BLT

Brendan Sadlier	Liam Smyth	Thomas Coogan
20402884	20321311	20356556

Synopsis:

Describe the system you intend to create:

The system we intend to create is a real-time betting platform focused on horse racing. This system will facilitate dynamic interactions between various services to manage the races, calculate odds for the horses in a race and enable clients to place bets on horses.

What is the application domain?

The application domain for this system is online gambling, specifically targeting the niche of horse race betting.

What will the application do?

The application takes in a list of horses from a database and generates races using these horses. The odds of each horse is dynamically calculated based on the score of each horse with regards to the scores of the other horses in the same race. These races are then sent to a bookie who shows the odds and the races to the client. The client can bet on any horse in the race they are shown. The client can choose what horse to bet on and how much to bet on the horse, or they can choose not to bet on the current race at all. Races are run every minute, and the winner of the race is broadcast to every client that bet on that race. A new race is then generated and broadcast to all connected clients and the process repeats.

Technology Stack

- **REST**: Used for general APIs that the microservices use to exchange data and make requests.
- **WebSockets***:* Used to send details and the winners of the races to the Client.
- **MongoDB:** Database for horses
- **Docker:** Container management for the services and database
- **Kubernetes:** Container orchestration for scaling and load balancing

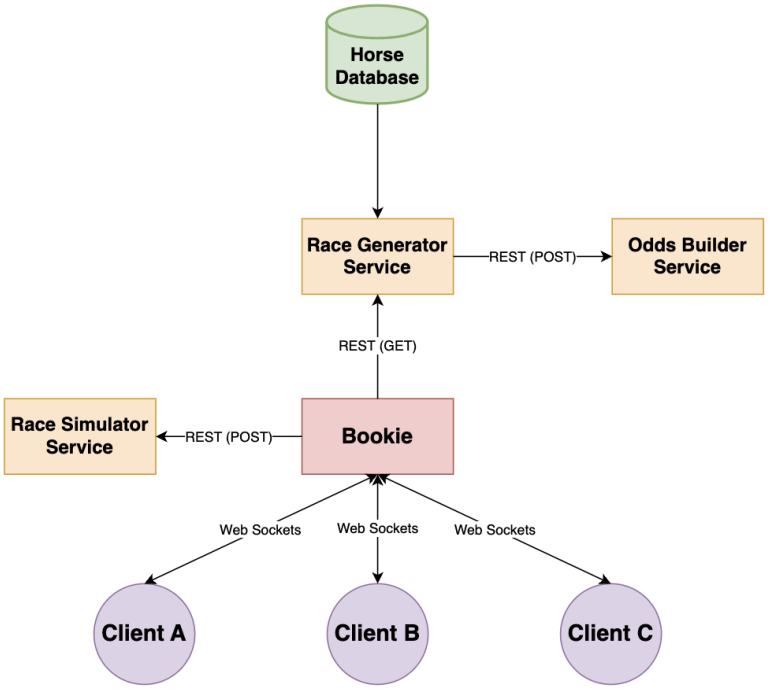
We used **REST** due to its simplicity and ability to use HTTP methods such as GET and POST. It allows the RaceGenerator, OddsBuilder, RaceSimulator and Bookie to communicate effectively and manage resources like the race and odds data. The fact that REST is stateless simplified the design of the server components and improved scalability by eliminating server-side sessions. REST is not effective for real-time communications that require persistent connections, which is why we didn't opt to use REST for communication between the Bookie and the Client.

We decided to use **WebSockets** because it provides a real-time communication channel between the Bookie and the Client. The race results and the new race updates need to be sent to the Client in real time, without the client having to request the data from the Bookie itself every time. WebSockets also have significantly less latency compared to traditional HTTP requests, which is important because we want the client to receive race updates as fast as possible. WebSockets are more complex to implement than standard HTTP communication, and managing a large number of open WebSocket connections can be challenging from a scalability perspective.

We chose to use **MongoDB** because it is a NoSQL database, meaning that we can store documents in a flexible, JSON like format. This is essential for managing the data for the horses. MongoDB also has quick retrieval capabilities which enhance the performance of generating the race, ensuring the client does not experience any latency between each race. MongoDB also supports horizontal scalability through sharding, enabling the distribution of data across multiple machines. However, MongoDB's performance depends heavily on having sufficient RAM, which can lead to high memory usage, and managing its scaling and backup strategies can be complicated.

