Concept clear of firebase

What is Firebase?

- A product of Google
- Helps developers to build apps faster and securely.
- No programming is required on the firebase side which makes it easy to use its features more efficiently.
- Provides services to android, ios, web, and unity. It provides cloud storage.
- Uses NoSQL for the database for the storage of data.

Firebase SDK

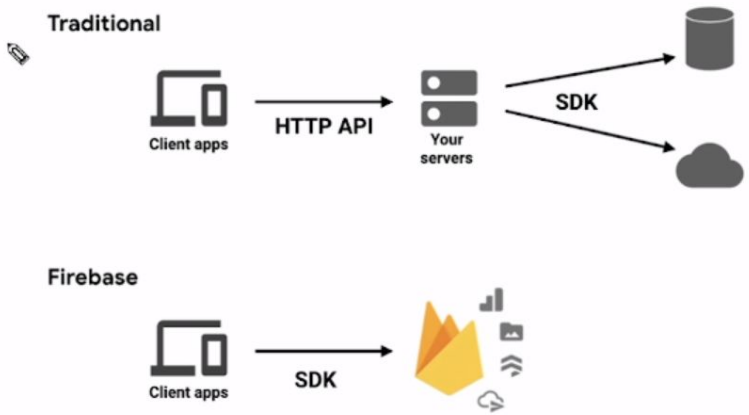
Software Development Kit একটা package -> একসাথে কতগুলো software development tools এর collection -> যা নির্দিষ্ট application তৈরীর কাজকে সহজ করে দেয়। Firebase SDK- এক বা একের অধিক sign-in methods এর সমন্বয় ঘটিয়ে Authentication System কে handle করার কাজ সহজ করে দেয়।

Firebase SDK

- The SDKs provided by Firebase, directly interact with backend services.
- There is no need to establish any connection between the app and the service.
- If you're using one of the Firebase database options, you typically write code to query the database in your client app.

Firebase SDK

- The traditional app development process requires writing both frontend and backend software. The frontend code just implements the API endpoints exposed by the backend, and the backend code actually does the work.
- With Firebase products, the traditional backend is bypassed, putting the work into the client.
- **Serverless**



Firebase Authentication

- Firebase Authentication provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app.
- It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more.
- Firebase Authentication integrates tightly with other Firebase services, and it **leverages industry standards like OAuth 2.0**

Firebase Authentication

- OAuth 2.0, which stands for "Open Authorization", is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user.
- OAuth 2.0 provides consented access and restricts actions of what the client app can perform on resources on behalf of the user, without ever sharing the user's credentials.

Since Firebase Authentication follows industry standards like **OAuth 2.0,** let's discuss more about OAuth2.0 to know how firebase authentication works behind the scene...

