Coding Challenge

Introduction

There is no time limit on this challenge, but we don’t want you to spend too much time on it.
We value:

- functional paradigm
- non-blocking and asynchronous flow
- clean and readable code
- good documentation

You are expected to take some design decisions and explain them during the code review.

The exercise is left open intentionally: things like error management, logging, is left to your judgment.
You are expected to write at least one unit test for the functionality that you see most required.

Bet History

You will create a simplified service that handles events related to a user placing bets and expose an API to retrieve betting history.

Coding

Part 1: services

You have to implement a few simple services that you will use to retrieve data or notify the user about a won bet.

OutcomeService

Let's start with OutcomeService. Please use the following definition:

```
case class Outcome(id: String, fixtureName: String, outcomeName: String)

case class OutcomeServiceError(msg: String, throwable: Option[Throwable])

trait OutcomeService {
 def getOutcome(outcomeId: String): Future[Either[OutcomeServiceError,
Outcome]]
}
```

Outcome - is the item the user is betting on. An example of an outcome might be:

```
Outcome(
   id = "outcome-0001",
   fixtureName = "Real Madrid vs Barcelona",
   outcomeName = "Barcelona wins match"
)
```

In this case, the user is betting that Barcelona will win a match against Real Madrid.

You are given a text file `outcomes.csv`, please use it as a static data source for the OutcomeService implementation. You can simply put this file under `src/main/resources`.

AccountService

Please use the following definition:

```
case class Account(id: String, name: String, phoneNumber: String)

case class AccountServiceError(msg: String, throwable: Option[Throwable])

trait AccountService {
 def getAccount(accountId: String): Future[Either[AccountServiceError,
Account]]
}
```

Identically to the OutcomeService, you're given a text file `accounts.csv` that you should use as a data source for the implementation.

NotificationService

Please use the following definition:

```
case class Notification(phoneNumber: String, message: String)
```

```scala
case class NotificationServiceError(msg: String, throwable:
Option[Throwable])

trait NotificationService {
 def send(notification: Notification):
Future[Either[NotificationServiceError, Unit]]
}
```

The implementation should be extremely simple - just log or print the notification to stdout.

Part 2: event sources

You have to implement sources for betting events. You have to use Akka Streams Source.
Please use the following definition:

```scala
sealed trait BetPlacementEvent

case class BetPlaced(
 betId: String,
 accountId: String,
 outcomeId: String,
 payout: Float,
 timestamp: Instant
) extends BetPlacementEvent

sealed trait BetSettlementEvent

case class BetWon(
 betId: String,
 timestamp: Instant
) extends BetSettlementEvent

case class BetLost(
 betId: String,
 timestamp: Instant
) extends BetSettlementEvent

trait EventSources {
 def betPlacementEvents: Source[BetPlacementEvent, NotUsed]
 def betSettlementEvents: Source[BetSettlementEvent, NotUsed]
}
```

You are given two text files `bet-placement-events.csv` and `bet-settlement-events.csv`, please use them as a static data source for the implementation.

To mimic the complexity that comes with event-driven asynchronous systems, we ask you to shuffle (randomize) events before streaming them and add a random delay of 1ms to 2000ms between every emitted event.

Part 3: event handling

Once the application is started you should start reading events from the `EventSources`, enrich them with data from `AccountService` and `OutcomeService`, and persist everything in memory. This part is completely up to you, but have in mind that you will use this data for the next part, the “API endpoint”.
Also have in mind the following:

- the order of the events is not guaranteed (you shuffled them and added random delay between every event)
- events can be duplicated or contradictory, e.g. you might receive BetWon and BetLost for the same bet ID. Events with the latest `timestamp` should override the others
- processing of the events must be non-blocking

If you receive a `BetWon` event you should send a notification to the user to congratulate him using the `NotificationService` - something like “`$name, congratulations you just won $payout on $fixtureName, $outcomeName !`”. You should not send a notification twice to the same user if you receive multiple `BetWon` events for the same bet.

Part 4: API endpoint

Expose an HTTP endpoint that allows fetching betting history for a user. Bets should be sorted by timestamp (the latest placed shows first). API should have support for cursor-based pagination (forward and backward). Unsettled bets should have a status “Open”.
It should take the following params:

name	description	is required
accountId	the id of the user account	required
first	number of betting events to return	optional, default 10

after	return betting events that come after this cursor	optional, cannot be used together with “before”
before	return betting events that come before this cursor	optional, cannot be used together with “after”

The actual representation is up to you, but we expect the request to look something like:

```
GET /bets/acc-0001/first=10
GET /bets/acc-0001/after=05a83161
```

And the response:

```
{
 "nodes": [
   {
     "betId": "bet-0001",
     "payout": 10.54,
     "status": "Lost",
     "fixtureName": "Malaga vs Madrid",
     "outcomeName": "Malaga wins",
     "cursor": "05a83161"
   },
   {
     ...
   }
 ],
 "navigation": {
   "hasNextPage": true,
   "hasPreviousPage": true,
   "firstCursor": "05a83161",
   "lastCursor": "c96aa025"
 }
}
```

Delivery

Push your work to a private GitHub repository and give read access to the bluelabseu-bot user. Afterward, we will review the code and invite you to the next stage of the recruitment process. The app should be containerized, meaning we should be able to test it via Docker invoking a command like:

```
$ cd bet-history && run.sh
```

This should start the application as a docker container that exposes the REST endpoint on `localhost:8080`

Please add a Readme file with the steps required to run your project.