Furniture Inventory Management System

Coriciuc Tudor, Staci Ana – Iulia, Tănase Iosif – Daniel

Project Summary

- **Core Features:**
  - *User Registration & Authentication* – Secure sign-up and login system using JWT
  - *Furniture Inventory Management* – Add, update, delete and view furniture items + view low - stock items
  - *Supplier Management* – Add, update, delete and view suppliers + view furniture items by supplier
  - *Category Management* – Add, update, delete and view furniture categories
  - *Stock Movement Management* – Add, update, delete and view stock movement
  - *Monitoring Dashboard* – Real-time system and API performance indicators

- **Technologies used:** Java + SpringBoot, MongoDB, Docker, Jenkins, Grafana + Prometheus + Alertmanager

API Endpoints

Operations for *Authentication*:

- POST/auth/register
- POST/auth/login

CRUD Operations for *Users*:

- GET/users
- GET/users/{id}
- PUT/users
- DELETE/users/{id}
- DELETE/users

CRUD Operations for *Category*:

- GET/category/{code}
- GET/category
- POST/category
- PUT/category/{code}
- DELETE/category/{code}
- DELETE/category

CRUD Operations for *Furniture*:

- GET/furniture/{sku}
- GET/furniture
- POST/furniture
- PUT/furniture/{sku}
- DELETE/furniture/{sku}
- DELETE/furniture
- GET/furniture/low-stock – returns all furniture that is low in stock

API Endpoints

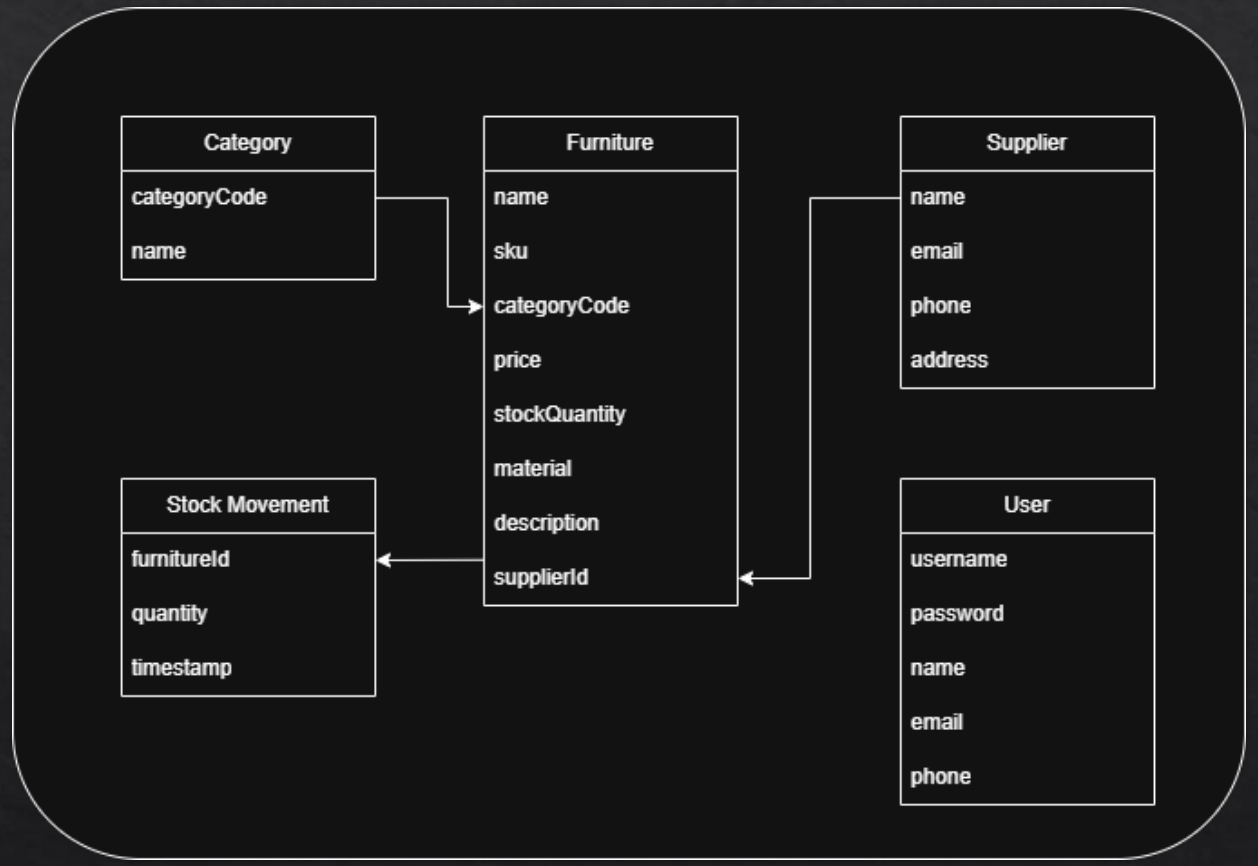
CRUD Operations for *Stock Movement*:

