Class 23 - Course Project (HTTP) && Authentication and Route Protection in Angular

Class 23 Course Content

Lesson Outline

Today we will learn how to:

1. **Authenticate users using Firebase**
2. **Understand the !! operator**
3. **Work with Refresh Tokens & JWT**
4. **Create AuthGaurds and use HTTP Interceptors**
5. **Add Secret Enviornment Variables**
6. **Implement User Authentication in Angular**

Lesson Notes

- **Authentication:** *Authentication* is the process of verifying the identity of an individual. Authenticating a user results in user information and access to certain actions or pages that would be restricted otherwise.
- **RESTful API:** A *REST API* (Representational State Transfer Architectural Application Programming Interface) is an architectural style and a set of rules for working with an API in a specific way. Using "RESTful Architecture", every incoming API request for the same source should ook the same, no matter where it is coming from.
- **JSON Web Token:** A *JSON Web Token* is a compact and self-contained way for securely transmitting information between parties as a JSON Object. They are an encrypted tokens that can be sent from client to server safely.

Course Project - Authentication Steps

STEP 1: Adding the Auth Page

Terminal:

- Inside the "shared" folder, create a new component called "AuthComponent".

```
ng g c shared/auth
```

shared/auth/auth.component.html file:

- Create a row with a column that contains a form inside. The form should have two inputs (email and password) and two buttons (Sign up and Sign in).

```
<div class="row">
  <div class="col-xs-12 col-md-6 offset-md-3">
    <form>
      <!-- EMAIL -->
      <div class="form-group">
        <label for="email">Email</label>
        <input
          type="email"
          id="email"
          name="email"
          class="form-control"
          email
        />
      </div>

      <!-- PASSWORD -->
      <div class="form-group">
        <label for="password">Password</label>
        <input
          type="password"
          id="password"
          name="password"
          class="form-control"
        />
      </div>

      <!-- BUTTONS -->
      <button class="btn btn-primary mr-3">Sign Up</button>
      |
      <button class="btn btn-primary">Login</button>
    </form>
  </div>
</div>
```

app-routing.module.ts file:

- Add a new route when a user is on "our-website.com/auth" that renders the "AuthComponent".

```
// . . .
{ path: 'auth', component: AuthComponent }
```

shared/navigation/navigation.component.html file:

- Create a new link that sends us to the Auth page.

```
<!-- . . . -->
<li class="nav-item">
  <a class="nav-link" routerLink="/library" routerLinkActive="active"
    >Library</a
  >
</li>
<li class="nav-item">
  <a class="nav-link" routerLink="/auth"
routerLinkActive="active">Auth</a>
</li>
```

STEP 2: Auth-Mode Toggling

shared/auth/auth.component.ts file:

- Create a new variable "isLoginMode" and set it equal to true by default.
- Create a new method called `onSwitchAuthMode()` and have it toggle the "isLoginMode" variable to the opposite of it's current value.

```
isLoginMode = true;

onSwitchAuthMode() {
    this.isLoginMode = !this.isLoginMode;
}
```

shared/auth/auth.component.html file:

- Set the types for both buttons.
- Use the "isLoginMode" variable and a ternary operator to dynamically set the text on both buttons.
- Add a `(click)` listener on the second button to call our `onSwitchAuthMode()` function.

```
<!-- BUTTONS -->
<button class="btn btn-primary" type="submit">
  {{ isLoginMode ? "Sign In" : "Sign Up" }}
</button>
|
<button class="btn btn-primary" type="button"
(click)="onSwitchAuthMode()">
  Switch to {{ isLoginMode ? "Sign Up" : "Sign In" }}
</button>
```

STEP 3: Handling the Form Input

shared/auth/auth.component.html file:

- Using the "Template-Driven-Approach", setup our form using "ngModel".
- Create a referrence on the form and an `(ngSubmit)` function.
- *Note*: Firebase requires us to have a password of at least 6 characters. Add this validation.
- Create a submit button that is disabled if the form isn't valid.

```
<form #authFormRef="ngForm" (ngSubmit)="onAuthFormSubmit(authFormRef)">
  <!-- EMAIL -->
  <input
    type="email"
    id="email"
    name="email"
    class="form-control"
    email
    ngModel
    required
  />
  <!-- . . . -->

  <!-- PASSWORD -->
  <input
    type="password"
    id="password"
    name="password"
    class="form-control"
    minlength="6"
    ngModel
    required
  />
  <!-- . . . -->

  <!-- BUTTONS -->
  <button class="btn btn-primary" type="submit"
[disabled]="!authFormRef.valid">
    {{ isLoginMode ? "Sign In" : "Sign Up" }}
  </button>
  <!-- . . . -->
</form>
```

shared/auth/auth.component.ts file:

- Create the `onAuthFormSubmit(formObj: NgForm)` function.
- Print to the console the "formObj.value".
- Reset the form.

```
onAuthFormSubmit(formObj: NgForm) {
    console.log('Form Values:', formObj.value);
    formObj.reset();
}
```

STEP 4: Setting Up Firebase Authentication

Your-Firebase-Project-Realtime-Database:

- Go to your firebase project and click on "Realtime Database".
- Click on the "Rules" tab and change the ".read" and ".write" rules.
- Publish the new rules.
- *Note*: Now if we try and "Fetch" data, our application will throw us a *401 (Unauthorized)* Error.

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null",
  }
}
```

Your-Firebase-Project-Authentication:

- Using the side navigation bar, click on "Authentication" and then the "Getting Started" button.
- Under the "Sign-in Method" tab, under "Native providers", click "Email/Password".
- Enable the top toggle and leave the bottom one disabled.
- Click Save.

Firebase-Auth-REST-API-DOCS:

- Navigate to the Firebase Rest Auth Docs - Sign Up with Email / Password
- Find the URL that looks like this: https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=(API_KEY). This will be the API we need to use in our application to sign up using Google/Firebase.
- Navigate to the Firebase Rest Auth Docs - Sign Up with Email / Password
- Find the URL that looks like this: https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=(API_KEY). This will be the API we need to use in our application to sign in using Google/Firebase.

Your-Firebase-Project-Setting:

- To get your "API_KEY", go the to the gear icon next to "Project Overview" on the side navigation bar.

- Click "Project Settings" and copy your "Web API Key".

STEP 5: Creating the Auth Service Sign Up Logic

shared/auth/auth.service.ts file:

- Create a new service inside the "shared/auth" folder called "auth.service.ts".
- Use `Injectable({})` to provide the service in the "root" of the application.
- Inject the "HttpClient" into the constructor.
- Create a variable that holds your API_KEY. You can also create variables for the sign up and sign in urls.
- *Note*: We will place our API Keys in the enviornment folder eventual, as it is best-practice.
- *Note*: Also, don't try and copy the API_KEY in this tutorial. You will need to find your own!
- Create a method `signUp(email: string, password: string) {}` that uses the "HttpClient" to post to the URL we saved earlier.
- *Note*: Use `(``)` (backticks) to dynamically put in your "API_KEY" variable in the right place.
- The "POST" request also requires you to pass in a body object containing the "email", "password" and "returnSecureToken" boolean variable.

```
import { HttpClient } from "@angular/common/http";
import { Injectable } from "@angular/core";

const AUTH_API_KEY = "AIzaSyC1PGMcG"; // Use your api key!
const SIGN_UP_URL =
  "https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=";
const SIGN_IN_URL =
  "https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?
key=";

@Injectable({
  providedIn: "root",
})
export class AuthService {
  constructor(private http: HttpClient) {}

  signUp(email: string, password: string) {
    return this.http.post(SIGN_UP_URL + AUTH_API_KEY, {
      email,
      password,
      returnSecureToken: true,
    });
  }
}
```

STEP 6: Creating the Response Interface

shared/auth/auth.service.ts file:

- Above the class declaration, create a new interface called "AuthResponseData".
- This interface should be an object containing all the response items and their respective types.
- Since the "post" request is an observable, add the interface as a type using the "<>" brackets: `<AuthResponseData>`.

