That’s So Maven

Bryan Agar

21204892

Jack Mulligan

21201412

Michael Scallon

21205085

Synopsis

Application Domain

The world football industry has been rapidly increasing over the last twenty years with the English football industry alone being valued at over four hundred and fifty billion and individual players being sold for over one hundred million. For clubs to produce income it is imperative that clubs both perform to a high enough standard to compete for tournaments such as the European champions league for tournament prize money while also having “superstars” to sell club merchandise and jerseys. With such a demand for this quality it has become incredibly important for smaller clubs to find undervalued players based on their underlining statistics and for bigger clubs to ensure that the sum of money they are paying for a player is a fair price. Statistics have also become a powerful tool to completely dictate a transfer policy with examples such as Brentford football club deploying a “moneyball” approach to the signing of players which was used to help the club get promoted and maintain a place at the top table of English football in the premier league.

While these club by club scouting networks are extremely customisable and can be as in depth as desired, smaller clubs do not have the access to the necessary funds to deploy their own and therefore miss out on one of the most important facets of modern football. The concept behind our application is to build and provide a detailed global scouting service which could be deployed by any team of any size and remove the expense of a custom scouting network which would in theory produce income by charging a subscription fee. This concept would also take some of the administerial and technical challenges of deploying their own application from smaller clubs without the experience or expertise to achieve this.

Application Functionality

The application works by having four main scouting networks around the world detailing the results and performances of each player from each game. These scouting networks are updated by local field scouts who watch and report on each game and allow a collection of data which can be stored, manipulated and viewed. The user can access our interface which allows for the user to define what kind of player they desire by limiting age, searching for certain statistics such as goals scored and look for players which may be in undervalued markets. The list of players fitting these criteria can then be returned and examined by the user.

Another alternative way in which a user can use this service is to search a player which they have seen live in one or more occasions but do not know any information about their production in the longer term. While a player might pass “the eye test”, access to information about their career can tell the user that the performance they observed was not a one-off great performance and is instead an example of their ability or to show that a player may not be as high quality as they have looked at first glance.

Planning

Before sitting down as a group we decided to research into various technologies, platforms and project ideas before we met so when we met as a group we could finalize what we wanted to achieve within the project and also within the timeframe given. It was important to us as a team to have a clear goal in mind while knowing we were under a time restraint to finish the project.

Due to the world cup and the team members having a strong interest in the sport of football, we decided to carry out and complete a project that involved this sport. Other ideas such as cloud storage, food delivery and online shopping system were also explored but due to many factors the football scouts project option was decided on.

As this was an extensive coding project and had multiple unique parts, we decided to split the individual components up into 3 sections, these included, client, broker and field scouts/endpoints. This approach allowed us individually to work on our own part of the project while having collaboration with other teammates in order that our individual component fitted together nicely just like a jigsaw puzzle.

To facilitate collaboration on coding components, it was important to use version control systems, we used tools such as Git and Gitlab, to track and manage code changes. It was also helpful to establish clear communication channels, we used whatsapp for informal messaging and allowed us to set up zoom meetings where more detailed topics needed to be discussed.

Technology Stack

The following distribution technologies were used to build the project:

- Netflix Eureka – Scaling out the number of scouts in the system.
- Redis – Caching data from the scouts at the broker.
- Docker – Containerisation.
- MySql – Database management.
- Rest – Communication between application and database.

The primary reason this particular set of technologies was chosen was their strong support for use with Java and Spring Boot, a Java framework. With the decision to build a decision which relies on REST to communicate, using Spring Boot was the obvious choice for a Java based system. Netflix Eureka is built on Spring Cloud, making it the obvious choice for application discovery. While there are many key-value caching solutions available, Redis was found to have the simplest integration to Java in the Jedis library. After doing research about support for RDBMS solutions in the javax.persistence library, MySQL was found to be the simplest option. Finally, Docker Hub contains a range of images featuring different versions of the Java Development Kit. This was an essential requirement, as Java 1.8 needed to be used to ensure good functioning of many parts of the system, particularly Netflix Eureka.

Netflix Eureka

Netflix Eureka is middleware designed for the discovery of services and web applications. It relies on REST to facilitate communication between Eureka servers and clients. Although Netflix Eureka was originally developed for managing the load balance and failover of mid-tier services, it has a number of properties that also make it a suitable candidate for managing services which are distributed globally. First and foremost, it is capable of registering services which are on different network internet protocol (IP) addresses and is not restricted to the use of different port numbers on the same IP address. This was proven in practice through the dockerisation of the service via docker compose, with each container being given its own network address within the Docker Network. Secondly, it is developed for use in Spring Cloud, with a Spring Boot service only requiring a single annotation around its entrypoint to register itself with the Eureka server. This makes the barrier to entry for our use case far lower when compared with alternative solutions.

Redis

Redis is an in-memory data structure store, which uses key-value pairs to store data using different data structures (e.g. lists, maps, sets). A popular use case for Redis is for storing data within in-memory cache. Although other key-value cache options are available (e.g. Couchbase, AWS DynamoDB), Redis was chosen for a number of reasons. Firstly, it has a well-maintained library for Java (Jedis). Secondly, it offers easy-to-use APIs for storing multiple data entries under a single key, which suits our player data well. Finally, although this wasn't explored in the current development of the system, it also offers support for out-of-memory storage. This feature could be exploited to facilitate broker replication.