- GET/stock_movement
- GET/stock_movement/{id}
- POST/stock_movement
- POST/stock_movement/batch
- PUT/stock_movement/{id}
- DELETE/stock_movement/{id}
- DELETE/stock_movement

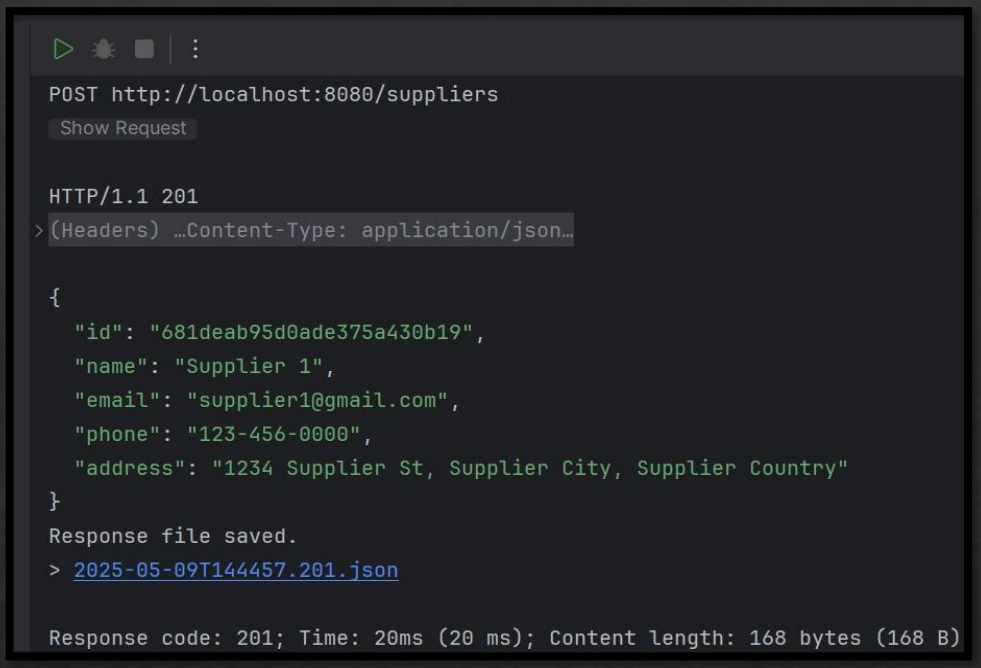
CRUD Operations for *Suppliers*:

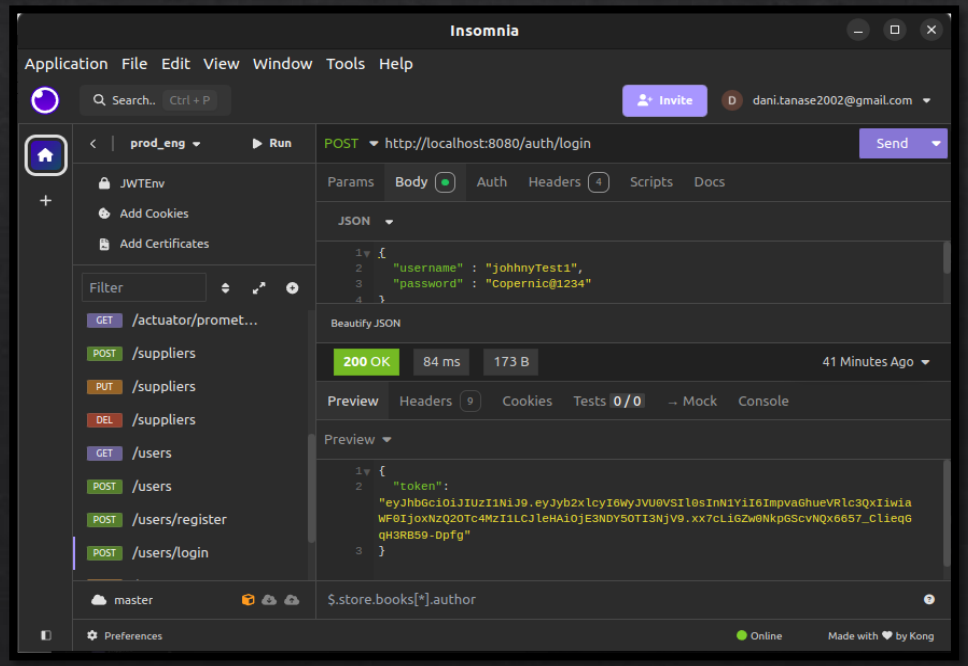
- GET/suppliers
- GET/suppliers/{name}
- POST/suppliers
- POST/suppliers/batch
- PUT/suppliers/{id}
- DELETE/suppliers
- DELETE/suppliers/{id}
- GET/suppliers/product-counts – returns the total of each product a supplier has

Database Diagram



API Testing





- For the API testing, we used HTTP Client in IntelliJ IDEA and Insomnia. These are used to make HTTP requests and to analyze the output so that we can determine if the API requests are working as intended.

