Web Application 2 2020-2021

Lab 2: Wallet Service of an e-commerce micro-service application

A distributed e-commerce service is made by many parts which perform some independent tasks. All of these parts, communicating with each other, contribute to make the entire system works.

One of them is a service that represents the Wallet. Each customer, in order to make orders related to one or more products, should have a Wallet. It handles the customer money and keeps track of customer transactions.

1. Create a Spring Initializr project. You can create it from *https://start.spring.io/* or directly from Intellij IDEA in *New Project* section. Include dependencies as Spring Web, Spring Data JPA and MariaDB driver.
2. Go to Database section of IntelliJ IDEA, type on "+" icon, go to Data Source, and add a new MariaDB datasource. If needed, download the necessary Driver Files. In "General" section assign an User and Password and press the "Test Connection" button for the connection on *jdbc:mariadb://localhost:3306.* Click on Ok.
3. In the same Menu of "+" button, click on "Jump to Query Console" icon and type "CREATE DATABASE ecommerce;", or whatever name you prefer. Check if the DB was created.
4. Go to the *application.properties* file and add the properties related to the datasource. Setting *spring.jpa.show-sql* to true, enables the logging of all SQL operations performed . For example, when you modify your domain model, you will see on console something like this: *Hibernate: alter table transaction add column creation_date_time datetime*

```
spring.datasource.url=jdbc:mariadb://localhost:3306/YOUR_DB_NAME
spring.datasource.username=YOUR_USERNAME
spring.datasource.password=YOUR_PASSWORD
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MariaDBDialect
```

5. You can set also *spring.jpa.hibernate.ddl-auto* property, which influences how the schema tool management, i.e. Hibernate, will manipulate the database schema at startup. You can assign it one of these values:
    - *update:* The update operation will attempt to add new columns, constraints,.., but will never remove a column or constraint that may have existed previously, but no longer does as part of the object model. Used for example in development
    - *create-drop:* during cleanup, the schema objects are dropped, leaving an empty database. Useful in test case scenarios
    - *validate*: only validates whether the tables and columns exist, otherwise throws an exception
    - *none:* for migration scripts for database change. Used mostly in production

Now let's create the structure of the project.

6. Create these packages in your main kotlin package: *controllers, domain, dto, services, repositories.*

We know that each **Customer** could have zero or more **Wallets.** A Customer, in addition to the wallets it is related to, has a *name, surname*, a *delivery address* and an *email,* which is unique.

A Wallet is characterized by the *current amount* of money that the customer has (which has to be a positive or zero value). Moreover, it is related to the owning customer and two lists of **Transactions:** those that represents purchases and contribute to reduce the amount of money in the wallet, and those that represents recharges, that instead contribute to increase the money in the wallet.

Each transaction contains the *amount of money transacted,* the *time instant* the transaction was performed and the reference to the two involved wallets: the one which the money was taken from, and the one which money was given to.

The amount of money, at any time, in a wallet, is the difference between the sum of all recharges and the sum of purchases and should never become negative.

If the customer tries to make a purchase spending more money that that is in his wallet, the transaction will be rejected by the system.

Draw, on a piece of paper, the db schema with its relationships.

7. In the *domain* package, create *Entities* which represents the data domain model with appropriate relationships mappings. For this reason, you need some Kotlin plugins. Add in your Gradle / Maven plugin section *org.jetbrains.kotlin.plugin.jpa*. It creates no-arg constructors for classes annotated with @Entity. As JPA requires, @Entity classes should have a default (non-arg) constructor to instantiate the objects when retrieving them from the database.
8. In order to make lazy fetching working as expected, entities should be open as described in a Kotlin issue KT-28525. We are going to use the Kotlin allopen plugin for that purpose. Add, in build.grandle.kt:

```
plugins{
  //...
  kotlin("plugin.allopen") version "1.4.31"
}
allOpen{
  annotation("javax.persistence.Entity")
  annotation("javax.persistence.Embeddable")
  annotation("javax.persistence.MappedSuperclass")
}
```