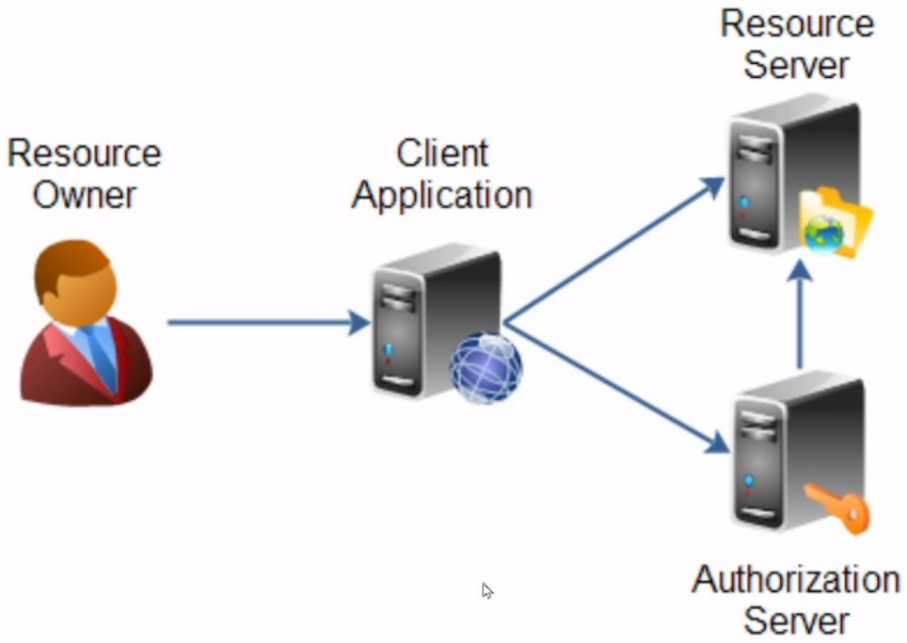
Principles of OAuth 2.0

- OAuth 2.0 is an authorization protocol and NOT an authentication protocol.
- **Authentication** verifies the identity of a user or service, and **authorization** determines their access rights.
- OAuth 2.0 is designed primarily as a means of granting access to a set of resources, for example, remote APIs or user's data.
- OAuth 2.0 uses Access Tokens. An **Access Token** is a piece of data that represents the authorization to access resources on behalf of the end-user.

Principles of OAuth 2.0

- The OAuth 2.0 authorization framework contains **authorization server**
- **Authorization Server** -receives requests from the Client for Access Tokens and issues them upon successful authentication and consent by the **Resource Owner(the user or system that owns the protected resources and can grant access to them.)**. The authorization server exposes two endpoints: the Authorization endpoint, which handles the interactive authentication and consent of the user, and the Token endpoint, which is involved in a machine to machine interaction.

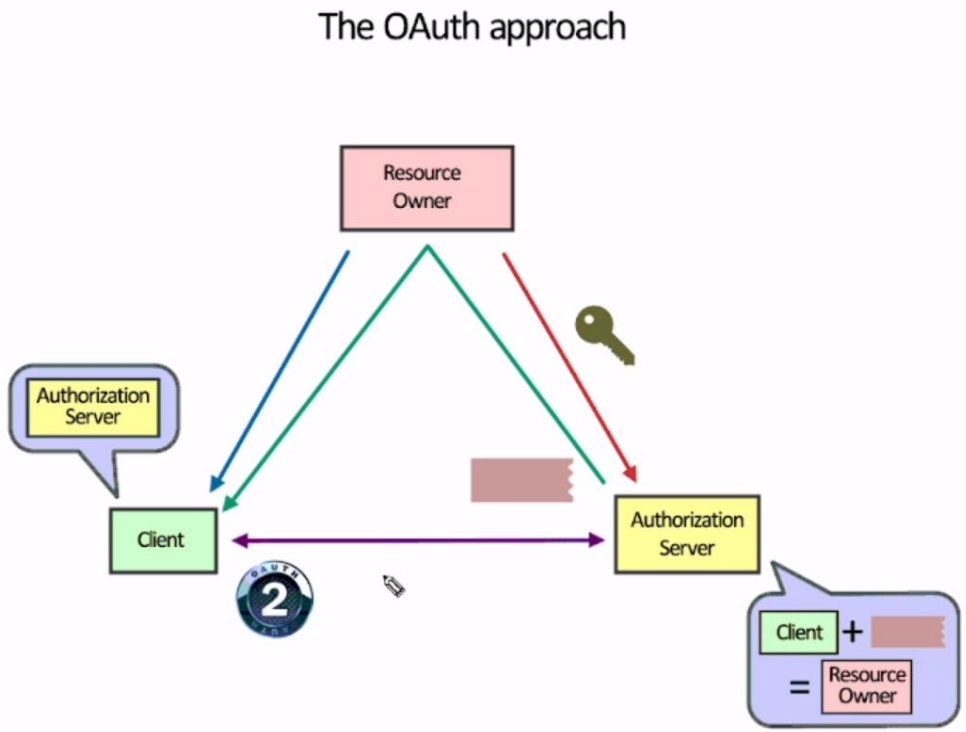
OAuth roles at a glance



How OAuth 2.0 works?

1. The Client requests authorization (*authorization request*) from the Authorization server, supplying the **client id** and **secret** to as identification; it also provides the scopes and an endpoint URI (***redirect URI***) to send the Access Token or the Authorization Code to.
2. The Authorization server authenticates the Client and verifies that the requested scopes are permitted.
3. The Resource owner interacts with the Authorization server to grant access.
4. The Authorization server redirects back to the Client with either an Authorization Code or Access Token, depending on the grant type.
5. With the Access Token, the Client requests access to the resource from the Resour

