

First and lastname:

Programme: **Applied Computer Science**

Minor: **AIBP / CSP / IBC / SE**

Course: **Object Oriented Architectures & Secure Development**

Lecturer/lecturers: **M. Blomme, M. De Wael, F. Vlummens**

Proofread by: **Matthias Blomme**

Academic year: **2022-2023**

Semester: **3**

Date and time: **16/08/2023, 13:30**

Session: **2**

Exam time: **3h15min + 45min extra time (=inclusive exam facility). Handing in at 17:30 the latest.**

EXAM

RESULT /20

SOURCE FILES

All required source files which are referenced in this assignment can be found under Leho under the header **“EXAM AUGUST”**. Here you will also find a **screencast** that gives more details about the application to be developed.

IMPORTANT

Every form of (on-line) communication between students or other parties is strictly forbidden.

This is an individual assignment. Should an irregularity (such as phone usage, cheating, copying, hacking, use of social media clients, ...) occur, this will lead to a notification of both the student(s) involved and the chair of the examination board, as defined in the Education and Examination Regulations (EER).

After completing this exam, it is forbidden to publish this assignment or the solution using any means possible, except for in your individual repository on the Gitlab server, which has been created for that purpose.

Read the assignment carefully and completely before beginning the exam.

ALLOWED SOURCES

You may use your own class notes, slides, books, syllabi, and other materials, including the internet. Communication with fellow students or third parties is strictly forbidden (see above). Make sure that any applications that might pop up or launch automatically or may be interpreted as an attempt of fraud (Outlook, Messenger, Facebook, ...) are closed!

HANDING IN

Handing in your complete project is done through the ACS department's Gitlab server. You have received an individual repository through the address <https://git.ti.howest.be/TI/2022-2023/s3/object-oriented-architectures-and-secure-development/exam/08-august/firstname.lastname> (own first and lastname, no accents on letters, spaces replaced by dots, e.g. Frédéric Vlummens → frederic.vlummens).

Make sure you commit on **remote main**, as this is the branch that will be graded by the lecturers.

You are required to commit and push on a regular basis:

- At least one commit every 20 minutes
- Each commit starts with your initials (e.g. Frédéric Vlummens → FV)
- At least one push per hour
- A final commit to complete handing in (see below)

Add a final commit (and push) to officially hand in.

The final commit message must contain the following message:

I, *lastname firstname*, hereby agree that this is my final version of the exam and that I can no longer modify it. I have completed this exam in good conscience as was required and without any fraudulent behaviour.

(replace *lastname firstname* by your own last and first name).

Verify for yourself that your project has been submitted on remote main in the correct repository.

When finished with this, head towards an available lecturer to **go over the following run-time checklist together**.

Finally, hand in this assignment and sign the presence list before leaving the exam room.

RUN-TIME CHECKLIST

Note: this is only part of the total evaluation!

Criterion	OK?
Successful compilation	
Login works	
Sending a message	
Message arrives on server	
Retrieving new messages using button	
Messages persisted in text file	
Number of tests	
Number of tests that pass	

SCORING / 20

- Run-time checklist / 5
- Code, code quality and (secure coding) best practices
 - UI / 3
 - Service and logic / 5
 - Data / 5
- Testing / 2

EXAM ASSIGNMENT

1. Concept of the application “MessageBroker”

You are developing a chat application to transmit messages between two parties. Communication is done through a server, which you will get as a JAR file, downloadable from Leho. How to execute the JAR file is described at the end of this assignment in appendix 1 and has been practiced prior to the exam.

How the application works:

Alice wishes to send a message to Bob. The following can be observed:

- Alice logs into the application through an FX login screen, using her username and associated password.
- She gets to see a second screen, containing two tabs: one for the “incoming” messages and one for the “outgoing” messages.
- In “outgoing” all messages that have been sent in the **current session** by the logged in user are displayed (so in this example, Alice).
- In “incoming” all messages the user has **ever gotten** (so not only for the current session) are displayed. In this example therefore all messages Alice received from other users, also for example from the day before.
- To make sure the “incoming” messages are refreshed, a Refresh button is to be provided. When clicking the button, the application retrieves newly incoming messages from the server (see below) and adds them to the list (make sure that the new messages are also ‘remembered’ for later).

2. Message persistence

As described before, “incoming” messages may not get lost. Therefore, the client stores them in a text file called **username.txt**, with *username* the name of the receiver. Messages for Alice are therefore stored in a file **Alice.txt**.

This needs to be done line based. Since the receiver is always the logged in user, there’s no need to write that name towards the file. What you will need to store on one line is:

- The unique timestamp
- The name of the sender
- The actual message

Upon launch you therefore retrieve all existing (old) messages from this file, but also do not forget to append newly received messages at the end of the file.

