Ghent University

Cloud Applications and Mobile (E736010)

Lab 5 Egress: Part2 - Securing Web Services

Professor:
Prof. dr. ir. Sofie Van Hoecke

Assistants:
Dr. Jennifer B. Sartor
Ing. Tom Windels

2024–2025

Table of contents

1 Introduction

During this lab session, we will extend the egress API and secure it using a token-based authentication.

First you will configure Ktors' JWT library for our egress, responsible for generating JSON web tokens (JWT). These tokens can authenticate a user, just like a username and password would. You will have to still implement two new endpoints, `/register` and `/login`, because a user needs to register and login before a token will be given to them. We'll then have the existing endpoints of our egress API protected by these tokens, a.k.a., a user has to authorize himself by providing a valid token to issue GET commands on our egress endpoints.

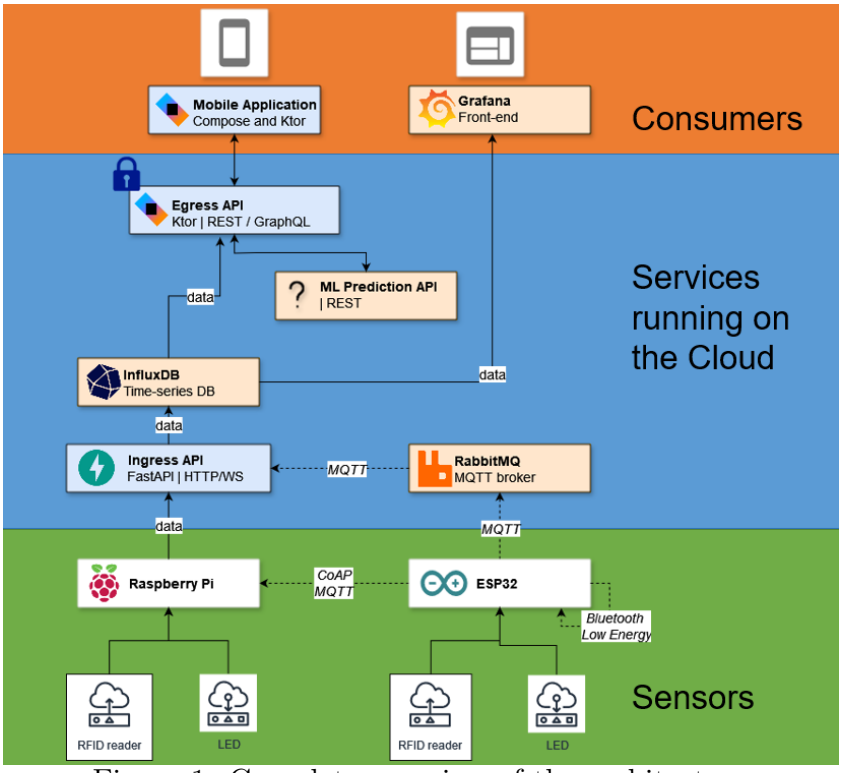

Figure 1: Complete overview of the architecture

2 Goals

1. Provide basic authentication to your egress API using JSON Web Tokens (JWT).
2. Register and log users in using a database in the cloud.

3 Preparation

3.1 Accessing the MariaDB database in the cluster

In this lab, we will need users to authenticate themselves, and thus we need a database to hold our users list. That database needs to be in the cloud.

To enable data persistence regarding user registration and authentication, a MariaDB service has been deployed within our Kubernetes cluster. This database can be accessed from within the cluster itself. To test your access to the MariaDB database used throughout this lab session, execute the following commands in your terminal using your own credentials:

```
# Creating a MariaDB client pod in your namespace and logging into it
kubectl run mariadb-client --rm --tty -i --restart='Never' --image
 docker.io/bitnami/mariadb:10.5.7-debian-10-r0 --command -- bash

# Logging into the MariaDB database from within that pod
mysql -h mariadb.database.svc.cluster.local -u USERNAME -p DATABASE
```

Note: Restart policy "Never" might not work on Windows devices. In case an error occurs, simply omit the restart parameter.

