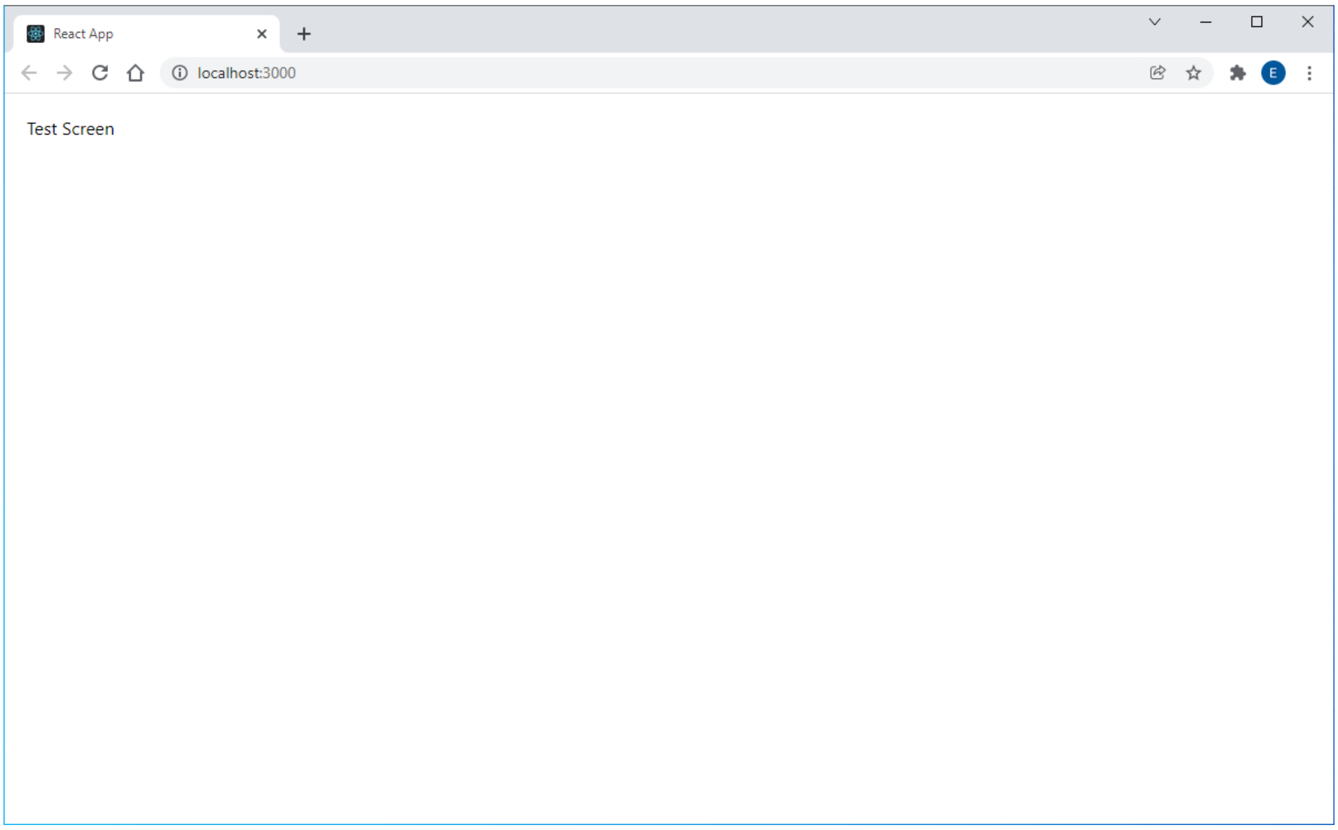
Even though `TestScreen.tsx` is meant to be a screen, we can treat it like a component. Let's display it in our `App.tsx` page so that it shows up as soon as we launch our app. This will make development a little faster because we're going to add to that Test Screen as we build components and see how they look.

To add `TestScreen`, first `import` it and then display it like you would any component. Let's also delete all the other stuff in `App.tsx` since we won't need it. While we're at it, let's go ahead and delete all the styles out of `App.css` as well.