Unit Testing - JUnit

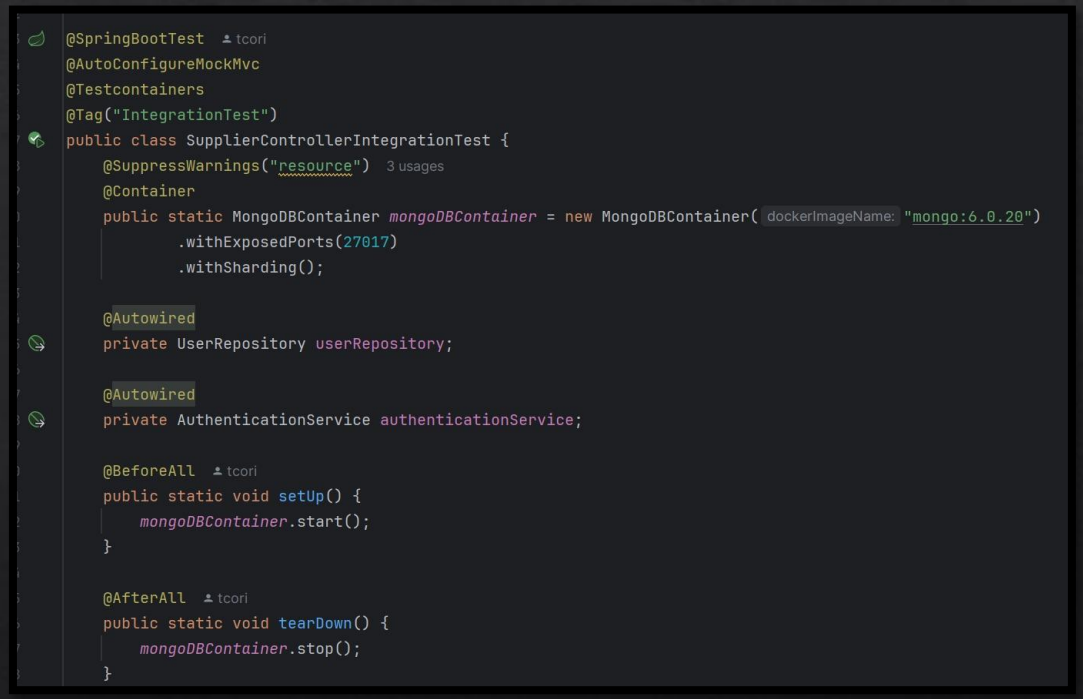
- To validate the functionality of a method, we implemented unit tests using JUnit. A unit test focuses on verifying the behavior of a single, isolated component of the application, without involving external dependencies, by mocking them. This way we try to ensure that all possible cases are covered.
- As a result, we achieved a **test coverage of 78% for the application's methods**.

```java
class CategoryControllerTest {  Ana Iulia Staci

    @Mock  14 usages
    private CategoryService categoryService;

    @InjectMocks  1 usage
    private CategoryController categoryController;

    private MockMvc mockMvc;  14 usages

    @BeforeEach  Ana Iulia Staci
    public void setUp() {
        MockitoAnnotations.openMocks( testClass: this);
        mockMvc = MockMvcBuilders.standaloneSetup(categoryController).setControllerAdvice(new GlobalExceptionFilter()).build();
    }

    @Test  Ana Iulia Staci
    void test_getCategoryByCode() throws Exception {
        // Arrange
        CreateCategory category = new CreateCategory( code: 1,  name: "Category 1");
        when(categoryService.getCategoryByCode(1)).thenReturn(category);

        // Act & Assert
        mockMvc.perform(get( uriTemplate: "/categories/1"))
                .andExpect(status().isOk())
                .andExpect(jsonPath( expression: "$.categoryCode").value( expectedValue: 1))
                .andExpect(jsonPath( expression: "$.name").value( expectedValue: "Category 1"));
    }
```

Element	Class, %	Method, %	Line, %	Branch, %
ro.unibuc.hello	84% (38/45)	78% (282/3...	80% (790/...	61% (59/96)
config	0% (0/3)	0% (0/24)	0% (0/63)	0% (0/4)
controller	100% (7/7)	100% (43/...	99% (146/1...	95% (23/24)
data	85% (6/7)	66% (61/92)	68% (131/1...	100% (0/0)
dto	100% (14/14)	95% (115/1...	95% (236/...	100% (0/0)
enums	100% (1/1)	100% (2/2)	100% (5/5)	100% (0/0)
exception	100% (2/2)	66% (2/3)	66% (4/6)	100% (0/0)
filters	50% (1/2)	33% (2/6)	24% (8/33)	0% (0/10)
service	87% (7/8)	87% (57/65)	90% (260/...	62% (36/58)

