Using Firebase for Front- and Backend Authentication

There is no doubt that Firebase changed the way authentication is performed in all sorts of applications and ever since its acquisition by Google this has been no exception, allowing anyone with a google account to easily login to any website. So in this tutorial, I want to cover how to use firebase for both Front- and Backend authentication using the JWT tokens generated by the Firebase API.
We will be using Java with Spring Boot for our Backend Application and Angular for the frontend.

Please note that this can look like a complicated process but don't worry since I will explain much of what I used to complete this tutorial. I tried to make it as short as possible while providing you with the most amount of information.

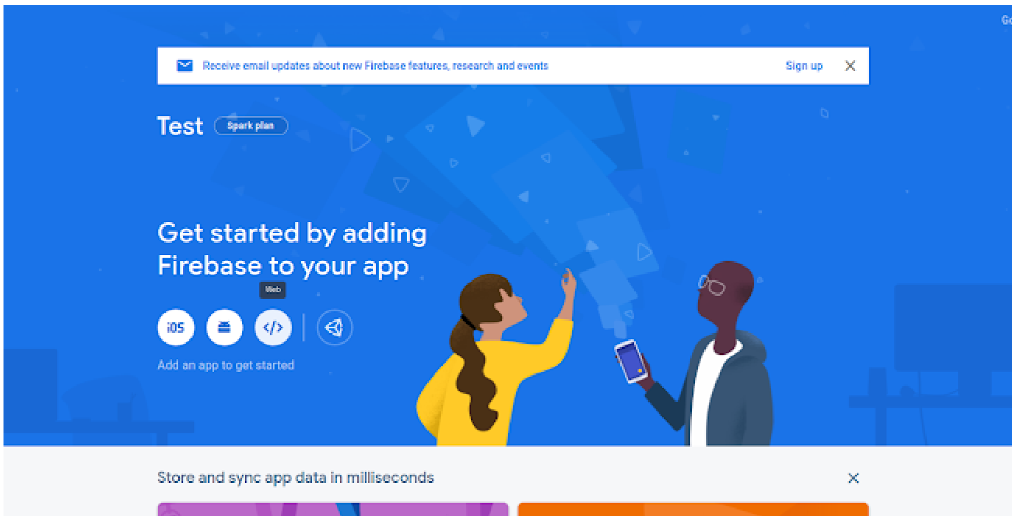
Also please note that **all of the code I show throughout this tutorial will be on GitHub** at the time of writing this article, so please head on over to GitHub if you want to follow along with your favorite IDE beside you. You can find the link at the end of the tutorial in the links section.

So without further ado, let's start configuring what you will need for your project.

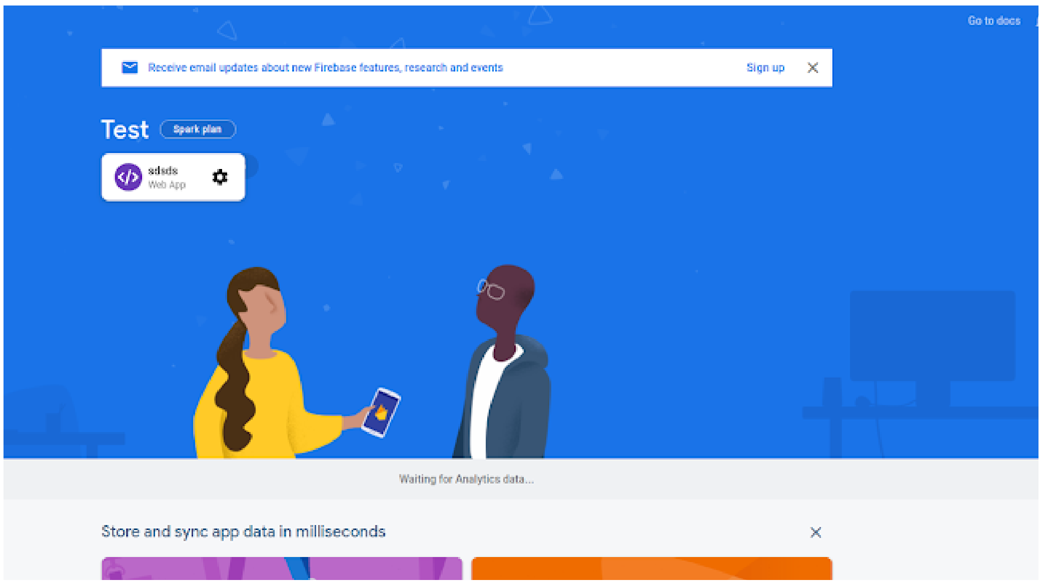
(At the time of writing this tutorial I am using java 11.0.8 and Angular CLI 11.1.4 with Node 14.15.1)