App.tsx

```
import React from 'react';
import './App.css';
import TestScreen from './screens/TestScreen';

function App() {
  return (
    <div>
      <TestScreen />
    </div>
  );
}

export default App;
```

Save and your application should automatically refresh and if your app looks like this, then it worked!



Test Screen

2.5. Commit Your Changes

Now that you have a basic application setup with a Test Screen that you can use to try things out, it seems like a good idea to commit your code to Github to take a "snapshot" that you can always revert back to if needed.

Make sure that your `.gitignore` is ignoring the `ApiKeys.ts` file that stores your API keys. In general, you never want to commit API keys to a repository since (most of the time) they should be kept secret.

```
git add *
git commit -m "Initial setup"
git push
```

Chapter 3. Data Model

Since we are building a gas mileage tracking app, we know we're going to need some data related to calculating gas mileage. Let's add a `/model/models.ts` file to store our data types.

During our project planning, we decided that we expect our app to be used whenever a user fills up their gas tank. At this time, some of the data the user might have access to is:

- Current Date
- Current Mileage on Vehicle
- Amount of Gas (in gallons) Purchased
- How Much User Paid for Gas

Although the last one (how much user paid) isn't relevant to calculating gas mileage, let's keep track of it anyway because maybe we'll want to use it at some point. Now we can model this data in our `/model/models.ts` file.

/model/models.ts

```
export interface IGasFillup {
    date: Date;
    currentMileage: number;
    gallonsOfGas: number;
    cost: number;
}
```

We can also add an interface for the user data we want to collect. For now, let's just stick with a name and email.

/model/models.ts

```
export interface IUser {
    firstName: string;
    email: string;
}

export interface IGasFillup {
    date: Date;
    currentMileage: number;
    gallonsOfGas: number;
    cost: number;
}
```

Chapter 4. Components

We've got our basic data structure so let's create some components so that we have something to look at and a way to interact with our data. Although the styling of our components may change later, or the data structure itself may change, it's ok to still build out these components because they won't take very long and they will most likely give us a good starting point that we can adjust in the future. Even if we have to throw them away altogether, it will still help us to build them because it may give us a better understanding of the needs of our project.

To start, we will create two components: one for viewing the data from an `IGasFillup` instance and one for editing the data from an `IGasFillup` instance.

Let's first start by creating a concrete class `GasFillup` in our `/model/models.ts` file and have it implement the `IGasFillup` interface. I'll make the constructor take in every property as a parameter, because I don't want to be able to create an instance of `GasFillup` without supplying all of the data. (A `GasFillup` instance wouldn't be very useful if it didn't include how many gallons of gas were used, for example) I also don't want to have to worry about some of the properties being `undefined`, so I'll just require them to all be set any time a `GasFillup` object is constructed.

/model/models.ts

```
export interface IUser {
    firstName: string;
    email: string;
}

export interface IGasFillup {
    date: Date;
    currentMileage: number;
    gallonsOfGas: number;
    cost: number;
}

export class GasFillup implements IGasFillup {
    public date: Date;
    public currentMileage: number;
    public gallonsOfGas: number;
    public cost: number;

    constructor(_date: Date, _currentMileage: number, _gallonsOfGas: number, _cost:
number) {
        this.date = _date;
        this.currentMileage = _currentMileage;
        this.gallonsOfGas = _gallonsOfGas;
        this.cost = _cost;
    }
}
```

4.1. View Component

We will start by building a view component for our `GasFillup` data, which is the simpler of the two. In our `/components` directory, let's create a `GasFillupView.tsx` file.

/components/GasFillupView.tsx

```
import React from 'react';
import { IGasFillup } from './../model/models';

interface Props { gasFillup: IGasFillup };
const GasFillupView: React.FC<Props> = (props) => {
    return (
        <div className="GasFillupView-container">
            <strong>{props.gasFillup.date.toLocaleDateString()}</strong>
            <p>Mileage: {props.gasFillup.currentMileage.toLocaleString()}</p>
            <p>Gas (gallons): {props.gasFillup.gallonsOfGas.toFixed(2)}</p>
            <p>Cost: ${props.gasFillup.cost.toFixed(2)}</p>
        </div>
    );
};

export default GasFillupView;
```

We add the `"GasFIllupView-container"` class so we can add styles to our component, and in `App.css`, let's add some basic styling.