Integration Testing

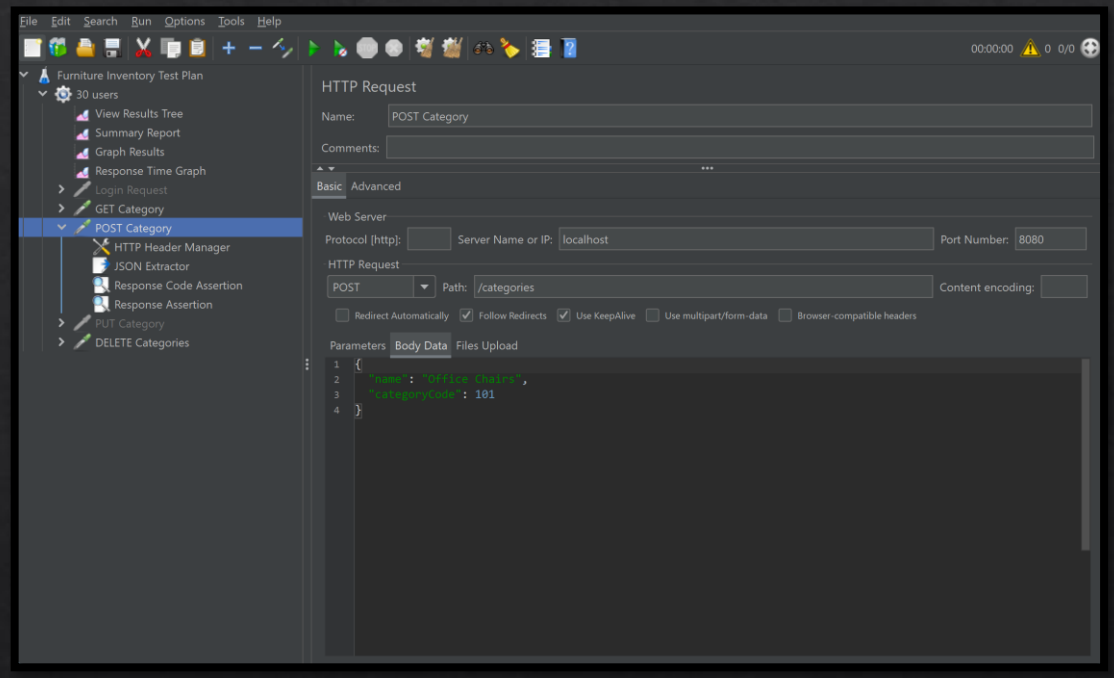
- Integration tests verify how multiple components work together in a real environment. Using **Testcontainers**, we dynamically start a MongoDB container to provide an isolated and consistent database for testing. This ensures that API endpoints, services, and database interactions are properly validated.



