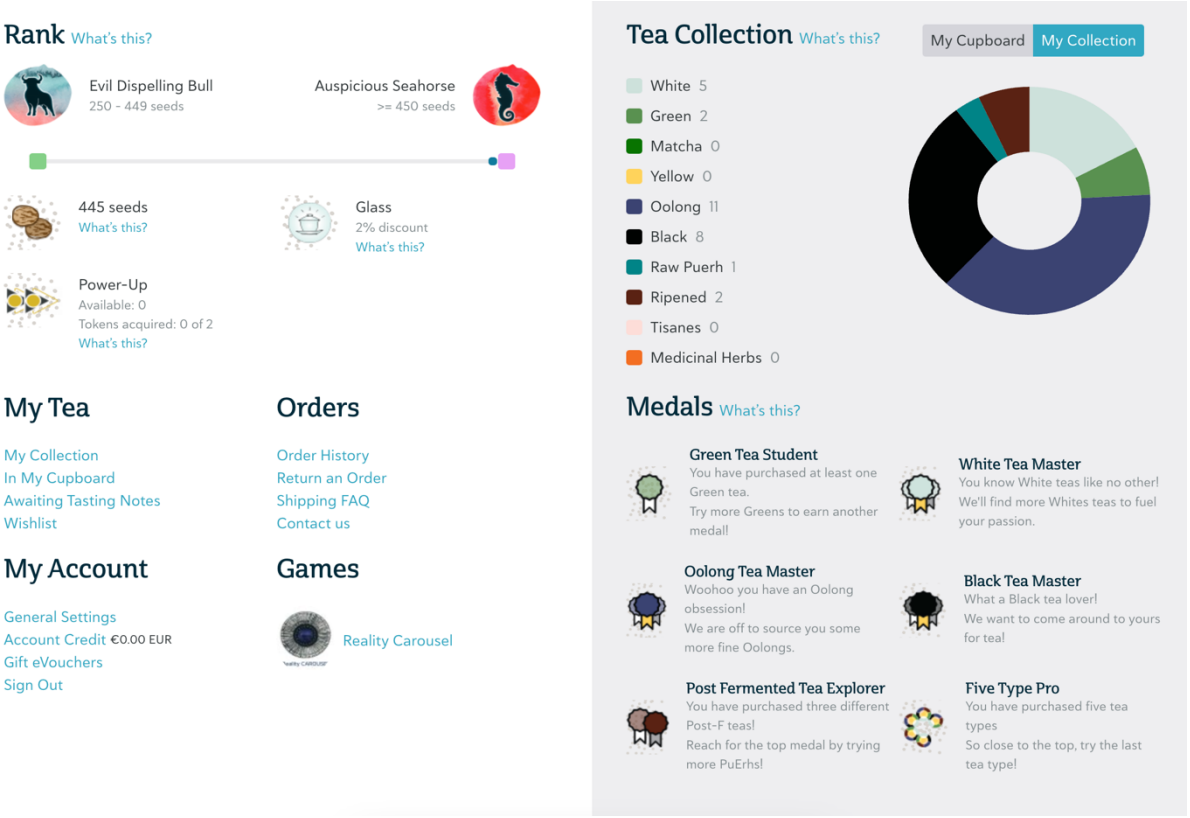
Leistungsbeurteilung 151 Datenbanken in Web-Applikationen einbinden

Basierend auf LBV 151-1 seitens Ecole des Métiers Fribourg (EMF Informatique)

Kernaufgabe Mei Leaf (40 Punkte, geschätzter Aufwand 40 Stunden)

"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 flavour 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)

Das Unternehmen Mei Leaf verkauft über ihren Webauftritt verschiedenste Teesorten zusammen mit entsprechendem Zubehör. Über ein persönliches Dashboard (https://meileaf.com/dashboard) können Kunden die Anzahl gekaufter Tees der jeweiligen Sorten einsehen. Auch werden ihnen für jeden Kauf sogenannte Seeds gutgeschrieben. Mit diesen erreichen sie höhere Ränge und Vergünstigungen. Medaillen gewinnt man im Wesentlichen durch das Verfassen von Bewertungen und das Erwerben resp. Probieren spezifischer Tees.



Obwohl vorliegende Kernaufgabe nur einen Teil der Dashboard Funktionalitäten von Mei Leaf umsetzt und auf ein Frontend gänzlich verzichtet, zeigt die obige Visualisierung auf, wie die Applikation E2E umgesetzt werden könnte.

Als Basis für die Implementation nachfolgender Anforderungen wird das Github Repository https://github.com/yagan93/SpringBoot_Security_JWT_5.7 zur Verfügung gestellt. Für die Persistenz der Daten ist PostgreSQL zu verwenden. Das Datenbankschema hat der 3 Normalisierungsform zu entsprechen.