Firebase Configuration

If you have never set up a Firebase Project you will need to set one up now. Don't worry, it won't take long.
Head over to the ***Firebase Console*** website and create a Google account or use one of your existing accounts to sign in. Please have in mind that if you are creating a website it is a good idea to keep your account and your website's account separate, so feel free to create a google account solely for that purpose.
Once you're done, add a new project and go through the steps of creating it. It will automatically create a new project for you. After you have completed all the steps, you will be presented with the console where you can check out all the stats about your account. Nice!
Now let's add a new Web App to serve your Website (select the **</>** icon to create one).
Here is an example of how to do so:


Receive email updates about new Firebase features, research and events
Sign up
Test
Spark plan
Get started by adding Firebase to your app
Web
iOS
Add an app to get started
Store and sync app data in milliseconds

You can click next on all the steps for now, since you can change everything through the console later.
Now let's check out the settings of our newly created app. You can do it as shown in the image below.


Go to docs
Receive email updates about new Firebase features, research and events
Sign up
Test
Spark plan
sdsds
Web App
Waiting for Analytics data...
Store and sync app data in milliseconds

Now, head over to the ***Service Accounts*** tab and click on ***Generate new private key***. It will download a JSON file containing all the credentials you need for your backend. It will look something like this:

```
{
  "type": "service_account",
  "project_id": "name of your project",
  "private_key_id": "the keys id",
  "private_key": "Private Key for your website",
  "client_email": "clients email",
  "client_id": "clients id",
  "auth_uri": "authentication uri",
  "token_uri": "token uri",
  "auth_provider_x509_cert_url": "auth provider for certificate",
  "client_x509_cert_url": "place where the certificate is stored"
}
```

You can save this for now since you will need to use it later for your java backend.

We can also generate the necessary configuration information for the frontend that we will use later for the Angular frontend. To do so, head over to the ***General*** tab and scroll down until you find the ***Firebase SDK snippet***. Select the ***Config*** option that will give you all the information you need to set up the angular firebase project later. In angular we will include this in the environment.ts file so it should look something like this:

```
 firebaseConfig: {
     apiKey: "key",
     authDomain: "domain",
     databaseURL: "url",
     projectId: "id",
     storageBucket: "storage",
     messagingSenderId: "message id",
     appId: "app id",
     measurementId: "measurement"
 }
```

Save the `firebaseConfig` for later, since you will need it for the Firebase authentication in your frontend.

And that should be it for the Firebase Console! Now let's focus our attention on the Java backend Application.

