PG6100 Group Exam

The exam needs to be done in groups of 3 students, over a period of 2 months. Only in very special cases it will be allowed a different number of students per group. Student groups have to be defined before the exam starts, and will not change during the course, unless in very special situations (eg, certified sickness).

The expected amount of work from this exam will be linearly proportional to the number of students in the group. On average, it is expected that each student spends around 80-120 hours on this delivery, i.e., 10-15 working days. Grades will be based on what can be expected from 3rd year students working 80-120 hours on a project. Students can of course work less or more hours. The range "80-120" is just to give a high level indication of what to expect.

All students in the same group will receive the same grade. In very special cases, students in the same group might get different grades: e.g., if 2 students do all the work whereas the 3rd does nothing. It might well happen that in the same group some students get an **A** whereas others get an **F**. Such assessment will be based on the Git history of the project. For example, during the marking of a project, if an examiner suspects that a student did only little compared to the rest of the group, the examiner can check every single Git commit from that specific student to assess his/her individual contribution to the project, and possibly give a different grade (usually an **F** or an **E**).

There is a *zero-tolerance* policy for slackers. Students are allowed to contact the lecturer if they believe that some of their team members are doing little or nothing by the time of the delivery. All such communications will be treated *fully confidentially*. Such "complains" will have **no** direct impact on the grades. The **only** thing that they would achieve is that the examiners might spend extra care when evaluating each student's contribution. Note: having students in the same group obtaining different grades does **not** imply that there was any complain in the first place. Different grades can be given based on what delivered, and especially depending on the Git history.

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 microservice that you need to build will be composed by different REST APIs. Each student has to be the main responsible for *at least* one service. This means being *the only student* that developed it. A consequence is that, in a group of 3 students, there should be at least 3 REST services. You can write more services if you want. Those extra services can be written in group. Note: the writing of a service should not take 100% of a student's time, as there are some other tasks that will need to be done collectively, e.g., writing documentation and end-to-end tests. It is perfectly fine that students will do different tasks in the group, e.g., one student focusing on the most difficult service while another one is doing more documentation or testing. Still, each single student must be responsible for at least one full service, even if smaller compared to the others developed in the group.

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 (eg from *StackOverflow*) to achieve some special functionalities (but recall to write a code comment about it, e.g., a link to the *StackOverflow* page).

Development **MUST** be done using Git. Make sure to make "sensible" commits, with "sensible" comments. Do not spam your Git history of meaningless commits just for the sake of increasing your number of commits. Recall: an examiner can look at the Git history of your project when deciding whether some students should be graded differently from the rest of the group. It is recommended to use Bitbucket as Git host provider, as it has free private repositories (you can easily move it to GitHub afterwards if you want). However, once the deadline for the submission is passed, it is recommended (but it is NOT a requirement), to open your repository and publish it as an open-source project (e.g., LGPL or Apache 2 license). This is particularly important if you want to add such project in your portfolio when applying for jobs. However, wait at least two weeks before doing it, as some students might have deadline extensions due to medical reasons.

Each student should use only a single Git account / user-name when making commits. Therefore, in a group of 3 students, when looking at the Git history, there should be only 3 different user-names. If a student does *screw up* and make Git commits with different user-names, write about it in the *readme.md* file, specify all the user-names belonging to each specific student.

Pair-programming is discouraged, but not forbidden (apart from the 3 base REST APIs), nor penalized. However, recall that, when students are evaluated individually, what counts is the user-name in the Git history. If 2 students want to do pair-programming, make sure to alternate user-names between commits. Otherwise, it will result like one student did everything while the other did nothing.

You should have a single Git repository for the whole project. You MUST use Maven, where each service is in its own Maven module. You must use Docker-Compose to start your whole system. The project must be self-contained, in the sense that all third-party libraries must be downloadable from Maven Central Repository and NPM (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 project. Compilation failures will heavily reduce your grade.

You must set up a proper "*.gitignore"* file. For example, "*.iml*", "*.idea*", "*node_modules*" and "*target*" must not be on the Git history.

If a student writes more than 1 REST service, there is no special requirement about those extra services. For example, you can write them in any language/framework (e.g., Scala or NodeJS), as long as you configure your services to run in Docker. Note, however, that your entire project MUST be buildable with a single command from Maven (e.g., there are Maven plugins to build JavaScript projects). Furthermore, any number of students can work on the same extra service.

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.

Although it is a requirement to have a GUI, the main topic of this course is the back-end. Therefore, indicatively, you should not spend more than 20-40% of your time on the front-end compared to the back-end.

Documentation is going to play a significant role in the marking of your project, i.e., **20%** of it. You need to have a *readme.md* file (in Markdown notation) in the root folder of your project. To avoid having a too long file, you can have extra "*.md*" files under a "*doc*" folder, linked from the *readme.md* file.

In the documentation, you need to explain what your project does, how it is structured, how you implemented it and which different technologies you did choose to use. Think about it like a "pitch sale" in which you want to show a potential employer what you have learned in this course. This will be particularly important for when you apply for jobs once done with your degree. Furthermore, in the *readme.md* you also MUST have the following:

- Link to where you have your Git repository (make sure examiners can have read access to it after the submission deadline is passed).
- 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.
- Any expected tool that should be installed before your application can be run.
- 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.
- For each student, a brief description of your individual contributions to the project. Also make sure to specify your Git user-names so examiners can verify what you wrote based on the Git history. Furthermore, make sure that, for each student, you specify which is the main REST service s/he is responsible for.

