Assignment 3 Report

Software Architecture and Platforms

Iorio Matteo, Furi Stefano

March 24, 2025

1 Introduction

We have developed our assignment based on the solution proposed for the Assignment 2, meaning that the Ubiquitous Language and the bounded context identified are kept inside this solution. This assignment is then a refinement of the previous one, which explores advantages and challenges in adapting an Event Driven Architecture, applied to two main domain boundaries as a use case: Users and E-Bikes.

2 Event Driven Architecture Implementation

2.1 What is Kafka

Apache Kafka is a distributed data streaming platform for publishing, storing, and processing data streams. It is designed to manage diverse data sources from various producers. In our project we decided to use the Apache Kafka along with Kubernetes, in order to deploy it inside its corresponding `StatefulSet`: this is required primarily due to the requirements for stable network identities, persistent storage, and ordered operations which are critical for Kafka and Zookeper. In other terms, Kafka and Zookeeper must be considered as stateful applications, thus needing a `StatefulSet` instead of an ephemeral `Deployment`.

Kafka integration in Vehicle Service (Spring): Now that we understand how to start Kafka, let's discuss how we integrated Kafka into the **vehicle service**. First, we needed to add all the required dependencies for Apache Kafka in *Spring*. Next, we created a specific configuration file for Kafka in which we defined all the necessary details about the service. In particular it was necessary to define:

- `bootstrap-server`: Comma-delimited list of host:port pairs to use for establishing the initial connections to the Kafka cluster.
- `group-id`: Unique string that identifies the consumer group to which this consumer belongs.

Now that Spring is aware about the Kafka cluster, It was necessary to define the Kafka Factory. The factory It is used for defining different configuration properties. In order to do that we created a new class `EventConsumer`, tagged with the `@Configuration`, because this class will define two different `@Beans`:

- `consumerFactory`: bean in which we define all the different properties that will be used by Kafka, the first this to specify is the `bootstrap-server`, the `group-id` and which deserializer for the key and the value of the input messages;
- `kafkaListener`: this bean creates a new Kafka listener using as input configuration the ones defined in the `consumerFactory` method.

After defining this class, all we had to do was define different handlers for the incoming topics. In order to do that, we defined the class `MessageConsumer` and tagged It with `@Component` annotation. This class uses the `EbikeService`, where all the different operations are defined to interact with the database. This class also defines two different methods:

- `listenUpdateLocation`: this method is using Kafka and It is listening on the `update-location` topic, all the messages that will be received on this topic will change the location for a specific ebike.

- `listenUpdateBatteryLevel`: this method is using Kafka and It is listening on the `update-battery` topic.

All the messages that are flowing inside Kafka are serialized as String and we decided to use the JSON format to easily parse the incoming data. Every time a new message is received in one of the two handlers, the message will be parsed by the `Deserializer` class. This class structured with the strategy pattern, define two methods, one used for unmarshall the **update location message** and another one for unmarshall the **update battery level**. Both methods return an *Optional*, in this way if the message is to well formatted the input message can be handled easily. In this way after the unmarshalling the incoming message the `MessageConsumer` class can use the `EbikeService` class for applying possible changes to a specific ebike.

2.2 User Service

User Service communication's model slightly differs from the EBike one, meaning that all communication of state changes regarding users are made directly publishing an event onto user's topic ("`users-update`"). While routes for getting a user by its id are kept in place (through REST APIs), the user service subscribes to events regarding mainly credit changes (for the sake of simplicity of this assignment), which are sent mainly by the `ride-service` in order to track user's credit consumption during rides.

