Web Applications 2

Lab 2: the basics of HTTP server scalability

Learning Objectives

By successfully completing this lab, students will develop the following skills:

- Using GitHub as a repository and a collaboration tool
- Creating and deploying a simple web application
- Generating and validating Json Web Tokens (JWTs) using different libraries
- Writing unit and integration tests and interpreting their results
- Assessing the scalability of of a web end-point and have a better understanding of the Universal Scalability Law

Description

The most demanding operation in a digital ticketing system, from the scalability point of view, is ticket validation. This operation, in fact, is executed each time passengers enter into or exit form a subway station, which, for a large town, may account to several thousands requests per second during rush hours.
We will assume that a digital ticket consists of a QR-code encoding a JSON Web Token (RFC 7519: https://tools.ietf.org/html/rfc7519, tutorial, docs and libraries: https://jwt.io/). A JWT is a dot-separated string consisting of an header (a base64-encoded json object describing the signature algorithm and token type), a payload (a base64-encoded json object describing the token claims), and a signature (a base64-encoded byte array).
Automatic gates at subway stations are equipped with an optical scanner that allows passengers to show their mobile phone displaying the ticket QR-code, read it and post a request to the subway ticketing system containing the station zone (a letter) and the scanned code. Upon reception of the request, the ticket-validation service which will perform the following actions:

1. decode the received http body content and validate the signature of the token
2. access the payload and control that the expiry timestamp (named “exp”) is still valid
3. access the payload and control that the ticket validity zones (named “vz”) includes the reported station zone
4. return a success code (200, Ok) if all checks succeed, or a failure (403, forbidden), if any check fails.

Steps

1. Select one group member as coordinator and ask her/him to create a GitHub private repository and to grant access permissions to all other group members. You may need to create free personal accounts, if you don’t have one, yet. All produced code has to be committed to this repository.

2. Run IntelliJ IDEA and select File → New → Project from Version Control…: Select enter the repository URL in the dialog box and proceed.

The repository will contain two modules: a Spring Boot web application named “server” and a node.js benchmarking application named “benchmark”. Create the first module via File → New → Module…: choose Spring Initializr, name it “server”, opt for the Kotlin language and the Gradle build tool. In the next screen select the Spring Web library dependency. Commit and push it, by adding a suitable message.
Create a second module: choose JavaScript / Node.js and name it “benchmark”. Commit and push.

3. Implement the given system by creating a Spring Web application that accepts post requests to the /validate endpoint. It will accept a json encoded object shaped as follows:
```
{
    zone: '1',
    token: "eeyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. //continues…
    eyJzdWIiOjEyMzQ1Njc4OSwidnoiOiIxMjMiLCJpYXQiOjE2NDkyMzkwMjJ9.
       HzJ1j3PFcpXkokSN_8SgP3mQfhT8QQjPgIoHUyV3eAI"
}
```
   and will return an empty payload with either code 200 or code 403.
   In order to manipulate the JWT, it is convenient to use the Java JWT library (https://github.com/jwtk/jjwt). Please note that there exists a previous version of the library (0.9.1) which is NOT compatible with the newer versions. When searching for code examples, please mind the version number.
   Commit code and push it to the repository.

4. Write a set of tests that let you control that all conditions are properly enforced.
   This means creating two kinds of test: a set of unit tests, assessing the proper behavior of the business logic, and a set of integration tests that look at the created service as a black box and only access its public interface.
   Unit tests operate on a white-box basis: they typically operate on single objects defined inside the application and verify that their methods conform to the expected behavior.
   Here is an example of a Spring Boot unit test verifying an hypothetical object named TicketingService:

```
@SpringBootTest
class UnitTests() {
    @Autowired
    lateinit var ticketingService: TicketingService

    @Test
    fun rejectInvalidJWT() {
        Assertions.assertThrows(ValidationException::class.java) {
            ticketingService.validateTicket("1", "aaa.bbb.ccc")
        }
    }
}
```

Integration tests, conversely, evaluate the compliance of a sub-system or component with specified functional requirements. Here is an example of a Spring Boot integration test verifying the behavior of the whole module:

```
@SpringBootTest(
  webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
class ServerApplicationTests() {
    @LocalServerPort
    protected var port: Int = 0

    @Autowired
    lateinit var restTemplate: TestRestTemplate

    @Test
    fun rejectInvalidZone() {
        val baseUrl = "http://localhost:$port"
        val request = HttpEntity(TicketInput("","aaa.bbb.ccc"))
        val response = restTemplate.postForEntity<Unit>(
            "$baseUrl/verify",
            request )
        assert(response.statusCode == HttpStatus.FORBIDDEN)
    }
}
```

Verify that all tests pass, then commit and push.

5. In order to test the scalability of the server, the *loadtest* tool will be used (https://www.npmjs.com/package/loadtest). This is an Apache Bench (ab) clone written in JavaScript and running in NodeJs, that allows you to customize the payload sent to the server by writing a simple javascript common module (also known as NodeJs module) exporting a single, zero-arguments, function. It will be invoked once per each request, in order to get a custom payload to submit to the server. In the benchmark module, create a function that will return a random, valid or invalid, request to the server. (Use Npm module 'jsonwebtoken' to create a JWT).
Configure the *loadtest* tool to run 10,000 requests using respectively 1, 2, 4, 8, 16, or 32 threads and record the reported number of Requests Per Second.
By using the usl4j library ("com.codahale:usl4j:0.7.0") create a csv file reporting the expected system behavior at increasing concurrency level (a real system has several thousands gates that might fire requests concurrently) and make a plot of it.

6. Modify the server, adding an extra constraint: check that the same ticket (as identified by the "sub" field in the jwt payload) has not yet been used. (Caution: the system is concurrent, and the check must be thread safe).
Add tests to enforce the condition. Check tests, commit
Estimate again the scalability of the server, using the same procedure as above and create a new plot. How does it compare to the previous one? Why?

7. Try tweaking with the various http flags (connection keep-alive, time limit, …) offered by the *loadtest* tool. How do they impact the results? Why?

Submission rules

Download a copy of the zipped repository and upload it in the “Elaborati” section of “Portale della didattica”. Label your file “Lab2-Group<N>.zip” (one copy, only - not one per each team member).
Work is due by Friday April 1, 23.59 for odd numbered teams, and by Friday April 8, 23.59 for even numbered ones.