Notes - Data Stores - Firebase Authentication

- Dr Nick Hayward

A general intro and outline for using Firebase authentication with client-side and mobile apps.

Contents

Intro Authentication needs to be explicitly set relative to the datastore.

However, we may also specify privileges for user access to given parts of the current database. This allows access per user to their data or any shared data in the database.

Login form For most applications, we will need to create a login form or explicit login page.

This form may include a simple single sign-in button, or a full form with input fields for a user's username and password.

A login form will then allow us to query Firebase, and handle the user's login request.

The HTML for this type of form might be as follows,

```
<form id="fb-login">
    <input type="text" value="add your username" />
    <input type="password" />
    <button id="submit-login">login</button>
</form>
```

For single sign-in with Google, for example, we may add just a login button,

```
<button id="submit-login">login</button>
```

test form logic We may then add some initial JavaScript logic to test the form and the submit login button.

For example, we need to test a click event listener for the button, and define a callback for successful login and error handling.

```
document.getElementById('submit-login').addEventListener('click', () => {
    console.log('login button clicked');
});
```

Auth routing Another requirement for authentication is correct routing of the authentication request.

For example, if a user's login is successful, they need to be redirected back to the app. If not, then the user will customarily be redirected back to the login page with an appropriate error message.

Another consideration for routing is authenticated access to an app's content. A user should be able to view all public material, and any other material appropriate to their authenticated status.

Firebase Auth Firebase authentication requires initial configuration of settings in the online console, and certain properties defined in the host app.

sign-in method To work correctly with Firebase authentication in an app, we need to modify the default config to enable this feature.

To enable this feature in the online Firebase Console, select *Authentication* in the left menu for the required database. Then, select the tab for *Sign-in Method*, which provides various options for user authentication.

For example, we might start by selecting the *Google* authentication option for a test application.

After selecting this option, the console will update to show this sign-in method as *enabled*.

provider In our JavaScript config file for Firebase, we need to define the required *provider* for our app.

For example, we've enabled the *Google* sign-in method on the Firebase console, so we need to define a provider for this service in our app.

```
// AUTH - define provider
const googleProvider = new firebase.auth.GoogleAuthProvider();
```

This usage is defined in the Firebase docs as follows,

- Firebase Auth docs - Google Provider

We may also see similar examples for Facebook, GitHub, Twitter, &c.

auth state change Firebase provides various methods for working with authentication.

Relative to the `firebase` object in our app's JavaScript, we may call `auth()` with various additional methods to check a user's login state.

For example, we might add the following to check the state of a user's authentication request and return.

```
// provides listener for user authenication
// checks if a user is logged in or not...
firebase.auth().onAuthStateChanged((user) => {
    if (user) {
        console.log('user logged in');
    } else {
        console.log('user logged out');
    }
});
```

This example provides a listener, which logs to the console the state of user logins to the application.

As we start our app, we will initially see the result of a query to Firebase for the current user's logged in state. This query is performed to check whether the current user is already logged in, which prevents the authentication from being re-submitted.

e.g. as the app starts, and the user is not logged in, it will simply log to the console 'user logged out'.

auth login We may then call the following function to start the login process for Google authentication.

```
// start login call to return  sign-in...
const startLogin = () => {
    return firebase.auth().signInWithPopup(googleProvider);
};
```

If a user is not currently logged in, this function will show a popup window with the option to login with a user's Google account. After login, a user will be redirected back to the current application.

The app's auth state will again be checked, and the `user logged in` will be logged to the console.

So, we may then test this login call with the login button in our app,

```
document.getElementById('submit-login').addEventListener('click', startLogin);
```

A successful login will redirect the user back to the app, as expected, and the auth state listener, as defined above, will log to the console. This successful login will persist for the cache of the application.

You may also check authenticated users in the Firebase console for your app. Simply select the *Authentication* option, and then the *Users* tab. This will then list all of the currently authenticated users for this app, i.e. those users that have signed into your app using the above options.

auth logout To allow a user to logout, we'll start by adding an explicit logout button.

```
<button id="submit-logout">logout</button>
```

We can set this to only show when a user is logged in, and then hide after logging out of the authentication service.