App.css

```
.GasFillupView-container {
  padding: 10px;
  border: 1px solid black;
}

.GasFillupView-container p {
  margin: 0;
}
```

We can try out our component by displaying it in our `TestScreen.tsx`. There are a few things we'll need to do to get this working.

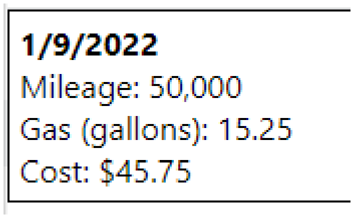
1. Import `useState` so we can use our `GasFillup` instance as a React state
2. Import the `GasFillup` class
3. Import the `GasFillupView.tsx` component
4. Initialize an instance of `GasFillup` and use that as our initial state
5. Render the component

`TestScreen.tsx` should now look something like this

/screens/TestScreen.tsx

```
import React, { useState } from "react";
import { GasFillup } from "./../model/models";
import GasFillupView from "./../components/GasFillupView";

function TestScreen() {
    const testGasFillup = new GasFillup(new Date(), 50_000, 15.25, 45.75);
    const [gasFillup, setGasFillup] = useState<GasFillup>(testGasFillup);

    return (
        <div style={{padding: "20px" }}>
            <GasFillupView gasFillup={gasFillup} />
        </div>
    );
}

export default TestScreen;
```

And when we look at the output in our browser, it should look like this:



1/9/2022
Mileage: 50,000
Gas (gallons): 15.25
Cost: $45.75

It's not the most beautiful component in the world, but it gets the job done (displays the data in our object), so we can use it and clean it up later. Since we've already created a component for it, it will be really easy for us to find the component (since it's in our `/components` folder) and update the styling for it, which should apply across our entire application as long as we use this component any time we want to display a `GasFillup` object.

Remember: We are aiming for a balance between quick iteration and organized structure.

4.2. Edit Component

Now we'll build a component that will allow us to edit a `GasFillup` object. This will be slightly more complex than creating `GasFillupView.tsx`, but we will follow the same procedure.

Let's create the `GasFillupEdit.tsx` component inside our `/components` directory. To make your development even faster, you could just copy `GasFillupView.tsx` and rename everything to `GasFillupEdit`.

/components/GasFillupEdit.tsx

```
import React from 'react';
import { IGasFillup } from './../model/models';
```

```
interface Props {
    gasFillup: IGasFillup;
    onDateChange(date: Date): void;
    onCurrentMileageChange(currentMileage: number): void;
    onGallonsOfGasChange(gallonsOfGas: number): void;
    onCostChange(cost: number): void;
};

// Format a date in a way that an <input type="date"> can handle...
function formatDate(date: Date) {
    const sYear = date.getFullYear().toString();

    let month = date.getMonth() + 1;
    const sMonth = (month < 10) ? "0" + month : month.toString();

    let day = date.getDate();
    const sDay = (day < 10) ? "0" + day : day.toString();

    return `${sYear}-${sMonth}-${sDay}`;
}

const GasFillupEdit: React.FC<Props> = (props) => {
    return (
        <div className="GasFillupEdit-container">
            <div>
                <label>Date: </label>
                <input type="date"
                    value={formatDate(props.gasFillup.date)}
                    onChange={(event) => {
                        const val = (event.currentTarget as HTMLInputElement).value;
                        props.onDateChange(new Date(val));
                    }} />
            </div>

            <div>
                <label>Mileage: </label>
                <input type="number" min="0" step="1"
                    value={props.gasFillup.currentMileage}
                    onChange={(event) => {
                        const val = (event.currentTarget as HTMLInputElement).value;
                        props.onCurrentMileageChange(parseInt(val));
                    }} />
            </div>

            <div>
                <label>Gas (gallons): </label>
                <input type="number" min="0"
                    value={props.gasFillup.gallonsOfGas}
                    onChange={(event) => {
                        const val = (event.currentTarget as HTMLInputElement).value;
                        props.onGallonsOfGasChange(parseFloat(val));
                    }} />
            </div>

            <div>
                <label>Cost: </label>
                <input type="number" min="0"
                    value={props.gasFillup.cost}
                    onChange={(event) => {
                        const val = (event.currentTarget as HTMLInputElement).value;
                        props.onCostChange(parseFloat(val));
                    }} />
            </div>
        </div>
    );
};

export default GasFillupEdit;
```

