# Complete React Developer in 2022 (w/ Redux, Hooks, GraphQL)

## Section 2: React Key Topics

* Don’t toch the DOM. I’ll do it.
* Build websites like lego blocks.
* Unidirectional data flow.
* UI, The rest is up to you.

## Section 4: Capstone Project: Intro + Setup

### Scaffolding Our Capstone Project

The following command will create a new react project

**npx create-react-app crw-clothing**

This will create a basic react project.

### Setting Up Our Categories

We add an array of categories that is mapped over to display all categories. The map() method used a lot for this.



### Adding Sass

Adding sass to a create-react-app is as simple as installing the sass package using the following command:

**yarn add sass**

A create-react-app is setup in a way that it will automatically utilize the sass syntax.

### Adding Fonts

Fonts can be added using the Google Fonts website.

## Section 5: Routing + React-Router

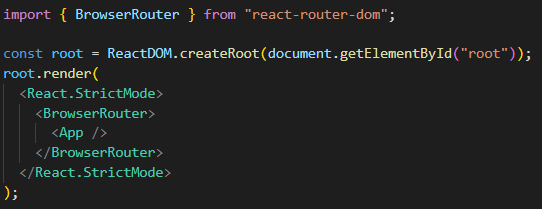
The library used for implementing routing is the react-router package. We can add this library using the following command:

**yarn add react-router-dom@6**

### React Router Outlet

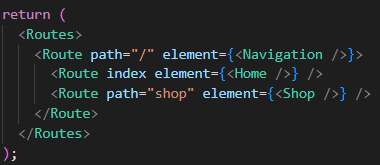
The following example shows the implementation of the router. This requires a few steps.

1. Wrap the App component with the BrowserRouter component provided by the react-router.



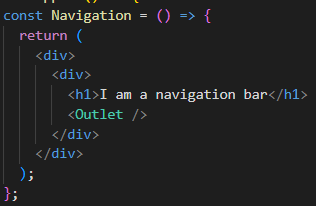
1. Create the navigation using the Routes component and each route using the Route component.





The first root route nests 2 routes. By setting the index property on the Home route, we basically tell the router to use this component as the default component to load along side the Navigation component.

The following code shows the Navigation component which utilizes the Outlet component.

By placing the Outlet component under the h1, we load the selected component (route) under there.

### React Router Link

In order to actually navigate to certain pages, we can use the Link component provided by react-router.

The Link component will render an anchor tag. It requires a to property which tells the anchor where to navigate to.

### Styling for Navigation + Logo

For the logo an svg file is used. We can add an svg file as follows.



Import the logo as a component and then also use it as a component.



Keep in mind that this only works because of the create-react-app scaffolding. This will provide the functionality needed to transform the svg to a react component.

## Section 6: Authentication + Firebase

### Setting Up Firebase

We will be using the firebase platform to add a backend to our project. Firebase allows us to create a backend projects and provides tools that we can use to further develop our application (authentication, database, hosting, storage, …).

In order for us to use this backend, we need to create a project on the platform and add the firebase library to our application.

**yarn add firebase**

### Authentication flow

We are going to be implementing an authentication using google sign in. Essentially if we have a gmail account, google already knows who we are and they can vouch that we actually are who we say we are.