9. For the Entities representations, use classes instead of Kotlin data classes with val properties, because JPA is not designed to work with immutable classes or with the methods generated automatically by data classes.
10. Annotate the attribute id of your entities with annotations @Id and @GeneratedValue. By default, the AUTO generation strategy is used: Hibernate will determine values based on the type of the primary key attribute.
11. For each Entity, several annotations can be used to specify the DB schema:
    - The name of the table in the DB, via the *@Table* annotation. For example *@Table(name="my_table_name")*. If we do not use the *@Table* annotation, the name of the entity will be considered the name of the table.
    - We can use the *@Column* annotation in front of a property to mention the details of a column in the table. The *@Column* annotation allows to specify details like column name, length, nullable, and uniqueness constraints. The name element specifies the name of the column in the table. The length element specifies its length. The nullable element specifies whether the column is nullable or not, and the unique element specifies whether the column must be unique. If we don't specify this annotation, the name of the field will be considered

    - the name of the column in the table, and no other restrictions will be placed. For example, *@Column(name="myName", unique=true).*
    - The SQL type to use for a property, if not derivable directly from the corresponding type.
12. Entities can be related to other entities. This is specified adding properties which are direct or indirect references (lists, sets, ...) to other entities and prefixing them with @OneToOne, @OneToMany, @ManyToOne or @ManyToMany, depending on the cardinality of the relationship.
    - one-to-many mapping means that one row in a table is mapped to multiple rows in another table; the Kotlin type of this property must be a collection (set, list) of the other entity.
    - In the table of the element with cardinality N, we use the *@ManyToOne* annotation and *@JoinColumn(name="column_name", referencedColumnName="id")* on the referenced attribute (for example, *@ManyToOne(fetch=FetchType.LAZY) and @JoinColumn(name="target_entity_id", referencedColumnName="id") val targetEntity: TargetEntity*). With the *JoinColumn* annotation we configure the name of the column in the table with cardinality N (the column that contains the foreign key) that maps to the primary key in the target entity (which has the cardinality 1).
    - The entity with cardinality 1, is labelled with the annotation *@OneToMany(mappedBy="target_entity_join_column_name", targetEntity=YourTargetEntity::class)*, to reflect the opposite relationship in the database.
13. In order to support checking field-level constraints, two dependencies must be added: *"org.hibernate.validator:hibernate-validator",* which enables the validation framework, and *"javax.validation:validation-api",* which enables the use of annotations such as *@NotNull, @Min, and @Max*, in front of bean properties.
14. Run your project, check that no exception is thrown, then go to the Database section, right-click your db, select Diagrams/Show Visualization, and see the generated schema. Compare it with that you drew before. Do the two schemas represent the same thing? The relationships reflect those you drew? Remove the mappedBy attribute of the @OneToMany annotation of one of the target entities and re-run the project: do you see any difference in the generated schema?

Now we want to handle some operations through our Wallet. We want to:

- Add a wallet to a Customer by customerID
- Get a wallet by walletID
- Perform a transaction moving money between two wallets
- Get wallet's list of transactions by walletID
- Get all the wallet's transactions in a given date range by walletID

It means that we need to perform some business logic with our domain data, to handle the CRUD operations. This task is usually performed by a Service, with wired Repositories to access to the data layer.

12. Create, in the repositories package, the Repositories interfaces which implements the CrudRepository<T, ID>. You can create a repository for Customer, Wallet and Transaction.
13. Create the *WalletService* interface declaring the needed methods with the parameters they should accept and the returning values. If those parameters/returning values are objects (or collections of them), they should be DTOs, which represent the abstraction of the application domain.
14. Implement, in the dto package, the Kotlin data classes for your DTOs
15. Create the *WalletServiceImpl* class which implements the *WalletService* interface. Annotate it with *@Service* and *@Transactional* (the last one in order to enable transaction management).
16. Implement the business logic for the overriden methods. Handle operations with repositories to check for example the existence of data and to perform the CRUD operations on the DAO objects.