OAuth 2.0 approach visualization

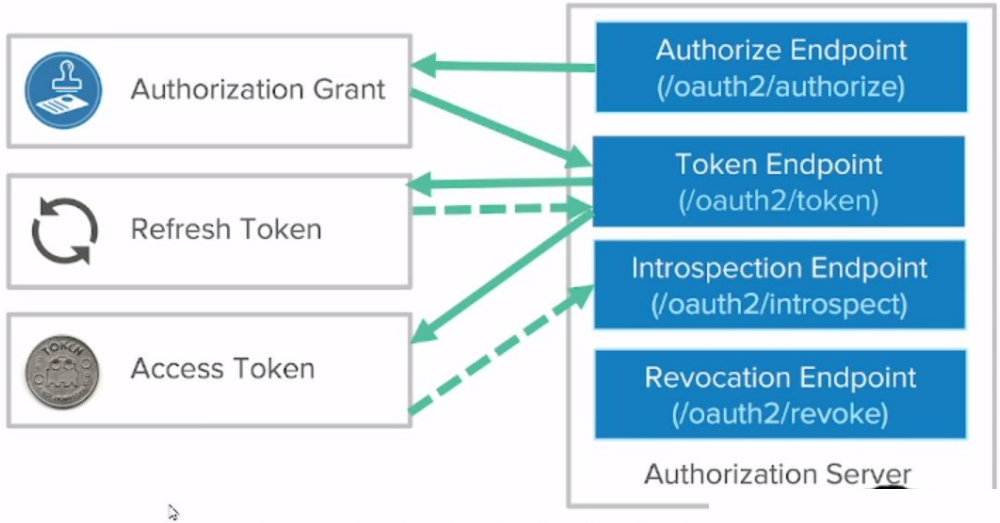


OAuth access tokens and authorization code

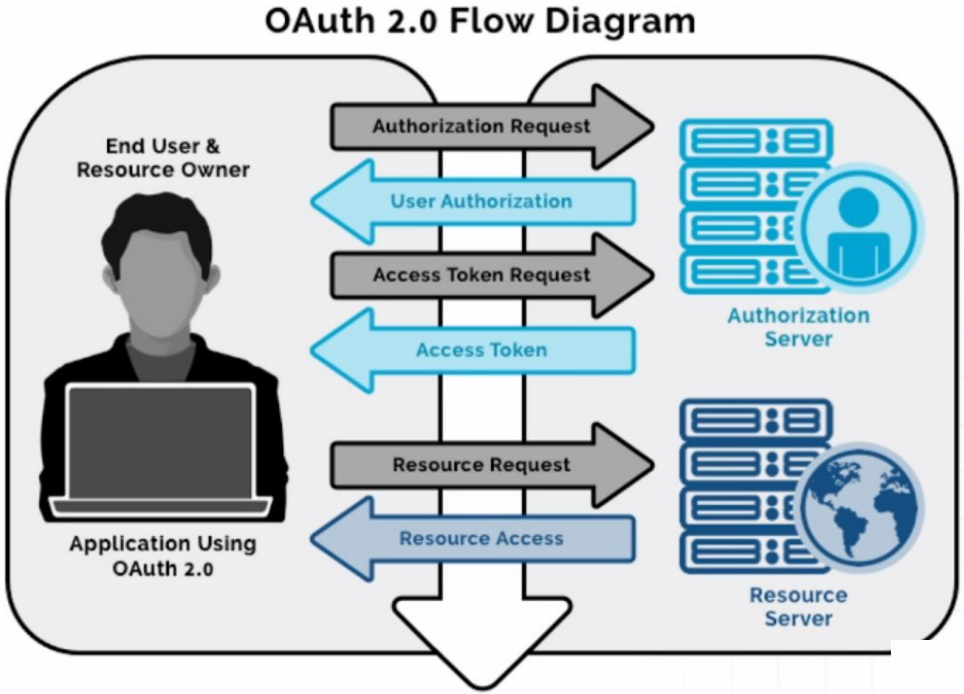
Authorization Code may be returned, which is then exchanged for an Access Token.

The Authorization server may also issue a Refresh Token with the **Access Token**.