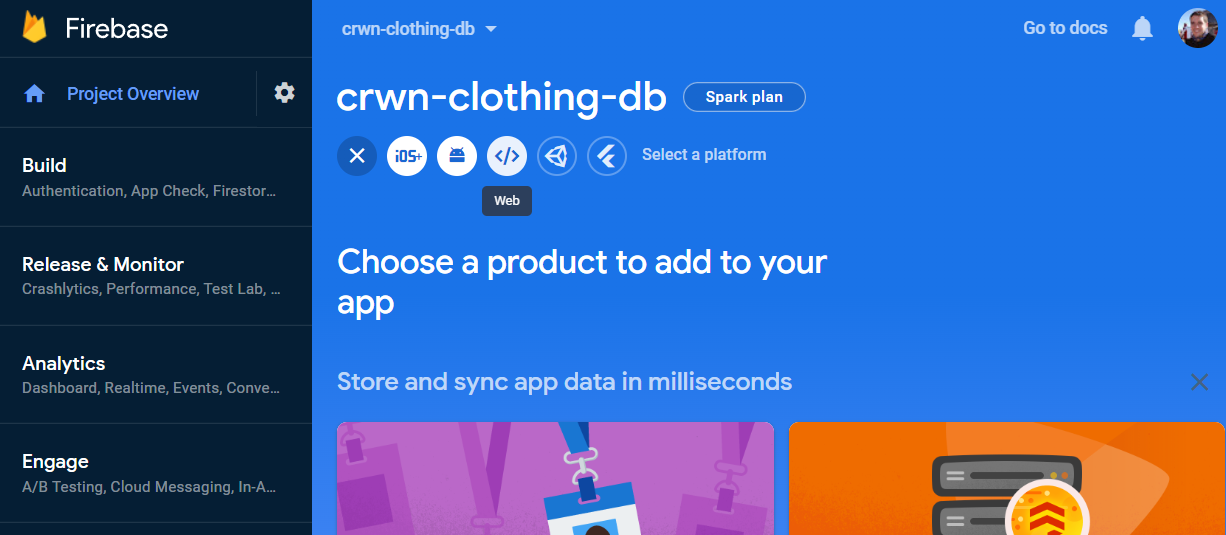
The authentication flow goes as follows:

1. We send an authentication request using our gmail credentials to the google servers.
2. Google will verify (or deny) the credentials and send back an auth\_token.
3. This auth\_token is then send from our application to the firebase backend.
4. Firebase will send this auth\_token to the google servers asking if this token is valid.
5. If a valid token is sent, the google servers will send back a verification\_token.
6. Firebase will then create an access\_token specifically for the user logging in with the correct authorizations.
7. Firebase will send this access\_token back to our application.
8. We need to store this access\_token and sent it along any request we sent to the firebase backend.

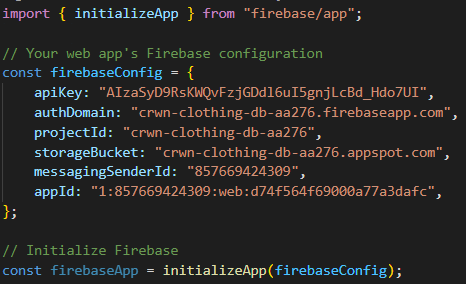
### Authentication With Firebase

Using authentication with firebase is pretty abstract. We need to first configure the way we provide authentication. Let’s go over some of the steps required to get things working.

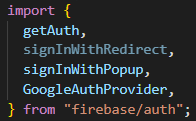
Before we can actually connect to our backend, we need to register our application on the backend. We can do this by pressing the following button.

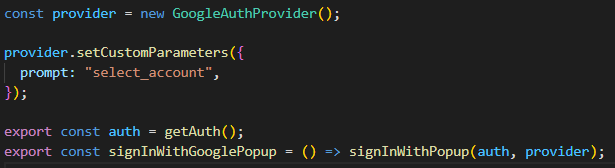


By pressing the </> button, we can register a web application. Whenever we registered an application, firebase will provide us with the necessary configuration to connect our application to firebase.

This code will basically abstract away all code that is required to correctly communicate with the firebase backend.

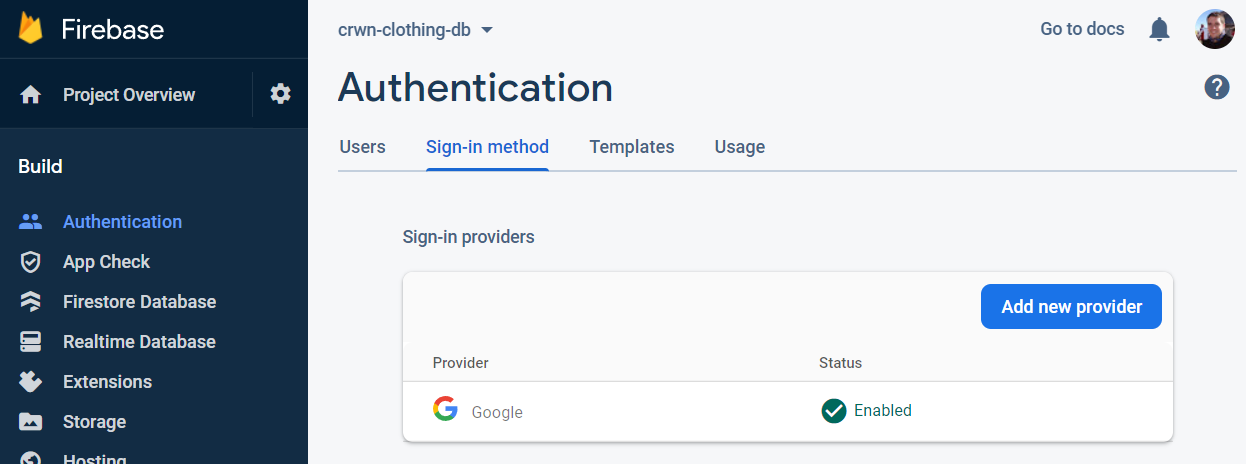
Next, we need to tell our application how to authenticate to firebase (in this case using the Google Sign In method).