3. Chat server

The chat server is provided and is to be queried to read the most recent messages. Also sending messages is done through the server.

To allow for all of this, the server supports different request types. You were already able to practice the HELLO request prior to the exam, the following request types are of use for this exam:

- SEND
- RECEIVE

The server is also line based. A request towards it always consists of one line. The response can consist of one or more lines, with one message per line. Per line, content is tab delimited.

SEND

Sending a message to another user, has the following format:

SEND <sender> <payload>

1. First comes the word SEND in capital letters.
2. Next a tab (in a Java string you use `\t` for this).
3. Followed by your own name (=the name of the logged in user).
4. Next another tab.
5. And finally the payload.

The payload is the username of the (existing) receiver, followed by a tab and then the actual message. We call this the payload because this data (full string consisting of receiver + tab character + message) is encrypted using the **sender's** password (the logged in user). Use the **JaSypt StrongTextEncryptor** for this.

Note that it is **not** necessary to transmit the timestamp: this is appended by the server upon receipt.

Two situations may now arise:

- A. Transmission works. You will get as answer from the server a line starting with the string **SEND success**.
- B. Transmission fails. You will get as answer from the server a line starting with the string **FAILED**

SEND <message-or-cause> (tab-delimited).

The most common reasons for failure are:

- One of the users (sender/receiver) does not exist.
- The password of the receiver with which the payload is encrypted, is incorrect (or the encryption did not go as intended).

RECEIVE

Requesting new messages from the server has the following format:

RECEIVE <receiver> <epoch> <nano>

1. First the word RECEIVE in capital letters.
2. Followed by a tab.
3. Next the name of the receiver (therefore the logged in user).
4. Followed by another tab.
5. And finally, the time as two longs (see description of **Instant** in appendix 2), also separated by tabs.

Two situations may arise:

- A. When the receive works, you get one line **per** message (only the messages **after** the time specified using the two longs). If there are no new messages, you do not get any lines, but the server immediately closes the connection.

Each message is one line, but the full line is encrypted by the password of the receiver. After decryption, you get the following format:

<epoch> <nano> <sender> <receiver> <message> (tab-delimited).

- B. When receiving fails, you get a line as answer starting with **FAILED RECEIVE <message-or-cause>** (tab-delimited).

5. Non-functional requirements

Write clean code: the fact that some messages come from a text file, whereas others from a server, should not be visible in the client (code). Furthermore, it should be simple to switch from text file code to for example database code or to replace the server code with database code (or something else). Indeed, both the text files and server are message sources, be it in a different 'version'. Although no grades will be assigned for it, writing a mock version can help with the structured completion of this assignment.

6. Unit testing

Provide at least one unit test (according to the style and steps studied) which tests the communication with the real server by your code. This way, you won't lose endless time launching your FX/GUI and sending/receiving messages in case a small detail should not work.

The test passes when the communication works and fails if it doesn't work.