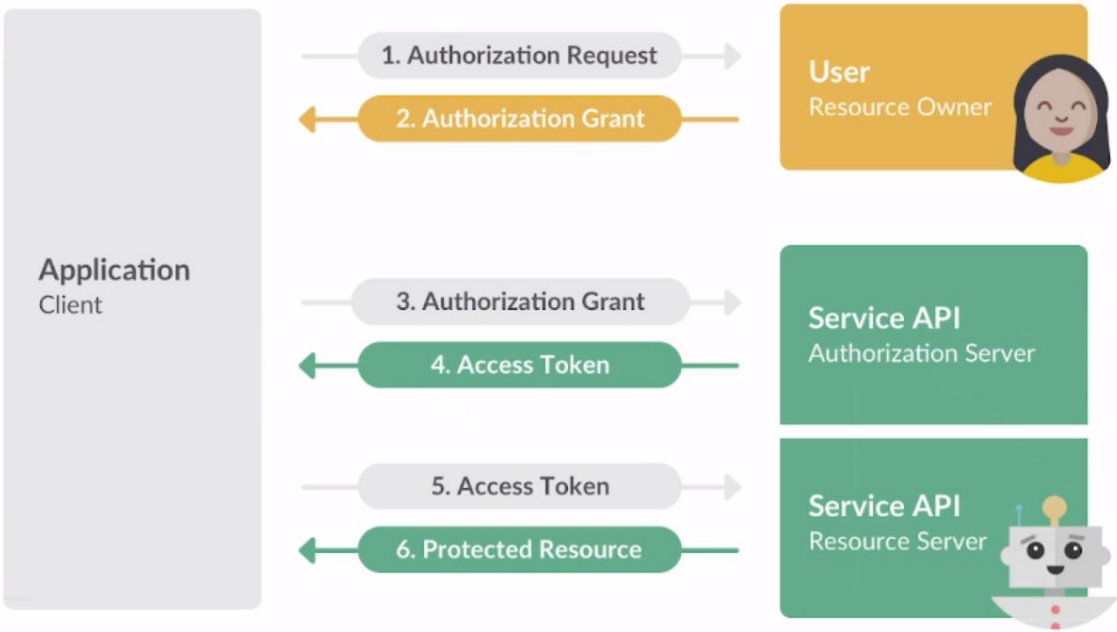
Refresh Tokens normally have long expiry times and may be exchanged for new Access Tokens when the latter expires. Because Refresh Tokens have these properties, they have to be stored securely by clients.



OAuth 2.0 working Flow diagram



OAuth 2.0 working for slack



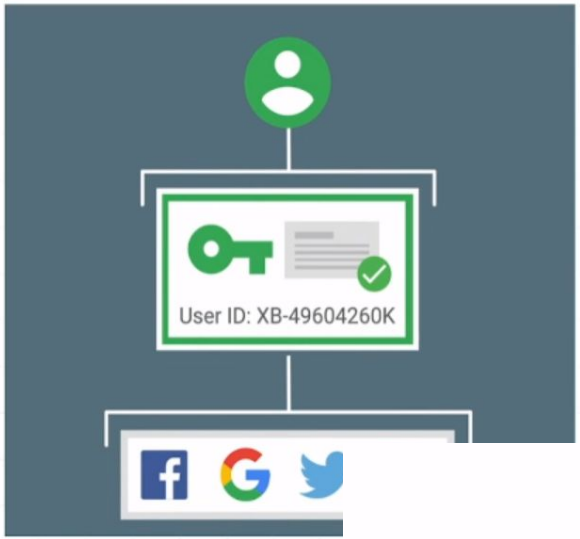
Firebase Authentication

- Once a user authenticates, 3 things happen:

  1) Information about the user is returned to the application via callbacks to allow us to personalize our app's user experience for the specific user

  2) The user information contains a unique ID which is guaranteed to be unique distinct across all providers

  3) This unique ID is used to identify the user and what parts of the backend system they are authorized to access.



Some Firebase Reference

- **initializeApp()**- Creates and initializes a FirebaseApp instance.
- **FirebaseApp** - A FirebaseApp holds the initialization information for a collection of services.
- **getAuth(app)**- Returns the Auth instance associated with the provided FirebaseApp. If no instance exists, initializes an Auth instance with platform-specific default dependencies.
- **OAuthProvider** - Provider for generating generic OAuthCredential.
- **OAuthCredential** - specify the details about each auth provider's credential requirements.

Why use an OAuth Provider?

