P5100 Enterprise 1, Mock Exam

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 48 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 *pg5100_<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. *pg5100_701234.zip*. No "*rar*", no "*tar.gz*", etc. You need to submit all source codes (eg., *.java* and *.xhtml*), 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.

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

- Have production code that is exploitable by SQL injection.
- Submit a solution with no test at all, even if it is working.
- Submit your delivery in a different format than *zip*. For example, if you submit a *rar* or *a tar.gz* format, then an examiner will give you an **F** without even opening such file.
- Submit a far too large zip file. Ideally it should be less than 10MB, unless you have (and document) very, very good reasons for a larger file. Zipping the content of the "*target*" folders is forbidden (so far the record is from a student that thought sending a 214MB zip file with all dependencies was a good idea…). You might want to run "*mvn clean*" before crating your *zip* file.

- If you use the library *Lombok*.

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

- Submit code that does not compile.
- Do not provide a *readme.md* file (more on this later) at all, or with missing parts.
- Skip/miss any of the instructions in this document.
- Your *zip* file is not named as instructed, e.g., submit something called *exam.zip*.
- Some parts of the exam are not completed, but it is not specified in the "*readme.md*" file.
- Application fails to start from an IDE (more on this later).
- Home page is not accessible at: *http://localhost:8080/*
- Having bugs in your application when examiners run and play with it.
- If you use mocking frameworks like *Mockito.*
- If you use empty spaces " " in any file/directory name. Use "_" or "-" to separate words instead.

Once you have finished your exam, and the submission deadline has passed, it is recommended (but not compulsory) to publish your solution on a public repository, like GitHub. This is useful to build your portfolio for when you will apply for developer jobs. However, wait *at least* two weeks from the submission deadline before doing it. The reason is that some students might get extensions due to medical reasons. In such cases, those students should not be able to access deliveries made by the other students.

The goal of the exam is to build a simple web application, given a specific theme/topic (discussed later). The application must be implemented with the technologies used during the course. In particular, you must build a *SpringBoot* application that uses JPA to connect to a SQL database, and must use JSF for the GUI. For this exam, you do NOT need to write any CSS or JavaScript, although you will have to edit some HTML. You must **NOT** use any EJB, *Arquillian* nor *Wildfly*. Using *Docker* is a plus, but not a requirement.

The project must be structured in 3 *Maven* submodules, in the same way as shown in class in some of the exercises: "*backend*" (containing Entity, Service, and all other needed classes), "*frontend*" (JSF beans and XHTML) and "*report*" (for aggregated *JaCoCo* report). Note, if you copy and paste those *pom.xml* files from the course repository, make sure that the root *pom.xml* file of your project is self-contained (i.e., no pointing to any parent).

For testing, you can use an embedded database (e.g., H2), or start a real one with *Docker* (or both). End-to-end tests must use *Selenium* with *Chrome*. You can make the assumption that the *Chrome* drivers are available under the user's home folder (as done in class and in the Git repository). Tests must be independent, i.e., they should not fail based on the order in which they are run. NOTE FOR EXTERNAL EXAMINERS: you must have such drivers on your machine when evaluating the project.

You must provide a "*readme.md*" file (in Markdown notation, in the same folder as the root *pom.xml*) where you briefly discuss your solution, providing any info you deem important. You are **NOT** allowed to write it in other formats, like *.docx* or *pdf*.

In the *readme.md* file, you must provide the name of a class that can be used as entry point for testing/debugging your application (e.g., like the *LocalApplicationRunner* used in the course). Your application **MUST** be runnable from an IDE (e.g., IntelliJ) by just right-clicking on such class. Note: this

entry point might not be the main production settings (e.g., which could be set for working on a cloud provider like Heroku), and it **MUST** provide some default data already present in the database (e.g., automatically initialized with a SQL script or a service bean). When running such *LocalApplicationRunner* from an IDE, the database must be automatically started (e.g., you can use an embedded database like H2). The home-page **MUST** then be accessible by pointing a browser to **localhost:8080**.

If you do not attempt to do some of the parts/tasks of this exam, you **MUST** state so in the "*readme.md*" file, e.g., "*I did requirements R1, R2 and partially R3. Did not manage to do R4 and R5*". **Failure to do so will further reduce your grade.**

The Application

In this exam, the topic/theme of the web application is regarding an online travel agency. The application shows a list of available trips, where logged-in users can book them.

R1: (Necessary but not Sufficient for E grade) Backend

In the "*backend*" module, you need *at least* the following 3 JPA entities:

- *User*: having info like name, surname, hashed-password, email, etc.
- *Trip*: having info like title, description, cost, location, time of the year, etc.
- *Purchase*: having info like which user booked which trip, and when.

