Software Design 236700 — Assignment 2

Teaching Assistant: Dor Brekhman <brekhman.d@campus.technion.ac.il>

May 20, 2022

1 General remarks

- For any questions that are not personal please use Piazza: `piazza.com/technion.ac.il/spring2022/236700`.
  If you want to e-mail the TA please include "software design hw1" and your name in the subject of the e-mail.
  You are REQUIRED to follow the piazza for announcements.
- Please think of the environment and *do not* print out this document!
- This assignment is written in English. If you have any trouble understanding the intention, please contact the TA.
- We encourage you to read all relevant documents & files from the course websites before starting work on this assignment. Please also note the existence of an FAQ, I'll try to keep it updated.
- Submission is electronic via the course website.
- You *must* work in pairs. The number of team-members is two (2) for each team. A team with a single member is not enough, and three members are too much—four is right out. Two students for each submission, no more and no less.
- You are allowed to switch homework partners, simply correct `README.md` .
- The course staff are most familiar with IntelliJ, therefore will only able to provide technical support for that environment. However, we will still attempt to assist you if you choose to forgo an IDE and use only the command-line tools.
- Any updates and changes to the assignment will appear *in this document*, and will be tracked in section A on page 10. You should occasionally refresh this document from the course website, and be sure to go through it before your final submission. Also note the date shown at the top of this page.
- This purpose of this assignment is learning how to use Monads,
  in particular `CompletableFuture<T>` .

- Your library and app subprojects will be used in the next assignment by other students to make the client side, so make they includes all the needed functionality of SifriTaub, in addition to your high-level storage abstraction.
  The pair with the most used library will get bonus points.

Contents

2 Changes to the Architecture

The base architecture remains the same, with a project for tests (to be used by course staff), a project for the main application implementing the business logic, a project for a `library` implementing the underlying data-structures, and an external library implementing read/write functionality.

The various sub-projects are connected to one another via GRADLE, but as before you are allowed to modify the build files. For more information, consult the previous assignment.

3 The SIFRITAUB™ Application

3.1 Refactoring

The existing API was modified so that if a method previously returned an instance of `T`, it now returns an instance of `CompletableFuture<T>`. You'll need to change your code and tests to match this new interface. Even the methods that previously returned `Unit` now return `CompletableFuture<Unit>`.
The expected semantics are that when the future is ready, the operation is finished. For example, if `SifriTaub™::authenticate()` is called, then when the returned future is ready then the login operation is complete, including any writes to the external storage.

3.1.1 Exceptions

Some exceptions require asynchronous access to the database, and will therefore be thrown from inside the `CompletableFuture<T>`. If an exception does not require asynchronous operations and can be checked outside the monadic context, it should be thrown instantly from the calling method. In general, all exceptions should be thrown as soon as possible: From the calling method, if possible, and otherwise from inside the monadic context.

Hint Look at `CompletableFuture::exceptionally` to see how CompletableFutures handle exception throwing in the monadic context. You can look at the supplied tests to gain some insight into this mechanism if you do not yet fully understand it.

3.2 New Features

Our system now supports user management and adding books to the library, and we are finally ready to start working on the loaning, to allow users to loan multiple books for their project.

3.2.1 Loan request

A loan request is a request for one or more books, where all the books have unique ids, meaning that there is only one copy from a specific book. A loan can be **obtained** only **when all the books are available**.

3.2.2 Loan Request Queuing

The updated interface now includes methods for submitting, canceling, and reviewing loan requests.
The system manages a queue of loans and once a user's loan is at the top, the user gets an obtained loan object. This object has list of "book approvals" which the user uses to physically get the book from the library.

IMPORTANT listBookIds() now should list only the available books (that have one or more copy).

3.3 External library

The external library implementing the database has been upgraded, and now uses `CompletableFuture<T>` :

```
interface SecureStorageFactory {
    fun open(name: ByteArray): CompletableFuture<SecureStorage>
}

interface SecureStorage {
    fun write(key: ByteArray, value: ByteArray): CompletableFuture<Unit>
    fun read(key: ByteArray): CompletableFuture<ByteArray?>
}
```

The semantics are exactly the same as the previous exercise, with the additional guarantee that the library is thread-safe—you may safely call read from multiple threads. Note that when writing the new value is only guaranteed to be available to future reads when the `CompletableFuture` is done. As usual, a dummy implementation is supplied along with the skeleton.

