Bookies / Online Betting Service

1. Overview of Problem

The problem we are trying to solve is that of an online sports betting service.

In this industry, several parties are involved in the creation of a transaction. Bookies set the odds of the event, event organisers arrange, execute and transmit the results of the event, while a person places a bet with the bookie on the outcome of a chosen event.

We will use a distributed computing solution to solve this problem for several reasons.

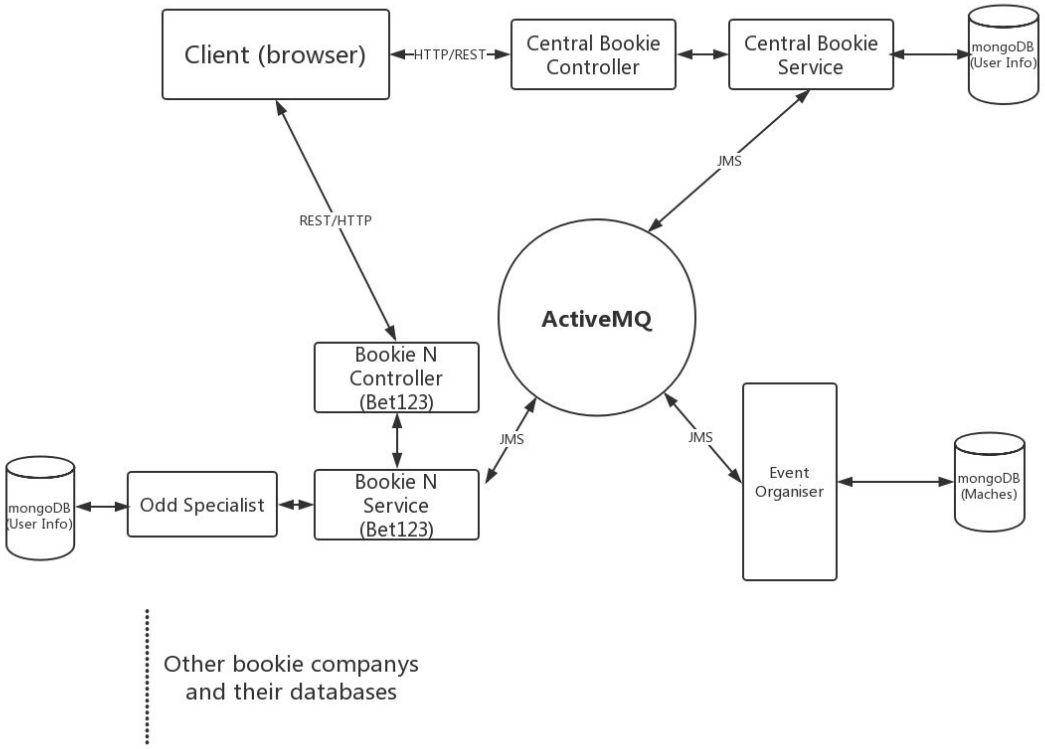
It is clear that for this type of business to be successful, prompt and reliable means of communication between parties, regardless of their location, must be undertaken so the necessary information reaches all of those concerned in a timely fashion. By definition, distributed computing focuses on effective message passing between several entities on different networks, so it was the obvious choice in first attempting to solve this problem as each component does not have to be in the same location/on the same network, yet they still need to communicate with the others.

Also, despite the bookie communicating with several other entities the system other than the person making the bet, like the event organiser, this backend-communication should be invisible to the person making the bet . The online betting system should appear as one computer to the user, a defining characteristic of distributed computing.

Finally, there is no need for the person aspiring to make the bet to be tied to one bookie or one type of sporting event. Distributed computing provides us with the ability of horizontal scaling, which in this case of seamlessly adding more events and/or bookies to the system.

As you can see, our decision to make use of distributed computing in order to solve this problem of an online betting service is well justified.

2. System Architecture



The 3 technologies at the heart of our application are JMS (provided by ActiveMQ), Spring (using Spring Boot) and REST. By utilizing JMS we are able to seamlessly broker messages between the components of our system and through the use of REST with Spring, we easily transmit an object's dependencies and inject them while keeping the components decoupled.

