Exam for PG5100, 2018

11:55am Wednesday 2nd May – 11:55am Friday 4th May

This assignment is worth 100% of your final grade for PG5100. The assignment has to be submitted on Its Learning by Friday 4th of May morning at 11:55am, under the "Exam delivery" folder.

The exam assignment will have to be zipped in a zip file with name pg5100_<id>.zip, where you must replace <id> with your own student id, eg pg5100_123456.zip. No "rar", no "tar.gz", etc. You need to submit all source codes (eg., .java and .xml), and no compiled code (e.g., .class and .jar) or IDE files (e.g., .idea). In the past, I got a 175MB and a 214MB files as a submission, containing everything, compiled code included… you can guess what grade the student got… an F. You really want to make sure to run a "*mvn clean*" before submitting.

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 compliable 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: I will run such command on my 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, for a total of 100 points.

Note: during the exam period, I will NOT answer to any email. However, I will answer questions on the discussion forum of the course. I will answer questions regarding possible misunderstanding in the exam's instructions, or more general questions like "can we use library X?". Questions like "How can we do Y?" will be of course left unanswered… At any rate, during the exam you should check the discussion forum often, as I might post some clarifications or other messages of interest.

You can (and should when appropriate) reuse code from:

https://github.com/arcuri82/testing_security_development_enterprise_systems

for example, the pom.xml files. You will also notice that the exam has similarities with the exercises you have done during the course. You are allowed to re-use / adapt your own code, but of course not the one from other students...

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 as a rar file instead of a zip (yes, I do hate rar files)
- 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.

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

- submit code that does not compile
- do not provide a readme.txt file (more on this later) at all, or with missing parts
- skip/miss any of the instructions in this document (e.g., how to name the zip file)
- application fails to start from an IDE (more on this later)
- having bugs in your application that I find when I run and play with it

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. The only exception is questions to me in the PG5100 discussion forum, as discussed earlier. Failure to comply to these rules will result in an F grade and possible further disciplinary actions.

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 need to build a SpringBoot application that uses JPA to connect to a SQL database, and using 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 should use an embedded database (e.g., H2). 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.

You must provide a “*readme.txt*” file (in same folder as the root pom.xml) where you briefly discuss your solution, providing any info you deem important. 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 should provide some default data already present in the database (e.g., automatically initialized with a SQL script or a service bean). The home-page must then be accessible by pointing a browser to **localhost:8080**.

If you do not attempt to do some of the exercises, you must state so in the “*readme.txt*” file (this will make me correct your exam much faster), e.g. “I did not do exercises X, Y and Z”. Failure to do so will reduce your grade.

UsedBooks Application

In this exam, the topic/theme of the web application is the trading of used books in a university. The application should show a list of books used in different courses at the university. A logged in user should be able to mark if s/he has a given book and s/he is willing to sell it. A logged in user should be able to express interest (e.g., by sending a message) in the buying of specific books.

(E1, 20pt) Backend

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

- *User*: having info like name, surname, hashed-password, email, etc.
- *Book*: having info like title, authors, course in which it is used, etc.
- *Message*: having info like the text of the message, the sender, the receiver, etc.

The actual fields (i.e., table columns) in these entities are up to you, and you will be evaluated on your decision (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 book
- (un)mark that a user is willing to sell a copy of a specific book
- retrieve all books
- retrieve all books for which there is at least one student that can sell a copy of it
- send a message from a user to another user

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.

(E2, 15pt) 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 with Maven, you must achieve a code coverage of at least 70% statement/line coverage on the whole "backend" module. Add new tests until such threshold is 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.

(E3, 20pt) Frontend

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

- Homepage: display all books (info summaries). If the user is logged in, then display a welcome message, and, for each book, enable possibility to register the wishing of selling it. Each book entry must show how many sellers it has, and link to a "book details" page.
- Book detail page: show list of sellers (if any), and provide message form to contact them (only if current user is logged in).
- Message page: for a logged in user, show all received and sent messages, sorted chronologically. Provide message form to reply to messages.
- 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 you design these pages, it is NOT so important how nice they look. The functionalities that you implement are more important. For example, it goes without saying that, when you display a sent message, you should show the name of the receiver. Another example: if the list of seller is empty, instead of showing a table with just headers and no rows, you could rather display a message stating there is no seller yet.

(E4, 15pt) 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 I can easily find them in your project):

- *testDefaultBooks*: go to home page, and verify that at least 2 books are displayed (there should be at least 2 books by default initialized in the database).
- *testRegisterSelling*: from home page, without being logged in, verify you cannot mark the selling of a book. Do log in with a user. Verify that you can mark the selling of a book. Mark the selling of a book. Verify the counter of sellers for that book has gone up by 1, whereas all the other books stayed the same. Unmark the selling of such book. Verify the counter went down to its original value. Mark it again, and verify the counter goes up by 1. Do logout, and verify the mark stays the same.
- *testBookDetails*: log in with a user X (actual id is not important) and mark a book for selling. Go to the book details page for that book, and verify that X is listed as seller. Logout. Do log in with a different user Y. Go to the same book details page. Verify that X is still listed as seller.
- *testMessages*: make sure that both user X and Y exist (actual ids are not important). Log in with X, and mark one book for selling. Log in with Y, and send a message to X (actual content is not important). Verify such message appears in the message page. Log in with X, and verify that the message from Y is visible in the message page. Send a reply. Log in with Y. Verify that both messages (the one sent to X and the reply) are visible and sorted correctly.

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 90%. 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.

(E5, 30pt) Extra

In the eventuality of you finishing all of the above exercises, and only then, if you have extra time left you should add functionalities/features to your project. Those extra functionalities need to be briefly discussed/listed in the "*readme.txt*" file (e.g., as bullet points). Each new visible feature must have at least one Selenium test to show/verify it. Note: in the marking, I might ignore functionalities that are not listed in the readme. What type of functionalities to add is completely up to you. Example of a possible new feature and description: "possibility to register books from the GUI, but only for administrators. The Selenium test for it is called X and can be found in file Y".

THIS MARKS THE END OF THE EXAM TEXT