Rest

Rest was the primary method used for querying the databases associated with the project. Rest is an architectural style for providing standards between systems on the web and spring-boot was used to create these restful API's. There were three main spring dependencies used – spring web ,spring data jpa and hibernate alongside other such as spring fox and MySQL driver which connects to the MySQL database to the application

MySQL

MySQL was chosen as the data management language for this project. There were many reasons why this was chosen for each of the main regional databases such as its dependency and scalability. It allowed for a system which was easy to manage for the scale of this project while also allowing for expansion if the project was to be taken further in the future. The tables were created using data from a csv file and a SQL query to convert this into the manually made tables while ensuring that each feature was assigned a correct type and that all columns that were present in the csv file were also present in the SQL table. Each of the four tables for each region were stored in the same database.

Docker

Docker uses OS-level virtualisation to deliver software in packages called containers. Containers are similar to virtual machines, in that they allow for different pieces of a complex system to be divided into self-contained units. In contrast to a virtual machine, with Docker there is no need to provision a guest OS in each unit. Instead, interactions between the host OS and software inside the container is facilitated by the Docker Engine.

The software was selected for use in this project as it assures that the system runs across different devices and different operating systems. Docker compose is a particularly attractive solution in this regard – it allows for a complex system consisting of many different pieces of software to be stood up using a single command from the terminal. Docker images are also available for all the principal components of the system (e.g. the Java Development Kit, MySQL server).

System Overview

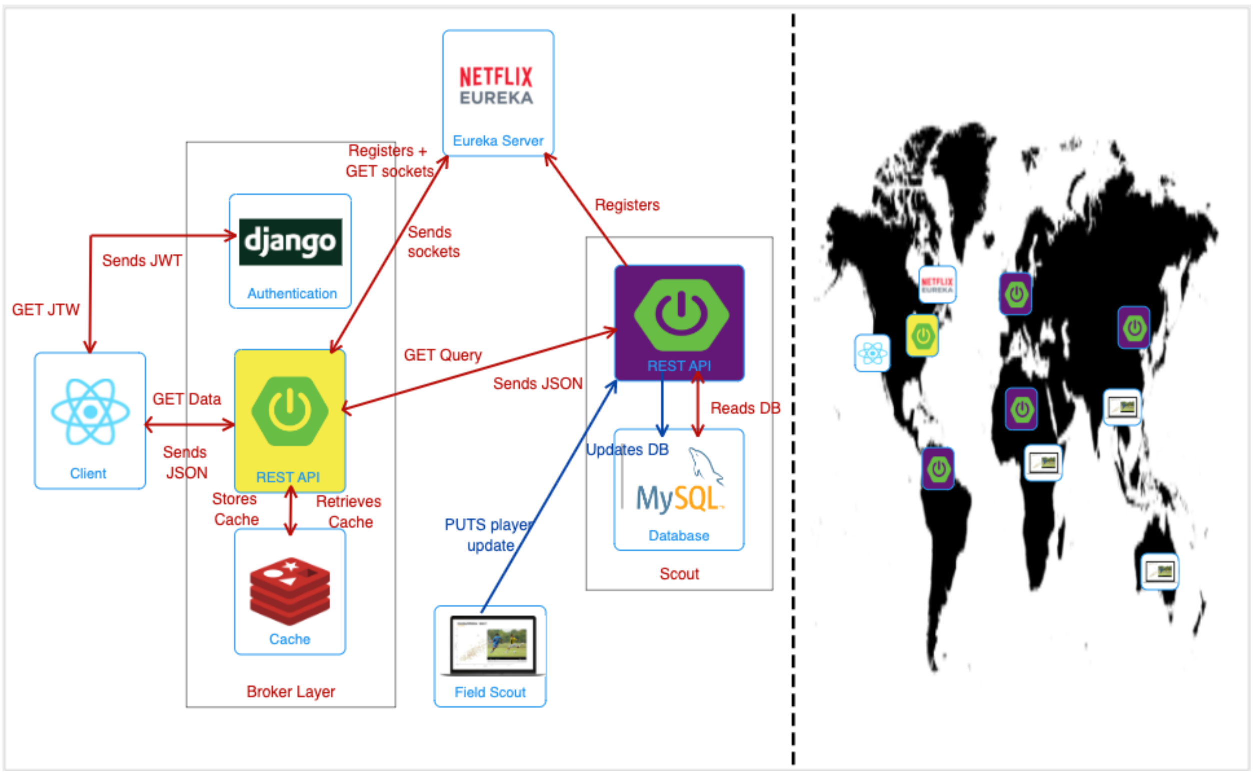

Figure 1: System Architecture Diagram (left) and Distribution of Components (right)

Client

The client component was two services combined and dockerised into one docker compose file. The client consists of a backend for user authentication and a frontend for user interaction and data being shown from the broker. Django was used for the backend, this is a python based Model view controller framework. This allowed us to scale the web application quickly and allowed us to focus on building the other parts of the system once this was done. The built in django user accounts authentication was used.