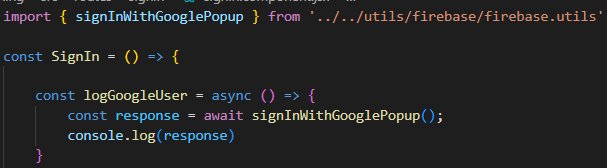


For starters, we will use a popup that will trigger the authentication flow explained previously.

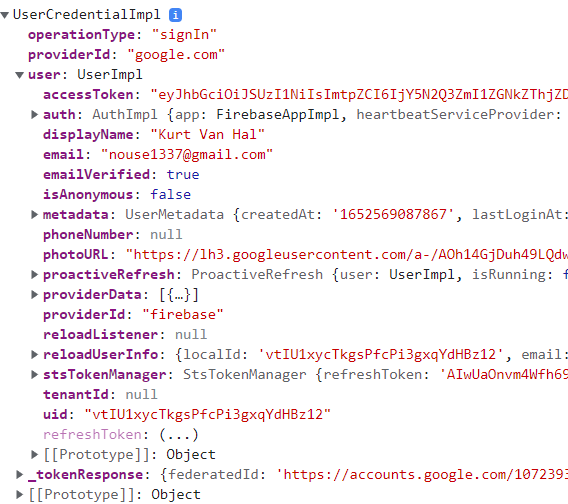
**Important!** We need to enable the Google sign-in method first.



The only step we need next is to call the exported signInWithGooglePopup function we created in our sign in page.



This will open up a popup that will automatically handle the authentication flow using our own google account. The response provided is shown below (if authentication was successful).

As shown in the authentication flow, the accessToken is the part that we need to store in order for our application to be able to retrieve data from our backend.

Obviously depending on the authorization granted to the user, will we be able to perform certain actions.

### Introducing Firestore Data Model

The firestore data model can be compared to a mongodb database. We have collections containing documents with data in them. The collections can be viewed as tables, the documents as a unique identifier and the data as the data (row) for that document.

### Setting Up User Documents

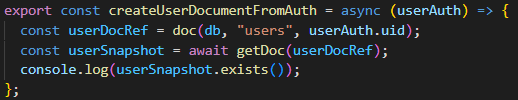
Firestore utilizes a special way of handling reading or writing data to our database. First we need to initialize the firestore database. The code below shows how this is handled.





This will return an object that is used to interact with the firestore database.

The next code snippet is shown to provide some additional information to how we can read/write to our database.



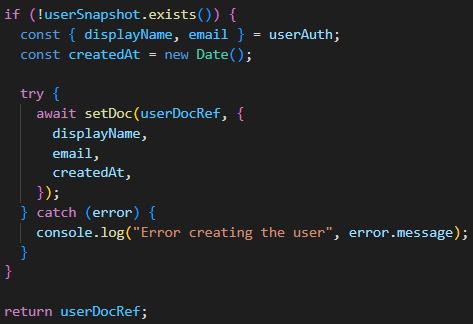
We are going to be storing a newly logged in user (using google signin) into the firestore db. The way firestore works is as follows.

1. We first get a reference to a specific document. If this document does not exist, firestore will create it as a placeholder.
2. By using the getDoc method we retrieve a snapshot of that particular document.
3. This snapshot provides us with additional methods such as the exists() method to see if this document actually exists.