Initially, a client interacts with a central bookie service using REST/HTTP. It is with this service that users register their details and log-in. MongoDB is used to support these two actions. First, a user's details are stored upon a user registering. Secondly, they are retrieved to authorize a login.

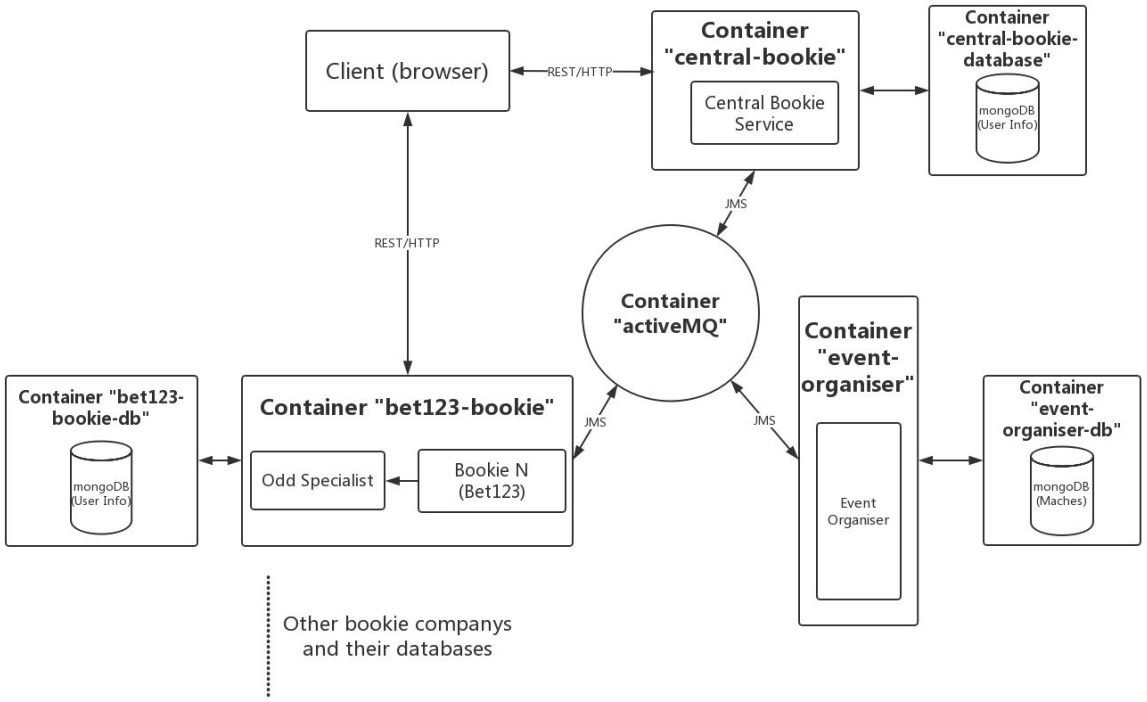
After a client logs in, the central bookie service lists all of the registered bookies and communicates the clients information to these bookies to inform them of the current user. When a user decides what bookie they want to bet with, they are redirected to its homepage.

Each bookies controller is linked to three common components - its own service, a mongo database and an odd specialist. The bookie service will communicate with the Event Organiser, through the message broker (ActiveMQ), to retrieve all the current events (Basketball, Football) which are stored in a database The bookie service will use this information to generate odds for the client.

After a client finishes placing their bet, the chosen bookies database is populated with the bet and all its corresponding information such as the amount. This is used to present a table of bets the user has placed throughout their session.

Finally, in order to demonstrate that different services in our project can be hosted on different machines with horizontal scalability, we used docker and docker network to deploy our system.

There are seven containers we need to open in the current scenario. Which should be three different services: "central-bookie", "bet123-bookie" (and any further bookies that may be added) and "event-organiser". Each one of these has a separate mongo database connected. These databases are also wrapped in their own container. Despite that, we also need to create a container with a activeMQ inside. The whole docker architecture is shown down below.



Notice that all the containers will be run under the same network with different names and IPs.

