PG6101, Enterprise 2, Home Exam, November 2019

The exam should **NOT** be done in group: each student has to write the project on his/her own. During the exam period, you are not allowed to discuss any part of this exam with any other student or person, not even the lecturer (i.e., do not ask questions or clarification on the exam). In case of ambiguities in these instructions, do your best effort to address them (and possibly explain your decisions in the readme file). Failure to comply to these rules will result in an **F** grade and possible further disciplinary actions.

The students have **72** hours to complete the project. See the details of submission deadline from where you got this document.

The exam assignment will have to be zipped in a *zip* file with name *pg6100_<id>.zip*, where you should replace *<id>* with the unique id you received with these instructions. If for any reason you did not get such id, use your own student id, e.g. *pg6100_701234.zip*. No "*rar*", no "*tar.gz*", etc. You need to submit all source codes (e.g., *.kt* and *pom.xml* files), and no compiled code *(.class*) or libraries (*.jar*, *.war*).

The delivered project must be compilable with Maven 3.x with commands like "*mvn package -DskipTests*" directly from your unzipped file. The project must be compilable with Java **8**. The project must be self-contained, in the sense that all third-party libraries must be downloadable from Maven Central Repository (i.e., do not rely on SNAPSHOT dependencies of third-party libraries that were built locally on your machine). All tests **MUST** run and pass when running "*mvn clean verify*". Note: examiners will run such command on their machine when evaluating your delivery. Compilation failures will heavily reduce your grade.

The assignment is divided into several parts/exercises. The parts are incremental, i.e., building on each other. You can (and should when appropriate) reuse code from:

https://github.com/arcuri82/testing_security_development_enterprise_systems

for example, the *pom.xml* files. However, every time a file is reused, you must have comments in the code stating that you did not write such file and/or that you extended it. You **MUST** have in the comments the link to the file from GitHub which you are using and/or copying&pasting. For an external examiner it **MUST** be clear when s/he is looking at your original code, or code copied/adapted from the course.

The exam consists in building an enterprise application using a microservice architecture. The main goal of the exam is to show the understanding of the different technologies learned in class. The more technologies you can use and integrate together, the better.

The application topic/theme of the exam will vary every year, but there is a set of requirements that stays the same, regardless of the topic/theme of the application.

The application has to be something new that you write, and not re-using existing projects (e.g., existing open-source projects on *GitHub*). If you plagiarize a whole project (or parts of it), not only you will get an **F**, but you will be subject to further disciplinary actions. You can of course re-use some existing code snippets (e.g., from *StackOverflow*) to achieve some special functionalities (but recall to write a code comment about it, e.g., a link to the *StackOverflow* page).

There is a requirement to build a Front-End GUI. One of the goals of this course is to learn how to integrate a GUI in a microservice architecture, but not building the GUI itself. Therefore, you can choose whatever technology you like, e.g., JavaScript frameworks like React or Angular running in NodeJS. However, the build of the frontend **MUST** still be initiated from Maven (i.e., the entire project **MUST** be buildable with a single Maven command, as seen in class).

Besides the *JDK* and *Maven*, you can assume that the examiners will have *Docker*, *NodeJS*, *NPM* and *YARN* installed on their machine.

In the *readme.md* you also **MUST** have the following:

- If you deploy your system on a cloud provider, then give links to where you have deployed it.
- Any special instruction on how to run your application.
- If you have special login for users (eg, an admin), write down login/password, so it can be used. If you do not want to write it in the documentation, just provide a separated file in your delivered zip file.

Note about the evaluation: when an examiner will evaluate your project, s/he will run it and manually test it. Bugs and crashes will **negatively** impact your grade.

Easy ways to get a straight **F**:

- Have production code (not the test one) that is exploitable by SQL injection.
- Submit a project with no test at all, even if it is working.
- Submit your delivery in any format different than *zip*.
- Submit a far too large zip file. Ideally it should be less than 10MB, unless you have (and document) very good reasons for a larger file (e.g., if you have a lot of images). Zipping the content of the "*target*" or "*node_modules*" folders is **ABSOLUTELY FORBIDDEN** (so far the record is from a student that thought sending a 214MB zip file with all compiled jar files was a good idea…). You really want to make sure to run a "*mvn clean*" before submitting and preparing the zip file. You might also want to make sure the "*.git*" folder does not end up in *zip* file (in case you are using Git during this exam).

Easy ways to get your grade **strongly** reduced (but not necessarily an **F**):

- Submit code that does not compile. (You might be surprised of how often this happens in students' submissions…)
- If you do not provide a "*readme.md*" at all.
- Skip/miss any of the instructions in this document.

Necessary but **not sufficient** requirement to get at least an **E** mark is:

- Write a new REST API using *SpringBoot* and *Kotlin*.
- Have **AT LEAST** one endpoint per main HTTP method, i.e., GET, POST, PUT, PATCH and DELETE.
- PATCH **MUST** use the JSON Merge Patch format.