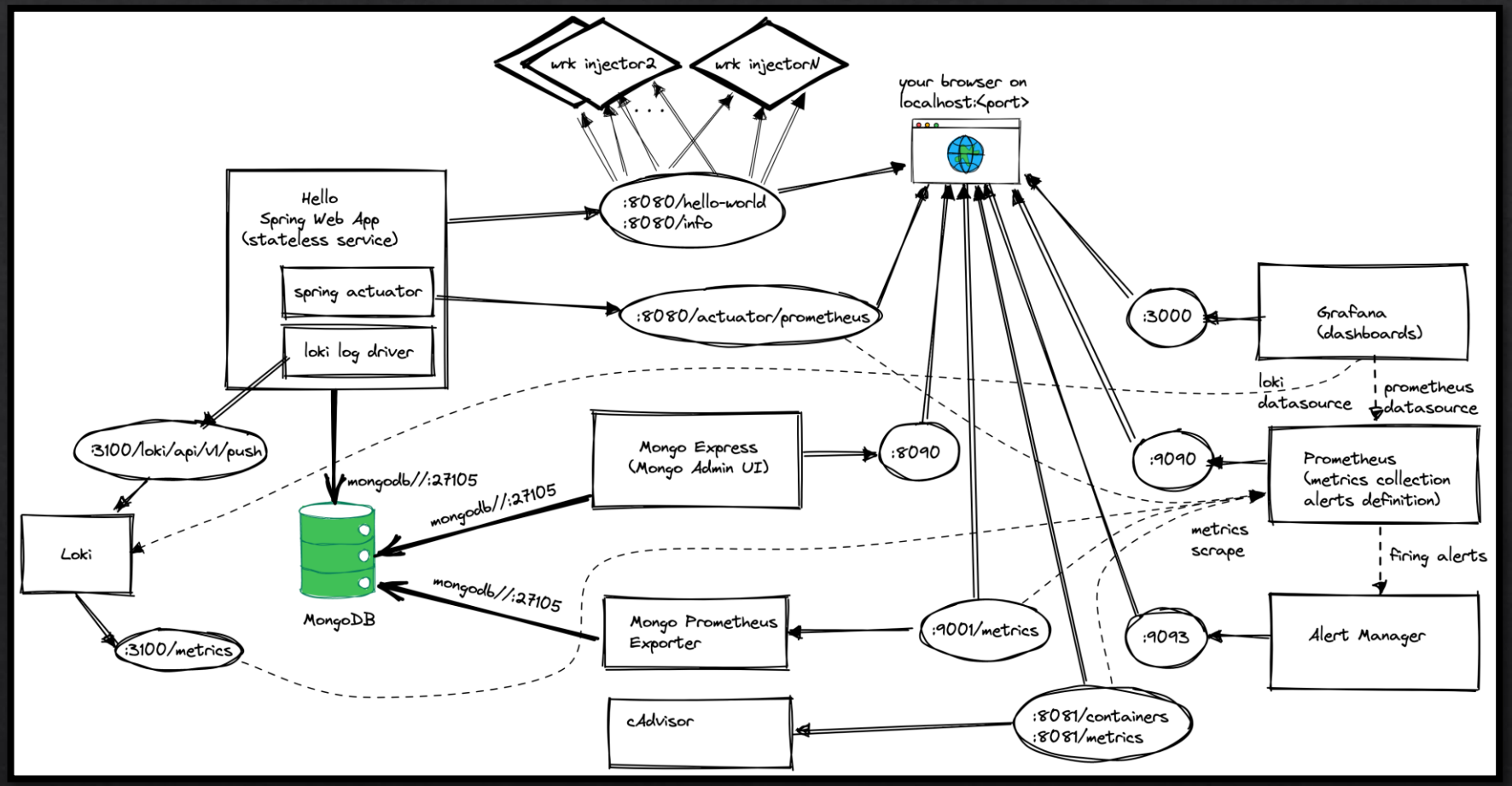
```java
@SpringBootTest  tcori
@AutoConfigureMockMvc
@Testcontainers
@Tag("IntegrationTest")
public class SupplierControllerIntegrationTest {
    @SuppressWarnings("resource")  3 usages
    @Container
    public static MongoDBContainer mongoDBContainer = new MongoDBContainer( dockerImageName: "mongo:6.0.20")
            .withExposedPorts(27017)
            .withSharding();

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private AuthenticationService authenticationService;

    @BeforeAll  tcori
    public static void setUp() {
        mongoDBContainer.start();
    }

    @AfterAll  tcori
    public static void tearDown() {
        mongoDBContainer.stop();
    }
```

Performance Testing - JMeter

- We performed performance testing using Apache JMeter to assess how the system handles concurrent API requests. The screenshot shows a test case for the *POST /categories* endpoint, where we simulate creating a new furniture category under load. This helps measure response times and system stability during data insertion operations.