This is quite a bit more code! But if you read through it, you'll see that much of it is just boilerplate code for listening to `onChange` events on `<input>` elements.

A few notes on this component:

- Our `Props` interface now includes events for when all of the data elements change. This will notify the user of this component whenever a value is changed.
- We created the `formatDate(date: Date)` function because the `<input>` expects the date to be formatted a specific way.
- There is potentially a slight bug with how the `Date` gets calculated that could result in the date being off by one day depending on the timezone. You may have to make adjustments or consider using a date selector from a library.

Let's add some styles to our `App.css` file to make the component look a little bit better.

App.css

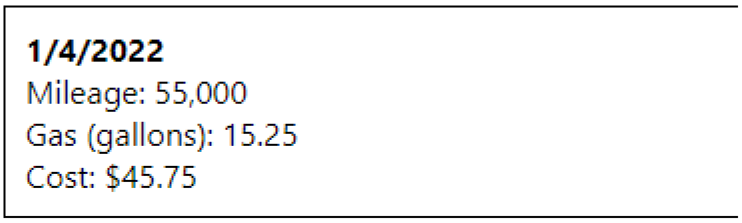
```
.GasFillupEdit-container {
  padding: 10px;
  border: 1px solid black;
}

.GasFillupEdit-container label {
  display: inline-block;
  width: 100px;
}
```

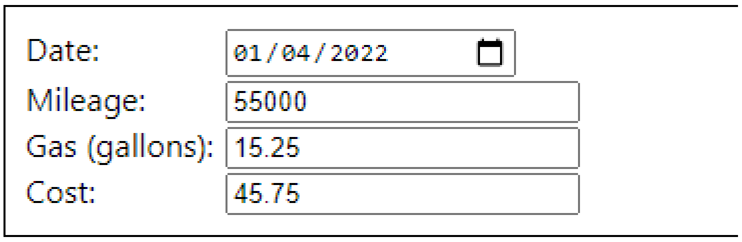
Finally, we'll display it in our `TestScreen` by importing it and then rendering it. For the events, we will just update our state object according to which propery was changed.

/screens/TestScreen.tsx

```
import React, { useState } from "react";
import { GasFillup } from "./../model/models";
import GasFillupView from "./../components/GasFillupView";
import GasFillupEdit from "./../components/GasFillupEdit";

function TestScreen() {
    const testGasFillup = new GasFillup(new Date(), 50_000, 15.25, 45.75);
    const [gasFillup, setGasFillup] = useState<GasFillup>(testGasFillup);

    return (
        <div style={{padding: "20px" }}>
            <GasFillupView gasFillup={gasFillup} />
            <br /><br />

            <GasFillupEdit gasFillup={gasFillup}
                onDateChange={(date: Date) => {
                    setGasFillup({ ...gasFillup, date: date })
                }}
                onCurrentMileageChange={(currentMileage: number) => {
                    setGasFillup({ ...gasFillup, currentMileage: currentMileage })
                }}
                onGallonsOfGasChange={(gallonsOfGas: number) => {
                    setGasFillup({ ...gasFillup, gallonsOfGas: gallonsOfGas })
                }}
                onCostChange={(cost: number) => {
                    setGasFillup({ ...gasFillup, cost: cost })
                }}
            />
        </div>
    );
}

export default TestScreen;
```

And our `TestScreen` should now look similar to the image below. Note that as we make changes to the data in our edit component, the view component is updated to reflect those changes since they are both using the same state instance of `GasFillup`.


