ChangeLog

20th Sep 2018: Initial release
22nd Sep 2018:

- **There are no buy offers in the orderbook anymore. So, the user can only buy BTC. This should make it a bit less tedious, as this removes half of the REST APIs and functions.**
- **There is only one user for P2**
- **The REST API should only be used for communication with the outside world. The internal messaging between the actors should take place using Akka's messaging mechanism.**

Learning Objectives

- Implementing actor model using Akka
- Integrating play framework with Akka
- RESTful web service
- Two-phase commit protocol

Story

We are implementing a prototype for a cryptocurrency exchange. The system has two roles: **UserActor** and **MarketActor**. For our prototype, users can trade in only one currency, Bitcoin (BTC).

Orderbook

The orderbook is a list of all the sell offers that the market has currently available. These offers are **not made by our users**; just assume they are there to begin with. Our users cannot *add* offers to the orderbook, but they can *accept* offers. Here is what an orderbook might look like:

Sell offers		
Rate (USD per BTC)	Amount (BTC)	Offer ID
100	5	431671cb
80	2	16b961ed
50	12	1e06381d

Use the above data to initialize your orderbook

As you can see, the orderbook lists the rate of the offer, number of BTC on offer, and a unique Offer ID. Also note that the entire orderbook is arranged in the **descending order of rate**.

The UserActor

The UserActor aims to assist users in making the best possible purchase choice. So, when buying BTC, the actor will try to buy at the lowest possible price. (Examples in a bit.)
The UserActor also maintains user's account balance in both BTC and USD.
For P2, assume there is only one user.

The MarketActor

The MarketActor's job is to maintain the orderbook by making changes to it as offers are accepted. It will also disallow any invalid trades. The actor responds to **Hold/Confirm** requests.

Examples

Refer to the above orderbook while reading these examples.

Example 1

Say the user wants to:

- Buy 5 BTC
- Pay a maximum rate of $80 per BTC

To **buy** BTC, the UserActor must look at the **sell offers** available. In our orderbook, the lowest available rate is $50 per BTC, and 12 BTC are available to buy at that rate. So, the UserActor will try to buy 5 BTC at a rate of $50 each. Here is what happens:

- The UserActor checks the orderbook, and determines the optimal buy is 5 BTC @ $50 each. The actor makes sure it has $50 * 5 = $250 available.
- The actor sends a Hold request to the MarketActor, with the Offer ID (1e06381d) and the number of BTC to hold (5).
- The MarketActor makes sure the given Offer ID has >=5 BTC on offer. The actor decreases the amount of BTC available for that offer from 12 to 7 in the orderbook.
- The UserActor sends a Confirm after receiving "success" from the market actor.
    - If the user actor didn't confirm, the market actor would time out the Hold request, and restore back the held BTC in the orderbook.
- The UserActor and deducts $250 from its USD balance, and adds 5 BTC to its BTC balance, after it receives "success" from the MarketActor.

Example 2

The user wants to:

- Buy 13 BTC
- Pay a maximum of $100 per BTC

In this case, the transaction cannot be confirmed using only the lowest available sell offer (since it only has 12 BTC). So, the UserActor will make **two separate Hold requests**:
- 12 BTC @ $50 per BTC (Offer ID 1e06381d)
- 1 BTC @ $80 per BTC (Offer ID 16b961ed)

It will send a **Confirm** request **only if it both Holds are successful**, otherwise it will return a failure status.

Example 3

The user wants to:
- Buy 10 BTC
- Pay a maximum of $10 per BTC

The UserActor returns failure. This isn't possible because the lowest sell offer is $50.

Example 4

The user wants to:
- Buy 15 BTC
- Pay a maximum of $80 per BTC

The UserActor returns failure. This isn't possible because not enough sell offers at <=$80.

This project is inspired by the paper Pardon & Pautasso, 2014. This paper may help you understand the scenario and techniques, especially the two-phase commit protocol. However, this project is different. Please closely read the instructions.

Deliverables & Grading Parameters

NOTE: The following parts are mainly for grading purposes. Please submit your whole project as one .zip file.

Part 0: Valid Submission (5%)

A valid submission means your submitted project executes out of box. In other words, it can be executed using 'sbt run' on a clean computer with only JDK and sbt installed. Make sure your project:
- Is submitted completely
- Has no compiling error
- Includes essential libraries, either in your code repository or configured as remote maven libraries in build.sbt script
- Includes no absolute path

A safe way is to test your program on another computer before you submit.

Please submit your whole project as ONE .zip file via Moodle. Refer to Guidelines section for guides of reducing submission size.

Part 1: A Play Framework based web service with RESTful APIs (10%)

Your program should provide specified RESTful APIs. That is, it should respond to specified RESTful requests and its responses comply with the API specification. Refer to the API section for the specification of APIs.

Part 2: Akka actors with debugging APIs (25%)

Your program should construct a UserActor and a MarketActor along with their debug APIs. Refer to the API section for specification of the debug APIs. You will risk losing credit for this part if you do not provide the debug API. The business logic is not considered for grading this part.