Adapting an Event Driven Architecture in order to handle user's state makes it easy now to adapt the **event sourcing design pattern**. In this way we can easily keep track of every change that occurred to a given user, and also recreating a user's state in a given moment in time re-applying all changes that occurred until the given time. With these considerations, it is possible to design a user-related event as easy as:

```
interface UserCreditEvent {
    val userId: Int
    val timestamp: LocalDateTime
    val creditUpdate: Int
}
```

Producing Events In this way, all gathered events are stored in a data structure that is indexed with a given user's Id, and appends events to a queue sorted by the timestamp. In order to retrieve a user's current state, it is just a matter of applying a reduction onto this collection. This method could waste quite a lot of resources for computing every time a given user's state, and so a simple cache is implemented in order to keep the latest version of an already requested user's state when no new events occurred. Other forms of optimizations could be performed, but we think they're well outside the scope of this assignment.

Being polling a blocking operation, Kotlin coroutines have been used in order to schedule this operation on an I/O bound optimized thread pool

(`Dispatchers.IO` coroutine context). In this way, the polling operation is not blocking the main thread. The coroutine responsible for polling, after collecting the records, emits each one of them through a Kotlin `Flow`, in order to be consumed if only there are actual consumers who consume this flow (*cold flow*). All of this grants a way to make the event consumption process asynchronous to the process of actual events collection.

Producing Events When it comes to producing a user event, we have defined a simple method extension that made less verbose sending an event:

```
fun <K, V> KafkaProducer<K, V>.send(
    topicName: String,
    key: K,
    value: V,
): Future<RecordMetadata>
```

In this way, the `UserProxy` created for the Assignment 2, which backs off the user implementation in the ride service, needs just a quick refactor of the two main methods for increasing and decreasing the credit:

```
fun increaseUserCredit(userId: Int, amount: Int) =
    producer.send("user-update", userId, amount)

fun decreaseUserCredit(userId: Int, amount: Int) =
    producer.send("user-update", userId, -amount)
```

With these simple refactor, the ride service now correctly sends events to the topic on which the user service is subscribed to.

2.3 Event Sourcing

Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic. The application reconstructs an entity's current state by replaying the events.

Applications persist events in an event store, which is a database of events. The store has an API for adding and retrieving an entity's events. The event store also behaves like a message broker. It provides an API that enables services to subscribe to events. When a service saves an event in the event store, it is delivered to all interested subscribers.

2.3.1 Event Sourcing in the Vehicle Service

First thing first It was necessary to implement an event repository, so in order to do that we have created the `EventRepository` class, where the repository is implemented with an HashMap, where as key we use the vehicle id and as value

we have a list of class `Event`. The `Event` class is an Abstract class with only two different attributes:

- `id`: the ID for the current event that is created using the `UUID` class.
- `eventDate`: the date in which the event occurred.

Then after realizing this class we defined the event `NewEBikePositionEvent`, this events represent an update on the ebike position, this class extends from the abstract class event and defines two attributes, the ebikeId and the new ebike position. In this way every time an ebike changes position this event will be triggered and sent to the repository. Back to the `EventRepository`, are defined three different methods for interacting with the repository, the first method is the `createNewEbike`, where we add the input ebike inside the repository, in this way It will be possible to access this ebike in the future by using Its id. Then we have the `updateEBikePostion` that creates a new event `NewEBikePositionEvent` for the input ebike id, inside this method we will check if the id exists and only if exists then we will add the event inside the list of all the events. And finally we have the `getLatestPosition` method, where It is possible to query the respository and return the latest position for the input ebike. If the ebike exists then this method will return the latest position otherwise this method will return an empty Optional. In order to trigger this repository we use REST APIs, more in particular if the vehicle service receives a request for updating the ebike position from the route: `"/update/location/id"`, It will internally create the `NewEBikePositionEvent` for the input ebike and in this way It would be possible to get the latest position for an ebike by reading from the `EventRespository`.

3 Autonomous Bike

3.1 Domain

In order to implement the autonomous bike, it is important to highlight all the different components that will be necessary for implement and deploy this. Firstly, we have to define the concept of smart city, namely the digital representation of the environment on which bikes, stations and users are located. With this representation we can simply infer that all those objects must at least keep a reference to their position in space. In this way it is possible to update users or bikes locations based on other agents in space, by means of querying the environment. Those agents becomes aware of the environment, and can perform actions based on sensing it. When it comes to the autonomous bike, this agent will be the digital twin of the physical bike. This agent will keep a reference to the world it is contained into, making it possible to query it in order to decide, for example, where is the closest station it can reach or in which direction is located the user requesting that bike.