- Each endpoint **MUST** use Wrapped Responses.
- Endpoints returning collections of data **MUST** use Pagination, unless you can convincedly argue (in code comments) that they do not deal with large quantity of data, and the size is always small and bounded. Example: an endpoint that returns the top 10 players in a leader-board for a game does not need to use Pagination.
- **MUST** provide Swagger documentation for *all* your endpoints.
- Write **AT LEAST** one test with *RestAssured* per each endpoint.
- Add enough tests (unit or integration, it is up to you) such that, when they are run from IntelliJ, they **MUST** achieve **AT LEAST** a 70% code coverage.
- If the service communicates with another REST API, you need to use *WireMock* in the integration tests to mock it.
- You **MUST** provide a *LocalApplicationRunner* in the test folder which is able to run the REST API independently from the whole microservice. If such REST API depends on external services (e.g., Eureka), those communications can be deactivated or mocked out (or simply live with the fact that some, but not all, endpoints will not work). It is **ESSENTIAL** that an examiner **MUST** be able to start such class with *simply* a right-click on an IDE (e.g., IntelliJ), and then see the Swagger documentation when opening http://localhost:8080/swagger-ui.html in a browser.
- In "*production*" mode, the API **MUST** be configured to connect to a *PostgreSQL* database. During testing, you can use an embedded database (e.g., H2), and/or start the actual database with Docker.
- You **MUST** use *Flyway* for migration handling (e.g., for the creation of the database schema).
- Configure Maven to build a self-executable uber/fat jar for the service.
- Write a Docker file for the service.

Necessary but **not sufficient** requirements to get at least a **D**:

- Your microservices **MUST** be accessible only from a single entry point, i.e., an API Gateway.
- Your whole application must be started via Docker-Compose. The API Gateway **MUST** be the only service with an exposed port.
- You **MUST** have at least one REST API service that is started more than once (i.e., more than one instance in the Docker-Compose file), and load-balanced with *Eureka/Ribbon*.
- In Docker-Compose, **MUST** use real databases (e.g., *PostgreSQL*) instead of embedded ones (e.g., H2) directly in the services.
- You **MUST** have at least 1 end-to-end test for each REST API using Docker-Compose starting the whole microservice.

Necessary but **not sufficient** requirements to get at least a **C**:

- You **MUST** provide a frontend for your application. You can choose whatever framework and language you want, although *React* is the recommended one.

- You need to make sure that all the major features in your application are executable from the frontend.
- Note: there is no requirement on the *design* of the pages. However, a bit of CSS to make the pages look a bit nicer will be appreciated and positively evaluated.

Necessary but **not sufficient** requirements to get at least a **B**:

- You **MUST** have security mechanisms in place to protect your REST APIs (e.g., although GET operations might be allowed to everyone, write operations like POST/PUT/PATCH do require authentication and authorization).
- You **MUST** set up a distributed session-based authentication with *Redis*, and you **MUST** setup an API for login/logout/create of users. Note: most of these can be a copy&paste&adapt from the course examples.
- The frontend **MUST** have mechanisms to signin/signup a user.

Necessary but **not sufficient** requirements to get at least a **A**:

- Besides the required REST APIs, you **MUST** also have a *GraphQL* one. It **MUST** have at least one *Query* and one *Mutation*.
- You **MUST** have at least one communication relying on *AMQP* between two different web services.

Application Topic

The application topic for this exam is about a social-media website (i.e., a very simplified Facebook).

Necessary but **not sufficient** requirements for E:

- REST API to handle user details: e.g., name, surname and email address.
- Need to handle "friendship" requests: eg, a user asking another for friendship, and this other deciding whether to accept it or not
- Need to be able to create new messages on the "timeline" of a user
- When the API starts, it must have some already existing data

Necessary but **not sufficient** requirements for C:

- In the GUI, should be possible to...
- ... search/display existing users
- ... register a new user
- ... visualize the current user details
- ... create a new timeline message for the current user
- ... display all timeline messages of a user, sorted by time

- ... create/accept friendship requests

Necessary but **not sufficient** requirements for B:

- From GUI, must be able to login/logout users. In the backend, this should be handled in a new REST API.
- A logged-in user should see a welcome message
- A user should be able to create new messages only on his/her timeline (**add a test to verify it**)
- A user should be able to see the timelines of only his/her friends (**add a test to verify it**)

Necessary but **not sufficient** requirements for A:

- A new GraphQL API should handle advertisement messages, to show on home-page
- Every time there is a new friendship request accepted in the REST API, such message should be sent to RabbitMQ. The GraphQL API should subscribe to such events, and use the received information somehow (completely up to you) to decide which ads to show next

If you have any time left, add any extra feature relevant to such type of system involving a social network. Remember to discuss any extra feature in the readme file.