1/4/2022
Mileage: 55,000
Gas (gallons): 15.25
Cost: $45.75



Date:	01/04/2022
Mileage:	55000
Gas (gallons):	15.25
Cost:	45.75

Chapter 5. Client API

Now that we've got a way to view and edit our primary data structure, let's start writing a client API to interact with the data. To start, our client API will need to do two things for us:

1. Provide the basic logic for our app (in our case, gas mileage calculations) so that our screens don't need to implement that logic
2. Provide a way for us to interact with the database (save data to the database, read data from the database, etc.)

> I'm going to organize the API modules in the way that I personally prefer to organize them, which some people may find a bit unusual. It is not essential that you organize your code in any specific way. I do recommend using the `/model` directory that we created, but inside of that, please feel free to organize your code in whatever way is most comfortable to you.

5.1. Application Logic API

Within our `/model` directory, let's create a new directory called `/GasMileageTracker` and then inside of that, we will make a file called `Util.ts` that will hold our utility functions. These are the functions that will do some calculations that we determined during our initial planning would be needed by our application.

In our case, we want to provide our current `GasFillup` and our previous `GasFillup` and calculate the gas mileage we got on our most recent tank of gas. We need the previous `GasFillup` so we can determine what the mileage was at that time, and with the current `GasFillup`, we can see how many miles we've driven since then as well as how much gas we put into the tank, which will tell us how much gas we've used since the last fillup (assuming we always fully fill the tank).

If the logic of these functions (in our case, just one function for now) is too complicated, it is acceptable to simply create the stub functions and come back and actually implement them later. This is similar to how we created really bare components and will come back and clean them up later. The point is to setup our infrastructure early so our application is easy to work with as it becomes more complex. The gas mileage calculation isn't particularly complex, so we will go ahead and implement it now so we don't need to jump around to different files in this guide.

/model/GasMileageTracker/Util.ts

```
import { IGasFillup } from "./../models";

/** Computes the gas mileage since the previous fillup */
export function getGasMileage(prevGasFillup: IGasFillup, currGasFillup: IGasFillup):
number {
    const miles = currGasFillup.currentMileage - prevGasFillup.currentMileage;
    const gallons = currGasFillup.gallonsOfGas;
    return miles / gallons;
}
```

You may be wondering why we don't just add this function to the GasFillup class. That is certainly a valid thing to do. However, often when we pull data from a database (such as Firebase), we are just getting the data itself and not instances of an actual class and therefore any functions attached to that class will not be present. You could add static functions to your GasFillup class if that is how you prefer to organize your code.

5.2. Database API

Now we will add some functions that allow us to read and write to the database. We will stick with the basic CRUD functionality, which stands for Create, Read, Update, Delete. Those will all be useful functions that users will likely need to do.

Let's start by adding a directory in /model/GasMileageTracker called /Firebase. Within that directory, let's add two files called Read.ts and Write.ts. I personally like to separate my read and write modules so it's more clear to me when I read my code if I am actually manipulating the database (writing to it), but once again: feel free to organize the code however you'd like.

We will add the stub functionality for now and then go back and update our code later.

/model/GasMileageTracker/Firebase/Read.ts

```
import { IGasFillup } from "./../../models";

export async function getGasFillupsForUser(userId: string): Promise<IGasFillup[]> {
    // TODO: Implement this later
    return [];
}
```

/model/GasMileageTracker/Firebase/Write.ts

```
import { IGasFillup } from "./../../models";

export async function createGasFillup(userId: string, gasFillup: IGasFillup):
Promise<void> {
    // TODO: Implement this later
}

export async function updateGasFillup(userId: string, gasFillupId: string, gasFillup:
IGasFillup): Promise<void> {
    // TODO: Implement this later
}

export async function deleteGasFillup(userId: string, gasFillupId: string):
Promise<void> {
    // TODO: Implement this later
}
```

5.3. API Organization

Let’s also add `index.ts` files in our directories to make importing the functionality from our components and screens more easy.