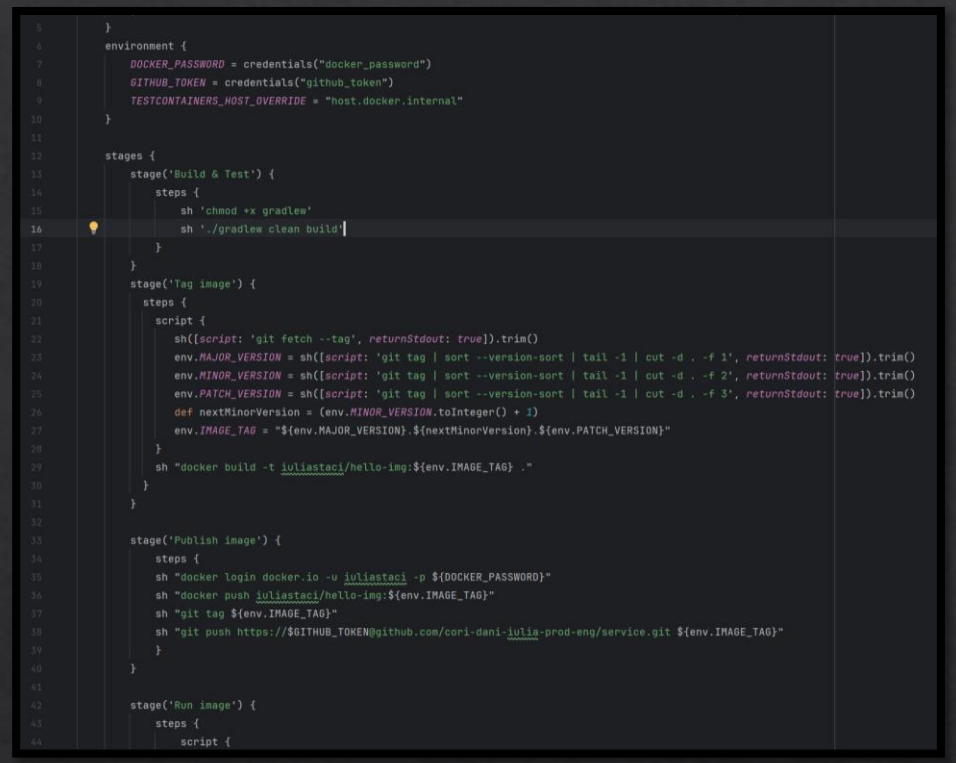
Monitoring System Diagram



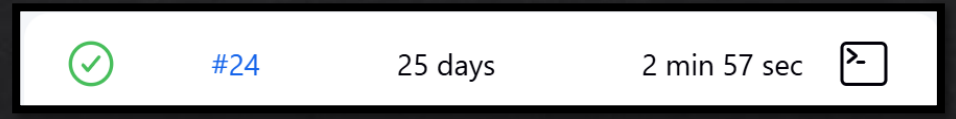
https://github.com/UNIBUC-PROD-ENGINEERING/service/wiki/Lab-8

CI/CD - Jenkins

- We implemented a CI/CD pipeline using Jenkins to automate the build, test, and deployment processes. The pipeline is defined in a Jenkinsfile and includes the following stages:
  - *Build & Test:* Compiles the application and runs unit tests using Gradle.
  - *Tag & Publish Docker Image:* Automatically tags the Docker image based on semantic versioning and pushes it to Docker Hub.
  - *Run Image:* Starts the application and its dependencies using Docker Compose to simulate the deployment environment.
  - *Run Integration Tests:* Executes integration tests to verify the correct interaction between deployed components.
- The second screenshot shows the last successful pipeline execution, demonstrating that the process completed correctly in under 3 minutes. This automation ensures faster delivery and consistent deployments.