Anforderungen

- A1 Registrierung Kunde (5 Punkte)
  Ein potenzieller Kunde muss sich unter /users/register registrieren können. Dabei wird ihm die Rolle 'CLIENT' sowie die Autoritäten 'CAN_PLACE_ORDER', 'CAN_RETRIEVE_PURCHASE_HISTORY' und 'CAN_RETRIEVE_PRODUCTS' zugewiesen. Auch erhält er den tiefsten der 5 möglichen (Bronze, Silver, Gold, Platinum, Diamond) Ränge. Nebst den Credentials muss für jeden Kunden mindestens der Vor- sowie Nachname, das Geburtsdatum sowie die Adresse hinterlegt werden. Die Attribute gilt es entsprechend zu validieren. Deren Fehlermeldungen müssen in DE, FR, EN bereitgestellt werden.

  Nebst der Umsetzung von A1 wird erwartet, dass folgende Frage beantwortet wird:

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

  Als Hashing Funktion des Passwortes wird bcrypt verwendet. Gegen was schützt ein Salt und wie unterscheidet sich dieser zu einem Pepper?

- A2 Bereitstellung von Produkten (4 Punkte)
  Es sind die Teesorten 'White', 'Green', 'Oolong', 'Black' sowie 'Medicinal Herbs' zur Verfügung zu stellen. Für jede dieser Teesorten gilt es 2 Produkte zu hinterlegen. Diese haben die Attribute Bezeichnung, Herkunftsland (China, Taiwan, Japan), Einkaufs- sowie Verkaufspreis pro 100g und das Datum der Ernte aufzuweisen. Über die URL /products sollen Kunden mit der Autorität 'CAN_RETRIEVE_PRODUCTS' die angebotenen Produkte einsehen können. Der Payload sollte per @RequestParam pagable sein. Hilfestellung: yagan93/SpringBoot_Cascading

- A3 Einzelnes Produkt kaufen (6 Punkte)
  Ein bestehender Kunde mit der Autorität 'CAN_PLACE_ORDER' muss in der Lage sein eine Bestellung für ein einzelnes Produkt in der gewünschten Quantität (Gramm) zu tätigen. Der Preis der Bestellung ergibt sich aus dem Verkaufspreis, der gewünschten Quantität sowie der Verbilligung des Ranges (Bronze 0%, Silver 4%, Gold 7%, Platinum 9%, Diamond 11%). Basierend auf dem verrechneten Preis werden dem Kunden Seeds gutgeschrieben (1 Seed pro 2 CHF). Falls die Gutschrift zu einer Rangänderung führt, muss diese entsprechend vorgenommen werden. (Bronze 0 Seeds, Silver 20 Seeds, Gold 60 Seeds, Platinum 140 Seeds, Diamond 300 Seeds).

- A4 Statistiken (3 Punkte)
  Ein Benutzer mit der Rolle 'ADMIN' kann über einen bereitgestellten Endpoint einsehen, mit welchem Kunde im vergangenen Monat am meisten Umsatz generiert wurde (deaktivierte Kundenkonten werden nicht in die Analyse miteinbezogen). Auch kann er über einen bereitgestellten Endpoint einsehen, von welchem Herkunftsland die Kunden in den vergangenen X Tagen am meisten Produkte bestellten (per @RequestParam oder @PathVariable).

- A5 Getätigte Bestellungen (3 Punkte)
  Ein bestehender Kunde mit der Autorität 'CAN_RETRIEVE_PURCHASE_HISTORY' kann eine Zusammenfassung seiner eigens getätigten Bestellungen einsehen. Hierbei wird aufgezeigt, wieviele Produkte pro Teesorte in der Vergangenheit bestellt wurden. Auch ist die Anzahl gutgeschriebener Seeds über alle Bestellungen pro Teesorte ersichtlich.

- D1 Physisches Datenbankschema (1 Punkt)
  Als Dokumentation gilt es ein physisches Datenbankschema (Notation UML) zu erstellen.

Bereitstellung

Bis anhin nutzten wir Docker lediglich für die PostgreSQL Datenbank. Nun sollte jedoch auch für die Spring Boot Applikation ein entsprechendes Image erstellt und per Container bereitgestellt werden.

- B1 JAR erstellen
  In einem ersten Schritt gilt es ein JAR der aktuellen Spring Boot Applikation zu erstellen. Dies wird per Befehl *./gradlew clean bootJar* erreicht. Das .jar File sollte nun unter build/libs auffindbar sein.

- B2 Image erstellen (1 Punkt)
  Nun erstelle man ein File 'Dockerfile' im Root der Spring Boot Applikation:

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

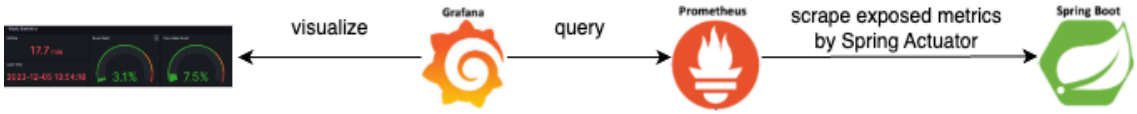
  Je nach verwendetem JDK gilt es `FROM openjdk:17` entsprechend anzupassen. Anschliessend kann das Image der Spring Boot Applikation über den Befehl *docker build -t spring-app .* erstellt und per *docker images* aufgelistet werden.