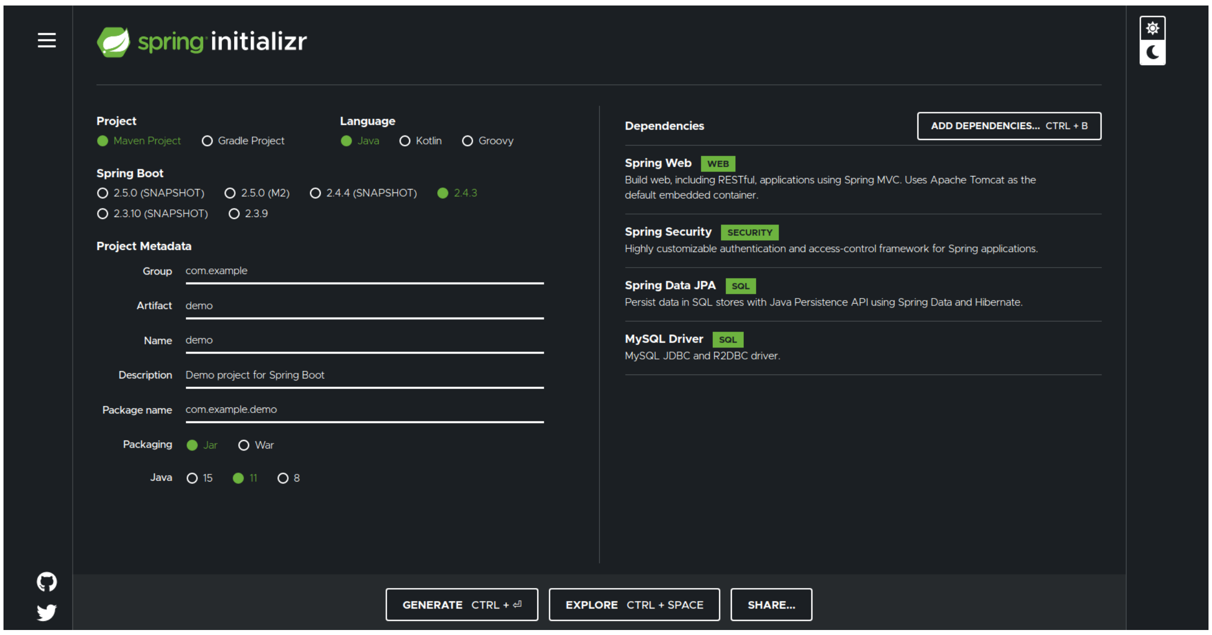
Java Backend

Setup

Let's start by introducing our Java backend.
To create our java backend application I use ***Spring Initializr*** at https://start.spring.io/ since it makes it easy to generate a project that already includes most of the dependencies we need for spring. If you are familiar with this process you can skip this part. Just be sure to add all the dependencies in your project's `pom.xml`.
On the Spring Initializr website, we will need to add some dependencies: ***Spring Web*** used to build an API with REST, ***Spring Security*** for securing our application, the ***MySQL Driver***

for communicating with the database, and ***Spring Data JPA*** since I will use JPA to handle all the communication with the database. Alternatively, you can use something else but I decided to use JPA since it makes database communication much easier.
Don’t forget to name your project as well, since it is much easier to do it now than later. Once you are done, you can generate the project and it will create a zip file containing your newly created Spring project.

And here is an example of how it will look once you are finished:



Project
[✓ Maven Project] Gradle Project

Language
[✓ Java] Kotlin Groovy

Spring Boot
2.5.0 (SNAPSHOT) 2.5.0 (M2) 2.4.4 (SNAPSHOT) [✓ 2.4.3] 2.3.10 (SNAPSHOT) 2.3.9

Project Metadata

Group	com.example
Artifact	demo
Name	demo
Description	Demo project for Spring Boot
Package name	com.example.demo
Packaging	[✓ Jar] War
Java	15 [✓ 11] 8

Dependencies

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Security SECURITY
Highly customizable authentication and access-control framework for Spring applications.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

MySQL Driver SQL
MySQL JDBC and R2DBC driver.

Now, there are some dependencies you can’t add through Spring Initializr so we will add them manually. So open up the project with your favorite IDE and let’s start coding!

To add these dependencies find the `pom.xml` file located in the root folder of your project. You will find a section called dependencies, where you want to insert the following dependencies:

```xml
<!-- https://firebase.google.com/docs/server/setup -->
<dependency>
      <groupId>com.google.firebase</groupId>
      <artifactId>firebase-admin</artifactId>
      <version>7.1.0</version>
</dependency>

<!-- Firebase Authentication -->
<dependency>
      <groupId>com.google.firebase</groupId>
      <artifactId>firebase-server-sdk</artifactId>
      <version>3.0.3</version>
</dependency>
```

The first dependency will allow you to communicate the Firebase database to access its storage. The second one allows you to access the Firebase Admin interface. We will use the second dependency the most since we will need to verify tokens and access user information.
I decided to add the first dependency in case you want to later use firebase for more than just user storage. This way, you can contact the Firebase database from your backend.

That should complete our Java backend setup! We can now focus on the code that makes up our backend application.

Code

Let's take a look at what we will need in terms of code to make it all work. To do this, I recommend creating a package called **auth** and inside that package create another one called **firebase** in your project where all the firebase components will reside. I would also create a package called **config** to store all the classes we will need for Web security configuration.
Your folder structure should now look something like this:

```
> auth
  > firebase
> config
```

Let’s start with the `config` package. Here we want to create a java class to deal with all incoming requests and also to allow us to bypass CORS, the Cross-Origin Resource Sharing, to allow our application to accept incoming requests. We can call this class `WebConfig`. Below is an implementation of that class:

config/WebConfig.class

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackageClasses = {WebConfig.class})
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("*")
                .allowedMethods("HEAD", "OPTIONS", "GET", "POST", "PUT",
"PATCH", "DELETE");
    }
}
```

You can see that we are allowing all origins to contact your application and accepting all methods one could send to it, which will prevent any CORS errors when you later try to connect your Angular frontend to your java backend.
You can also see that there are some annotations in the method. The @Configuration annotation tells the Spring container to scan for Beans and service requests to those beans. In this case, WebMvcConfigurer has @Bean definition methods so we need to include that annotation. @EnableWebMvc is used to import Spring MVC configuration and @ComponentScan(basePackageClasses = {WebConfig.class}) is used to allow Spring to auto-detect beans. The basePackageClasses annotates which classes should be scanned for decorated beans.

We also need a class to configure Web Security. We can create a new class in the config package called WebSecurityConfig. This class will act as a filter for all incoming requests to the application.

config/WebSecurityConfig.class

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled =
true)
@Order(SecurityProperties.BASIC_AUTH_ORDER)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Value("${rs.pscode.firebase.enabled}")
    private boolean firebaseEnabled;

    @Autowired
    private UserService userService;

    @Autowired
    private FirebaseAuthenticationProvider authenticationProvider;

    @Bean
    @Override
    public AuthenticationManager authenticationManager() throws
Exception {
        return new
ProviderManager(Arrays.asList(authenticationProvider));
    }

    private FirebaseFilter tokenAuthorizationFilter() {
        return new FirebaseFilter(userService);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        if(firebaseEnabled) {
            http.addFilterBefore(tokenAuthorizationFilter(),
BasicAuthenticationFilter.class)
                    .authorizeRequests()
                    .and()
                    .cors()
                    .and()
                    // we don't need CSRF because our token is
invulnerable
                    .csrf().disable()
                    // All urls must be authenticated (filter for token
always fires (/**)
                    .authorizeRequests()
                    .antMatchers(HttpMethod.OPTIONS).permitAll()
                    .antMatchers("/auth/**").authenticated()
                    .and()
                    // don't create session

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELE
SS); //.and()
        } else {
            http.httpBasic().disable()
                    .csrf().disable()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELE
SS)
                    .and()
                    .authorizeRequests()
                    .anyRequest().permitAll();
        }
    }
}
```

Let's quickly analyze this class. The ***configure*** method will filter out any HTTP requests and in case we enable firebase through the **`firebaseEnabled`** variable, it will set the ***FirebaseFilter*** class to act as a filter for any incoming requests, which we will create later. In this tutorial, I won't go much in-depth about the rest of the configuration for HttpSecurity. If you want a separate tutorial on that let me know over on the **Contacts** section of the website (→https://somnus.ddns.net/contacts).
Here we are also creating an **`authenticationManager`** Bean that uses a class called **`FirebaseAuthenticationProvider`** which we will also create later.
If you are wondering about the **`firebaseEnabled`** variable, I decided to store its value in the **`application.properties`** file, so that if you later want to separate your developing and production environments you can very quickly disable it in development for testing purposes. Here is a snippet from the application.properties file:

src/main/resources/application.properties

```
# MySQL driver
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

## Jpa
spring.jpa.database-platform=org.hibernate.dialect.MySQL5Dialect

# Enable Firebase config
rs.pscode.firebase.enabled=true

security.oauth2.resource.filter-order=3

# Jpa
spring.jpa.hibernate.ddl-auto=update

server.port=8080
```

Now let's head over to the **auth/firebase** package to create all the classes needed by Firebase to interact with your backend.

Let’s start by taking a look at the **`FirebaseFilter`** class. Below is the code for that class:

auth/firebase/FirebaseFilter.class

```
public class FirebaseFilter extends OncePerRequestFilter {

    private UserService userService;

    public FirebaseFilter(UserService userService) {
        this.userService = userService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        String xAuth = request.getHeader("X-Authorization-Firebase");
        String loginHeader = request.getHeader("firebaseToken");
        if (StringUtils.isBlank(xAuth) ||
!StringUtils.isBlank(loginHeader)) {
            filterChain.doFilter(request, response);
            return;
        } else {
            try {
                FirebaseTokenHolder holder =
FirebaseParser.parseToken(xAuth);

                String userName = holder.getUid();

                Authentication auth = new
FirebaseAuthenticationToken(userService.loadUserByUsername(userName),
holder);

SecurityContextHolder.getContext().setAuthentication(auth);

                filterChain.doFilter(request, response);
            } catch (BadCredentialsException e) {
                throw new SecurityException(e);
            }
        }
    }

}
```

Our custom filter extends the **`OncePerRequestFilter`** class, which, as the name suggests, is a filter that will be applied only once for every request made to the server.
This class imports another class called **`UserService`**, which is a service created to handle information about users. This is passed through the class **`WebSecurityConfig`** class, which I have shown you earlier.
This class has a **`doFilterInternal`** method which we will override to create our own custom JWT token filter. Let's analyze the method so that you can understand it.
This method will look for a header field with the key **`X-Authorization-Firebase`**, which is the name I gave to the header field but you can customize it at will. This field contains the JWT token for firebase and it will be used for authentication purposes. This method will also look for a header field with the name **`firebaseToken`**. This field will be used by the **`UserController`**, which I will introduce later, to authenticate the user for the first time he authenticates with the backend. All the remaining requests will only come with the X-Authorization-Firebase field. The reason we need the first condition:

```
StringUtils.isBlank(xAuth) || !StringUtils.isBlank(loginHeader)
```