Part 3: MarketActor (25%)

Implement the business logic of MarketActor, which includes but is not limited to:

- Maintaining the state of the orderbook
- Responding to Hold and Confirm requests
- Cancelling holds on timeouts
- Debug APIs for MarketActor

Part 4: UserActor (25%)

Implement the logic of the user actor, which includes but is not limited to:

- Responding to buy or sell requests, trying to satisfy the request with the best available offer (as illustrated in the examples)
- Performing transactions using two-phase commit protocol
- Linking RESTFul APIs to user actor
- Debug APIs for user actor

Part 5: Error handling (10%)

You should deal with errors occurring in different components:

- FAIL response from actors
- Timeout in an actor's response.

In the user actor, you should keep an event log in a DB for recovery purposes. Record at least the following events:

- Responses of Hold requests, including:
    - Success or failure
    - Expiration time
- Responses of Confirm requests

- Partially failed requests, which happen when
    - one confirm fails but at least one confirm succeeds in the current transaction
    - one Hold expires without successful confirm but some confirms succeed in the current transaction

Record a timestamp and a transaction ID with each log entry. The transaction ID should be unique for each transaction.
You are not required to implement the recovery mechanism itself, just record the events.

Ingredients

- JDK 8 or higher
- sbt (ver 1.2.1 or higher)
- Play (ver 2.6.18 or higher) [Don't need to download manually]
- Akka (ver 2.5.4 or higher)
- sqlite-jdbc (ver 3.23.1 or higher)
    - May need to add the following to build.sbt:
    - libraryDependencies += "org.xerial" % "sqlite-jdbc" % "3.8.11.2"
- Jackson JSON library (ver 2.8.7 or higher)

API

RESTful API

Request	Response	Description
POST /addbalance/usd/:amount	{ "status": "success"}	Add the given amount to the user's USD balance.
GET /getbalance	{ "status": "success", "usd": <amount>, "btc": <amount>}	Return the user's USD and BTC balance.

GET /transactions	{
	"status": "success",
	"transactions": [<list of transaction IDs>]
	}
		Get a list of successful transactions.

GET /transactions/:transactionID	{ "status": "success", "amount":<in BTC>,	Get details of a given transaction.

	“rate”:<in USD per BTC>
	}
	or
	{
	“status”: “error”,
	“message”: <error message>
	}	

GET /selloffers	{
	“status”: “success”,
	“offers”: [<list of sell Offer IDs>]
	}
		Get a list of sell Offer IDs.

GET /selloffers/:offerid	{
	“status”: “success”,
	“rate”: <in USD per BTC>,
	“amount”: <in BTC>
	}
	or
	{
	“status”: “error”,
	“message”: <error message>
	}
		Get sell offer details.

POST /buy/:maxrate/:amount	{
	“status”: “success”,
	“transactionID”: <transaction ID>
	}
	or
	{
	“status”: “error”,
	“message”: <error message>
	}
		Request a buy transaction, for the given BTC amount at the given maximum rate.
		
		For the transactionID, use a scheme of your choosing.

Debugging API

These APIs are for debug purpose, mainly for mocking fail and timeout scenarios of actors.

Request	Response	Description

POST /debug/confirm_fail	{ “status”: “success”}	After this request is posted, the market actor will reply fail to subsequent Confirm requests without actual processing.
POST /debug/confirm_no_response	{ “status”: “success”}	After this request is posted, the market actor will not reply to subsequent Confirm requests and do no actual processing.
POST /debug/reset	{ “status”: “success”}	After this request is posted, the actor will reset to normal operation.

The REST APIs should only be used for communication with the actors from the outside world (example, by a human user through their browser).

Internal communication between the actors must happen through Akka’s messaging system, and the details of it are left up to you.

Guidelines

The guidelines are mainly for Java; many of them may also apply to both Java and Scala. You are free to choose between Java and Scala.

Set RESTful Routes

Take GET /selloffers/:offerid as an example. First, add following line into routes.conf:

GET /selloffers/:offerid controllers.HomeController.getSellOffer(offerid: String)

In HomeController, add a function:

```
public Result getSellOffer(String offerid){
    return ok("sell offerid: " + offerid);
}
```

Try to visit http://localhost:9000/selloffers/testing
You should see: sell offerid: testing

Refer to: http://www.baeldung.com/rest-api-with-play

- Use Akka

Akka is already part of the Play Framework, so you do not need to install it. Just import it when necessary:
import akka.actor.*;
import akka.japi.*;
For quick Akka examples, refer to:
http://developer.lightbend.com/guides/akka-quickstart-java/?_ga=2.96077404.1561246931.1504127451-2077373114.1503939704

- Talk to Akka actor in Play

Refer to: https://www.playframework.com/documentation/2.6.x/JavaAkka

- Reduce the size of submission

The size of submission should be far less than the locker limit if you set and clean it properly. Here are some tips for size reduction:

- Do not include binary libraries in your project. If you need to use a third party library, add it into build.sbt as a dependency. For example:

  libraryDependencies += "org.xerial" % "sqlite-jdbc" % "3.23.1"
  The library will be downloaded and installed from maven repository when next time `sbt run` being executed.

- Remove target and intermediate files. For a sbt project, this can be achieved by deleting nested target directories, such as ./target, ./project/target, and ./project/project/target. Refer to:
  https://stackoverflow.com/questions/4483230/an-easy-way-to-get-rid-of-everything-generated-by-sbt