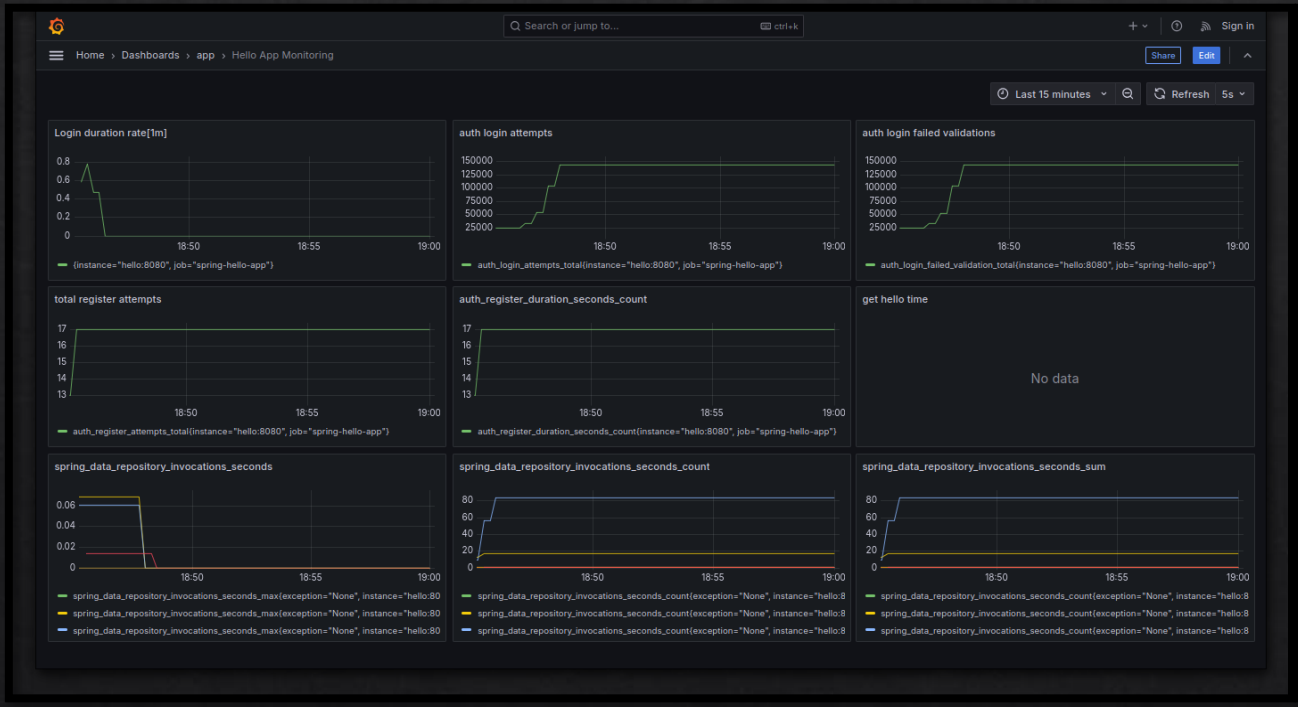


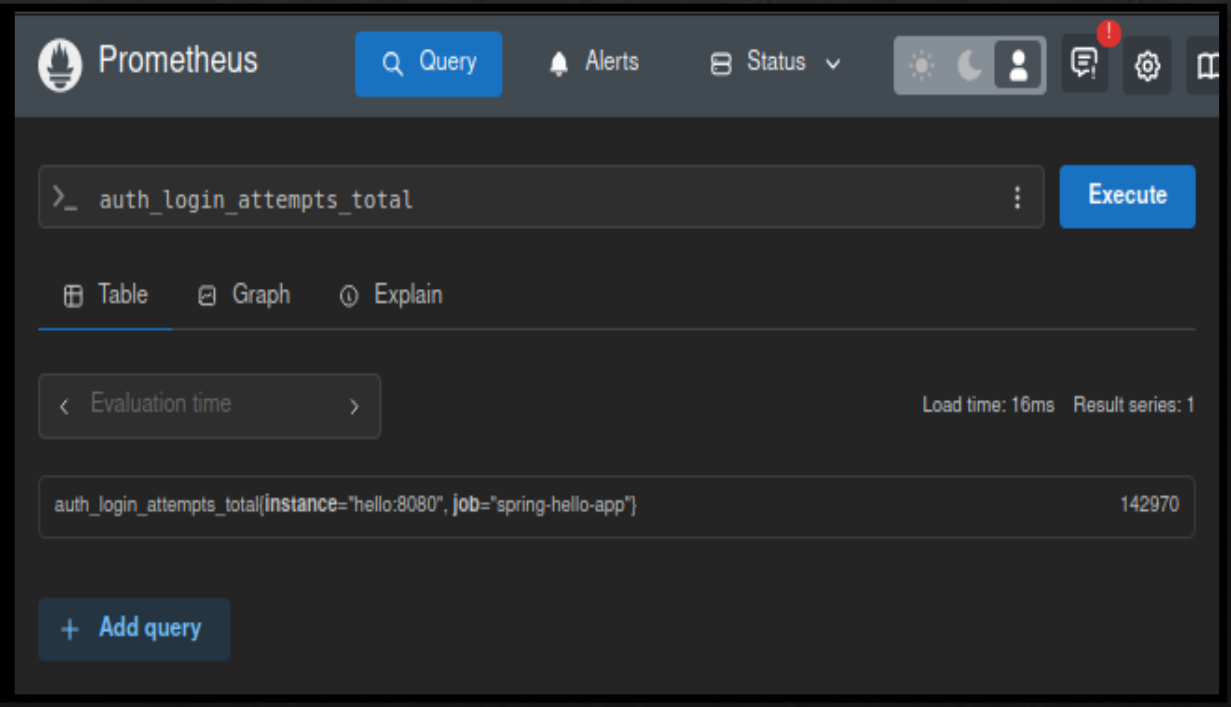
```
    }
    environment {
        DOCKER_PASSWORD = credentials("docker_password")
        GITHUB_TOKEN = credentials("github_token")
        TESTCONTAINERS_HOST_OVERRIDE = "host.docker.internal"
    }

    stages {
        stage('Build & Test') {
            steps {
                sh 'chmod +x gradlew'
                sh './gradlew clean build'
            }
        }
        stage('Tag image') {
          steps {
            script {
              sh([script: 'git fetch --tag', returnStdout: true]).trim()
              env.MAJOR_VERSION = sh([script: 'git tag | sort --version-sort | tail -1 | cut -d . -f 1', returnStdout: true]).trim()
              env.MINOR_VERSION = sh([script: 'git tag | sort --version-sort | tail -1 | cut -d . -f 2', returnStdout: true]).trim()
              env.PATCH_VERSION = sh([script: 'git tag | sort --version-sort | tail -1 | cut -d . -f 3', returnStdout: true]).trim()
              def nextMinorVersion = (env.MINOR_VERSION.toInteger() + 1)
              env.IMAGE_TAG = "${env.MAJOR_VERSION}.${nextMinorVersion}.${env.PATCH_VERSION}"
            }
            sh "docker build -t iuliastaci/hello-img:${env.IMAGE_TAG} ."
          }
        }

        stage('Publish image') {
            steps {
            sh "docker login docker.io -u iuliastaci -p ${DOCKER_PASSWORD}"
            sh "docker push iuliastaci/hello-img:${env.IMAGE_TAG}"
            sh "git tag ${env.IMAGE_TAG}"
            sh "git push https://$GITHUB_TOKEN@github.com/cori-dani-iulia-prod-eng/service.git ${env.IMAGE_TAG}"
            }
        }

        stage('Run image') {
            steps {
                script {
```