is because we don't want to process the firebase token from the user if he hasn't logged in yet.
The rest of the code just processes the token. The class **`FirebaseParser`** parses the token, then we get the user from the database through its username and pass it to the security context so that we can later inject a Principal in any controller, which will allow us to get the user if we need to. Don't worry, I will give some examples later.
Lastly, you'll want to pass the request to the **`filterChain`**. This is so that the request can continue to be processed by any other filters that come after the one we've just created.

We can also quickly take a look at the FirebaseParser class. This class parses the Firebase token and verifies if it is still valid. Here is a snippet from that:

auth/firebase/FirebaseParser.class

```
@Component
public class FirebaseParser {
    public static FirebaseTokenHolder parseToken(String idToken) {
        if (StringUtils.isBlank(idToken)) {
            throw new IllegalArgumentException("FirebaseTokenBlank");
        }
        try {
            FirebaseToken firebaseToken =
FirebaseAuth.getInstance().verifyIdToken(idToken);

            return new FirebaseTokenHolder(firebaseToken);
        } catch (Exception e) {
            throw new SomnusDemoException("Invalid Auth token");
}
    }
}
```

We also need a FirebaseTokenHolder class that will hold the token information for Firebase. This allows you to wrap the token in an object parsed by the `FirebaseParser` for easier access and maintainability.

auth/firebase/FirebaseTokenHolder.class

```
public class FirebaseTokenHolder {
      private FirebaseToken token;

      public FirebaseTokenHolder(FirebaseToken token) {
            this.token = token;
      }

      public String getEmail() {
            return token.getEmail();
      }

      public String getIssuer() {
            return token.getIssuer();
      }

      public String getName() {
            return token.getName();
      }

      public String getUid() {
            return token.getUid();
      }

      public String getGoogleId() {
            String userId = ((ArrayList<String>) ((ArrayMap) ((ArrayMap)
token.getClaims().get("firebase"))
                        .get("identities")).get("google.com")).get(0);

            return userId;
      }
}
```

Then we can create the **FirebaseAuthenticationToken** class, which will be used to store the authentication information in the security context in java. You can see it being used in the **doFilterInternal** method I have shown you before.

auth/firebase/FirebaseAuthenticationToken.class

```
public class FirebaseAuthenticationToken extends
AbstractAuthenticationToken {
      private static final long serialVersionUID =
-1869548136546750302L;
      private final Object principal;
      private Object credentials;

      /**
       * This constructor can be safely used by any code that wishes to
create a
       * <code>UsernamePasswordAuthenticationToken</code>, as the
       * {@link #isAuthenticated()} will return <code>false</code>.
       *
       */
      public FirebaseAuthenticationToken(Object principal, Object
credentials) {
            super(null);
            this.principal = principal;
            this.credentials = credentials;
            setAuthenticated(false);
      }

      /**
       * This constructor should only be used by
```

```
	 * <code>AuthenticationManager</code> or
<code>AuthenticationProvider</code>
	 * implementations that are satisfied with producing a trusted
(i.e.
	 * {@link #isAuthenticated()} = <code>true</code>) authentication
token.
	 *
	 * @param principal
	 * @param credentials
	 * @param authorities
	 */
	public FirebaseAuthenticationToken(Object principal, Object
credentials,
							Collection<?
extends GrantedAuthority> authorities) {
		super(authorities);
		this.principal = principal;
		this.credentials = credentials;
		super.setAuthenticated(true); // must use super, as we
override
	}


	public Object getCredentials() {
		return this.credentials;
	}

	public Object getPrincipal() {
		return this.principal;
	}

	public void setAuthenticated(boolean isAuthenticated) throws
IllegalArgumentException {
		if (isAuthenticated) {
			throw new IllegalArgumentException(
					"Cannot set this token to trusted - use
constructor which takes a GrantedAuthority list instead");
		}

		super.setAuthenticated(false);
	}

	@Override
	public void eraseCredentials() {
		super.eraseCredentials();
		credentials = null;
```

```
      }

}
```

We will also need the `FirebaseAuthenticationProvider` class which will authenticate the user when he tries to access the system with a token.