```
export interface AuthResponseData {
  kind: string;
  idToken: string;
  email: string;
  refreshToken: string;
  expiresIn: string;
  localId: string;
  registered?: boolean;
}
// . . .
return this.http.post<AuthResponseData>(//. . .//
```

STEP 7: Sending the Sign Up Request

shared/auth/auth.component.ts file:

- Inject the "AuthService" inside the constructor.
- Add form validation at the beginning of the `onAuthFormSubmit()` method.
- Grab the "email" and "password" from the form.
- Create a conditional that checks on the "isLoginMode" boolean.
- Use the `AuthService.signUp(email, password)` method and pass in the email and password from the form.
- Subscribe to the response data of the `signUp()` method and print to the console the response. Also, print the error if we have one.

```
constructor(private authService: AuthService) {}
// . . .
onAuthFormSubmit(formObj: NgForm) {
  // Validation check
    if (!formObj.valid) return;

// Destructure the form input values
    const { email, password } = formObj.value
```

```
// Conditional to see what mode we are in
    if (this.isLoginMode) {
      // Sign In Logic
    } else {
      // Sign Up Logic
      this.authService.signUp(email, password).subscribe(
        (res) => {
          console.log('Auth Response Success:', res);
        },
        (err) => {
          console.error('Auth Res Error:', err);
        }
      );
    }

    // Observable logic with error handling

// Reset the form
    formObj.reset();
}
```

Stopping Point:

- Check your console and make sure you are getting the correct message printed out.
- Check you Firebase Auth Users tab and see if your email/password made it through.

STEP 8: Handling Sign Up Errors

shared/auth/auth.component.ts file:

- Create a new variable "errMsg" and set it to null by default.
- Inside the subscription error callback, set the local "errMsg" variable to the err we receive.
- Inside the success callback, if we have an error, clear it.

```
errMsg: string = null;
// . . .
this.authService.signUp(email, password).subscribe(
  (res) => {
    console.log("Auth Response Success:", res);
    if (this.errMsg) this.errMsg = null;
  },
  (err) => {
    console.error("Auth Res Error:", err);
    this.errMsg = err.message;
  }
);
```

shared/auth/auth.component.html file:

- Add a paragraph below the form that displays our "errMsg" if there is indeed an "errMsg".

```
<!-- . . . -->
</form>

<!-- ERROR -->
<p class="alert alert-danger mt-3" *ngIf="errMsg">{{ errMsg }}</p>
```

STEP 9: Adding the Sign In Logic

shared/auth/auth.service.ts file:

- Create the `signIn(email: string, password: string) {}` method that takes in a string and password and returns an Observable from an "HttpClient POST" request.
- Make sure the "AuthResponse" Interface has an optional boolean field called "registered".

```
export interface AuthResponseData {
  // For Sign In Only
  registered?: boolean
}
// . . .
signIn(email: string, password: string) {
    return this.http.post<AuthResponseData>(
        SIGN_IN_URL + AUTH_API_KEY,
        {
        email,
        password,
        returnSecureToken: true,
        }
    );
}
```

shared/auth/auth.component.ts file:

- Inside of the `this.isLoginMode` conditional, use the "AuthService" to login. Subscribe to the data and print the result to the console.

```
if (this.isLoginMode) {
  // Sign In Logic
  this.authService.signIn(email, password).subscribe(
    (res) => {
      console.log("Auth Sign In Response:", res);
      if (this.errMsg) this.errMsg = null;
    },
```

```
      (err) => {
        console.error("Auth Res Error:", err);
        this.errMsg = err.message;
      }
    );
  }
```

- Refactor this component to use an Observable.