If anything else goes wrong during this process, please contact one of the lab supervisors.

3.2 Getting familiar with Ktor's JWT

For the token-based authentication throughout this lab session, we will generate JSON Web Signatures (JWS). In essence, JWS allows us to encrypt a JSON-based data structure using a digital signature. So we will encrypt a JSON data structure to generate our API tokens. We have to make sure our tokens are unique, so we will use the user's username as our data structure that will be encrypted, and JWS uses extra headers to make sure the resulting token is unique (a configuration secret key that we will provide and a timestamp).

In Kotlin using Ktor, this can easily be achieved using their implementation, named JSON Web Tokens (JWT). You should read through this JWT tutorial thoroughly to know how to program in Ktor with these tokens.

4 Tutorials

4.1 Patching security into the project

Your existing egress implementation has to be patched to add the initial security code. The following subsection goes over the code that is added. The patch can be found on the online learning platform. Applying this patch through IntelliJ is straightforward: navigate through "Git" > "Patch" > "Apply Patch" or "VCS" > "Apply Patch" and select the downloaded file. A new menu opens up showing all files and it's contents that will be added by applying the patch. Simply press "OK" with all changes selected to import the patch into your code base.

The patch includes changes to `build.gradle.kts`, the configuration file describing the build progress. This requires a "Gradle Sync" before the IDE can provide features like the ability to run and auto-complete. Syncing the project can be done by clicking the "Gradle Sync" button typically located at the top right of the IDE, or clicking "Gradle" (elephant logo) > "Reload

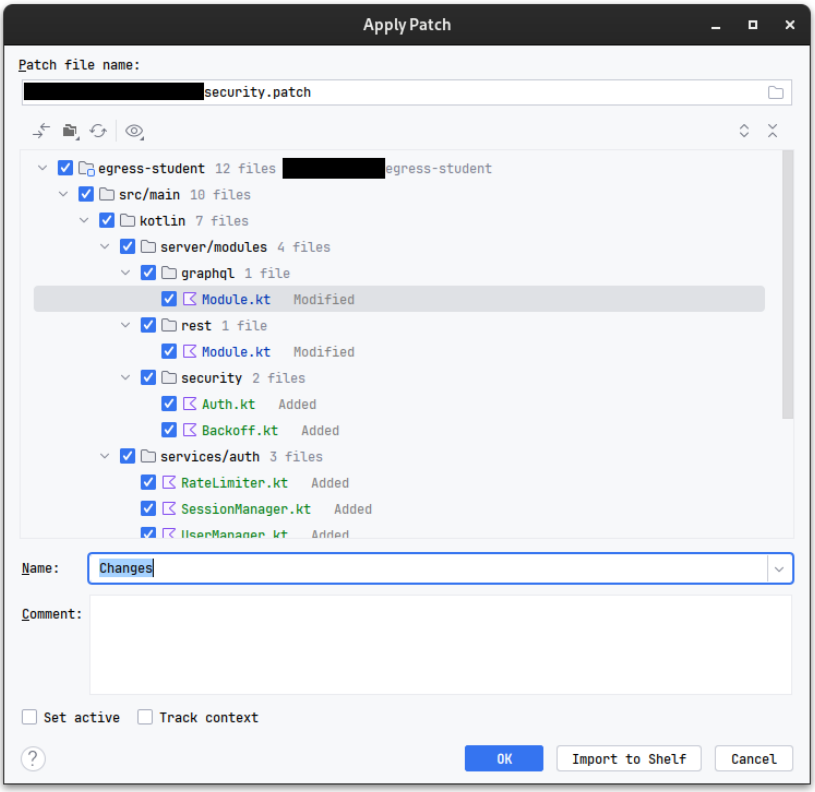

Figure 2: Applying the security patch through IntelliJ

All Gradle Projects" (refresh icon or download icon depending on your IDE version) found typically in the right toolbar.