auth/Firebase/FirebaseAuthenticationProvider.class

```
@Component
public class FirebaseAuthenticationProvider implements
AuthenticationProvider {

      @Autowired
      private UserService userService;

      public boolean supports(Class<?> authentication) {
            return
(FirebaseAuthenticationToken.class.isAssignableFrom(authentication));
      }

      public Authentication authenticate(Authentication authentication)
throws AuthenticationException {
            if (!supports(authentication.getClass())) {
                  return null;
            }

            FirebaseAuthenticationToken authenticationToken =
(FirebaseAuthenticationToken) authentication;
            UserDetails userDetails =
userService.loadUserByUsername(authenticationToken.getName());
            if (userDetails == null) {
                  throw new SomnusDemoException("Firebase user does not
exist");
            }

            authenticationToken = new
FirebaseAuthenticationToken(userDetails,
authentication.getCredentials(),
                        userDetails.getAuthorities());

            return authenticationToken;
      }
}
```

Last but not least we will create a class called **`AuthenticationSecurity`**. This will extend the **`GlobalAuthenticationConfigurerAdapter`** class which sets the global authentication provider to be the **`FirebaseAuthenticationProvider`** which we created earlier.

<u>auth/AuthenticationSecurity.class</u>

```java
@Order(Ordered.HIGHEST_PRECEDENCE)
@Configuration
public class AuthenticationSecurity extends
GlobalAuthenticationConfigurerAdapter {

    @Autowired
    private UserDetailsService userService;

    @Value("${rs.pscode.firebase.enabled}")
    private Boolean firebaseEnabled;

    @Autowired
    private FirebaseAuthenticationProvider firebaseProvider;

    @Override
    public void init(AuthenticationManagerBuilder auth) throws Exception
{
        auth.userDetailsService(userService);
        if (firebaseEnabled) {
            auth.authenticationProvider(firebaseProvider);
        }
    }
}
```

That should do it for the Firebase and authentication part. Now let’s tie up some loose ends.

Users need to log in somehow, therefore we will create a **`UserController`** that will receive the authentication request from the frontend. We can create a **<u>users</u>** package to better organize our code.
The **`UserController`** class can be found below.

users/UserController.class

```
@RestController
@RequestMapping(value = "/user-api")
public class UserController {

    @Autowired
    private UserService userService;

    @PostMapping(value = "/authentication/authenticate-user")
    public UserDto authenticateUser(@RequestHeader String firebaseToken,
@RequestBody UserDto userDto){
        return userService.authenticateUser(firebaseToken, userDto);
    }
}
```

As you can probably notice you will need to create a UserService as well, so let's create that as well.

users/UserService.class

```
@Service
public class UserService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;


    @Override
    public User loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userRepository.findByUserName(username);
        if (user == null)
            throw new SomnusDemoException("No user Found");

        List<RoleEntity> grantedAuthorities = new ArrayList<>();
        for (GrantedAuthority role : user.getAuthorities()) {
            grantedAuthorities.add(new RoleEntity(role.getAuthority()));
        }

        user.setAuthorities(grantedAuthorities);
        return user;
    }

    public UserDto authenticateUser(String firebaseToken, UserDto
userDto) {
        if (StringUtils.isBlank(firebaseToken)) {
            throw new IllegalArgumentException("Blank Firebase Token");
        }
        FirebaseTokenHolder firebaseTokenHolder =
FirebaseParser.parseToken(firebaseToken);

        // add user to repository if it does not exist already
        User user =
userRepository.findByUserName(firebaseTokenHolder.getUid());
        if (user == null) {
            Pair<String, String> firstAndLastName=
parseFirstAndLastName(userDto);
            String firstName = firstAndLastName.getFirst();
            String lastName = firstAndLastName.getSecond();
            user = new User(firebaseTokenHolder.getUid(),
firebaseTokenHolder.getEmail(),
                    userDto.getDisplayName(), firstName, lastName,
rolesHandler.getRole(Role.USER),
                    Role.USER);
            userRepository.save(user);
        }

        // get user roles
        // List<String> roles = new ArrayList<>();
        // user.getAuthorities().stream().forEach(auth ->
roles.add(auth.getAuthority()));

        // create new UserDto
        return new UserDto(user.getId(), user.getUsername(),
user.getEmail(), user.getDisplayName(),
                user.getFirstName(), user.getLastName(),
user.getRole().name);
    }

}
```

We can quickly go over the **`UserService`** class.
The loadUserByUsername method is used to get a user from the database given its username. It returns an Exception in case no user is found. This method was used in the **`doFilterInternal`** method to get the users.
We will also need to create the **`authenticateUser`** method, which will allow us to authenticate a user. It will verify if the given token exists and parse it in case it does. It then tries to get a user from the repository and in case no user is found, it will create a new user with the information provided by the **`userDto`**. It will then return the user retrieved from the database back to the frontend in the form of a **`UserDto`** object.
The reason why the authentication method receives a **`userDto`** is that we need to receive

and store user information passed by the frontend, for example if the user is new to the system and wants to register. We then save the data to a local database which might not sound important now, since all user information is currently handled by firebase, but if we later want to add roles to the users or even just include them in other objects in our backend as attributes we will need to retrieve them from the database.
Furthermore, we want to return a `UserDto` if we later need to send more information about the user to the frontend, which might be the case later down the line.

