Performance assessment 151 Integrate databases into web applications

Based on LBV 151-1 from Ecole des Métiers Fribourg (EMF Informatique)

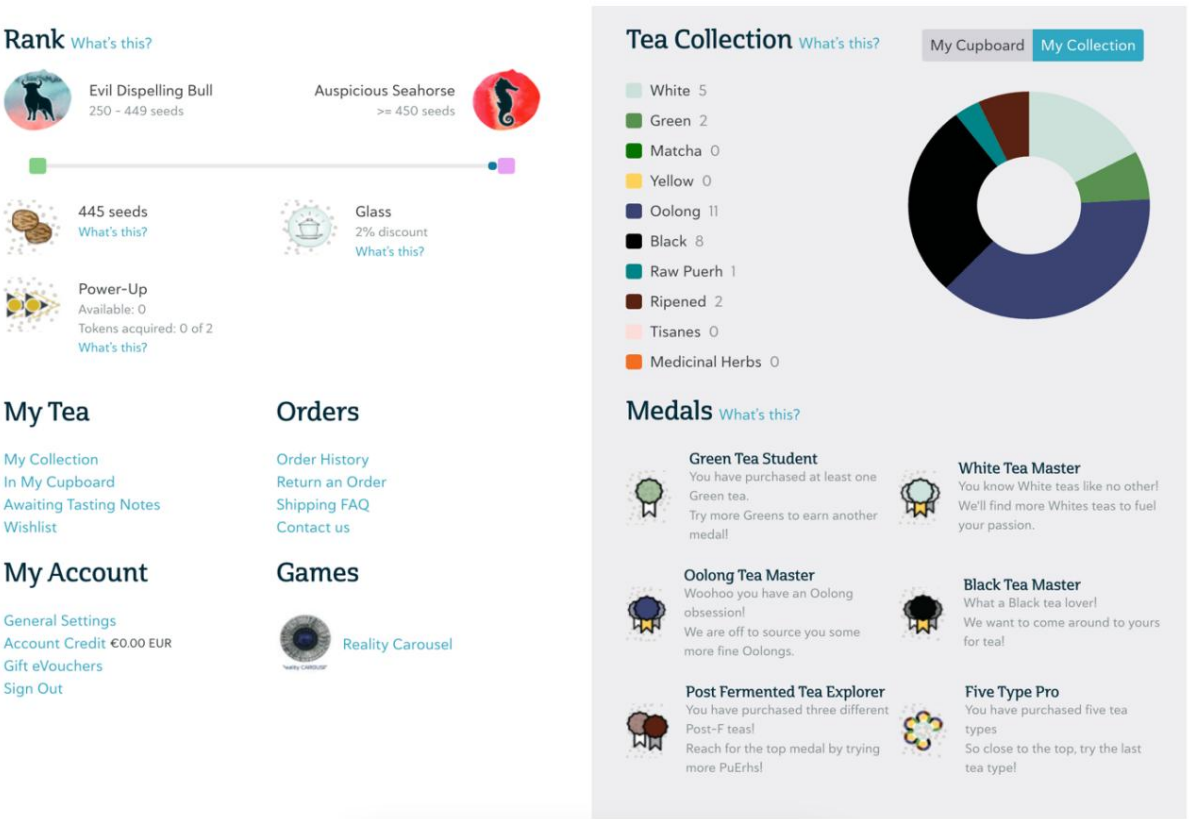
Core task Mei Leaf (40 points, estimated effort 40 hours)

"Push the reset button on your understanding of tea. Forget the stuffy tea rooms and silver service. Forget the scented nonsense mis-sold as 'tea' to yummy mummies. Forget those awful tea bags dunked in milky water. Let us introduce you to true tea.
Tea is a rare thing - an indulgence of flavor and effect that is really good for you too! It is the convergence of all the great things in life - natural, healthy, functional, artisan and delicious. It is a journey with no end of learning.

Mei Leaf was established in London in 2006 (previously called chinalife) to represent true tea culture. Don and his team tirelessly explore the mountains of the East to find the most delicious teas on the planet. These are pinnacle teas made by masters from the perfect terroirs and picked at the perfect season" - About Mei Leaf (https://meileaf.com/about)

The company Mei Leaf sells a wide variety of teas via its website
together with appropriate accessories. Customers can view the number of teas of each type purchased via a personal dashboard (https://meileaf.com/dashboard). You will also be credited with so-called seeds for every purchase. With these you can achieve higher ranks and benefits. You essentially win medals by writing reviews and purchasing them. Try out

specific teas.



Although this core task only implements part of the dashboard functionalities of Mei Leaf and completely dispenses with a front end, the visualization above shows how the E2E application could be implemented.

The Github repository https://github.com/yagan93/SpringBoot_Security_JWT_5.7 is provided as a basis for implementing the following requirements. PostgreSQL must be used for data persistence. The database schema must correspond to the 3 normalization form.