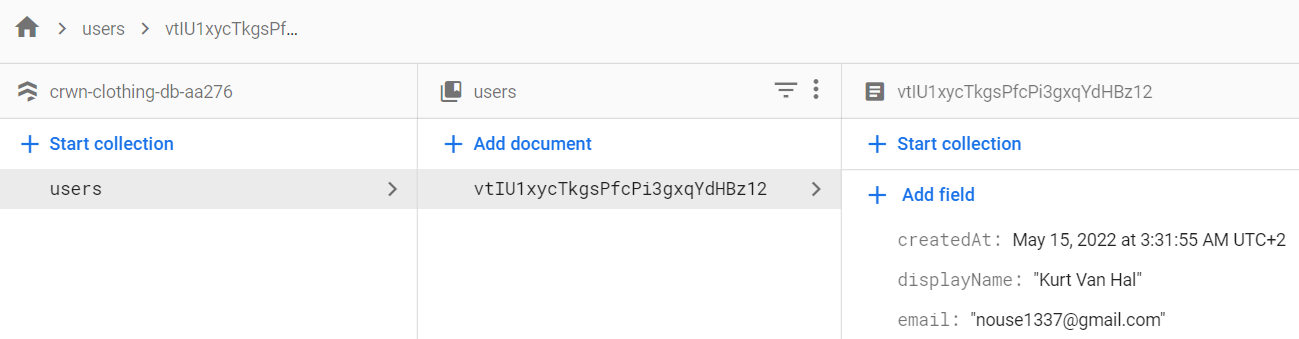
Next we will see how this works when we want to store users.

### Finish Creating User Documents

In order for us to save a new signed in user in our application firestore database, we need to see if it exists. If it does not exist, the setDoc() method can be used to save data to our database.



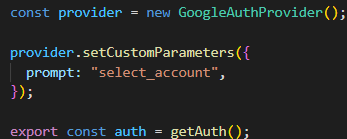
It requires the userDocRef reference and the object that needs to be stored. The database will look like this



### Sign In With Redirect

As an example we are going to implement an additional signin method, still using the google signin but this time utilizing the redirect method.

The steps required are as follows:

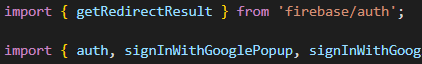
1. As with the popup, we need to import the signInWithRedirect method from the firebase/auth library.
2. This method requires an auth singleton and a provider, same as the signInWithPopup  
   



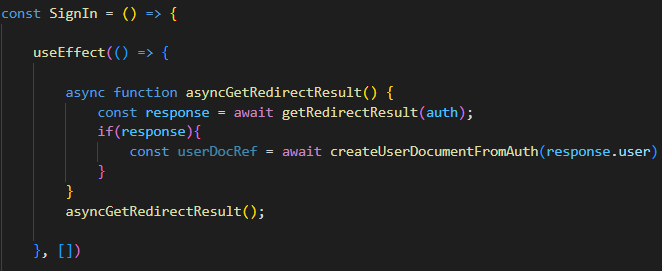
1. There is a huge caveat when using this functionality. Since we are redirected to the google authentication page, we basically loose track of our application state, since when returning from this page, we reinitialize our application and loose track of where we were in our code.

Retrieving the response requires some additional steps.

Firebase provides a solution for this. First we import the following:



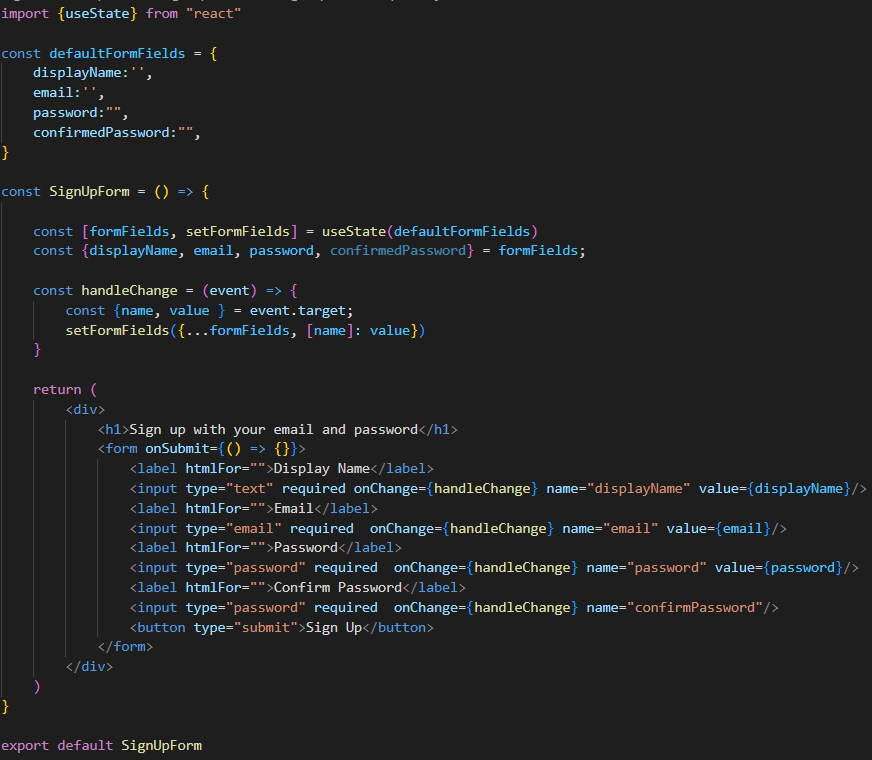
Important are the getRedirectResult and the auth singleton. Think of the auth singleton as an object that keeps track of our entire authentication state for our firebase backend and application.

We use the useEffect method to try and retrieve the redirect results. 

Here we use the getRedirectResult function and pass it the auth singleton to retrieve potential results of a signin. If we have results, we again create a new document from the new user, if he/she doesn’t already exist.

### Sign Up Form Pt.1

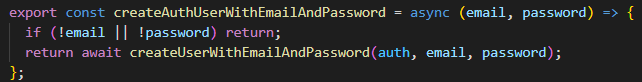
We will be leveraging the useState hook to handle our form data. The code below shows the form field for signup. Currently the state is linked to the form, additional logic needs to be added to actually handle signing up.



A generic function is used to update any of the formfields. By using the name property on our input field, we can update our state appropriately.

### Sign Up Form Pt. 2

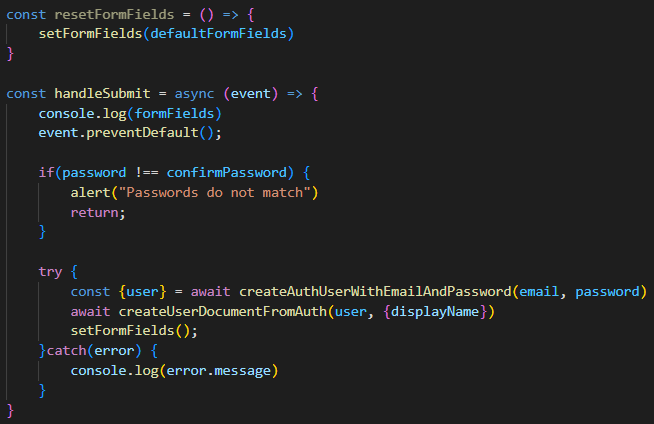
Next, we are going to use the createUserWithEmailAndPassword method provided by google firebase.



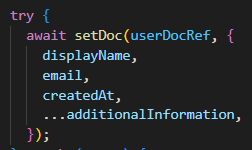
What this will do is authenticate our user to the firebase authentication service. We still need to save the returned created user in our firestore database.

### Sign Up With Email + Password

In our signup component, we need to handle the submission of our signup. The code below shows how this is handled.



Keep in mind that the creation of a new user in the authentication service will not return a displayName. It is our responsibility to provide this displayName to the createUserDocumentFromAuth. We do this by expanding the function to also receive an object that will override the displayName when present.



### Generalizing Form Input Component

Since the form input field will be used multiple times, a separate component is created for it.

The input should receive a label property and will spread all other properties provided to the component.

Calling this component is shown below. We ca also pass an object containing the props and spread these out in the component.



### Custom Button Component

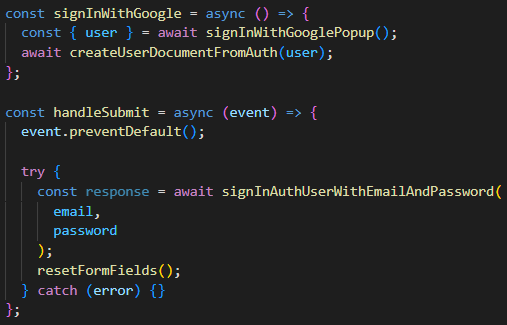
A custom button is added that can receive 3 different styles, a default, an inverted and a google sign in style. The component is shown below as well as the way we would call it.