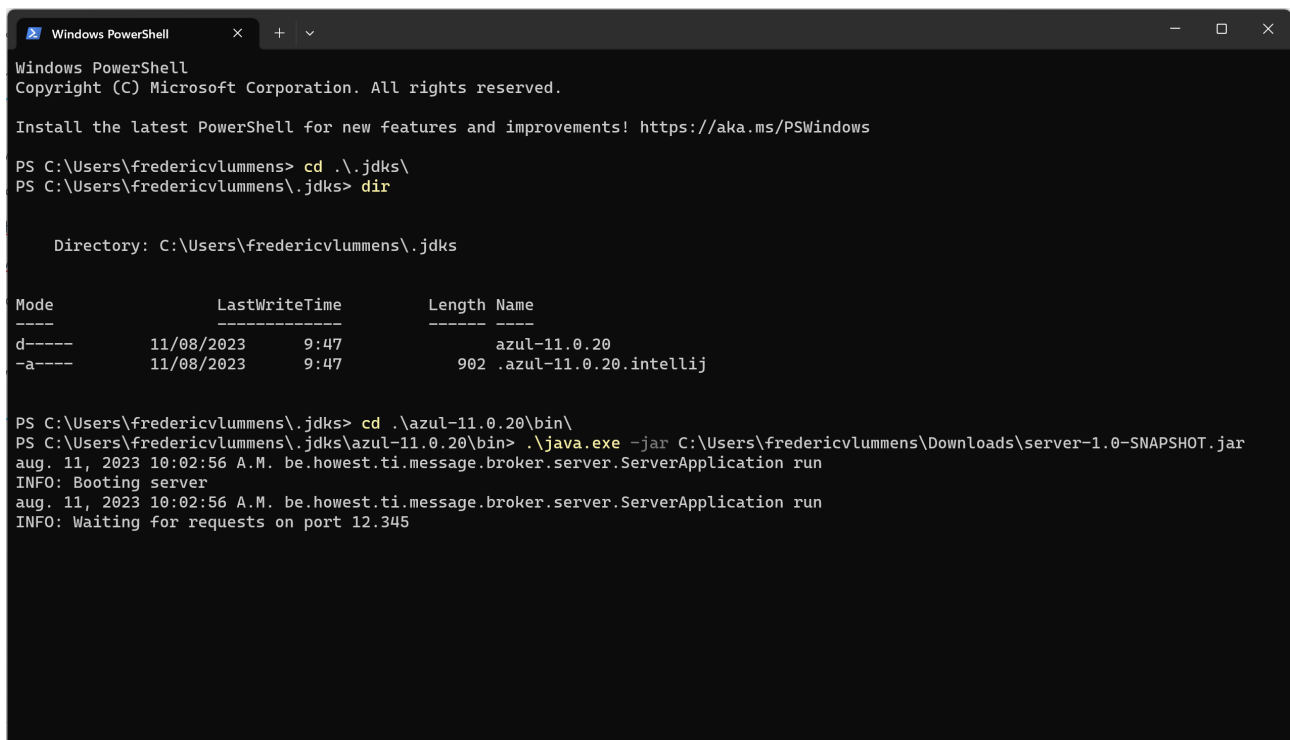
7. General

- Allow for clean and readable code
- Always apply the *best practices* as well as *secure coding guidelines* as studied in class.
- Use the Stream API where relevant.
- All error messages for the user are displayed in a control part the user interface, destined for this purpose.
- No technology specific exception (DB, file, network, ...) is allowed to reach the end user. Translate all these exceptions towards your own exception. However, do log the original exception for the *dev(ops)* people.

APPENDIX 1 – Executing the JAR file

1. Download the JAR file and place it somewhere on your computer.
2. Next, open a PowerShell and navigate to the folder `C:\Users\your-username\.jdk`
3. In this folder, you will find a subfolder for each JDK you have installed earlier using IntelliJ.
4. Navigate to the bin subfolder in the desired JDK.
5. Next, execute the command `java -jar C:\path-to-the-jar\server-1.0-SNAPSHOT.jar`

Below you see an example on a Windows test environment:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\fredericvlummens> cd .\.jdk\
PS C:\Users\fredericvlummens\.jdk> dir

Directory: C:\Users\fredericvlummens\.jdk

Mode                LastWriteTime         Length Name
----                -
d-----          11/08/2023   9:47             azul-11.0.20
-a-----          11/08/2023   9:47           902 .azul-11.0.20.intelli

PS C:\Users\fredericvlummens\.jdk> cd .\azul-11.0.20\bin\
PS C:\Users\fredericvlummens\.jdk\azul-11.0.20\bin> .\java.exe -jar C:\Users\fredericvlummens\Downloads\server-1.0-SNAPSHOT.jar
aug. 11, 2023 10:02:56 A.M. be.howest.ti.message.broker.server.ServerApplication run
INFO: Booting server
aug. 11, 2023 10:02:56 A.M. be.howest.ti.message.broker.server.ServerApplication run
INFO: Waiting for requests on port 12345
```

The server is now running locally, and as you can see listens on port 12345.

Accounts and passwords

On the server, the following accounts have been predefined.

Account	Password
Alice	Alice123
Bob	Bob123
Cheryl	Cheryl123
David	David123

APPENDIX 2 – Code tips and tricks

Timestamps:

To get a unique timestamp in Java, use the built-in class **Instant**. Every “instant” is uniquely defined by two longs, the epochSecond value and the nano value. You can request them as follows:

```
timestamp.getEpochSecond();  
timeStamp.getNano();
```

To convert two longs back to an Instant somewhere else, use a *factory method* as follows:

```
Instant.ofEpochSecond(e, n);
```

Splitting on <tab>:

To split a String in pieces based on <tab> signs, it can be useful to know that in the method split of String you need to use “\\t” (so two slashes instead of one).

APPENDIX 3 – Roadmap

Writing a working application all at once (GUI-file-server-...) is nearly impossible. Therefore, support the techniques we studied and practiced to structure and implement your code. The roadmap below is a suggestion that can help you further to tackle this exam assignment in a structured way. Feel free to change the order of items 2-3-4 according to your own strengths or habits, or you can use your own roadmap...

1. Read and understand the assignment, make a schema/design of the entire application for yourself.
2. Develop a working JavaFX application according to the best practices, making sure all buttons and features work, but with fake or no data.
3. Implement reading (getAll) and writing (add) messages to file. Tip: write one (or more) unit test(s) to test this code. This way, you are sure it is working.
4. Implement reading (getAll) and sending (add) messages through the server using encryption. Tip: write one (or more) unit test(s) to test this code. This way, you are sure it is working and you also complete the requirements for §6.
5. Combine 2-3-4 into one working application.