```
authObsrv: Observable<AuthResponseData>;
// . . .
onAuthFormSubmit(formObj: NgForm) {
    // Validation check
    if (!formObj.valid) return;

    // Destructure the form input values
    const { email, password } = formObj.value

    // Conditional to see what mode we are in
    if (this.isLoginMode) {
      // Sign In Logic
      this.authObsrv = this.authService.signIn(email, password);
    } else {
      // Sign Up Logic
      this.authObsrv = this.authService.signUp(email, password);
    }

    // Observable logic with error handling
    this.authObsrv.subscribe(
      (res) => {
        console.log('Auth Res Success:', res);
        if (this.errMsg) this.errMsg = null;
      },
      (err) => {
        console.error('Auth Res Error:', err);
        this.errMsg = err.message;
      }
    );

    // Reset the form
    formObj.reset();
}
```

Stopping Point:

- Make sure you can Sign Up with a new email.
- Make sure you can Sign In using that email and password.
- When you Sign In, check to see if you have the correct response in your developer tools console. (It should have "registered: true") as the last key/value pair in the response object.)

STEP 10 - Modeling, Creating, and Storing Users

shared/auth/User.model.ts file:

- Inside the "shared/auth" folder, create a new file called "User.model.ts".
- export a "User" class and inside the constructer, create public variables for every field we want to have access to globally and private variables for more sensative data.
- Create a "getter" function inside this model that returns the user's "_token".

```
export class User {
  constructor(
    public email: string,
    public id: string,
    private _token: string,
    private _tokenExpirationDate: Date
  ) {}

  public get token() {
    // Validation to ensure we have a expDate and it is not past the
current date
    if (!this._tokenExpirationDate || new Date() >
this._tokenExpirationDate)
      return null;

    // Send the user's token
    return this._token;
  }
}
```

shared/auth/auth.service.ts file:

- Create a "BehaviorSubject" variable that stores the authenticated user set to null by default.
- Inside the `signUp()` method, add the "pipe" and "tap" operators to call a new method we will create `handleAuth(email, localId, idToken, expiresIn)`.
- Inside the `signIn()` method, add the "pipe" and "tap" operators to call a new method we will create `handleAuth(email, localId, idToken, expiresIn)`.
- Create the `handleAuth(email: string, userId: string, token: string, expiresIn: number)` method that creates the "expirationDate", a "new User", and saves that user to local storage.

```
export class AuthService {
  currentUser = new BehaviorSubject<User>(null);

  // . . .
```

```
  signUp(email: string, password: string) {
    return this.http
      .post<AuthResponseData>(SIGN_UP_URL + AUTH_API_KEY, {
        email,
        password,
        returnSecureToken: true,
      })
      .pipe(
        tap((res) => {
          // Use "Object Destructuring" to get access to all response
values
          const { email, localId, idToken, expiresIn } = res;
          // Pass res values into handleAuth method
          this.handleAuth(email, localId, idToken, +expiresIn);
        })
      );
  }

  signIn(email: string, password: string) {
    return this.http
      .post<AuthResponseData>(SIGN_IN_URL + AUTH_API_KEY, {
        email,
        password,
        returnSecureToken: true,
      })
      .pipe(
        tap((res) => {
          const { email, localId, idToken, expiresIn } = res;
          this.handleAuth(email, localId, idToken, +expiresIn);
        })
      );
  }

  handleAuth(email: string, userId: string, token: string, expiresIn:
number) {
    // Create Expiration Date for Token
    const expDate = new Date(new Date().getTime() + expiresIn * 1000);

    // Create a new user based on the info passed in the form and emit
that user
    const formUser = new User(email, userId, token, expDate);
    this.currentUser.next(formUser);

    // Save the new user in localStorage
    localStorage.setItem("userData", JSON.stringify(formUser));
  }
}
```

STEP 11: Reflecting the Auth State in the UI

shared/auth/auth.component.ts file:

- Inject the Angular "Router" inside the constructor.
- Inside the successful response data has been logged, use the router to navigate the user to the "/bookshelf" page.

```
constructor(private authService: AuthService, private router: Router) {}
// . . .
(res) => {
        console.log('Auth Res Success:', res);
        if (this.errMsg) this.errMsg = null;

        this.router.navigate(['bookshelf']);
    },
```

STEP 12: Updating the Navigation

shared/navigation/navigation.component.ts file:

- Inject the "AuthService" inside the constructor.
- Create a new variable "isAuthenticated" and set it to false by default.
- Inside `ngOnInit()`, setup a subscription to the "currentUser" we are emitting in the "AuthService".
- Inside the subscription, use the `!!` operator to set the local "isAuthenicated" variable to the boolean value of the user we get from the subscription.
- Implement `OnDestroy()` and unsubscribe to this when the component is destroyed.