You can find the classes for the `User` and `UserDto` over on github, so make sure you check those out.

Conclusion

Phew… That was a lot to take in. Don't worry, I also didn't learn it in one go, so take your time and do things at your speed. Let's now focus on the frontend part of this tutorial. Now that will be fun!

Angular Frontend

Now we can focus on the frontend part of our tutorial, the Angular project setup. Since you are doing authentication with Angular, I will assume you know a thing or two about Angular itself. So I won't go much into the basics on how to configure a new Angular Project. I will have a demo-project over on github with all the things I mention here in this tutorial.

Setting up Firebase

To set up the angular project, we will just need to install firebase and rxjs-compat (used for mergeMap function later). To do so, just execute the following command on your angular project folder:

```
npm i firebase @angular/fire --save
npm install rxjs-compat
```

Now Firebase is going to install and you should be up and running!

Code

So now that you have Firebase installed let's go over the things you will need.
Let's start by adding the necessary Firebase information to your angular environment. I will add them to the development environment, so be sure that if you ever enter production that you also change your `environment-prod.ts`. We will add the `firebaseConfig` which you have retrieved from Firebase in the beginning of this tutorial. The `environments.ts` should look something like this:

src/environments/environment.ts

```
export const environment = {
      production: false,
      firebaseConfig: {
            apiKey: "key",
            authDomain: "domain",
            databaseURL: "url",
            projectId: "id",
            storageBucket: "storage",
            messagingSenderId: "message id",
            appId: "app id",
            measurementId: "measurement"
      }
};
```

So now that we added that to your angular development environment we can start adding all the necessary classes for the firebase Authentication.

Lets first go over the `authentication.service.ts`. This service manages the google authentication in your whole project. Every time you need to access user Information, you can access it through here.
This service also stores the user information that you have sent from your backend. It stores it onto local storage and I have created methods to get that user information in case you need it in your application.
It supports two login methodologies, login with a google account, which corresponds to the `loginWithGoogle()` method, and login with an email and a password, which corresponds to the `loginWithEmail()` method.
The `createUser()` method offers you the possibility of having a custom form, where you can add more fields to your user in case you want to save them in your backend. I added a first name and a last name just as an example, but you can have any fields you like.
There is also a `logout()` method for you to logout your user if you want to. Just remember to also change the `userModel`, which I will introduce after, if you also want to send that information to your backend as well. Also be sure to change your `UserDto` and `User` classes in the backend.
This class is a bit more complex and I ended up deciding not to take up a lot of pages with it. So I recommend you go over to the GitHub page of the project if you want to take a good look at the code.

You will also need to create a User model to represent your user data coming from the backend. That should be pretty simple to create, so let's do it quickly:

src/app/models/user.model.ts

```
export interface UserModel {
    id?: number;
    username : string;
    displayName: string;
    email: string;
}
```

Let's quickly also add a User controller service, so that our **`authentication.service.ts`** can use it to contact the backend to authenticate users. This service adds a header field called ***firebaseToken***, which we also added in our backend. This will be received by the **`UserController`** in our backend and it will be used by the **`UserService`** to authenticate the user.
The code can be found below.

src/app/services/controllers/user-controller.service.ts

```
@Injectable({
  providedIn: 'root'
})
export class UserController {

  protected httpOptions = {
    headers: new HttpHeaders({
      "Content-Type": "application/json"
    })
  };

  constructor(
    protected http: HttpClient,
  ) { }

  public authenticateUser(token: string, user: UserModel):
Observable<any> {
    this.httpOptions = {
      headers: new HttpHeaders({
        "Content-Type": "application/json",
        "firebaseToken": token
      })
    };
    // contacts the User controller we have created
    return this.http.post("localhost:8080/auth/auth-user", user,
this.httpOptions);
  }
}
```

Now we need something to intercept your http requests to add the token to transport it to your backend for authentication. That is where the `CustomHttpInterceptor` comes into play. It will intercept all of your http requests and add the authentication token with the same header name as we added in the backend (`X-Authorization-Firebase`).
This will call the user service to get the users token and, if present, adds the corresponding headers to your http request. Otherwise, it just sends a request without any authorization headers.
Below is an implementation of that class.