3. Technology Choices

Technology	Motivation for Use
REST	Familiarity with the REST API from use in practicals. Most logical, efficient and widespread standard in the creation of APIs for Internet services
MongoDB	Database required to store user and bet info. MongoDB used because one of our team members previous experience with it
Spring Boot	A great framework that could easily implement a web application. By using it, we could save a lot of time and focus on the main logic.
JMS	Java-based MOM that acts as an enabling layer between the client and the provider and standardises interactions with the provider. Sends and receives data in a reliable fashion.
ActiveMQ	Message broker fully implemented with JMS. Messages share a common format which makes integration between different applications easier
Docker	Demonstrate that our application can be deployed and run in a distributed manner. Each containers will have different IPs and hosted on different environment while still act as a whole.

4. The Team

Give details of your team members and their assigned tasks below.

Student Number	Name	Assigned Task(s)
15439382	Dylan Rowley	- Implement Database - Populate Database
15347791	Seán Grant	- Add multiple bookies [Scalability] - Use Gson to send/receive JMS messages as JSON
15205944	Zhang Qinyuan	- Create Back-end: - Bookie Service

		- Odds Generator - Event Organiser - Implement Docker - Implement Betting Logic

5. Task List

5.1. Add multiple bookies [Scalability]

In order to make sure that the application was scalable, it had to be simple to add multiple bookies without having to change the core application. In our application every bookie service functions in the same way; users can bet on either football or basketball games (provided they have sufficient funds), update their balance by entering (in this case fake) credit card details and view the current bets they have placed with the given bookie. The only place where bookies differ is in name and the value used to calculate odds.

So, in order to add scalability with multiple bookies a controller called *GeneralBookieController* was added as a superclass. This defined all of the functioning required for each bookie (as stated above), then, in order to add any new bookies to the server all that needed to be created was a new controller, e.g. *Bet123Controller*, which extends the GeneralBookieController superclass. Each subclass maps a request to the inherited methods, then calling that method of the superclass passing the bookie name as argument and calcNum (used for calculating odds) if necessary.

A challenge was faced in trying to scale this component simply and efficiently. We tried to avoid the need for creating a new controller for each bookie as it lead to unnecessary duplicated code (even with the general superclass).We had initially hoped we could create objects of the controller class passing the bookieName as an argument to the constructor. This was difficult as controllers couldn't be instantiated as objects and because the @RequestMapping value must be constant, leading us to believe it must be hardcoded for each individual bookie. However, at a late stage in the project and after further research into Spring, we realised that we likely could have improved the code having only one GeneralBookieController by passing the bookie name as a PathVariable, similar to *hrefInfo* with betting. This information will be kept in mind for future projects but due to time limitations we could not add it.

The use of MongoDB also aided scalability as no new databases had to be hardcoded for each new bookie. When Mongo tries to get a database or collection, it either retrieves it if it already exists, otherwise it dynamically creates a new one. All that had to be done then was get the database "<*bookieName*>DB".

Despite the improvements we've identified, we still believe the application is scalable as it is still remains simple to add new bookies and it does not require changing the core code on each new addition.

5.2. Use Gson to send/receive JMS messages as JSON

JMS and ActiveMQ are used in the project for sending the lists of events (football and basketball matches) from the *EventOrganiser*, retrieved from the eventOrganiserDB, to the *GeneralBookieService*.

In implementing JMS we decided to send messages to the JMS Queue as a String in JSON format. We decided this to keep a more uniform format of the type of messages are being sent. Regardless of whether the JMS message is a List of FootballMatchInfo objects, a List of BasketballMatchInfo objects or a String email, the message will always be in the form of a JSON string.

Gson is a great library for converting Java objects into JSON and converting JSON back into Java objects. Once the list of events was retrieved in *EventOrganiser*, it would be converted to JSON using Gson, that JSON String would then be sent to the JMS Queue (ActiveMQ) and then retrieved from the Queue using a JMSListener in the *GeneralBookieService* where it would be converted back to a List of events by Gson.

5.3. Create initial application prototype and framework

By using Spring Boot and Maven, we could easily set up our application. However, all the things that are mentioned above are new to us and we had to educate ourselves in each technology using its documentation. Luckily, each proved clear and concise and presented no major technical challenges

Firstly, by using spring boot, we could define the service and controller easily by adding "@Controller" or "@Service" at the beginning of the java files. The controllers then can easily access services by using "@Autowired". A new service instance will be created and ready to be used. Secondly, writing RESTful APIs is easier in spring boot comparing to our previous assignment (assignment 4). Thirdly, maven helps us import packages quickly by adding few lines in pom.xml.