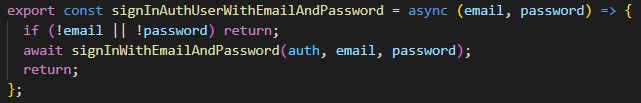




### Sign In Form

A new form is created allowing the user to authenticate using the email/password or google sign in method. Basically a copy is made from the signup form and edited to run the sign in methods.



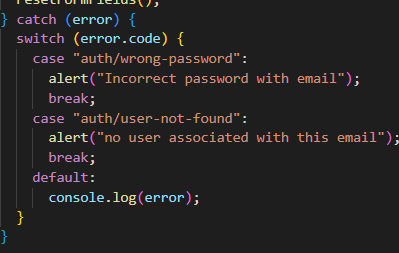


We use the signInWithEmailAndPassword method provided by firebase to authenticate.

### Finishing Authentication Page

A few improvements can be made to our authentication page.

* Give better error handling when submitting the form



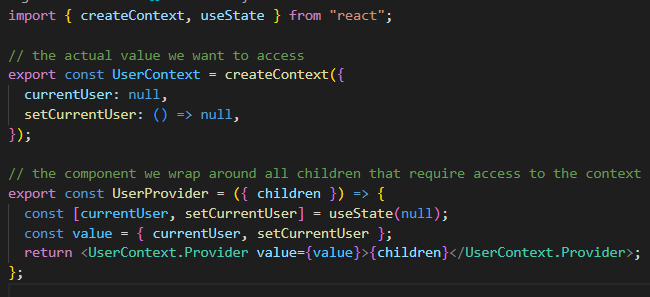
* Add type=”button” to the google sign in button. This prevents the default sign in method to run since it is the default behavior of a button.
* Add sass to the authentication page to center and flex.

## Section 7: React Context For State Management

### User Context

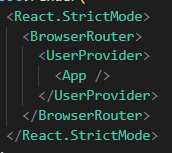
In order for our entire application to have access to the user object we receive from firebase, we can use a context.

First we need to build a context, the code is shown below.



A context requires 2 pieces, the actual context and the component we wrap our app in to allow it access to the context. Context leverages the useState to handle rerenders of components using the context.

Next we wrap our application in the UserProvider component.



In order to now actually retrieve or set the context, we use the “useContext” hook that will handle the rerendering whenever a change is made to the context.



We retrieve the setter function of the useState as shown above and are now able to set the currentUser within the sign in form.

Retrieving the currentUser is done in a similar way.



### Re-rendering From Context

It’s important to understand that any component that is hooked in the context using the useContext hook will get reran (not per se rerendered) whenever the context changes. Rerendering will only occur if the updated context is used within the jsx of our component.

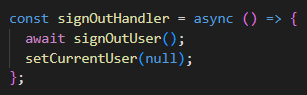
**!IMPORTANT** All code before the return statement of a component is ran whenever the context changes. This has some performance implications when using the context in a lot of components.

### Signing Out

Signing out is done using the signOut function provided by firebase. It requires the auth singleton and will sign out the signed in user.



After signing out we still need to reset our context.



### 

## Section 8: Observer Pattern

Currently, we are calling the setCurrentUser function in multiple locations. Firebase however, provides an easier method of handling authentication changes.

By using what is called an observer, we can tell our application to listen to any changes in regards to the authentication state and execute a callback whenever a change occurs.

This way we can centralize the handling of users signing in or out, since these actions are considered “changes to the authentication”.

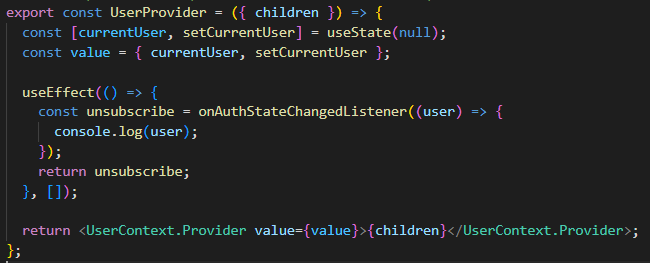
The method that can be used is:



We can then utilize this observer as follows.