Requirements

- A1 registration customer (5 points)
  A potential customer must be able to register at /users/register. It is assigned the role 'CLIENT' as well as the authorities 'CAN_PLACE_ORDER', 'CAN_RETRIEVE_PURCHASE_HISTORY' and 'CAN_RETRIEVE_PRODUCTS'.
  He also receives the lowest of the 5 possible (Bronze, Silver, Gold, Platinum, Diamond) ranks.
  In addition to the credentials, at least the first and last name, date of birth and address must be stored for each customer. The attributes need to be validated accordingly. Their error messages must be provided in DE, FR, EN.

  In addition to the implementation of A1, the following question is expected to be answered:

```
$2a$12$R9h/cIPz0gi.URNNX3kh2OPST9/PgBkqquzi.Ss7KIUgO2t0jWMUW
\__/\/ \____________________/\_____________________________/
Alg Cost      Salt                        Hash
```

  bcrypt is used as the hashing function for the password. What does a salt protect against and how does it differ from a pepper?

- A2 Provision of products (4 points)
  The tea varieties are 'White', 'Green', 'Oolong', 'Black' and 'Medicinal Herbs'
  To make available. For each of these types of tea you have to store 2 products.
  These have the attributes name, country of origin (China, Taiwan, Japan), purchase and sales price per 100g and the date of harvest. Customers with the authority 'CAN_RETRIEVE_PRODUCTS' should be able to view the products offered via the URL /products. The payload should be pagable via @RequestParam. Help: yagan93/SpringBoot_Cascading

- A3 Buy a single product (6 points)
  An existing customer with the 'CAN_PLACE_ORDER' authority must be in the
  Be able to place an order for a single product in the desired quantity
  (grams). The price of the order depends on the sales price, the desired quantity and the reduction in the rank (Bronze 0%, Silver 4%,
  Gold 7%, Platinum 9%, Diamond 11%). Based on the price charged, seeds will be credited to the customer (1 seed per 2 CHF). If the credit leads to a change in rank, this must be done accordingly.

  (Bronze 0 Seeds, Silver 20 Seeds, Gold 60 Seeds, Platinum 140 Seeds, Diamond 300 Seeds).

- A4 statistics (3 points)
  A user with the 'ADMIN' role can use a provided endpoint to see which customer generated the most revenue in the past month (deactivated customer accounts are not included in the analysis).

  He can also use a provided endpoint to see from which country of origin customers ordered the most products in the past X days (via @RequestParam or @PathVariable).

- A5 Orders placed (3 points)
  An existing customer with the 'CAN_RETRIEVE_PURCHASE_HISTORY' authority can
  View a summary of your own orders. This shows how many products per type of tea have been ordered in the past.
  The number of seeds credited across all orders per type of tea can also be seen.

- D1 Physical database schema (1 point)
  A physical database schema (UML notation) needs to be created as documentation.

Deployment

Until now we only used Docker for the PostgreSQL database. However, a corresponding image should now also be created for the Spring Boot application and made available via a container.

- Create B1 JAR
  The first step is to create a JAR of the current Spring Boot application. This is achieved using the *./gradlew clean bootJar* command . The .jar file should now be found under build/libs.

- Create B2 image (1 point)
  Now create a file 'Dockerfile' in the root of the Spring Boot application:

```
FROM openjdk:17
COPY build/libs/*.jar spring-app.jar
EXPOSURE 8080
ENTRYPOINT ["java", "-jar", "/spring-app.jar"]
```

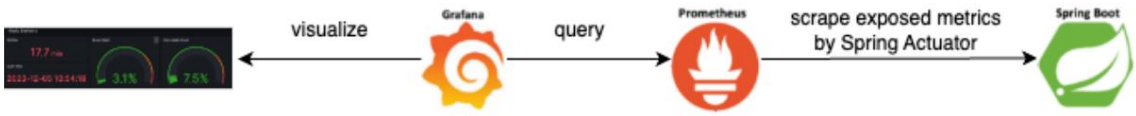
  Depending on the JDK used, FROM openjdk:17 needs to be adjusted accordingly.
  The image of the Spring Boot application can then be created using the *docker build -t spring-app command.* created and listed via *docker images .*

- Create B3 Compose (3 points)
  Instead of starting the two images (Spring Boot, PostgreSQL) individually, there is another file 'docker-compose.yml' in the root of the Spring Boot application
  create. Its content needs to be worked out as part of the core task at hand.