You are now also provided with a second external library, `sd-loan-service` , which provides the following interface.
This interface is a registery for the library and **must** be used when users loan or return books.

```
interface LoanService {
    /**
     * Marks a single book as loaned in the library (for registry)
     *
     */
    fun loanBook(id: String): CompletableFuture<Unit>

    /**
     * Marks the single book as available again (for registry)
     */
    fun returnBook(id: String): CompletableFuture<Unit>
}
}
```

A dummy implementation is provided (`LoanServiceImpl`) which is not bound to the interface (see Section 3.7 below). You can assume that all errors thrown from within this interface are thrown inside the monadic context

3.4 Robustness to system failures

As before, a system can restart abruptly in all cases. An exception is made for loan requests: The system is guaranteed to stay alive if the loan queue is not empty, or there are users that didn't return their books. This simplifies the workflow and you can ignores such cases in your testing.

3.5 Time limits & limitations

All the time limits discussed here are in regards to the time it takes the future to complete. A future should be returned immediately.

3.5.1 External library

The external library is still slow, and has the following properties:

1. The open method is linear in the number of databases created, 100ms for each database. So, after creating 10 databases with unique names, the method will complete after 1 second each time it is called to open one of those names.
2. The write method has constant time complexity. It completes immediately.
3. The read method is linear in the size of the returned value, 1ms for each byte returned. So, if the returned value is "softwaredesign" (16 bytes in UTF-8 encoding), the method will complete after 16ms. Again, the length of the key does not influence time complexity.
4. The loan service library is fast and all operations are $O(1)$ time complexity.

3.5.2 Test sizes

Each test can include up to one million (1,000,000) users and up to 1,000,000 books. There can be up to one million total loan requests (running + obtained) in the system.

3.5.3 Test time limits

The entire test suite has a time limit of about 10 minutes (this value may be adjusted according to the size of the test suite.). This includes loading users and adding books to the library.

Each test can contain up-to 5 read queries, a single create, update or delete call, and any number of constructor calls, and should take up to 10 seconds. The test may have a setup or teardown part which is not counted in the 10 seconds, but is counted for the overall time limit.

Tests that include listing book ids will have a more generous time limit of 30 seconds and will list at most 10 book ids at a time (But you can definitely pass them in under 10 seconds).

3.5.4 Concurrency

The tests will *not* be concurrent, i.e., the test will call a method, wait for it to complete (using the `CompletableFuture`,) and only then call another method. Methods will not be called simultaneously from different threads. The provided sample tests are written to guarantee this. Note however that if the work continues after the returned `CompletableFuture` is ready, you may *still* have concurrency related bugs.

3.6 Dummy implementation

For your benefit, a dummy implementation of the persistent storage layer and the loan service is provided. It acts the same as before. You are not required to mock the storage layer and are free to implement your tests however you wish. You should not implement a real system for loaning books for the loan service, but check in your tests if the loan service was called as specified in the docs.

3.7 Dependency injection and GUICE

You must use GUICE in this assignment. Write your classes to use dependency injection, and use `@Inject` in any class where dependency injection is relevant. You must implement `SifriTaubModule` so that it contains all of your bindings (you can use several modules if that is more useful to you, but our tests will use only `SifriTaubModule` .)

You are provided with `SecureStorageModule` , which provides a binding for `SecureStorageFactory` (a default, dummy implementation). That module and the fully-implemented module that we will test your code with have no binding for `SecureStorage` ; you should think about how to dependency inject `SecureStorage` using GUICE.

Note that our tests *only* use `SifriTaubModule` (see the staff test for an example.) You need to configure that module to include `SecureStorageModule` .

As for the loan service, you are provided with an interface and a GUICE module. Again, we do not explicitly load this module, and you should determine how to load it through the `SecureStorageModule` .