The marking will be strongly influenced by the *quality* and *quantity* of features you can implement. For each main feature, you **MUST** have an end-to-end test to show it, and also you need to discuss it in the documentation. *A feature without tests and documentation is like a feature that does not exist*. For B/A grades, the more features you implement and test in your project, the better.

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.

For the deliverable, you need to **zip** (zip, not rar or tar.gz files) all of your source code and Git history (i.e., the "*.git*" folder). Once unzipped, an examiner should be able to build it with Maven and run it on his/her machine using Docker Compose. You can assume that an examiner has installed JDK **8**, Docker, Chrome Selenium Drivers, NPM and Node. You must target JDK 8, and no other version.

During the exam period, you will need to check often the discussion forum on Canvas. There is not going to be any major change to this document, but clarifications might be posted there. Clarifications made on Canvas will be binding for the evaluation and grading of your exam. If you have questions, do NOT send private emails to the lecturer, but rather post on such discussion forum.

Throughout the course, once the 2-month exam starts, in the 2 hours after the 2-hour lectures, students are allowed to discuss and show their working prototype to the lecturer. It goes without saying that

groups that start early with implementing their systems will be at an advantage compared to the ones that wait until the last moment before starting.

The students have 2 months to complete the project. During such months, students will be busy also with other exams. This is known, and will be taken into account during the evaluation. However, if a group starts to work on the project only at the last moment (and this will be visible in the Git history, e.g., in the past I had groups that started just the day before the deadline…), that will not look so well for the evaluation, unless the project is good (and a very good project started just at the last moment will raise a red flag for extra checks regarding cheating). However, there is no strict requirement stating exactly when students must start, and how to distribute how much work they should do during the 2 months (e.g., if constant throughout the 2 months, or more concentrated in the last couple of weeks).

Students are allowed to re-use and adapt any code from the main repository of the course: https://github.com/arcuri82/testing_security_development_enterprise_systems

However, every time a class is reused, you must have comments in the code stating that you did not write such class and/or that you extended it. For an external examiner it must be clear when s/he is looking at your original code, or code copied/adapted from the course.

Note: if you copy&paste a whole (or large parts of a) REST API from the course, that will not count among the 3 base REST APIs that each student must write by their-self. Example: you might want to copy&paste&adapt the REST API dealing with users and authentication. You will still need to have the other 3 REST APIs.

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 as a rar file instead of a 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). The reason is that your delivery might be needed to be sent by email to external examiners. 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.
- Not including the Git history "*.git"* as part of the delivered zip file.
- Submit in a group that is not composed of 3 students, unless you had **formal** approval from the lecturer.

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 *for each student* to get at least an **E** mark is:

- Write one 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.
- 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 should 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 should 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.
- Among the at least 3 REST APIs (i.e., at least one per student), at least 1 of them must connect to a SQL database like PostgreSQL. During testing, you can use an embedded database (e.g., H2), and/or start the actual database with Docker. Each service responsible for a database must use Flyway for migration handling.
- 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 for the group 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.
- You need 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.
- In Docker-Compose, 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 for the group to get at least a **C**:

- You must provide a frontend for your application. You can choose whatever framework you want, although React in NodeJS is the recommended one.
- The REST APIs will be evaluated in isolation, but, to evaluate your whole microservice, the frontend is going to play a major role for the examiners. 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 for the group to get at least a **B**:

- You need to have security mechanisms in place to protect your REST APIs (eg distributed session-based authentication with Redis). The frontend must have mechanisms to signin/signup a user.
- You need at least one communication relying on AMQP.

Nice to have (but **not necessary**) for **A** grades:

- Deployment on a cloud-service (e.g., AWS or Google Cloud).
- Selenium tests with Page Objects for all the major features in your application.
- Besides the required REST APIs, also have GraphQL ones.

Application Topic

The application topic for this exam is about cinemas. You need to build a microservice representing a system to handle cinema booking of seats, and showing current movies out.

Possible ideas for web services:

- User authentication (basic id and password).
- User details (e.g., email, address, previously purchases of tickets, special discounts and subscriptions)
- Movies: which ones are scheduled this week (and in which rooms in the cinema, and at what times), which ones will be in the future, etc. Also would like to see some descriptions of the movies (e.g., title, director). Likely only admins should be able to edit such info. Could connect to an external real server with movie info to get realistic data (including images).

- Booking of tickets and choice of seats. Note: can simulate actual payment via a fake service that always authorizes and accept payment regardless of validity of credit cards (ie, do not need to handle real credit card payment, nor accessing PayPal).

Required features:

- As a user, on the home page I want to see the list of movies for this week, and be able to see what is in the future. Note: when starting applications with Docker-Compose, want data in databases (eg movie schedule) pre-loaded with some test data (so I can see an actual schedule, and make some bookings once registered a new user).
- Once logged in, should be able to buy a ticket for a movie, and choose a seat (or more if buying several tickets at the same time).
- Administrators should be able to edit the schedule of movies.

Add any extra feature relevant to such type of system involving cinemas. You can take inspiration from existing web sites (e.g., www.nfkino.no).