Monitoring

"Prometheus is a free software application used for event monitoring and alerting. It records metrics in a time series database (allowing for high dimensionality) built using an HTTP pull model, with flexible queries and real-time alerting. The project is written in Go and licensed under the Apache 2 License, with source code available on GitHub and is a graduated project of the Cloud Native Computing Foundation, along with Kubernetes and Envoy" - About Prometheus (https://prometheus.io/)

"Grafana is a multi-platform open source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources" - About Grafana (https://grafana.com/)



To expose the metrics of the Spring Boot application via Spring Actuator for Prometheus, the following steps must be taken:

- Add M1 dependencies

  The following 2 dependencies need to be added to the build.gradle file:

  ```
  implementation group: 'io.micrometer', name: 'micrometer-registry-prometheus',
   version: '1.12.0'
  implementation group: 'org.springframework.boot', name: 'spring-boot-starter-actuator', version: '2.7.18'
  ```

- Enable M2 Spring Actuator Endpoints

  ```
  management.endpoints.web.exposure.include=*
  ```

- Customize M3 SecurityFilterChain

  The Spring Actuator Endpoints must be released in the file core/security/WebSecurityConfig.java:

  ```
  .antMatchers(HttpMethod.GET, "/actuator/**").permitAll()
  ```

The activated Spring Actuator endpoints should now be visible at http://localhost:8080/actuator. One of these would be http://localhost:8080/actuator/health. Spring Actuator should also expose the necessary metrics for Promotheus via http://localhost:8080/actuator/prometheus. (1 point)

- Configure M4 Prometheus and Grafana

  Now it's time to copy the provided 'monitoring' folder (see Teams) into the root of the Spring Boot application. With the files datasources.yml as well
  prometheus.yml is used by Grafana or Prometheus configured.

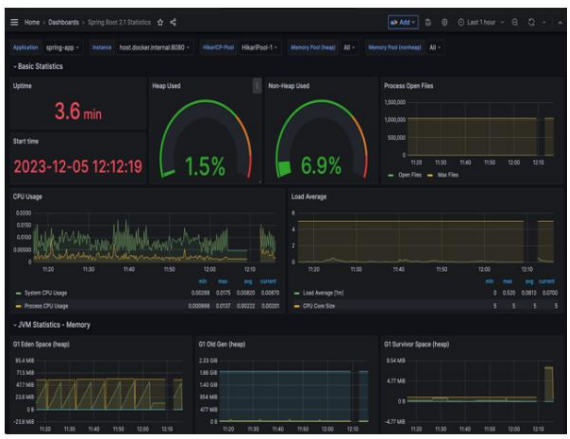
- M5 Start Prometheus, Grafana, Spring Boot, PostgreSQL

  The docker-compose.yml created in task B3 must now be supplemented with the following 2 services and started using the command *docker-compose up --build --force-recreate :*

```
Prometheus:
    image: prom/prometheus:v2.44.0
    container_name: prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./monitoring/prometheus/prometheus.yml:/etc/prometheus /prometheus.yml


grafana:
    image: grafana/grafana:9.5.2
    container_name: grafana
    ports:
      - "3000:3000"
    restart: unless-stopped
    volumes:
      - ./monitoring/grafana/provisioning/datasources:/etc
          /grafana/provisioning/datasources
```

- Create M6 Spring Boot Dashboard on Grafana (2 points)



After starting the above container, Grafana should be available as a
service at http://localhost:3000.
With the credentials admin/admin you can
you can log in and set your own password.
A new Spring Boot
Dashboard can be
found at http://localhost:3000/dashboards
to be created. To do this, navigate to New - Import and use the ID 10280. We use the Prometheus configured in M4 as the
Prometheus data source.

quality assurance

In order to finally ensure the necessary quality, the created Spring Boot application should be tested.
Here we limit ourselves to requirement A2. The repository yagan93/SpringBoot_Testing. can be used
to help with subsequent tasks
be used. The necessary dependencies can also be found from its build.gradle.

- Q1 Unit Test Weblayer (2 points)
  retrieveAll_requestAllProducts_expectAllProductsAsDTOS serves as a reference
  the class ProductControllerUnitTests.java. Please note the written comment above the method signature.

- Q2 Unit Test Service Layer (2 points)
  FindById_requestProductById_expectProduct from the ProductServiceImplUnitTests.java class
  serves as a reference.

- Q3 Unit Test Data Access Layer (1 point)
  FindAll_requestAllProducts_expectAllProducts from the ProductRepositoryUnitTests.java class
  serves as a reference. The test is carried out with database H2.

- Q4 Integration Test (2 points)
  RetrieveAll_requestAllProducts_expectAllProductsAsDTOS from the ProductIntegrationTests.java class
  serves as a reference.

- Q5 System Test Postman/Newman (3 points)
  The .json files in the System folder serve as a reference. It also makes sense to get into the
  Read the documentation at https://learning.postman.com/docs/collections/using-newman-cli/
  command-line-integration-with-newman.

- Q6 Static testing (1 point)
  The first step is to download the IntelliJ plugin SonarLint.
  This is then followed by a static analysis of the implemented code
  from requirement A1-A5. What optimization suggestions are
  given?