Consider also the possible edge cases, returning the appropriate values (for example throwing a NotFound exception if an element is not present in your DB).

17. If needed, in your repository, write custom queries following the correct conventions and keywords, or define, in JPQL, the queries through the @Query annotation.
18. Whenever you have to return an object, convert it to its DTO representation. You can use for this purpose the Kotlin Extension functions.

We want to expose our web resources and handle incoming requests through a series of actions. The endpoints to expose will return raw data back to the client (a JSON representation) instead of the View Model. For this reason, implement a Rest Controller, that indicates that the data returned by each method will be written into the response body instead of rendering a template. The endpoint to expose could be:

- /wallet [POST] → Create a new wallet for a given customer. In the Request Body there will be the Customer's ID for which you want to create a wallet. The wallet created will initially have no money. Once the wallet is created, return a 201 (created) response status and the wallet itself as the response body
- /wallet/{walletId} [GET] → Get the details of a wallet. The response body will be the requested wallet, and the response status 200 (ok)
- /wallet/{walletId}/transaction [POST] → Create a transaction taking the amount of money set in the body from the given wallet and transferring it to a second walletId, always defined in the body. Return the created transaction
- /wallet/{walletId}/transactions?from=<dateInMillis>&to=<dateInMillis> [GET] → Get a list of transactions regarding a given wallet in a given time frame
- /wallet/{walletId}/transactions/{transactionId} [GET] → Get the details of a single transaction

19. Create the WalletController class in the controllers package. Annotate it with *@RestController* and *@RequestMapping("/wallet")*, which, applied to class-level, maps the */wallet* request path to this controller.
20. The WalletController will use the wired WalletService to access to the data tier and perform the necessary business logic.
21. Implement the above listed endpoints, annotating methods with *@GetMapping / @PostMapping* with the url they maps (for example, *@GetMapping("/{walletId})* to obtain the wallet with the given id). Note that there is not present an endpoint related to the creation of a customer, because it is not a task of a Wallet Service. So insert it manually to the table from the Query Console. Moreover, the wallet created is initially empty and the money inside it will be taken from the system (the "bank"). In this lab, we'll not implement it, so to perform your tests, modify the wallet's amount directly from the wallet's table (or through an sql query in Query Console).
22. If is necessary to validate the Request Body, add the *@Valid* annotation before the parameter which represents the request body DTO. Decorate the fields of your DTOs with the validation annotations (for example *@Min)* if necessary. To retrieve the validation errors during an incoming request, add the parameter of type BindingResult: calling the BindingResult hasErrors() methods, you will know if it was performed a Bad Request. (for example, *fun addAPI(@RequestBody @Valid myDTO: MyDTO, bindingResult : BindingResult) ….).*
23. The endpoints should return a ResponseEntity which represents the whole HTTP response: status code, headers, body. So, for example in case of a Bad Request, you can build your response as *ResponseEntity.badRequest().body("Bad request message").*
24. Going to the *Endpoint* section in the IDE (usually one of the tabs below) you can see the list of exposed endpoints. Clicking on one of them, you can go to the HTTP client editor and try your endpoints.

To have more info on relationships, you can read these tutorials:

- Hibernate One to Many Annotation Tutorial, https://www.baeldung.com/hibernate-one-to-many
- Building web applications with Spring Boot and Kotlin, https://spring.io/guides/tutorials/spring-boot-kotlin/
- Defining JPA/Hibernate Entities in Kotlin, https://medium.com/swlh/defining-jpa-hibernate-entities-in-kotlin-1ff8ee470805
- Demo application: Kotlin + Hibernate powered by Spring Boot, https://github.com/s1monw1/hibernateOnKotlin