This button will be called with the standard event listener, which will execute the following,

```
// start logout call to return sign-out...
const startLogout = () => {
    return firebase.auth().signOut();
};
```

App usage So, we may now allow a user to login and logout of the application with the Google provider service with Firebase.

However, we need to setup our example app to use the authenticated status relative to permissions and access. i.e. defining what an authenticated user may access and view within the app.

redirecting a user As a user logs in and logs out of an application, we need to ensure they are redirected to the appropriate content, page, or screen for their authentication status.

For example, a user might be redirected to their account page after logging in, and then to the home page upon logout.

With explicit routing frameworks, including custom stack navigation, we may define such pages or screens. We may also restrict access relative to log in status.

user access Likewise, for a single page app, we may restrict certain content relative to a user's authentication status.

A simple test of this status may be executed in the state listener for Firebase authentication.

```
// provides listener for user authenication - checks if a user is logged in or not...
firebase.auth().onAuthStateChanged((user) => {
    if (user) {
        loginBtn.style.display = 'none';
        logoutBtn.style.display = 'inline';
        console.log('user logged in');
    } else {
        loginBtn.style.display = 'inline';
        logoutBtn.style.display = 'none';
        console.log('user logged out');
    }
});
```

In the above example, we are simply modifying the value of the display property for each button relative to a user's authentication status. This will then show the appropriate button to the user dependent upon their *auth* state.

Likewise, we may execute certain functions to modify a single page app for each *auth* state. Perhaps, we might show certain content and options for an authenticated user, or execute an async query for that user to the Firebase data store.

app content One option we may test is simply showing and hiding content relative to a user's *auth* state.

For example, a user logs into the app, and content is queried from the connected Firebase datastore. The app's UI is then updated with this content.

```
// provides listener for user authenication
// checks if a user is logged in or not...
firebase.auth().onAuthStateChanged((user) => {
    const output = document.getElementById('fb-content');
    if (user) {
        loginBtn.style.display = 'none';
        logoutBtn.style.display = 'inline';
        outputData(output);
        console.log('user logged in - data output');
    } else {
        loginBtn.style.display = 'inline';
        logoutBtn.style.display = 'none';
        clearData(output);
        console.log('user logged out - data output removed');
    }
});
```

In the above state listener, we check as usual for a user's login state. Then, we call the appropriate function either load data to the app, `outputData()` , or remove the data using the `clearData()` function.

As the data is loaded asynchronously from Firebase, it is only loaded in the app when a user has logged in successfully.

For example, we might load some data as follows,

```
// get ref in db once
// call forEach() on return snapshot
// push values to local array
// unique id for each DB parent object is `key` property on snapshot
function loadData() {
  // get data from FB
    const data = db.ref('egypt/ancient_sites')
      .once('value')
      .then((snapshot) => {
        const sites = [];
        snapshot.forEach((siteSnapshot) => {
          sites.push({
            id: siteSnapshot.key,
            ...siteSnapshot.val()
          });
        });
            return sites;
      });
        // return data Promise
        return data;
}
```

We may then call then function in the `outputData()` function to update the UI,

```
// prepare data from loadData() for rendering
function outputData(elem) {
    // use data Promise - append to DOM...
    const output = loadData().then((data) => {
        for (site in data) {
            const p = document.createElement('p');
            const title = document.createTextNode(data[site]['title']);
            p.appendChild(title);
            elem.appendChild(p);
        }
    });
    // return the generated output for rendering...
    return output;
}
```

We might abstract this further with separate functions and logic for *render* updates, element building, validation &c.

As a user logs out of the app, we also require a function to delete the rendered content. For example,

```
// check child nodes relative to passed element
function clearData(elem) {
    // check passed element for child nodes
    while (elem.firstChild) {
        // remove child...
        elem.removeChild(elem.firstChild);
    }
}
```

This function will check the passed element for child nodes. While they exist, it will simply remove them from the UI, thereby deleting the app's content.

References

- Firebase Auth docs - Google Provider