4.2 Security code added to Egress

We will add two new endpoints to your egress API. One to register and one to login to your secured egress. Registration, in this case, requires providing a unique username and password, which will be hashed, and making sure no user exists with those credentials. Logging in will entail checking the password provided against the hashed password stored in the database. After a user has been verified, a security token has to be generated.

```kotlin
object SessionManager {
    data object AlreadyExists: RuntimeException("Username already taken")
    data object InvalidCredentials: RuntimeException("Invalid credentials")

    @Serializable
    data class Credentials(
        val username: String,
        val password: String
    )

    @Serializable
    data class Token(
        val token: String
    )
    val secret = properties.resolve("jwt.secret")
    val issuer = properties.resolve("jwt.issuer")
    val audience = properties.resolve("jwt.audience")
    val expiry = properties.resolve("jwt.expiry").toLong()

    /**
     * Creates a user account using the credentials found in the body
     */
    suspend fun createAccount(credentials: Credentials): Result<Unit> {
        LOGGER.info("Incoming register attempt for `${credentials.username}`")
        TODO("Pass the credentials to UserManager to create an account,
        returning a failure if no account has been created")
    }

    /**
     * Creates a token result if the incoming call contains correct
     credentials, or a failure with matching exception
     */
    suspend fun processLogin(credentials: Credentials): Result<Token> {
        LOGGER.info("Incoming login attempt for `${credentials.username}`")
        TODO("Check the credentials using UserManager, only generating a token
        if valid")
    }
```

Listing 1: Code snippet from `services/auth/SessionManager.kt`

As mentioned above, we will be using Ktor's JWT library. We will use this library in `/services/auth/SessionManager.kt`, part of which is shown in Listing 1. You can see classes that represent the body of a login attempt or registration attempt, called `Credentials`, and a token. In the `createAccount` function, a user will register their new credentials. After a login operation is when you will have to generate a new security token. You will generate the token using the properties specified in `/resources/auth.properties`: `jwt.secret`, `jwt.issuer`, `jwt.audience`, and `jwt.expiry`. The secret used to sign the tokens should be unique, but can be any random string. Also notice that we have to specify an encryption algorithm to use, and we will use `HMAC256` with the secret. You will have to include the user's username in the token as well. The issuer is mandatory and is specified as a string that identifies the principal that issued the JWT. The audience identifies the recipients that the JWT is intended for. For our purposes, these two fields are mandatory to include, but we won't use their abilities to limit the scope of where the token can be used. The `jwt.expiry` field holds the number of milliseconds until this token will expire. You will have to look at the documentation to know how to configure and generate JSON web tokens.

In the file `/services/auth/UserManager.kt`, everything is handled to talk to our MariaDB in the cloud, and the important parts are shown in Listing 2. Notice we have a `UserTable` object, as shown below, created for the fields of a new database record that encapsulates a new user, with username and hashed password. In this file, we also see the logic to create a new user, check if a user's credentials are correct, and generally connect with the database.

```kotlin
object UserTable: Table() {
    val username = varchar("username", 50)
    val hash = binary("hash", 60)
    override val primaryKey = PrimaryKey(username)
}

/**
* Creates a user with the given `name` and `password`. This operation
may fail, in which case `false` is returned.
* The most common cause for a failure is name collision (name
already in use).
*/
suspend fun create(name: String, password: String): Boolean { ... }

/**
* Checks the provided `name` and `password`, returning `true` if
the credentials are correct.
*/
suspend fun check(name: String, password: String): Boolean { ... }
```

Listing 2: Code snippet from `services/auth/UserManager.kt`

In the `/server/modules/security/Auth.kt` file, you have to configure your token verifier and validate a particular payload. The verifier function allows you to verify a token format and its signature, while the validate function allows you to perform additional validations on the JWT payload (such as checking the username). You should also include a challenge function that allows you to configure a response to be sent if authentication fails.

Finally, your existing egress endpoints have been protected by wrapping them in a call to `authenticate`. You can see this in `/server/modules/rest/Module.kt` and `/server/modules/graphql/Module.kt`. In `/server/modules/security/Auth.kt` we can see in the `authenticate` block in `configureSessionEndpoints` that if authentication is successful, we can retrieve the username from the token.