Our user interface consisted of dynamic web pages created using Thymeleaf, a Java XML/XHTML/HTML5 template engine. We find that the it is a good technique that could be used to create dynamic web pages. Backend (like controllers) can get form's data from frontend by using "@ModelAttribute". Frontend can also generate dynamic information according to the data passed by the backend.

One of the biggest challenges is that all of the relative techniques are new to us. At the very beginning of the project, different kinds of configuration problems occurred frequently e.g

entry class cannot find relative controllers or services due to some wrong package scanning settings. It took me a long time to figure this out in order to generate a prototype of our project.

5.4. Implement Betting Logic

Implementing betting logic is one of the most important parts of our project. We need to make sure that all the information is successfully stored in a certain database. A domain class called “betInfo” was created to store the betting orders’ information from clients and pass them through different pages and services. Once the user filled in the form inside the betting web page, we used thymeleaf to wrap these up and send that to certain bookie company’s service, who will then use it to find other relative orders with the same account and display all of that in one single web page. User bets were then stored in the databases of each individual bookie.

However, it is better to contact the central bookie service to authorize the users’ information if it is a real-world business. In our cases, authority information including email and password need to be sent back to central bookie again. However, passing information (or message) through ActiveMQ has already implemented in other logic. That is why the betting logic is not fully implemented and might seems a little insecure.

5.5. Implement Docker

Trying to split and package our maven projects is important but difficult. In order to show that our application could be run on several machines, I did not use docker compose (it is hard as well). Docker network proved to be a good choice since it provides different IPs and names for different containers. Different containers can also access the others by using their names. In this case, I created a cluster with seven containers in total and two export ports to the localhost. Each business, database and message broker have their own containers. Central bookie business is opened on port 8081 while a sample bookie company, bet123, is opened on port 8082.

Figuring out how each container contacts each other was a great challenge. We found out that instead of using IPs, which might change every time, we could use containers’ names instead since they are all unique and persistent. Furthermore, there are many pieces in our system that utilize databases or message broker, trying to find out where and do not miss each one of them was a big problem as well.

In the future, we might find it possible to deploy our system in a larger form, to say, a larger machines cluster.

5.6 Implement Database.

In order to provide the application with a layer of persistence we utilized MongoDB.

MongoDB was chosen for two reasons. Firstly, due to the availability of a clear, simple Java API which would enable seamless integration with our application. Secondly, MongoDB greatly supported the scalability of our application because of its implicit creation of databases and their collections (collections in MongoDB are just SQL tables) i.e if a database (DB) not currently present was requested, and you attempt to add a table and input data to this currently unknown DB, MongoDB now assumes this is no mistake and creates the database for you. Therefore, once the API was written for one bookie, it was written for all.

Once MongoDB was installed on our machines, we began creating the connection with databases for the initial components of the application. Databases were first created for storing users' information in central bookie service. We created several data access objects (dao) to provide an interface to the required databases. This allowed the services interact with the necessary databases when required.

5.7 Populate Database.

In order to keep the content of events as varied as possible, the team set out to populate the collections of events with a large corpus of data.

Datasets for each type of event were retrieved from kaggle.com and the data (league name, team name, probability of winning etc) was extracted using simple Python scripts into text files. A collection for each type of event was then created and populated with the relevant data under the eventOrganiserDB.

This presented no technical challenges due to the simplicity of Python and the reuse of code from the initial set-up of our databases described above.

6. Reflections

In this section we will reflect on what we learned throughout the development of this project, discussing whether the technologies used were appropriate and what benefits and limitations they brought.

6.1 Benefits of Technologies Used

In section 3, we detail the technologies that have been utilized throughout this project. Here, we take an in-depth look at the benefits of these technologies

6.1.1 JMS

We believe that JMS was an appropriate technology to implement a distributed online betting service for several reasons.

Firstly, JMS ensures that messages sent are sure to be delivered. This guarantee proved extremely advantageous in the development of our project. It removed the need for a complex implementation of ensuring messages are successfully sent and received between components. For example, if we were to use socket programming to distribute this application, most of these message assurances now become a manual effort and components that communicate with each other will be tightly coupled. This is not something we want in a distributed solution as it hinders scalability.