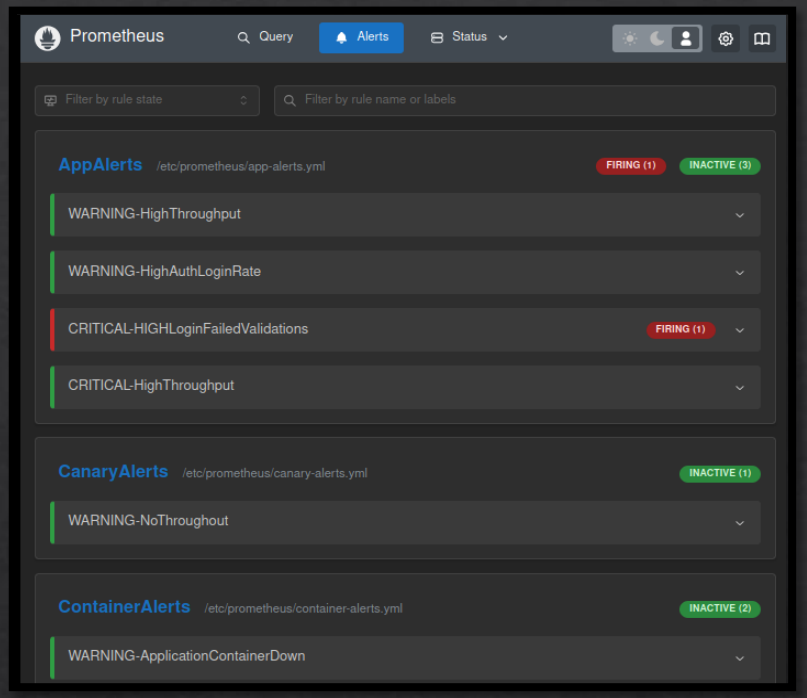


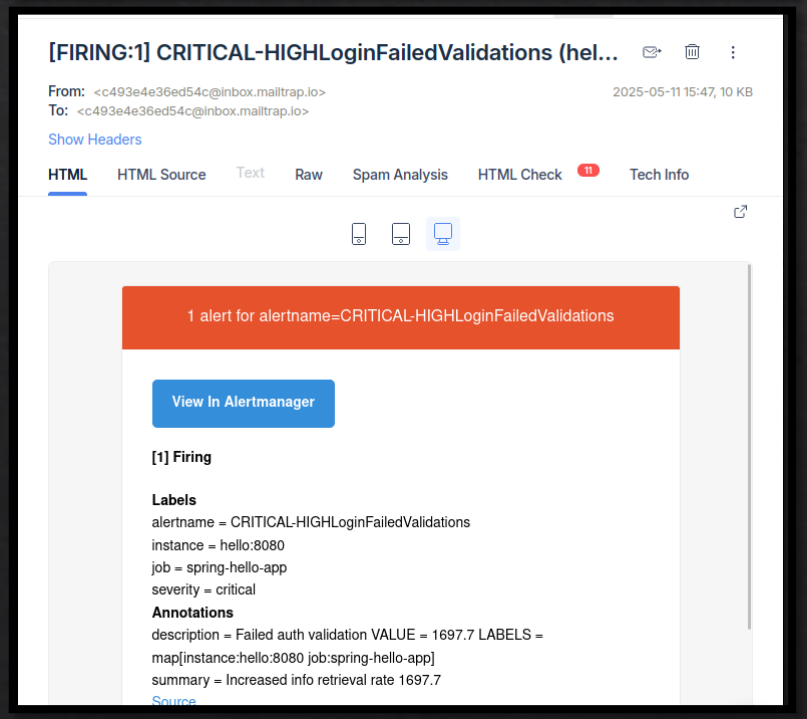
Monitoring & Alerting

- **Dashboards**: Built in Grafana to visualize critical auth metrics:
  - Login duration, login attempts, failed validations on logins, registration attempts, registration duration.
  - Data sourced from Prometheus for real-time monitoring.









Monitoring & Alerting

- **Alerting Rules**:
  - **WARNING – High Authentication Login Rate**: Triggers if login rate exceeds `1/sec` over 1m (10s duration).
  - **CRITICAL – High Login Failed Validations**: Triggers if failed validations exceed `2/sec` over 1m (10s duration).
- **Notification Flow**:
  - Alertmanager routes alerts to **Mailtrap**, sending email notifications with summaries and detailed descriptions.