5 Tasks

5.1 Task 1: Configure JWT

In this task, you will be configuring JWT so you will be able to generate security tokens and use them to secure your existing egress endpoints. For this, you will modify the `/server/modules/security/Auth.kt` file. Inside the `configureAuth` function, you have to add verify, validate, and challenge functions. These will configure a token verifier, validate the JWT payload, and define a response if authentication fails, respectively.

5.2 Task 2: Add a registration endpoint

In this task, you will add a registration endpoint. In the `/server/modules/security/Auth.kt` file, you will have to fill in the contents of the `post("/register")` function. This will have to call the `createAccount` function defined in `/services/auth/SessionManager.kt` that also needs to be implemented. Given the passed in credentials, the `createAccount` function should create the user's account by communicating with the database, making sure to do error checking.

We suggest testing with Postman. Follow the guide to download Postman, then send an API request to your API to make sure it works as expected, also testing faulty input.

5.3 Task 3: Add a login endpoint

In this task, you will add a login endpoint. In the `/server/modules/security/Auth.kt` file, you will have to fill in the contents of the `post("/login")` function. This will have to call the `processLogin` function defined in `/services/auth/SessionManager.kt` that also needs to be implemented. Given the passed in credentials, the `processLogin` function should check if this user is entering valid credentials, and if so, generate a token and return it. You can throw an error if the credentials are incorrect. You can also use Postman to test this endpoint.

5.4 BONUS Task 4: Implement IP ban or backoff to prevent attacks

It is possible for bad actors to spam various combinations of usernames and passwords to the login endpoint, trying to break into existing user's accounts. This causes two problems: it can find a match, resulting in an account being compromised, as well as overload the cloud infrastructure as the egress code will send out database requests for every attempt. Other bad calls, such as unauthorized requests or using bad query parameters, can also negatively impact the infrastructure. To mitigate this, a rate limiter plugin "Backoff" for our Ktor application will be created. The rules for when and how long hosts are banned from making requests are already implemented in `/services/auth/RateLimiter.kt`, and can be accessed through that class.

In `/server/modules/security/Backoff.kt`, the backoff plugin implementation is still missing, and for this bonus task you need to implement it. This plugin is required to do two things: observe call responses made by the contacted endpoint to see whether the call was a valid request, and intercept and stop incoming requests from hosts that are currently banned from making requests, responding with such a message instead of processing the request properly. It is important for the plugin to fully halt the processing of requests from banned hosts, as otherwise the infrastructure would still suffer from request spam.

6 Questions

Answer the following (open) questions in your lab report:

1. From a consumer point-of-view, e.g., a mobile app that wants to visualize the sensors exposed by the egress API, what has changed in the requests they make between the version of the egress API in Part1 and the extended version from this lab session (Part2)?
2. How would you protect the `/register` endpoint to prevent account creation spam as much as possible?
3. How would you handle banning users, meaning, revoking their tokens?

7 Summary

In part 2 of this lab, we looked at securing our Egress API. We looked into generating unique API tokens for users, and making users authorize themselves before they could access our egress endpoints. In this way, we are securing our egress web service before we open it up to be used by consumers.

8 Material to submit

Prepare a lab report that elaborates on what you have done for **both Part1 and Part2 of the Egress lab**. First, include any screenshots or explanations of problems that you had with the preparation and tutorial sections of both write-ups. Then explain each of the solved tasks (6 obligatory tasks from Part1 and 3 in Part2), including your code in the report and a short description of how you solved each task. Include proof, in the form of screenshots or videos, that you got each task working. Include the answers to the questions from both Part1 and Part2's write-up in your report. Only the **.pdf** extension is allowed for the report. Name your lab report **LAB5_FamilyName_FirstName.pdf**. Archive the report together with the source code for the exercises and videos, and name this archive **Lab5_FamilyName_FirstName.zip**. Hand everything in using Ufora. You can hand everything in multiple times, only the last file will be saved.