4 Tips & Tricks & Hints

1. This assignment in inherently asynchronous, and if you're not careful you'll accidentally operate in parallel, creating race condition bugs. Make sure you understand monads and use monad composition. Specifically, you should never let a `CompletableFuture` drop out of scope unless you've either composed it into another `CompletableFuture` or are somehow sure that it's completed.
2. Read the `CompletableFuture` documentation. There are a lot of useful things there. The KOTLIN extension method library linked in the tutorial is also nice.
3. Parallelism is nice when reading from the storage since you can wait for things in parallel.
4. Your laptop might not have enough cores to exhibit failures due to race conditions. Try running on `csl3` .
5. Don't use `join()` , except in tests.
6. Add a logger to your code. It's a very useful thing to have, and you can just turn off the messages before submitting.
7. Try to get GUICE to do things for you by smart configuration instead of doing things manually.

5 Documentation

In the next assignment, you'll use someone else's library and they'll be using yours. In order to prepare for this, document your entire project (`library` *and* `techwm-app` , except the tests). Follow KOTLIN's instructions on how to document code (https://kotlinlang.org/docs/reference/kotlin-doc.html). We have included a configuration for DOKKA, Kotlin's documentation engine (https://github.com/Kotlin/dokka), and you can run

```
gradle documentation
```

To generate the relevant documentation files under `build/docs` . The generated documentation should *not* be submitted, we'll re-generate it ourselves.

6 Dry Part: Monads

Prove that the AbstractFactory design pattern can be made into a monad (Reminder, an `AbstractFactory<T>` implements a single method `get()` that returns a `T`):

1. Define `of` , `flatMap` , and any other required methods,
2. Show that the monadic laws (from the tutorial) hold for this definition.

7 Administrivia

7.1 Late submission

The late submission penalty is $n \cdot 5$, where n is the number of days after the submission date. Late submission closes after 5 days.
The exceptions to this rule are reserve duty (Miluim), and very exceptional circumstances to be decided by the TA. In any case, special extensions must be requested well in advance. Last minute extensions are unlikely to be granted, and extensions *after* the submission date will *not be granted* at all, even for reserve duty.

7.2 Your grade

You will be graded based on the following criteria:

1. Number of tests passed, including old tests (Automatic.)
2. Light code-review. If your design is deeply problematic, you will be penalized for that. However, if your design could be better, you will not be penalized, and you'll get feedback (Manual).
3. The existence and quality of your tests, and that they pass (Manual).
4. Fixing any issues mentioned in the feedback from the previous assignment (Manual).

8 Submission Instructions

The project must be submitted as a ZIP file. We have provided automation for this purpose in the form of a GRADLE task that zips the entire project directory into a file named `submission.zip`. To use it, run the following command from the root directory of your project (the `assignment1` directory that contains everything else) or from INTELLIJ in the prompt that appears after pressing "Ctrl" twice:

```
gradle submission
```

In addition to writing your code, you should fill out `README.md` with your names, ID numbers, and project documentation, and add your answers to the dry questions in section 6 on the previous page as a PDF file named `dry.pdf`. Everything should be in the project root directory, and subsequently in the `submission.zip` file.

You *must* make sure the submission file is ***less than 1MB***. Larger submissions will be silently ignored! Use GRADLE to decrease the size of your submission. Make sure you do not submit any log files, compiled binary files (e.g., .CLASS or .JAR files) or other spurious baggage. Look at the `exclude` rules in the `build.gradle.kts` to change what is not included with the submission.

Make sure to *test* your submission after creating the zip file!

8.1 Submission checklist

- Did you correctly merge your old code into the new skeleton?
- Did you add all bindings to the GUICE module?
- Did you use the provided external library, remembering not to use any other persistence methods?
- Did you write tests for everything? How's your coverage?
- Do the provided minimal tests pass (with an implementation of the external library)?
- Did you answer the dry question and add a PDF?
- Did you fill out `README.md`?
- Did you extract the submission zip to a different location and make sure it compiles, including running `gradle test` *from the command-line*[1]?

If you answered yes to all, you're good to go! ☺

And most importantly, have fun!

[1] Your submission will be tested from the command-line, so make sure that works!

A Changelog