Secondly, JMS uses an asynchronous messaging system. This ensures that even if our JMS provider fails, the message will still be in the queue and sent once it is back functioning.

Another feature that is worth a quick mention is that distributed implementations using JMS benefit from language independence. It is only the message format that has to be persistent. Although we did not run into this problem in this current implementation due to solely relying on Java, the obvious advantage of this feature can be seen - if we decide to create other components in another language, the message passing between them is unaffected.

6.1.2 MongoDB

MongoDB had many benefits as well which led to us using it. Firstly, being a non-relational database system, it is schema-less. Meaning the number of fields, and size of documents stored can differ. This could be useful going forward as different types of events are added.

For example, BetInfo now stored in the database stores fields such as the amount of the bet, the odds, what team to win etc. But what if in future different bet options are added? For example accumulators. Then the BetInfo stored would have different fields than some other bets. Fortunately this won't be in issue with MongoDB.

As mentioned previously, MongoDB is easily scalable as well. There is no need to manually add new databases and collections(tables) when a new bookie is added, MongoDB will do this manually.

6.1.3 Docker

Finally, docker and docker network give us a good environment as different parts of system work distributed and cooperate with each other. It proves that our system can be hosted on different machines but still act as a whole. Other than that, by using docker, our system could be easily deployed on a new computer since all our images are pushed online in the docker's repository. Without any pre-installations or configurations, users can deploy our system easily by running a single script.

6.2 Limitations of Technologies Used

The way we made use of Spring for the bookies with a GeneralBookieService and GeneralBookieController in charge of each of the individual bookies certainly came with its limitations. With the request mappings for each bookie using a general controller and each bookie using the same general service, they were limited in the differences between them. As mentioned above, as the project is currently set up, the only thing that differs between each individual bookie is the name and value used for calculating odds. But what if in future a new bookie comes along that wants to say offer bets on horse racing, or not allow betting on football? Or really any new functionality not offered by the other bookies? Currently this wouldn't be possible without adding in a new Service or Controller. This hinders the desired scalability of the application as it's no longer as simple as adding in a new bookie differing only by name and odd calc value.

The JMS also brings us a limitation. Because it is an asynchronous system, users might not be able to see certain information while it is still on its way. In order to avoid this, we try to get messages earlier than users need to actually view them. It is clear that we should use threads or any other ways to solve the problem other than what we used. But it will definitely be more difficult to implement.

Because ActiveMQ as a broker holds on to messages until they are consumed, performance could decrease as the system grows if there are large amounts of bookies and users sending messages. This would be something to keep in mind in future.

The Docker also came with limitations. It was initially our intention to deliver the final version of the project using the docker, making it much easier to set-up and run. However, this proved to be a challenge as our code was always changing as new features were added and bugs were fixed. In the end, we decided to implement the docker using an older version of the code, just to show an example of how this technology might be used.

6.3 Thoughts on Distributed Solution

The distributed solution we used covers a lot of techniques and ideas. The three main things we used are Spring Boot, JMS and REST. These techniques ensure that our system could be separated into different pieces while they still act as a whole.

We also employed the docker to prove that the idea that our system is distributed. By using this, we split our project into three logical parts, three databases who are managed by each business, and one message broker who transfer JMS message for these businesses. Additionally, databases in our systems are independent, who do not need to synchronise with others. It is more simple to do, but very dangerous in the real-world scenario.

As for horizontal scalability, more companies could be added to our system as we have implemented in the main project where there are three different bookie companies and they can all be easily created and implemented.

Because our system does not involve any other layers, we do not consider the vertical scalability. However, some different features, which might relative to a real-world bookie business, require that. For examples, a real-time odds changing service or balance system that connects to banks' services.

6.4 What was learned?

Seán:

"Each of the technologies used in this project were relatively new to us. Although REST and Spring were covered during the semester we certainly gained a better understanding of their use and implementations. However, the use of JMS and ActiveMQ was completely uncharted territory for me. In our original proposal we had decided on using Swagger and GraphQL, so when we ended up using JMS and ActiveMQ I had to learn those technologies quickly. Some of the JMS had already been implemented when I started to build on it so I had to make sure I fully understood how JMS was working with ActiveMQ before changing anything.
Another useful technology learned was MongoDB. Until now my only experience with databases was with MySQL back in second year. MongoDB being a non-relational database was something entirely new to learn. I really enjoyed using MongoDB as it was very easily implemented with Java and the fact that it is non-relational meant there was no difficult search queries involved, thus easy to pick up.