The way that we’d like to use the functionality from a screen, for example, is something like this:

```
import * as GasMileageTracker from "./../model/GasMileageTracker";

//
// ...
const gasMileage = GasMileageTracker.Util.getGasMileage(...);

//
// ...
await GasMileageTracker.Firebase.Write.createGasFillup(...);
```

We can accomplish this by exporting our modules in the `index.ts` files as shown below. Whenever we add a new file, we’ll have to remember to update the `index.ts` files.

/model/GasMileageTracker/Firebase/index.ts

```
import * as Read from "./Read";
import * as Write from "./Write";
export { Read, Write }
```

/model/GasMileageTracker/index.ts

```
import * as Util from "./Util";
import * as Firebase from "./Firebase";
export { Util, Firebase }
```

Note that `Util` is actually a file and `Firebase` is a directory. That’s ok. The syntax will remain the same. When you specify a directory, the system will search for the `index.ts` file in that directory.

We will see soon in practice how this will make importing and using these modules much easier.

To summarize, you should now have a folder structure that looks something like this:

```
MyApp/
 ├── model/
 │    ├── GasMileageTracker/
 │    │     ├── Firebase/
 │    │     │    ├── Read.ts
 │    │     │    ├── Write.ts
 │    │     │    ├── index.ts
 │    │     ├── Util.ts
 │    │     ├── index.ts
 │    ├── models.ts
```

Chapter 6. User Authentication

6.1. Authentication API

We will add a new file, `Auth.ts` in our `/model/Firebase` directory and we can just copy the code directly from the guide. This API will allow us to perform the basic functionality of signing users in, signing them out, and creating new accounts, among other things. By putting this in a dedicated module, it makes it easier for us to add more functionality on top of those actions. For example, after a user creates an account, we may want to perform additional tasks.

/model/Firebase/Auth.ts

```
import firebase from "firebase/compat/app";
import "firebase/compat/auth";
type UserCredential = firebase.auth.UserCredential;

// Create a new user with the given email/password...
export async function createUserWithEmailAndPassword(email: string,
    password: string): Promise<void> {

    return firebase.auth().createUserWithEmailAndPassword(email, password)
        .then((userData: UserCredential) => {
            const user: firebase.User | null = userData.user;
            if (!user) { return; }

            // Send the email verification...
            return user.sendEmailVerification();
        });
}

// Sign-in user with given email/password...
export async function signInWithEmailAndPassword(email: string, password: string):
Promise<UserCredential> {
    return firebase.auth().signInWithEmailAndPassword(email, password);
}

// Sign-out current user...
export async function signOut(): Promise<void> {
    return firebase.auth().signOut();
}

// Send password reset email to given email...
export async function sendPasswordResetEmail(email: string): Promise<void> {
    return firebase.auth().sendPasswordResetEmail(email);
}

// Change current user's password...
export async function changeCurrentUserPassword(currentPassword: string,
    newPassword: string): Promise<void> {
```

```
    const currentUser = firebase.auth().currentUser;
    if (!currentUser) { return Promise.resolve(); }

    return reauthenticateCurrentUser(currentPassword).then(() => {
        return currentUser.updatePassword(newPassword);
    });
}

// Reauthenticates current user (only works if password-authenticated)...
export async function reauthenticateCurrentUser(password: string):
Promise<UserCredential> {
    const currentUser = firebase.auth().currentUser;
    if (!currentUser) { return Promise.reject("No current user"); }
    if (!currentUser.email) { return Promise.reject("Current user does not have an
email"); }

    const credential = firebase.auth.EmailAuthProvider.credential(currentUser.email,
password);
    return currentUser.reauthenticateWithCredential(credential);
}
```

And don't forget to update the `index.ts` file to export `Auth`.

/model/Firebase/index.ts

```
import * as Auth from "./Auth";
import * as Read from "./Read";
import * as Write from "./Write";
export { Auth, Read, Write }
```

6.2. Login Screen

In our `/screens` directory, let's create an additional directory called `/auth` to store all of our auth screens (`Login`, `Signup`, etc.).