The actual fields (i.e., table columns) in these entities are up to you, and you will be evaluated on your decisions (and how you structure them). You should read the whole text of this exam to see what kind of fields you might need. You **MUST** add *reasonable* constraints to all the fields of those entities (e.g., a name must not be blank or too long).

You need to write Spring *@Service* classes to provide *at least* the following functionalities:

- create a user
- create/delete a trip
- book a trip
- retrieve all trips, based on queries dealing for example with location, cost and time of the year, sorted by cost

Once the entities and services are finalized, you **MUST** use *Flyway* to initialize the schema of the database. Hibernate/JPA must be configured in the "*ddl-auto:validate*" mode.

R2: (Necessary but not Sufficient for E-D grade) Testing of Backend

Write integration tests for each of the *@Service* classes, using JUnit and *@SpringBootTest* annotation. You should have at least one test for each of the public methods in those services. Enable the calculation of

code coverage with *JaCoCo*. When the tests are run, you must achieve a code coverage of **at least 10%** (for **E** grade) or **70%** (for **D** grade and above) statement/line coverage on the whole "*backend*" module, both in *JaCoCo* (when run in *Maven*) and when run directly from *IntelliJ* with "*Run with Coverage*". Add new tests until such thresholds are reached. Note: it is important that you name the tests in meaningful ways. Tests should be easy to read and understand what they are actually testing.

R3: (Necessary but not Sufficient for C grade) Frontend

In the "*frontend*" module, you need to provide at least the following web pages:

- Homepage: display the top N (e.g., N=5) trips, with info summaries, and links to detail page. If the user is logged in, then display a welcome message. Homepage **MUST** have a query system in which a user can search for trips based on location, sorted by cost (other kinds of searches can be added as well as extra).
- Trip detail page: show all the details of the trip (e.g., description, cost and location). A logged in user should be able to book the trip (if s/he has not already done it).
- User login/signup page, based on *Spring Security* and storing of user info on the SQL database. It should be possible to logout from any of the pages (e.g., via a button). When a login/signup fails, you **MUST** show an error message.
- User account details: besides showing basic info like name and surname, also show the list of booked trips, if any.

When you design these pages, it is **NOT** so important how nice they look. The functionalities that you implement are more important.

Recall that, when running in the application in development mode, the application needs to have some fake/test data already present in the database. In this case, there should be several different trips (e.g., 10-20) with different properties, so that an examiner can test the different search options from the GUI in the homepage.

Note: if it is not possible to start the application from *LocalApplicationRunner* and see the homepage (even if it can be incomplete) at **localhost:8080**, then you will fully fail this exercise requirement regardless of the rest (i.e., you will not be able to get a grade higher than a **D**).

R4: (Necessary but not Sufficient for B grade) Selenium Tests

For each web page, implement a corresponding *Page Object*. Use such *Page Objects* to implement at least the following *Selenium* tests (use the same test names, so an examiner can easily find them in your project):

- *testDefaultTrips*: go to home page, and verify that at least N trips are displayed.
- *testDisplayTripDetails*: go to trip detail page for a trip, and verify you can see its details.

- *testDisplayUserInfo*: verify you cannot access the user info page. Do create a user and login. Go to the user info page, and verify the right data is visible there. Logout. Verify you cannot access the user info page any longer.
- *testBookTrip*: go to a trip detail page. Verify you cannot book it. Do log in with a user, and go back to that page. Do book the trip. Go to the user info page, and there do verify that the booked trip is now displayed.
- *testSearch*: from home page, do a specific query (e.g., based on location), and verify that all the trips matching that query are displayed (must be at least 1), and no other.

Enable *JaCoCo* to collect aggregate statistics of code coverage for the whole application in the "*report*" module. You need to achieve at least a **total code coverage of 80%**. Add new tests until such threshold is reached. What kind of tests to add is up to you, e.g., unit, integration or system tests. The report of the aggregate statistics should be generated when calling "*mvn verify*" from command-line. Recall that bugs in your application will significantly reduce your grade.

R5: (Necessary but not Sufficient for A grade) Extra

In the eventuality of you finishing all of the above exercises, *and only then*, if you have extra time left you should add new functionalities/features to your project. Those extra functionalities need to be briefly discussed/listed in the "*readme.md*" file (e.g., as bullet points). Each new visible feature **MUST** have at least one *Selenium* test to show/verify it, and this **MUST** be specified in the *readme.md* (i.e., which new feature is covered by which *Selenium* test). Note: in the marking, examiners will ignore new functionalities that are not listed in the readme document. What type of functionalities to add is completely up to you.

THIS MARKS THE END OF THE EXAM TEXT