NATIONAL UNIVERSITY OF SINGAPORE

Department of Computer Science, School of Computing

IT5100B—Industry Readiness: Stream Processing

Academic Year 2023/2024, Semester 2

ASSIGNMENT 2

REACTIVE PROGRAMMING AND EVENT STREAMING

Due: 23 February 2024, 11.59pm **Score**: [30 marks] (30%)

INSTRUCTIONS

For this assignment, you will need:

- An IDE or text editor to write, compile and execute Java programs with JDK 11+.
- Templates and supporting files (bundled together with this document).
- Docker (optional, for running your Apache Kafka® cluster).

This assignment is worth 30% of your overall grade. The objectives of this assignment are:

- To introduce you to the `Flux` and `Mono` APIs.
- To introduce you to reactive programming.
- To get you started with working with Apache Kafka®.

You will need the following dependencies for this assignment, which you can download from Maven through your IDE:

- `io.projectreactor.kafka:reactor-kafka:1.3.22`
- `io.projectreactor:reactor-core:3.6.2`
- `org.apache.logging.log4j:log4j-slf4j-impl:2.22.1`

You should only submit `ReactiveExercises.java`, `UserStateChangeEventProducer.java` and `UserStateChangeEventConsumer.java` to Canvas. Contact `yongqi@nus.edu.sg` if you have queries.

REACTIVE PROGRAMMING [15 marks]

Project Reactor gives us a nice set of tools to work with reactive data streams declaratively. Ensure all your programs are fully non-blocking and reactive—go reactive all the way, and don't break the chain! All your answers should be written in `ReactiveExercises.java`.

Question 1 (Happy Sum) [3 marks]. The happy sum of n is defined to be

$$n^2 + (n-1)^2 + \cdots + 4 + 1 + 4 + \cdots + (n-1)^2 + n^2$$

Complete the `happySum` method that receives `n` and produces its happy sum. You may assume that `n` is positive. You should make use of the `Flux::range` method to generate a `Flux` of integers from 1 to n (inclusive). Example runs follow:

```
happySum(1).subscribe(System.out::println); // 1
happySum(5).subscribe(System.out::println); // 109
```

Question 2 (Moving Averages) [4 marks]. In Assignment 1, the `Stream` API does not expose a simple windowing mechanism. It's a good thing for us that `Flux` does. Again suppose we have an IoT sensor recording air temperatures every day. What we can do is to transform a stream of temperatures into a stream of n-day moving average temperatures by:

1. Windowing the stream of temperatures with window size n.[1]
2. For each window, obtain the average.

Complete the `movingAverage` method that receives a `Flux` of temperatures and `n`, and produces a stream of `n`-day moving average temperatures. Example runs follow:

```
Flux<Double> temps = Flux.just(1.0, 2.0, 3.0, 4.0, 5.0);
movingAverages(temps, 3)
        .subscribe(System.out::println);
// 2.0, 3.0, 4.0
```

> **Tip**: Unlike `DoubleStream`s, `Flux` does not expose any mechanism to obtain averages. Thus, you might want to define a simple data structure that keeps track of sums and lengths, and use the `Flux::collect` method to collect each window into your data structure.

Question 3 (Cooking API) [4 marks]. Once again we are going to perform a simulation of our cooking example. Suppose we want to create a static method `chickenChop` that returns a chicken chop. Cooking a chicken chop requires several steps:

1. Get a chicken and grill it
2. Get a serving of fries and fry it
3. Assemble and deliver

In this simulation, you are provided with several helper methods in the `FoodService` class that achieve these:

- Getting chicken and fries from the refrigerator (`getRawChicken` and `getRawFries` respectively)

[1] Because `Flux` only places upper bounds on window sizes, remember to filter out windows with less than n elements.

- Grilling and frying chicken and fries (`grillChicken` and `fryFries` respectively)
- Assembling and cooking the fries (`assemble`)

> **Warn**: Do not directly call the `Chicken`, `Fries` or `ChickenChop` constructors or methods. Use the methods exposed in `FoodService` to do what you need to (imagine we are actually retrieving these data from external services like databases and external REST APIs).

Write the static method `chickenChop` that returns a chicken chop reactively. Example runs follow:

```
chickenChop().subscribe(System.out::println);
/* got raw chicken after 1.0s
   got raw fries after 1.0s
   got grilled chicken after 1.0s
   got cooked fries after 1.0s
   got chicken chop after 1.0s
   chicken chop */
```

> **Tip**: When composing reactive operations (do `f` **and then** do `g`), use `flatMap`.

Question 4 (Programming Style) [4 marks]. You will receive the remaining four points if your solution is well-written. All your code should be reactive and non-blocking.

EVENT STREAMING [15 marks]

In this section, we are going to try streaming events to an Apache Kafka® cluster and using it as a data store. First, let's set up the cluster:

- Start up an Apache Kafka® cluster with three brokers with external listeners at `localhost:9092`, `localhost:9093` and `localhost:9094`.
- Create a topic called `users` with three partitions and a replication factor of 2.
- For convenience, set the retention period of this topic to around one minute (`retention.ms=60000`).

In this part, you will be completing the `UserStateChangeEventProducer` and `UserStateChangeEventConsumer` classes.

Question 5 (Producer) [6 marks]. Complete the `send` method which receives a `UserStateChange` event and returns a `Mono` such that when subscribed, sends the event to your Apache Kafka® cluster.

> **Tip**: To see your producer in action, you can run a console consumer (`kafka-console-consumer.sh`) on one of the Apache Kafka® brokers, allowing you to see events being streamed into your cluster. You are provided with two classes, `Dashboard` and `RandomEventProducer` that help you produce events. You may run the `main` methods in either of these classes to see events being streamed into your cluster, assuming your `send` method works correctly.

Question 6 (Consumer) [6 marks]. First, fill in the `changeStateOfUserInMap` from Assignment 1 (you may just need to change the type of the ID to `UUID`, but your implementation should otherwise be identical). Then, complete the `main` method which consumes events from your Apache Kafka® cluster, such that every time an event is consumed from the cluster, the program shows the latest state of the data store. Note that your `main` method is allowed to `block` (or `blockLast`) so that the program continues to run while waiting for events to be streamed from Apache Kafka®.

Then, run the `main` methods in `RandomProducer` and `UserStateChangeEventConsumer` to see events being streamed into your cluster and seeing the updated data store every time an event streams into your consumer.

> **Tip**: Previously, we used a parallel `reduce` function to reduce the state change events into an `ImmutableMap`. This time, we do not want to `reduce` because we are streaming events from Apache Kafka® boundlessly. Instead, use the `scan` method which emits the `ImmutableMap` every time an event is produced by the consumer `Flux`.

Question 7 (Programming Style) [3 marks]. You will receive the remaining three points if your solution is well-written.

– End of Assignment 2 –