Web Applications 2

Lab 3: User registration

Learning Objectives

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

- Creating and deploying a stateful web application relying on a RDBMS
- Using JPA to save, retrieve, update, and delete entities
- Managing relationships between entities
- Creating a scheduled task to periodically clean entities
- Using the testcontainer library to support integration tests
- Using a rate limiter to prevent denial-of-service attacks
- Using docker to manage component deployment

Description

In order to provide personalized content, a digital ticketing system needs to manage a set of users, securely storing their credentials and providing a way to recover them in case they are lost. This, in turn, entails that users have the ability to request a new registration, by posting a minimal set of personal information (chosen username, password, email address) to the "/user/register" endpoint: as a consequence, some information will be stored and a validation process that will either lead to accept the supplied information, thus completing the registration, or to remove the request, dropping any supplied data, will be started.
Since the registration process is open to the general public, care has to be taken to prevent any form of abuse, in particular, the possibility of overloading the system by continuously submitting new pieces of information. To this purpose, a rate-limiter will be applied to the exposed end-point, so that - if more than a given number of requests is received in a second, they will be rejected with status code 429 - Too Many Requests.
In order to prevent inconsistent data to be submitted, several checks will be enforced:

- username, password, and email address cannot be empty;
- username and email address must be unique system-wide;
- password must be reasonably strong (it must not contain any whitespace, it must be at least 8 characters long, it must contain at least one digit, one uppercase letter, one lowercase letter, one non alphanumeric character);
- email address must be valid.

If any check fails, the request will be rejected with status code 400, Bad Request; otherwise an entry will be stored in a suitable db table, containing the uploaded data, together with the indication that the record is not yet active, since there is no proof, yet, that the applicant has control of the submitted email address. A provisional random ID will be returned with status code 202, Accepted.
In order to validate the given email address, when data have been recorded, a random activation code will be generated and stored in another db table, together with the provisional random ID, an expected activation deadline, an attempt counter, and a reference to the user record; moreover an email message will be sent to the provided address containing the activation code.

If, within the expected time, a registration completion request will be posted to the "/user/validate" endpoint containing the provisional random ID and the corresponding activation code, the user record will be transitioned to the active state, the corresponding record in the activation table will be removed, and status code 201, Created, will be returned, together with the definitive UserDTO. If the request will be received after the expiration of the deadline, the activation record will be removed from the activation table, and status code 404 will be returned. If the provisional random ID does not exist, status code 404 is returned. If the provisional random ID exists but the activation code does not match the expected one, status code 404 will be returned and the attempt counter will be decremented: if it reaches 0, the activation record will be removed together with the User entry.
Rate limiting will also be applied to this endpoint.
In order to prevent a malevolent client from submitting information that is never validated, a scheduled task will be periodically run, removing all expired validation records and the corresponding pending user entities.

Steps

1. Create a repository and start a new Spring Boot project including Spring WebMVC, Spring Data JPA, and Java Mail Sender. Commit and push.

2. Make sure to have Doker Desktop running (or install it if necessary) and create a container with the official postgres image (https://hub.docker.com/_/postgres). By default, all files created inside a container are stored on a writable container layer. This means that data doesn't persist when that container no longer exists. It may be convenient to create a local folder where the data table will be stored.
   After that, create the container adding a '-v' flag to map a container folder with the host folder created before, and a '-p' flag to map the host port with the container port in order to enable communications outside of the container (see https://sqlkitty.com/docker-windows-persistent-postgresql/). Using the database explorer panel, verify that the database is up and running and that it can be properly connected via the Query Console. Configure the application.properties file with the spring.datasource.* properties properly set for your database installation. Add also the following property which makes Hibernate generate better SQL for the chosen database:
   `spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect`
   Compile and run the project and verify that it can properly connect the DBMS and no exception is thrown. Commit and push.

3. In order to store the requested information, two tables are needed: one for storing user data, the other for storing activation codes. Create two classes, User and Activation, annotated with @Entity, and insert the necessary properties.
   The User class will have a @GeneratedValue @Id of type Long.
   The Activation class will have a property of type User, decorated with the @OneToOne annotation. Its @Id property will be of type UUID and it will represent the provisional id used during the registration process, moreover, it will contain an attempt counter, initially set to 5.

Set the spring.jpa.hibernate.ddl-auto property to “create”, compile and run the project and verify that the corresponding two tables have been created in the DBMS. Then, set the spring.jpa.hibernate.ddl-auto property to “validate”, restart the project and check that no exception is thrown. Commit and push.

4. Create two more classes, UserDTO and ActivationDTO and define two extension functions, User.toDTO() and Activation.toDTO(), in charge of converting an entity in the corresponding DTO.
   Define also two new interfaces: UserRepository and ActivationRepository, which both extend CrudRepository.
   Add a set of unit tests that check that the extension functions properly work as expected. Run the tests, check that everything is ok, then commit and push.