For the login screen, we can just copy the `LoginScreen` from the guide with minimal changes.

/screens/auth/LoginScreen.tsx

```
import { useState } from "react";
import { Auth } from "./../../model/GasMileageTracker/Firebase";

function LoginScreen(props: any) {
    const [isLoading, setIsLoading] = useState<boolean>(false);
    const [email, setEmail] = useState<string>("");
    const [password, setPassword] = useState<string>("");

    // Occurs when "Login" is clicked...
    const onLoginClicked = () => {
        setIsLoading(true);
        Auth.signInWithEmailAndPassword(email, password)
            .then(() => {
                setIsLoading(false);
                // Redirect user
            }, (error) => {
                setIsLoading(false);
                // Show an error message to user
            });
    }

    return (
        <div>
            <h2>Login</h2>

            <input type="email" value={email} placeholder="Email"
                onChange={(event) => { setEmail(event.target.value); }}
            /><br />

            <input type="password" value={password} placeholder="Password"
                onChange={(event) => { setPassword(event.target.value); }}
                onKeyPress={(event) => {
                    if (event.key === "Enter" || event.charCode === 13) {
                        onLoginClicked();
                    }
                }}
            /><br />

            <button onClick={onLoginClicked}>Login</button><br />

            {isLoading ? <span>Loading...</span> : null}
        </div>
    );
}

export default LoginScreen;
```

6.3. Signup Screen

The `SignupScreen` will be very similar to the `LoginScreen`.

/screens/auth/SignupScreen.tsx

```
import { useState } from "react";
import { Auth } from "./../../model/GasMileageTracker/Firebase";

function SignupScreen(props: any) {
    const [isLoading, setIsLoading] = useState<boolean>(false);
    const [email, setEmail] = useState<string>("");
    const [password, setPassword] = useState<string>("");
    const [passwordConfirm, setPasswordConfirm] = useState<string>("");

    // Occurs when "Signup" is clicked...
    const onSignupClicked = () => {
        setIsLoading(true);
        Auth.createUserWithEmailAndPassword(email, password)
            .then(() => {
                setIsLoading(false);
                // Redirect user
            }, (error) => {
                setIsLoading(false);
                // Show an error message to user
            });
    }

    return (
        <div>
            <h2>Signup</h2>

            <input type="email" value={email} placeholder="Email"
                onChange={(event) => { setEmail(event.target.value); }}
            /><br />

            <input type="password" value={password} placeholder="Password"
                onChange={(event) => { setPassword(event.target.value); }}
            /><br />

            <input type="password" value={passwordConfirm} placeholder="Password
(confirm)"
                onChange={(event) => { setPasswordConfirm(event.target.value); }}
                onKeyPress={(event) => {
                    if (event.key === "Enter" || event.charCode === 13) {
                        onSignupClicked();
                    }
                }}
            /><br />

            <button onClick={onSignupClicked}>Signup</button><br />
            {isLoading ? <span>Loading...</span> : null}
        </div>
    );
}

export default SignupScreen;
```

6.4. Forgot Password Screen

The `ForgotPasswordScreen` will be very similar to the `LoginScreen`.

/screens/auth/ForgotPasswordScreen.tsx

```
import { useState } from "react";
import { Auth } from "./../../model/GasMileageTracker/Firebase";

function ForgotPasswordScreen(props: any) {
    const [isLoading, setIsLoading] = useState<boolean>(false);
    const [email, setEmail] = useState<string>("");

    // Occurs when "Reset Password" is clicked...
    const onResetPasswordClicked  = () => {
        setIsLoading(true);
        Auth.sendPasswordResetEmail(email)
            .then(() => {
                setIsLoading(false);
                // Redirect user
            }, (error) => {
                setIsLoading(false);
                // Show an error message to user
            });
    }

    return (
        <div>
            <h2>Forgot Password</h2>

            <input type="email" value={email} placeholder="Email"
                onChange={(event) => { setEmail(event.target.value); }}
                onKeyPress={(event) => {
                    if (event.key === "Enter" || event.charCode === 13) {
                        onResetPasswordClicked();
                    }
                }}
            /><br />

            <button onClick={onResetPasswordClicked}>Reset Password</button><br />

            {isLoading ? <span>Loading...</span> : null}
        </div>
    );
}

export default ForgotPasswordScreen;
```

6.5. Index.ts

For convenience, we can export all of these screens from the `/screens/index.ts` file.