This function can then be used within the context file.



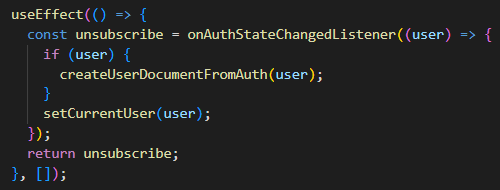
The onAuthStateChanged observer returns an unsubscribe function, that needs to be executed whenever the UserProvider component unmounts, since this is a memory leak.

We add the listener once whenever the component mounts and give it the callback function it needs to execute.

Next we can remove any call to the setCurrentUser within our separate components and use the callback to handle setting the context.

### Finalizing Auth Listener

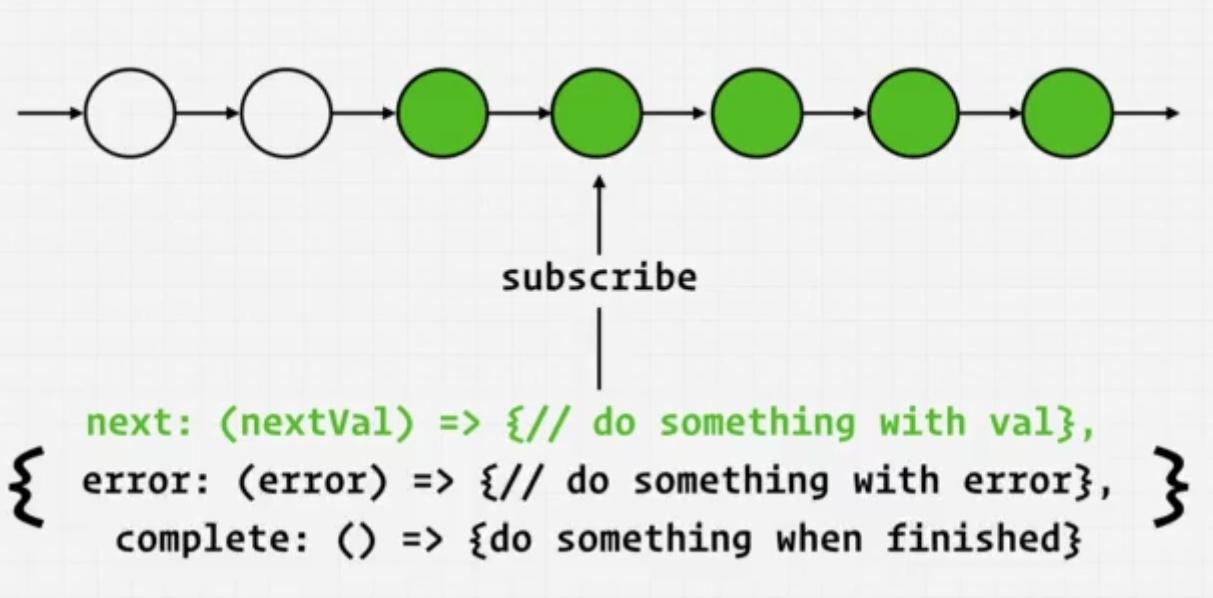
Since we can now update our context state in one location, we can use the setCurrentUser function within the callback of our observer.

****

We can also create a new user in our firestore. The createUserDocumentFromAuth will handle existing users.

### Observer Pattern

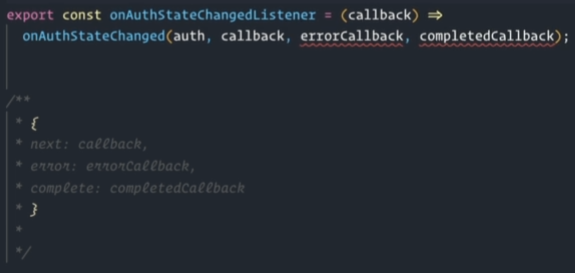
The observer pattern is relatively simple.



Basically, an observer stream is an asynchronous sequence of events. We need to subscribe to it using a listener. A listener is an object that has 3 callback functions.

* The next function will be triggered when a new event happens
* The error function will be triggered when the stream encounters an error
* The complete function will be triggered when the stream has ended.

The onAuthStateChanged function provided by firebase will create such a listener.



## Section 9: React Context Continued