5. The business logic is contained in a couple of services: EmailService, which is responsible for sending email messages, and UserService that manages the registration process. The latter will have the former as dependency, as well as the two repositories.
   Implement the two classes and write a set of unit tests, by properly mocking their dependencies, in order to be able to verify the correct implementation of the business logic. When a registration token expires, both the user record and the activation record must be removed from the DB. In order to periodically prune expired registration data, a scheduled job can be used (see https://reflectoring.io/spring-scheduler/) . Run the tests, check that everything is ok, then commit and push.

6. Implement the given system by creating a Spring Web REST controller that accepts post requests to the /user/register endpoint. It will accept a json encoded object shaped as follows:
   ```
   {
       nickname: “somename”,
       password: “Secret!Password1”,
       email: “me@email.com”
   }
   ```
   and will return, in case of success, an json object like this:
   ```
   {
       provisional_id: “eda6bff4-cc1e-46be-80fe-b5a59fcc75e3”,
       email: “me@email.com”
   }
   ```
   The controller will also accept post requests to the /user/validate endpoint: it will accept json objects like
   ```
   {
       provisional_id: “eda6bff4-cc1e-46be-80fe-b5a59fcc75e3”,
       activation_code: “123456”
   }
   ```
   and returns, in case of success objects like:
   ```
   {
       userId: 123,
   ```

```
    nickname: “somename”,
    email: “me@email.com”,
}
```

7. Write a set of integration tests that let you control that the system is behaving as expected. Since the system is now stateful and depends on the information stored in the database, it is necessary to configure it in a way that allows tests to be run against a known data set.
   To this purpose, the testcontainers library (https://testcontainers.org) is used: it relies on Docker to provide lightweight, throwaway instances of common databases and other infrastructural components that can be run in a Docker container.
   Add the following dependencies to the build.gradle.kts file:
   ```
   testImplementation ("org.testcontainers:junit-jupiter:1.16.3")
   testImplementation("org.testcontainers:postgresql:1.16.3")
   ```
   Add also the following block to the same file:
   ```
    dependencyManagement {
      imports {
       mavenBom("org.testcontainers:testcontainers-bom:1.16.3")
      }
    }
   ```
   This will provide the support for creating a data access layer integration test.
   The basic structure of a test class is reported here:

```
@Testcontainers
@SpringBootTest(webEnvironment=SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureTestDatabase(replace=AutoConfigureTestDatabase.Replace.NONE)
class DbT1ApplicationTests {
  companion object {
    @Container
    val postgres = PostgreSQLContainer("postgres:latest")

    @JvmStatic
    @DynamicPropertySource
    fun properties(registry: DynamicPropertyRegistry) {
      registry.add("spring.datasource.url", postgres::getJdbcUrl)
      registry.add("spring.datasource.username", postgres::getUsername)
      registry.add("spring.datasource.password", postgres::getPassword)
      registry.add("spring.jpa.hibernate.ddl-auto") {"create-drop"}
    }
  }
  @LocalServerPort
  protected var port: Int = 0
  @Autowired
  lateinit var restTemplate: TestRestTemplate
  @Autowired
  lateinit var userRepository: UserRepository

  @Test
  fun someTest() {
     //Write here your integration test
```

```
  }
}
```

Verify that all tests pass, then commit and push.

8. Since the two endpoints need to be exposed on the public internet, a rate limiting strategy will be adopted to restrict the number of API calls that a client can make within a given timeframe.
   Library Bucket4J (https://github.com/vladimir-bukhtoyarov/bucket4j) will be used.
   While this kind of constraint can be applied directly inside each controller method, we will leverage on the fact that the Spring Framework offers a more general solution to factor-out common handler code via interceptors: these are classes that implement the HandlerInterceptor interface or extend the HandlerInterceptorAdapter class. When the DispatcherServlet receives a request, it sends it to the targeted controller method, first querying a configurable set of interceptors.
   Create a class named RateLimiterInterceptor that extends HandlerInterceptorAdapter and override the preHandle(...) method. Following the instruction contained in this example (https://www.baeldung.com/spring-bucket4j), create a bucket and apply a rate limit of 10 requests per second.
   Add the interceptor to the InterceptorRegistry, by properly creating a class that implements the WebMvcConfigurer interface, specifying the URL “/users/**” as a path pattern.
   Add a set of integration tests that verify that the rate limiting conditions properly work.
   Commit and push.

Submission rules

Download a copy of the zipped repository and upload it in the “Elaborati” section of “Portale della didattica”. Label your file “Lab3-Group<N>.zip” (one copy, only - not one per each team member).
Work is due by Friday April 22, 23.59 for odd numbered teams, and by Friday April 29, 23.59 for even numbered ones.