```
isAuthenticated = false;

// . . .

constructor(private httpService: HTTPService, private authService:
AuthService) {}

// . . .

ngOnInit(): void {
    this.authService.currentUser.subscribe((user) => {
      // !! - Bang Bang, You're a Boolean.
      this.isAuthenticated = !!user;
    });
}

ngOnDestroy(): void {
    this.authService.currentUser.unsubscribe();
}
```

shared/navigation/navigation.component.html file:

- Using `*ngIf`, disable the "bookshelf" and "library" routes if there is no authenticated user.
- Create a new route for Signing Out and Signing In.

```
<li class="nav-item" *ngIf="isAuthenticated">
  <a class="nav-link" routerLink="/bookshelf" routerLinkActive="active"
    >Bookshelf</a
  >
</li>
<li class="nav-item" *ngIf="isAuthenticated">
  <a class="nav-link" routerLink="/library" routerLinkActive="active"
    >Library</a
  >
</li>
<li class="nav-item" *ngIf="!isAuthenticated">
  <a class="nav-link" routerLink="/auth"
routerLinkActive="active">Auth</a>
</li>
<li class="nav-item" *ngIf="isAuthenticated">
  <a class="nav-link" routerLink="/auth" routerLinkActive="active">Sign
Out</a>
</li>
```

STEP 13: Adding Tokens to Outgoing Requests

shared/auth/auth.service.ts file:

- Create a new variable "userToken" and set it to null by default.

```
userToken: string = null;
```

shared/http/http.service.ts file:

- Inject the "AuthService" inside of the constructor.
- Using the "pipe" and "take" operators from "rxjs", grab the "currentUser" from the "AuthService" and get acces to the stream of data and immediately unsubscribe.
- To combine these two Observables and have access to the token inside the http request, use the "exhaustMap" operator.
- Refactor the http method subscription code to now pass in "params" to set the new "auth-token" to the "user-token".

```
  // *INJECTIONS*
constructor(
    private http: HttpClient,
    private authService: AuthService,
    private bookshelfService: BookshelfService
) {}

//   . . .

// *METHOD* - Fetch books from Firebase DB
// ! NOTE: WE WILL NOT NEED THIS CODE SHORTLY!!!
fetchBooksFromFirebase() {
    return this.authService.currentUser.pipe(
      take(1),
      exhaustMap((user) => {
        console.log(user);
        return this.http
          .get(this.firebaseRootURL, {
            params: new HttpParams().set('auth', user.token),
          })
          .pipe(
            tap((books: Book[]) => {
              this.bookshelfService.setBooks(books);
            })
          );
      })
    );
}
```

STEP 14: Attaching the Token Using an Interceptor

shared/auth/auth-interceptor.service.ts file:

- Create a new service called "AuthInterceptorService" inside the "shared/auth" folder.
- Use `Injectable()` without passing any "providedIn" arguments...We will do this one a tad differently.
- Export the "AuthInterceptorService" that `implements HttpInterceptor`.
- Inject the "AuthService" inside the constructor.
- Add the `intercept(req: HttpRequest<any>, next: HttpHandler) {}` method which returns the request inside the `authService.currentUser) "pipe"/"exhauseMap" callback.
- Check to see if we have a user, if not, send back the unmodified request.
- Before returning the request, modify it to set the auth token.

```
import { AuthService } from "./auth.service";
import { Injectable } from "@angular/core";
import {
  HttpHandler,
  HttpInterceptor,
  HttpParams,
  HttpRequest,
} from "@angular/common/http";
import { exhaustMap, take } from "rxjs/operators";

@Injectable()
export class AuthInterceptorService implements HttpInterceptor {
  constructor(private authService: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    return this.authService.currentUser.pipe(
      take(1),
      exhaustMap((user) => {
        // Make sure we have a user
        if (!user) return next.handle(req);

        // Modify the req to have access to the token
        const modifiedReq = req.clone({
          params: new HttpParams().set("auth", user.token),
        });

        // Return the modified request
        return next.handle(modifiedReq);
      })
    );
  }
}
```

app.module.ts file:

- Inside the "providers" array, add an object that provides our "AuthInterceptorService".