As you make an app that accesses a solid web based back-end it's important to consider the following aspects of web security:

- Requiring strong passwords.
- The use of strong encryption.
- Ensuring secure communication (between client and server).
- Securing password storage within an encrypted database.
- Implementing password recovery.(Which also has to be secure).
- Adding 2 factor authentication. (Highly recommended extra layer of security)
- Including protection against man in the middle attacks.

Why use an OAuth Provider?

Having said all this even the simplest of applications may become fairly difficult to make when we factor in security.

Thankfully we won't have to worry about this if we are going to use an OAuth provider which will handle this for us. OAuth providers allow your users to login into your application using credentials from a trusted website such as Facebook, Github, Google+or even Twitter.

Send a div as children inside another component

```
<div className="countries-container">
                {
                    countries.map(country => <Country
                        key={country.capital}
                        country={country}
                    >
                        <div>
                            <h2>Sent as children inside prop</h2>
                        </div>
                    </Country>)
                }
</div>
```

```
const Country = (props) => {
    const { name, flags, region, population } = props.country;
    // console.log(props?.country?.capital[0]);
    console.log(props);
    return (
        <div className="country">
            <h4 >This is: {name.common}</h4>
            <img src={flags.png} alt="" />
            <p><small>Region: {region}</small></p>
            <p>Capital is: {props.country?.capital} </p>
            <p>population: {population}</p>
            <small>{props.children}</small>
        </div>
    );
};
```

Get div as children inside child component

Access Context value from anywhere inside app.js

```
import React, { useContext } from 'react';
import { AuthContext } from '../contexts/UserContext';

const Header = () => {
    const { user } = useContext(AuthContext);
    console.log('context', user);

    return (
        <div>

        </div>
    );
};
```

```
export const AuthContext = createContext();

const UserContext = (props) => {

    const user = {displayName: 'Akash'}

    const authInfo = {user: user}

    return (
        <AuthContext.Provider value={authInfo}>
            {props.children}
        </AuthContext.Provider>
    );
};
```

```
context ▶ {displayName: 'Aakash'}
context ▶ {displayName: 'Aakash'}
```

Access Context value from anywhere inside app.js

So you can get the context value from **any component you want** using **useContext()** hook

Now think ...

If you somehow can do anything that the context value get access of user login information?

```
export const AuthContext = createContext();

const UserContext = (props) => {

    const user = {displayName: 'Akash'}

    const authInfo = {user: user}

    return (
        <AuthContext.Provider value={authInfo}>
            {props.children}
        </AuthContext.Provider>
    );
};
```

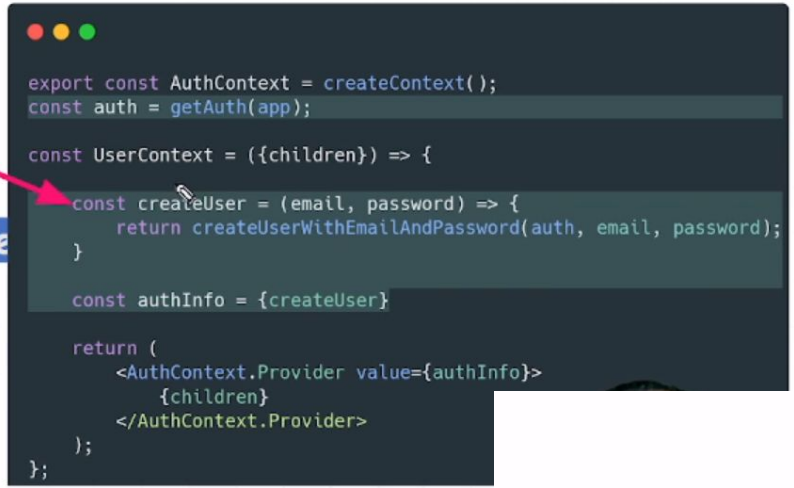
So What? Bingo!!!