- B3 Compose erstellen (3 Punkte)
  Anstatt die beiden Images (Spring Boot, PostgreSQL) einzeln zu starten, ist im Root der Spring Boot Applikation ein weiteres File 'docker-compose.yml' zu erstellen. Dessen Inhalt gilt es als Teil vorliegender Kernaufgabe auszuarbeiten.

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/)



Um die Metriken der Spring Boot Applikation über Spring Actuator für Prometheus zu exponieren, müssen folgende Schritte vorgenommen werden:

- M1 Abhängigkeiten hinzufügen

  Im File build.gradle gilt es folgende 2 Abhängigkeiten zu ergänzen:

```
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'
```

- M2 Spring Actuator Endpoints aktivieren

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

- M3 SecurityFilterChain anpassen

  Im File core/security/WebSecurityConfig.java sind die Spring Actuator Endpoints freizugeben:

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

Unter http://localhost:8080/actuator sollten die aktivierten Spring Actuator Endpoints nun ersichtlich sein. Einer dieser wäre http://localhost:8080/actuator/health. Auch sollte Spring Actuator über http://localhost:8080/actuator/prometheus die nötigen Metriken für Promotheus exponieren. (1 Punkt)

- M4 Prometheus und Grafana konfigurieren

  Nun gilt es den zur Verfügung gestellten Ordner 'monitoring' (siehe Teams) in den Root der Spring Boot Applikation zu kopieren. Mit den Files datasources.yml sowie prometheus.yml wird Grafana resp. Prometheus konfiguriert.

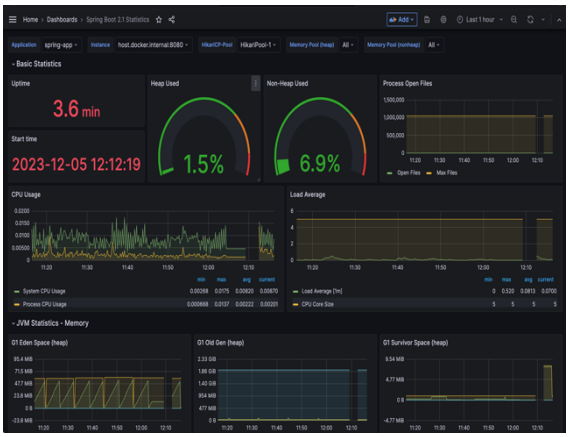
- M5 Prometheus, Grafana, Spring Boot, PostgreSQL starten

  Das in Aufgabe B3 erstellte docker-compose.yml ist nun durch folgende 2 Services zu ergänzen und per Befehl *docker-compose up --build --force-recreate* zu starten:

```
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
```

- M6 Spring Boot Dashboard auf Grafana erstellen (2 Punkte)



Nach dem Start obiger Container sollte Grafana als Service unter http://localhost:3000 erreichbar sein. Mit den Credentials admin/admin kann man sich anmelden und ein eigenes Passwort setzen. Ein neues Spring Boot Dashboard kann unter http://localhost:3000/dashboards erstellt werden. Hierbei navigiere man zu New - Import und benutzt die ID 10280. Als Prometheus Data Source benutzen wir den in M4 konfigurierten Prometheus.

Qualitätssicherung

Um abschliessend die nötige Qualität sicherzustellen, sollte die erstellte Spring Boot Applikation getestet werden. Hierbei beschränken wir uns auf Anforderung A2. Als Hilfestellung für nachfolgende Aufgaben kann das Repository yagan93/SpringBoot_Testing verwendet werden. Aus dessen build.gradle können auch die nötigen Abhängigkeiten entnommen werden.

- Q1 Unit Test Weblayer (2 Punkte)
  Als Referenz dient retrieveAll_requestAllProducts_expectAllProductsAsDTOS aus der Klasse ProductControllerUnitTests.java. Man beachte den verfassten Kommentar über der Methodensignatur.

- Q2 Unit Test Service Layer (2 Punkte)
  Als Referenz dient findById_requestProductById_expectProduct aus der Klasse ProductServiceImplUnitTests.java.

- Q3 Unit Test Data Access Layer (1 Punkte)
  Als Referenz dient findAll_requestAllProducts_expectAllProducts aus der Klasse ProductRepositoryUnitTests.java. Der Test wird mit der Datenbank H2 ausgeführt.

- Q4 Integration Test (2 Punkte)
  Als Referenz dient retrieveAll_requestAllProducts_expectAllProductsAsDTOS aus der Klasse ProductIntegrationTests.java.

- Q5 System Test Postman/Newman (3 Punkte)
  Als Referenz dienen die .json Files im Ordner System. Auch macht es Sinn sich in die Dokumentation unter https://learning.postman.com/docs/collections/using-newman-cli/command-line-integration-with-newman einzulesen.

- Q6 Statisches Testen (1 Punkt)
  In einem ersten Schritt gilt es das IntelliJ Plugin SonarLint herunterzuladen. Anschliessend ist mit diesem eine statische Analyse des implementierten Codes aus Anforderung A1-A5 durchzuführen. Welche Optimierungsvorschläge werden gegeben?