```
// . . .
providers: [
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptorService,
      multi: true,
    },
],
// . . .
```

shared/http/http.service.ts file:

- Refactor the `fetchBooksFromFirebase()` method to not send the token anymore because our "AuthInterceptorService" now takes care of that logic.
- Clean up the components imports, constructor, and unused code.

```
import { AuthService } from "./../auth/auth.service";
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { tap } from "rxjs/operators";
import { BookshelfService } from "./../../bookshelf/bookshelf.service";
import { Book } from "../book/book.model";

@Injectable({
  providedIn: "root",
})
export class HTTPService {
  // . . .

  constructor(
    private http: HttpClient,
    private bookshelfService: BookshelfService
  ) {}

  // . . .

  fetchBooksFromFirebase() {
    return this.http.get<Book[]>(this.firebaseRootURL, {}).pipe(
      tap((books) => {
        this.bookshelfService.setBooks(books);
      })
    );
  }
}
```

STEP 15: Adding Sign-Out Functionality

shared/auth/auth.service.ts file:

- Inject the Angular Router in the constructor.
- Create a `signOut()` method that signs our user out of the application.
- Reroute the user to the "/auth" page.

```
constructor(private http: HttpClient, private router: Router) {}

// . . .

signOut() {
    this.currentUser.next(null);
```

```
    this.router.navigate(['auth']);
}
```

shared/navigation/navigation.component.ts file:

- Create the `onSignOut()` method that uses the "AuthService" to sign out.

```
onSignOut() {
    this.authService.signOut();
}
```

shared/navigation/navigation.component.html file:

- On the Sign Out list-item tag, add a click listener that calls the `onSignOut()` method.

```
<li class="nav-item" *ngIf="isAuthenticated" (click)="onSignOut()">
  <a class="nav-link" routerLink="/auth" routerLinkActive="active">Sign
Out</a>
</li>
```

STEP 16: Adding the Auth-Gaurd

shared/auth/auth.gaurd.ts:

- Inside the "shared/auth" folder, create a new component "AuthGaurd".
- Use `Injectable({ providedIn: 'root' })` to provide this gaurd in the root of the application.
- Export the "AuthGaurd" class that `implements CanActivate`.
- Inject the "AuthService" and "Router" inside the constructor.
- Create the `canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot)` method. This method should contain all the logic on what should happen when a user tries to access the application (authenticated or not).

```
import { AuthService } from "./auth.service";
import { Injectable } from "@angular/core";
import {
  ActivatedRouteSnapshot,
  CanActivate,
  Router,
  RouterStateSnapshot,
} from "@angular/router";
import { map, take } from "rxjs/operators";

@Injectable({ providedIn: "root" })
```

```
export class AuthGaurd implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return this.authService.currentUser.pipe(
      take(1),
      map((user) => {
        const isAuth = !!user;

        if (isAuth) return true;
        if (!isAuth) return this.router.createUrlTree(["auth"]);
      })
    );
  }
}
```

app-routing.module.ts file:

- Add the `canActivate: [AuthGaurd]` as an attribute on the "bookshelf" and "library" pages.

```
{
    path: 'bookshelf',
    component: BookshelfComponent,
    canActivate: [AuthGaurd],
    // . . .
},
// . . .
{ path: 'library', component: LibraryComponent, canActivate: [AuthGaurd]
},
```

Extra Credit/Time - (Instructors Push these to Github!)

STEP 1: Adding Angular Environment Variables for API_KEYS

environments/environment.ts & environments/environments.prod.ts files:

- In the root of our application, there is a file titled "environments". Go inside there an open up the "environment.ts" file.
- Under the `production: false,` property and value, add a new property and value for the "firebaseAPIKey".
- *Note*: You will need to use your personal firebase API Key... The one in this code has changed since creating this tutorial.
- Copy this and place it in the "environments/environments.prod.ts" file.

