**Practice**

**Recommended Timeline**

The recommended timeline for the whole module is 2 weeks.

**Task**

**General requirements**

1. Code should be clean and should not contain any “developer-purpose” constructions.
2. App should be designed and written with respect to OOD and SOLID principles.
3. Code should contain valuable comments where appropriate.
4. Public APIs should be documented (Javadoc).
5. Clear layered structure should be used with responsibilities of each application layer defined.
6. JSON should be used as a format of client-server communication messages.
7. Convenient error/exception handling mechanism should be implemented: all errors should be meaningful and localized on backend side. Example: handle 404 error:
8. • HTTP Status: 404
9. • response body
10. • {
11. • “errorMessage”: “Requested resource not found (id = 55)”,
12. • “errorCode”: 40401
13. • }

where \*errorCode” is your custom code (it can be based on http status and requested resource - certificate or tag)

1. Abstraction should be used everywhere to avoid code duplication.
2. Several configurations should be implemented.

**Part 1**

Migrate your existing Spring application from a previous module to a Spring Boot application.

**Part 2**

**Business requirements**

This sub-module is an extension of REST API Basics, and it covers such topics as pagination, sorting, filtering and HATEOAS. Please imagine that your application has a lot of data, so when you make a GET request it will return, for instance, 1 million records. This will take much time to process such request and return the result to the consumer of your API. That is exactly what pagination, sorting, and filtering can solve. The other topic is HATEOAS what stands for the phrase "Hypermedia As The Engine Of Application State". When you are viewing a web page, you see data on it and can perform some actions with this data. In REST when you request a resource you get the details of the resource in the response. Along with it you can send the operations that you can perform on the resource. And this is what HATEOAS does.

The system should be extended to expose the following REST APIs:

1. Change single field of gift certificate (e.g. implement the possibility to change only duration of a certificate or only price).
2. Add new entity User.
   * implement only get operations for user entity.
3. Make an order on gift certificate for a user (user should have an ability to buy a certificate).
4. Get information about user’s orders.
5. Get information about user’s order: cost and timestamp of a purchase.
   * The order cost should not be changed if the price of the gift certificate is changed.
6. Get the most widely used tag of a user with the highest cost of all orders.
   * Create separate endpoint for this query.
   * Demonstrate SQL execution plan for this query (explain).
7. Search for gift certificates by several tags (“and” condition).
8. Pagination should be implemented for all GET endpoints. Please, create a flexible and non-erroneous solution. Handle all exceptional cases.
9. Support HATEOAS on REST endpoints.

**Application requirements**

1. JDK version: 8. Use Streams, java.time.\*, an etc. where it is appropriate. (the JDK version can be increased in agreement with the mentor/group coordinator/run coordinator)
2. Application packages root: com.epam.esm.
3. Java Code Convention is mandatory (exception: margin size –120 characters).
4. Apache Maven/Gradle, latest version. Multi-module project.
5. Spring Framework, the latest version.
6. Database: PostgreSQL/MySQL, latest version.
7. Testing: JUnit, the latest version, Mockito.
8. Service layer should be covered with unit tests not less than 80%.

**Part 3**

This sub-module covers following topics:

1. ORM
2. JPA & Hibernate
3. Transactions ORM stands for Object Relational Mapping. It’s a bit of an abstract concept – but basically it’s a technique that allows us to query and change data from the database in an object oriented way. ORMs provide a high-level abstraction upon a relational database that allows a developer to write Java code instead of SQL to create, read, update and delete data and schemas in their database. Developers can use the programming language they are comfortable with to work with a database instead of writing SQL statements or stored procedures. A JPA (Java Persistence API) is a specification of Java which is used to access, manage, and persist data between Java object and relational database. It is considered as a standard approach for Object Relational Mapping. JPA can be seen as a bridge between object-oriented domain models and relational database systems. Being a specification, JPA doesn't perform any operation by itself. Thus, it requires implementation. So, ORM tools like Hibernate, TopLink, and iBatis implements JPA specifications for data persistence. A transaction usually means a sequence of information exchange and related work (such as database updating) that is treated as a unit for the purposes of satisfying a request and for ensuring database integrity. For a transaction to be completed and database changes to made permanent, a transaction has to be completed in its entirety.

**Application requirements**

1. Hibernate should be used as a JPA implementation for data access.
2. Spring Transaction should be used in all necessary areas of the application.
3. Audit data should be populated using JPA features (an example can be found in materials).

**Application restrictions**

1. Hibernate specific features.
2. Spring Data

**Demo**

**Practical part**

1. Generate for a demo at least
   * 1000 users
   * 1000 tags
   * 10’000 gift certificates (should be linked with tags and users) All values should look like more -or-less meaningful: random words, but not random letters
2. Demonstrate API using Postman tool (prepare for demo Postman collection with APIs)
3. (Optional) Build & run application using command line

**Theoretical part**

Mentee should be able to answer questions during demo session.