/screens/index.ts

```
export { default as LoginScreen } from './auth/LoginScreen';
export { default as SignupScreen } from './auth/SignupScreen';
export { default as ForgotPasswordScreen } from './auth/ForgotPasswordScreen';

export { default as TestScreen } from './TestScreen';
```

Chapter 7. User Auth Navigation

We've created our authentication screens, but we currently can't access them because we haven't built a way for us to switch to other pages. This is where the `react-router-dom` library that we installed will come in handy. `react-router-dom` will allow us to create different "Routes" that we can navigate to.

Our application will show different pages depending on whether a user is logged in (or "authenticated") or not. So we will create two different navigation routers: one for if the user is logged in, and one for if the user is not logged in.

In the `/navigation` directory, create a file called `AuthNavigation.tsx`. This will be the router we use for when a user is not logged in. The user will be able to go to the `LoginScreen`, `SignupScreen`, and `ForgotPasswordScreen`, but will not be allowed to go to any screens that display a user's data.

/navigation/AuthNavigation.tsx

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import { LoginScreen, SignupScreen, ForgotPasswordScreen } from './../screens';

function AuthNavigation() {
    return (
        <Router>
            <Routes>
                <Route path="/" element={LoginScreen}></Route>
                <Route path="/login" element={LoginScreen}></Route>
                <Route path="/signup" element={SignupScreen}></Route>
                <Route path="/forgotPassword" element={ForgotPasswordScreen}></Route>
            </Routes>
        </Router>
    );
}

export default AuthNavigation;
```

We'll also make a navigation router for users who are logged in. For now, we just have links to our test screen, but you can imaging adding screens for the user's profile, viewing their gas fillups, etc.

/navigation/MainNavigation.tsx

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import { TestScreen } from './../screens';

function MainNavigation() {
    return (
        <Router>
            <Routes>
                <Route path="/" element={TestScreen}></Route>
                <Route path="/test" element={TestScreen}></Route>
            </Routes>
        </Router>
    );
}

export default MainNavigation;
```

Don’t forget to export them from the `index.ts` file.

/navigation/index.ts

```
export { default as AuthNavigation } from "./AuthNavigation";
export { default as MainNavigation } from "./MainNavigation";
```

7.1. Switch Routers When Authentication State Changes

Back in `App.tsx`, we will listen for authentication state changes and switch our router accordingly. The `onAuthStateChanged` event will fire when a user creates an account, signs in, signs out, etc.

App.tsx

```
import { useEffect, useState } from 'react';
import firebase from 'firebase/compat/app';
import 'firebase/compat/auth';
import ApiKeys from './constants/ApiKeys';
import { AuthNavigation, MainNavigation } from './navigation';

function App() {
  // Track user authentication...
  const [isAuthenticated, setIsAuthenticated] = useState(false);

  // Initialize Firebase app...
  if (!firebase.apps.length) { firebase.initializeApp(ApiKeys.FirebaseConfig); }

  // Listen for firebase auth changes and update authenticated status...
  useEffect(() => {
      const unsubscribe = firebase.auth().onAuthStateChanged((user) => {
          setIsAuthenticated(!!user);
      });
      return () => unsubscribe();
  }, []);

  // Render navigation based on authentication state...
  return (isAuthenticated)
      ? <MainNavigation />
      : <AuthNavigation />
}
```

7.2. Link To Pages

Now that everything is setup, we just need to add links to the various pages so we can navigate to them. A menu bar would be great, but until then, we can just add simple links in our `TestScreen`. We can link to a page like this.

```
<button onClick={() => {
    this.props.history.push("/login");
}}>Go To Login Page</button><br />
```

Conclusion