Then you can show logged in information anywhere

Add createUser custom function in Context API

So now do the step by step **auth integration process** in **userContext.js** to get **logged in user info**

You have to implement custom createUser, signIn, logOut function which will return the actual firebase authentication functions but you can get access of these real firebase functions from anywhere through importing the custom functions.



```
export const AuthContext = createContext();
const auth = getAuth(app);

const UserContext = ({children}) => {

    const createUser = (email, password) => {
        return createUserWithEmailAndPassword(auth, email, password);
    }

    const authInfo = {createUser}

    return (
        <AuthContext.Provider value={authInfo}>
            {children}
        </AuthContext.Provider>
    );
};
```

Import createUser custom function in Register.js

```
const Register = () => {
    const { createUser } = useContext(AuthContext);

    const handleSubmit = event => {
        event.preventDefault();

        const form = event.target;
        const email = form.email.value;
        const password = form.password.value;

        createUser(email, password)
            .then(result => {
                const user = result.user;
                console.log('registered user', user);
            })
            .catch(error => {
                console.error(error)
            })

    }

    return (
       <form onSubmit={handleSubmit} className="card-body">
          <input type="email" name="email"  required />
          <input type="password" name='password' required />
          <button className="btn btn-primary">Register</button>
       </form>
    );
};
```

createUser custom function

createUserWithEmailAndPassword() behind the scene

Hurray! that's the actual logic.

You can think like: "you sent a function as props in ema-john project. Every button activity in child component affects the main function in parent component."

The above custom function logic is something like that.

onAuthStateChanged

- onAuthStateChanged allows us to subscribe to the users current authentication state, and receive an event whenever that state changes.
- onAuthStateChanged adds an observer for changes to the user's sign-in state. The observer triggers when users sign in, sign out...
- The onAuthStateChanged method also returns an unsubscriber function which allows us to stop listening for events whenever the hook is no longer in use.
- Calling onAuthStateChanged() "adds an observer/listener for changes to the user's sign-in state" AND returns an unsubscribe function that can be called to cancel the lis

How to use onAuthStateChanged?

There are two ways for API Call Interaction:

1) Triggering an event by clicking a button…etc
2) useEffect

Since onAuthStateChanged is a side-effect and this function needs to be called without triggering any event, it needs to be incorporated inside the useEffect hook.

How to use onAuthStateChanged?

The state of the user comes from firebase authentication and to get that we need to first:

1) setup a listener to observe changes in auth state in the firebase auth provider

How to use onAuthStateChanged?

The onAuthStateChanged takes two parameters

a) auth - provides an Auth instance

b) observer- it is a callback function. It gets invoked **immediately** after registering the onAuthStateChanged observer with the current authentication state and whenever the authentication state changes.

How to use onAuthStateChanged?

2) To start listening to auth state changes when our application **mounts**, we need do this with a `useEffect` hook:

```
useEffect(() => {
        const unsubscribe =  onAuthStateChanged(auth, currentUser => {
            setUser(currentUser);
            setLoading(false);
            console.log('auth state changed', currentUser);
        })

}, []);
```

How to use onAuthStateChanged?

3) Finally, the onAuthStateChanged() function returns the unsubscribe function to unregister the onAuthStateChanged observer. We save this function in a variable and name it unsubscribe. At the end, we return this unsubscribe function for cleanup to avoid memory leaks.

```
useEffect( () => {
        const unsubscribe =  onAuthStateChanged(auth, currentUser => {
            setUser(currentUser);
            setLoading(false);
            console.log('auth state changed', currentUser);
        })

        return () => {
            unsubscribe();
        }

}, [])
```

Why do I need to unsubscribe to `onAuthStateChanged` in firebase?

Calling **onAuthStateChanged()** "adds an observer/listener for changes to the user's sign-in state" **AND** returns an unsubscribe function that can be called to cancel the listener.
So the goal of the following pattern

```
useEffect( () => {
      const unsubscribe =  onAuthStateChanged(auth, currentUser => {
          //code
      })

      return () => {
          unsubscribe();
      }

  }, [])
```

is to listen **only once** for changes to the user's sign-in state. The first time the listener is triggered it calls the unsubscribe function, so it doesn't listen anymore

Why do I need to unsubscribe to `onAuthStateChanged` in firebase?

- You unsubscribe to avoid memory leaks.
- When you initialise onAuthStateChanged() you create a listener. If you don't unsubscribe then this listener will continue to listen even after you stop using it. This will waste memory.
- In order to unsubscribe you need something to unsubscribe from. This is why yo the listener to a variable. This allows you to refer to the variable when you want unsubscribe.

Private Route

- The private route component is used to protect selected pages in a React app from unauthenticated users
- The **<PrivateRoute />** component will simply check the current user state from the user, destructured from **useContext** hook
- The private route component renders child components (children) if the user is logged in. If not logged in the user is redirected to the /login page with the return url pas Navigate component.

Thank you!