Docker simplifies the deployment and scaling of our services. Docker provides great portability, consistency and fault isolation for each of the services but it can be difficult to manage a large number of containers. We chose **Kubernetes** to orchestrate the containers because it automates scheduling, load balancing and fault tolerance for a live system without manual intervention. However, Kubernetes can cause a significant resource overhead, especially for smaller applications such as our Horse Betting app.

System Overview



Race Generator:

- Receives GET requests from the bookie to generate new races.
- Reads in horses from the database.
- Generates a race from a random predefined number of horses.
- Generate a Races object with a race name, time and list of horses.
- Sends this race object to Oddsbuilder to generate the odds before sending the complete Race object back to the Bookie.

Odds Builder:

- Receives POST requests from the Race Generator.
- Calculates the odds for each horse in a race. The odds for each horse is inversely proportional to the horses score/the sum of scores for all horses in the race. This means that horses with higher scores will have lower betting odds.
- Sends the race back to Race Generator with the newly calculated odds.

Race Simulator

- Receives POST requests from the bookie containing a race.
- Picks the winner of the race. Horses with higher scores are more likely to be picked as the winner.
- Sends the result of the race back to the Bookie.

Bookie

- Sends a GET request to RaceGenerator to get the next race.
- Broadcasts the details of the current race to all connected clients via a websocket topic.
- Each race is given a separate websocket topic where the results of that particular race will be published.
- Every 60 seconds, the current race is simulated and the winner is sent to the designated topic for that race.
- A new race is then generated via RaceGenerator and the process repeats.

Client

- Subscribes to the general Races topic on startup using websockets.
- Receives the race data for every new race and displays it to the user.
- If the client does not want to bet on the current race, then the Client simply waits for the data of the next race.
- If the user does want to bet on the current race, then they input the horse's name that they wish to bet on and how much they want to bet. The Client then keeps track of this bet and subscribes to the websocket topic for that particular race. The Bookie publishes the winner to that topic after the race has been simulated and the Client tells the user if they have won or lost, along with how much money they won if their horse won the race.

Our system handles scalability by using Docker and Kubernetes for managing and scaling the containers. Kubernetes handles load balancing which deals with the fault tolerance for our system, meaning that if a service goes down, it will automatically create a new container for that service and the system will stay running as normal.

Contributions

Brendan: RaceGenerator Service, Database Configuration, Docker, and Kubernetes

Liam: RaceSimulator, OddsBuilder Service, Websocket connections/message handling

Tommy: Bookie Service and Client

Reflections

What were the key challenges you have faced in completing the project? How did you overcome them?

The biggest challenge with this project was coming up with a method for the bookie to send race results to only the clients that had bet on the given race. Initially we tried to get the Bookie to send messages to each individual client, informing them of the result of their bet on that particular race, but this was a bad solution to the problem. We fixed this issue by creating a separate websocket topic for each generated race. Clients who bet on that race then subscribe to the topic for that race and the Bookie broadcasts the winning horse to all clients who are subscribed to that race's topic. The results of the bets and the amount of money won is then calculated on the Client side and displayed to the user.

The kubernetes implementation was more difficult than expected. The challenge was implementing the load balancer with the websockets. We overcame this by exposing the port 8080 in the bookie service pod

What would you have done differently if you could start again?

We would have spent more time planning out how the communication between the Client and the Bookie would work. The errors we encountered when building that part of the system proved to be the biggest bottleneck in the completion of this project, and could have been avoided if we spent a bit more time on planning out the communication process between these 2 components. The decision to use a microservice architecture was good because it allowed us to effectively split up the workload across multiple containers, leading to better service cohesion and better overall fault tolerance.

What have you learnt about the technologies you have used? Limitations? Benefits?

We learned that REST is not a good method for sending real-time updates to users about events that occur on the server side of the system. We initially planned on using REST for communication between the Bookie and the Client but quickly realised that this was a poor architectural decision for this system. REST did prove to be very effective for communication between the race services, where the service endpoints only needed to be used when they received a request from other services/bookie.

The use of Docker and Kuberenetes was a great architectural decision because it made it much easier for us to ensure fault tolerance and scalability within our application. Dockerising the application services and Bookie was very straightforward and could be done after all of the business logic for the application was coded, allowing us developers to better focus on the core objectives when coding each service.