Aside from learning new technologies, we were also taught a lesson in project management. With the majority of this project taking place outside of the college term we found ourselves in different parts of the country, unable to meet face-to-face and collaborate. With it also being over the Christmas break each team member had busy schedules and naturally were taking some time off from college work. Due to this, good project management and communication skills were required to ensure that everyone was kept on the same page of what was happening and that work was being done. Here unfortunately we faltered a little bit. Everyone was responsible for a lack of communication over the break leading to us being confused about what others had done and occasionally two people ended up working on the same thing. Naturally it is more difficult to communicate complicated issues when you're not meeting face-to-face as well. This was certainly a lesson learned and I feel we will all manage projects far more efficiently in future"

Dylan:

"This project proved to be an exceptionally good learning experience in several regards

Firstly, due to very poor foresight and organisational skills, I joined the development cycle later than I should have and found myself learning different and more technologies than initially expected. Unfortunately, I chose the route of attempting to understand the application alone, but not the technologies that underpinned it. This proved an inferior decision as although I could understand how the application worked, I could not extend it easily. In hindsight, I should have learned the technologies from a foundational level and not in the

context of the application.

Nevertheless, the introduction to JMS, Spring, MongoDB and Thymeleaf kept me on my toes and I'm leaving this project with a much broader understanding of all of these technologies.

But, the main lesson I will take away from this project is that frequent and informative communication among team members is absolutely vital. All team members, on more than one occasion, found themselves on different pages in regards to the stages of development and the overall direction of the project. This was a mountain to climb throughout the entire development cycle and was easily avoidable had we all adhered to constant communication.

Overall,I've gained valuable experience with several new technologies and have been forced to make revisions in how I work in teams. Something that will prove vital in day-to-day work.
"

***Tony**:*

"I learned a lot of new techniques that are totally unfamiliar before, including Spring Boot, Maven, JMS, MongoDB and Docker. Finding out how each one of these works and putting pieces together into a project is what I would say most valuable.

We started the project by using Spring Boot and Maven because it is said that those techniques are quite popular these days. It turns out coding a web application by using these are absolutely easier and faster comparing to some old techniques we tried before. They do have their reasons to be widely-used nowadays. Manipulating mongoDB and activeMQ is not easy at the very beginning. But once we successfully found out how to do, the rest goes smoothly as well. Docker is also popular nowadays and it took me a long time to find out how it works. I tried different ways to link different containers and finally I discovered that by using docker network and different names, containers can find others. However, later on I found out that docker compose is also an approach which I did not try before. Might give a try if still have time.

Apart from new techniques being learned, working with my teammates is also important and challenging since I am not a native speaker. It took me a long time to present my opinions or thoughts and to get what they want to say in some cases. Fortunately, we have a good pre-arrangement of our project at the very beginning thanks to my teammates. Because I don't normally prepare that well at the beginning stage of a project, I always end up messing things up before. That is one of the most important things I learned during this project.

Overall, working with my teammates and building a full business web application with several new techniques involved is challenging but full of fulfillments in the end."

7. Deployment

Main Application:

Deployment of the application requires two services to be installed and running; MongoDB and ActiveMQ.

Once those services are running the application is started by running the class DsBookieBusinessApplication, found in src/main/java/com/example.

The running application can be accessed in a browser at http://localhost:8080
The project also uses Maven dependencies so whatever IDE that is used to run the code will need Maven integration.

Docker Example (an older version of the main one):

First you need to install docker.
Once you installed docker, simply run the “demo-start.sh”.
It will deploy our system on your computer.
To stop and delete our system, please run the “demo-stop.sh” file.
Further information is in the “dockerize” folder in our source code.

References

[1] JMS in Spring Guide, *https://spring.io/guides/gs/messaging-jms/*
[2] JMS Tutorial, *https://j2eereference.com/advantages-java-message-service-jms/*
[3] Apache ActiveMQ, *http://activemq.apache.org/*
[4] MongoDB, *https://www.mongodb.com/*
[5] MongoDB with Java Guide, *https://www.baeldung.com/java-mongodb*