AUTHENTICATION HOOKS

All of the authentication hooks generally follow the same structure, with the exceptions of ‘useValidateUserInput’ & ‘useChangeEmail’, which I will give their own sections at the end.

Each hook returns an object with two properties. The first is a function and the second is an object which provides updates about the status of that function. For example if the function request is pending, has an error or has been completed successfully.

THE FUNCTION:
All of these returned authentication hook functions are asynchronous. This is because they are connecting with and updating the Firebase Authentication service, which takes time to do. These functions also update our User Context API, so that both the User Context and the Firebase Authentication service are in sync.
Within these functions we have some conditionals, which are dictated by the isCancelled piece of state. These are our clean-up conditionals, so that if our component dismounts the state will not be updated. It is also within these functions that if you were to update an external database, like MongoDB for example, you would add use of another hook to do so within this function.

THE OBJECT:
The object returned usually consists of three key value pairs:

1. *isPending* - This tells us if the function called in the hook is still in process.
2. *error* - This will display the error message should one be returned from Firebase.
3. *success* - This tells us wether our function was successful or not. Its original default value is null.

Using these three pieces of state we can conditionally display different jsx elements in our components depending on their values.

THE CLEAN-UP FUNCTION:
This is not returned but is present in our hook. It is the function ‘setIsCancelled(true);’ which is returned inside of the empty useEffect at the end of our hook. Since a useEffect clean-up function runs on mount and every subsequent re-render, at the start of our returned function we need to reset isCancelled to false.

EXAMPLE:
To give an example of how one of these hooks are used within a component I will use the ‘useLoginWithEmailAndPassword’ hook:

```
import { useLoginWithEmailAndPassword } from ‘../../hooks/authentication-hooks/
useLoginWithEmailAndPassword’;

const Component = () => {

	const { login, loginState } = useLoginWithEmailAndPassword();

	return (
            {!loginState.isPending ? <button onClick={() => login(email, password)}
>Login<button> : <button disabled>Logging in<button>}
      )

}
```

//

ROUTING

All routing can be found within the App.js file and uses React Router v6. So far there is only 3 different pages:

1. *The Home page*, which a user can only see when logged out.
2. *The Dashboard*, which only a logged in user can see.
3. *The Account Settings page*, which only a logged user can see.

All other authentication services happen with the use of modals.

ROUTE GUARDING:
The routing here so far is quite basic. For example the home path (‘/‘) will route you to a different page (either Home or Dashboard) depending on your login status. This basic route guarding also expends to the Account Settings page and should continue like this throughout the entirety app.

LOADING SCREEN:
When first loading the page connecting to Firebase to check wether a user is logged in or not can take some time. So that we do not have a blank screen during this period I have added a PreLoader component, which basically tells the user that the service won’t be long and shows a loading SVG. This should help with visitor retention.

MODALS:
Furthermore, you will see at the top of the ‘App’ div that there are multiple lines referring to the modalState. This is so that if the Modal Context says that a modal is to be rendered on the screen, it is here we create the portal. The portal leads to a div with the id of ‘modal-root’ in the index.html file.

ROUTE GUARDING USING MODALS:
This line of code:

```
{!modalState && user.user !== null && !user.user.emailVerified &&
ReactDOM.createPortal(<UnverifiedEmail />, document.getElementById(‘modal-
root’))}
```

Is route guarding, so that the user cannot use the websites’ services unless they have verified their email. This is to help prevent spam accounts.