```
// * environments.ts * \\
export const environment = {
  production: false,
  firebaseAPIKey: "AIzaSyC1PGMcG", // Put your key here!
};
```

```
// * environments.prod.ts * \\
export const environment = {
  production: true,
  firebaseAPIKey: "AIzaSyC1PGMcG", // Put same key here!
};
```

shared/auth/auth.service.ts:

- Import `{ environment }` from "src/environment/environment"
- Delete our AUTH_API_KEY and replace the instances with `environment.firebaseAPIKey`.

```
import { environment } from 'src/environments/environment';

// . . .

signUp(email: string, password: string) {
    return this.http
      .post<AuthResponseData>(SIGN_UP_URL + environment.firebaseAPIKey, {
// . . .
    });
}
// . . .
signIn(email: string, password: string) {
    return this.http
      .post<AuthResponseData>(SIGN_IN_URL + environment.firebaseAPIKey, {
//. . .
    })
}
```

STEP 2: Adding Auto-Login & Auto-Logout Functionality

GOAL: When the user refreshes the page, keep their authentication status active.

shared/auth/auth.service.ts file:

- Create a "UserData" Interface.
- Create a new method `automaticSignIn()` that grabs the "userData" from localStorage, checks to see if it is a valid user, sets a new refresh date, and authenticates this user again.

```
export interface UserData {
  email: string;
  id: string;
  _token: string;
  _tokenExpirationDate: string;
}

// . . .

automaticSignIn() {
    const userData: UserData =
JSON.parse(localStorage.getItem('userData'));

    if (!userData) return;
    const { email, id, _token, _tokenExpirationDate } = userData;

    const loadedUser = new User(
      email,
      id,
      _token,
      new Date(_tokenExpirationDate)
    );

    if (loadedUser.token) {
      this.currentUser.next(loadedUser);

    //   const expDuration =
    //     new Date(_tokenExpirationDate).getTime() - new
Date().getTime();
    //   this.automaticSignOut(expDuration);
    }
}
```

app.component.ts file:

- Inject the "AuthService" inside the constructor.
- Inside `ngOnInit()`, use the `authService.automaticSignIn()` method.

```
export class AppComponent implements OnInit {
  constructor(private authService: AuthService) {}

  ngOnInit() {
    this.authService.automaticSignIn();
  }
}
```

shared/auth/auth.service.ts file:

- Create a new private variable "tokenExpTimer" of type "any".

- Create an `automaticSignOut(expDuration: number)` method that signs the user out when their expirationDate is exceeded.
- In the `signOut()` method, remove the "userData" from localStorage, clear the "tokenExpTimer" and set it to null.
- In the `handleAuth()` method, before you set the "userData", call `this.automaticSignOut(expiresIn * 1000)` method to start the timer.
- In the `automaticSignIn()` method, once ensured the user is valid, call the `this.automaticSignOut(expDuration: number)` method.
- *Note*: We can check if this is working by temporarily setting the "expDuration" to 2000.

```
export class AuthService {
  private tokenExpTimer: any;

  // . . .

  signOut() {
    this.currentUser.next(null);

    localStorage.removeItem("userData");

    if (this.tokenExpTimer) clearTimeout(this.tokenExpTimer);

    this.router.navigate(["auth"]);
  }

  // . . .

  automaticSignOut(expDuration: number) {
    console.log("Expiration Duration:", expDuration);

    this.tokenExpTimer = setTimeout(() => {
      this.signOut();
    }, expDuration);
  }

  handleAuth(email: string, userId: string, token: string, expiresIn:
number) {
    // . . .

    // Sets a new timer for the expiration token
    this.automaticSignOut(expiresIn * 1000);

    // . . .
  }
}
```

Additional Notes

Resources

- StackOverflow - Javascript Client Side vs Server Side Validation
- Blog - RESTful API's
- JSON Web Tokens - Introduction