src/handlers/custom-http-interceptor.ts

```
@Injectable()
export class CustomHttpInterceptor implements HttpInterceptor {

  constructor(
    private authService: AuthenticationService,
    private router: Router
  ) { }

  intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
    return this.authService.getToken().mergeMap((token: string | null)
=> {
      if (token) {
        const authReq = req.clone({
          headers: req.headers.set('X-Authorization-Firebase', token)
            .append('Access-Control-Allow-Origin', '*')
        });
        return next
          .handle(authReq)
          .pipe(catchError(
            (err: HttpErrorResponse) => {
              console.log(err);
              if (err.status === 401) {
                this.router.navigate(['/']);
                this.authService.logout();
                console.log("You are not authorized to perform that
task!");
              } else {
                console.log(err.message);
              }

              return throwError(err);
            }
          )) as Observable<HttpEvent<any>>;
      }
      // if there is no token, don't send it
      const authReq = req.clone({
        headers: req.headers.set('Access-Control-Allow-Origin', '*')
      });
      return next
        .handle(authReq)
        .pipe(catchError(
          (err: HttpErrorResponse) => {
            console.log(err);
            if (err.status === 401) {
              this.router.navigate(['/']);
              console.log("You are not authorized to perform that
task!");
            } else {
              console.log(err.message);
            }

            return throwError(err);
          }
        )) as Observable<HttpEvent<any>>;
    });
  }
}
```

We also need something called an **AuthGuard** service. This service is used to manage user access to certain components in your angular website. In this case, we will make a simple AuthGuard that will only prevent non logged-in users to access certain components in our application. You can later also add roles to the AuthGuard Service, so that only users with a certain role can access that component. If you want to know more about how to set up roles with angular, I can have a separate tutorial on that as well. Let me know over on the contacts section.
The code for the **AuthGuard** can be found below.

src/app/services/authentication/auth-guard.service.ts

```
@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {
    constructor(
        private router: Router,
        private authenticationService: AuthenticationService,
    ) {}

    async canActivate(route: ActivatedRouteSnapshot, state:
RouterStateSnapshot) {
        const user = await this.authenticationService.isLoggedIn();
        if (user) {
            // authorised so return true
            return true;
        }

        // not logged in so redirect to home page page
        // here you could redirect your user to the login page for
example
        this.router.navigate(['/'], { queryParams: { returnUrl:
state.url }});
        return false;
    }
}
```

You can use the AuthGuard to restrict other component usages by using something like the example below in your `app-routing.module.ts`.

src/app/app-routing.module.ts

```
 {
     path: "page-path",
     component: NameOfYourComponent,
     canActivate: [AuthGuard]
 }
```

In this case the `NameOfYourComponent` component can only be accessed if your AuthGuard allows it. You can do this to any components in your application

That should be it for the Frontend. But I will leave you some extras as well.

Extras

I have also thrown a global error handler for you. You can find it at **src/handlers/global-error-handler.ts** if you search it on github. This will allow you to handle all errors in one place and help you debug your app better.

I have also provided a demo component over at **components/demo** to show you how you can import the user from any component in your angular project.

"One more thing …"

Now there are a couple of things I want to discuss with you just to make everything clear.

Remember at the beginning of the tutorial when I told you to generate the JSON file containing all the credentials that you need for your backend? That file can be used if you ever want to contact firebase to store any data or to read data from the firestore. Here we are just parsing tokens coming from the frontend. Those tokens already came from the firebase API as you saw from the `authentication.service.ts` I have shown you earlier.

Also if you want a better perspective of the whole code head over to GitHub to check it out, as it will not only give you a much better view of the whole code structure, but also have some extra things you need to get your code up and running.

I haven't created much of the UI part of Angular since my purpose was just to show you what happens behind the hood. If you would like a more in-depth tutorial on how to integrate everything with google material design and make good-looking login pages, I will leave some links below.

Also if you happen to come across some mistakes in this tutorial please let me know so that I can correct them. You can let me know of any mistakes by contacting me through the somnus website's contacts section (→ https://somnus.ddns.net/contacts)

Links

GitHub - https://github.com/somnus-admin/somnus-authentication-tutorial
Angular Material Design - https://material.angular.io/
Learn more about Angular Login and Registration -
https://jasonwatmore.com/post/2020/07/18/angular-10-user-registration-and-login-example-tutorial

Many Thanks to:
Savicprvoslav on github, for doing an amazing job at configuring Firebase with java backend. Without him I could never have done this tutorial.