By tackling this problem in the described way, we are able to define four main autonomous agents: Word, A-Bike, User and Stations.

3.2 Implementation

Our implementation for this task can be found inside the `Vehicle Service` package. More in particular we decided to realize different classes in order to accomplish our goal. First of all, thanks to our implementation, defining a new type of bike is very easy and straightforward, because all It needs is to implement the interface `Bike`. Then inside this class we created three different methods. The first one, `reachStation`, allows the Agent to automatically query the world, making the agent able to understand its position in the world, and then select the closest Station on which the agent will be directed to. The second method that we decided to implement is `reachUser`, which takes as input the user position: in this way, when a user asks for this bike, the agent knows where is the user. Then all it needs to do is to move towards the user's position. The last method we decided to introduce is `getStations`. This method asks to the World class all the stations given as input the Agent's position, and also a radius in which the station needs to be. The World class will return a list of all the stations, then the Agent will automatically take the closest one using a specific Distance operation. The second class we decided to create is `World`, which represents the knowledge of the entire digital world (i.e. the smart city). By using this class It is possible to access and query all the different stations located in the environment. This class exposes two APIs: the first one is `getStations` that accepts as input a position and a radius. Then we created a record for representing a `Station`, which mainly stores its position in space. Then the final class is the `Distance`, inside It we have defined only one method which is the code for calculating the `manatthanDistance` given two positions, used to get the closest station from an ABike who calls this method.

4 Deployment through Kubernetes

Firstly, in order to use Kubernetes, it is needed to build a tagged image for each service. It is very handy instead of building from source every time (for every Kotlin projects ~60 seconds build time), to publish our images on docker hub.

After publishing the images, we must distinguish two different kind of services in order to properly define Kubernetes deployments and services:

- Services with persistency;
- "Functional" services.

Regardless of the kind of service, we can identify three main components for each one of them:

- `Deployment`: actual service description;
- `Service`: specify type of service (e.g. `ClusterIP`);
- `ConfigMap`/`Secrets`: used for storing configuration variables (they replace `.env` files for configuration) in order to implement the Runtime Configuration (*Externalized Configuration*) pattern.

In our infrastructure, in order to keep the deployment as simple as possible, we avoided the usage of `ConfigMap`s and incorporating configuration variables directly inside the deployment. On the other side, their usage is strongly recommended in order to keep configuration-related variables/secrets in separate files (encouraging the *externalized-configuration pattern*).

When it comes to services with persistency (e.g `user-service`, `ride-registry`, etc.), alongside the deployment of the actual service (e.g. the deployment of the `user-service` web server and event consumer application) a separate `Deployment` and `Service` are then needed for the persistency layer (i.e. the database). In order to keep persistency even when the database are shut down, we should define a `PersistentVolume` and its corresponding `PersistentVolumeClaim` for each database, and then *mounting* them when the deployment is applied. For the same reasons of `ConfigMaps`, we have directly incorporated the definitions of volumes inside databases' deployment manifests, alongside database secrets.

On the other hand, the second kind of services are much simpler and their structure is composed of just a `Deployment` for the service itself and its corresponding `Service`.

With this configuration, the overall infrastructure should work as expected as long as services share the same *namespace*. If this is not the case, `ConfigMap`s must be simply changed accordingly to instruct services that now other services' hostnames will be prefixed by the service namespace (e.g. `<gateway-namespace>.api-gateway:4001` for the `api-gatway` service), taking advantage of the *Externalized Configration pattern*.

Finally, in order to make the overall deployment accessible from outside, *Kafka*'s service and The *API Gateway* must be exposed through port mapping, or change their internal network interface.