Web Applications 2

Lab 3: Ticket management

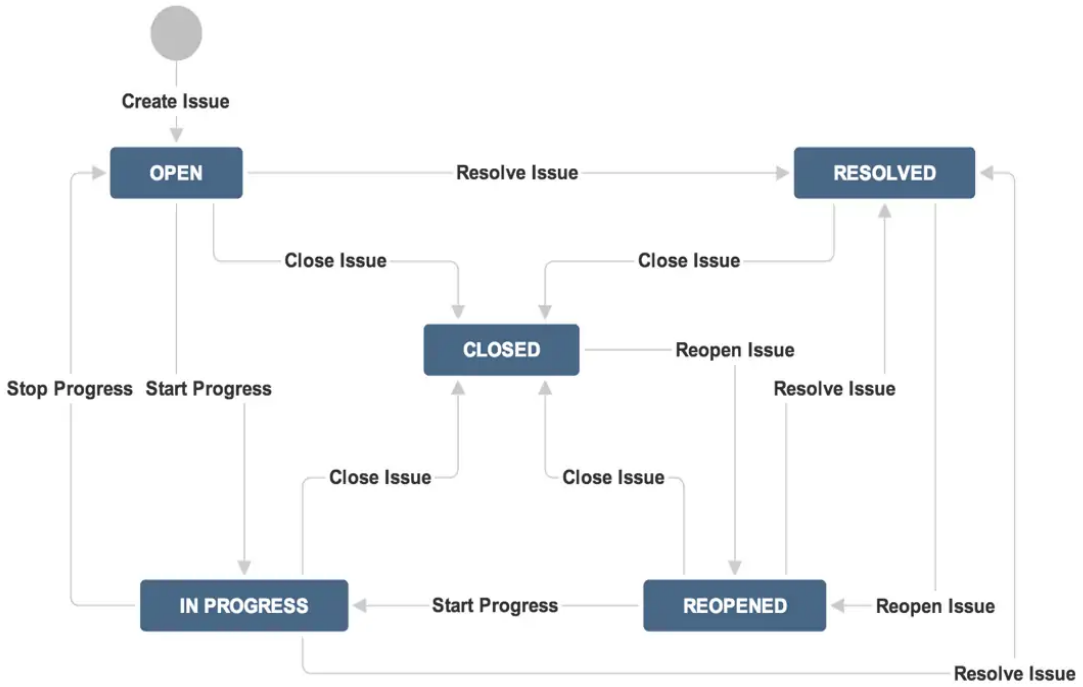
Learning Objectives

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
- Using the testcontainer library to support integration tests
- Using docker to manage component deployment

Description

In order to provide customer support, a digital ticketing system is needed. A customer creates a new ticket describing the issues encountered. Once the ticket is received, its status becomes IN PROGRESS and a priority level is setted when a manager assigns it to an expert. After being assigned, the two sides may start chatting. A chat is composed of multiple messages, and a message can contain some attachments. In this phase tickets can change status according to the following schema.



The history of state changes should be saved so that managers can evaluate the efficiency of the operators.

Steps

1. Starting from lab 2 repository, a member of the team creates a new module called ticketing. This module needs Spring Web, Spring Data JPA, PostgresSQL Driver. Commit following the Conventional Commits specification: https://www.conventionalcommits.org/en/v1.0.0/ and push.
2. Before starting to implement the new module it would be advisable for the team to start schematizing the APIs and data structures necessary to follow the functional requirements. Better to spend more time in the design phase than refactoring code and tests.
3. In order to store the requested information, some tables are needed. Create the needed classes annotated with @Entity. In case a table needs a primary key self-generated, use @GeneratedValue @Id annotations. Relationships between Java objects are provided through annotations (one-to-one, one-to-many, many-to-one, many-to-many)
   https://www.section.io/engineering-education/introduction-spring-data/
4. 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.

5. 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.

6. Microservices are deployed as containers, so you can launch them comfortably without having to manage dependencies. Create a docker image of your new microservice using jib https://cloud.google.com/java/getting-started/jib and run it locally in order to check if it works. 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 May 05, 23.59 for odd numbered teams, and by Friday May 12, 23.59 for even numbered ones.