Having user accounts allowed for the application not only to be used by guests but also allowed the end user to have a personalised experience. The user is able to login in and create an account using a valid email and that would store their info securely in the back-end by Django.

The user can stay logged in for up to 2 hours on the same browser as after that period the JWT tokens expire due to inactivity. JWT, or JSON Web Token, is an open standard used to share security information between two parties — a client and a server. Each JWT contains encoded JSON objects, including a set of claims. JWTs are signed using a cryptographic algorithm to ensure that the claims cannot be altered after the token is issued.

JSON Web Tokens (JWTs) are a popular way to handle authentication in modern web applications. Here's an overview of how JWTs is used to handle authentication in a React and Django application

1. The user logs in to the React application using their credentials (e.g., email and password).
2. The React application sends the login request to the Django backend, including the user's credentials.
3. The Django backend verifies the credentials and, if they are valid, generates a JWT. A JWT is a JSON object that is signed and encoded. It typically contains information about the user and a unique identifier for the token.
4. The Django backend sends the JWT back to the React application as a response to the login request.
5. The React application stores the JWT in local storage or a cookie.
6. From this point on, the React application will include the JWT in the Authorization header of any subsequent requests to the Django backend.
7. The Django backend will verify the JWT on each request and, if it is valid, allow the request to proceed. If the JWT is invalid or has expired, the Django backend will return an error.

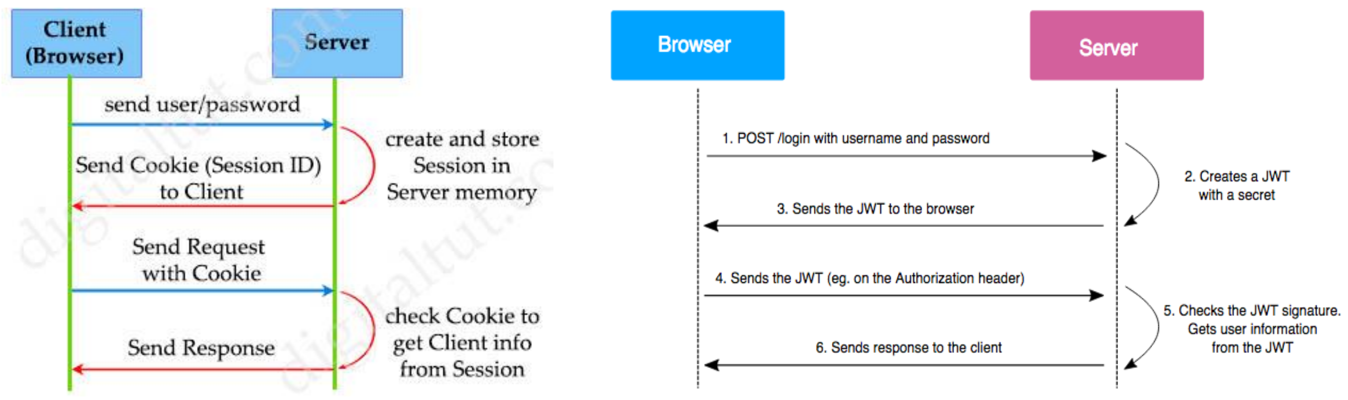
Client (Browser)
Server
send user/password
create and store Session in Server memory
Send Cookie (Session ID) to Client
Send Request with Cookie
check Cookie to get Client info from Session
Send Response
Browser
Server
1. POST /login with username and password
2. Creates a JWT with a secret
3. Sends the JWT to the browser
4. Sends the JWT (eg. on the Authorization header)
5. Checks the JWT signature. Gets user information from the JWT
6. Sends response to the client

Figure 2: Example of how JWT Tokens are Shared between Client and Backend

The frontend was written with the highly popular and industry used javascript library called React. React allows developers to make complex sites by creating reusable components that are all independent of each other, but communicate through hooks, ensuring a high performing application. Furthermore, the modularity and reusability of each component make styling far easier than through plain HTML and CSS, which we all agreed was a key factor to developing a successful application. React version 17 was used for the frontend.

The client through the frontend was given a list of inputs the request to the broker. These include Position, Age, Goals per game, assists per game and concessions per game. Once altered by the end user, the client would then be able to send the personalized query to the broker and a unique request would be returned. The get request was carried out by using the Axios external plugin library. you can use Axios to send HTTP requests to APIs or to other servers in order to retrieve or save data.

Using Docker can make it easier to manage and deploy complex applications like those built with React and Django, especially if you need to run the application in multiple environments or scale it up over time. It also allows you to take advantage of the benefits of containerization, such as improved security and isolation of resources.

Broker
The broker is a Spring Boot (v2.5.6) application running a single RESTful read endpoint (/players), which returns data about the players as JSON. Querying the broker without parameters returns all players from all scouts. The endpoint supports five optional parameters that allow for the data to be filtered. Data is returned using a PlayerReport format, containing nested PlayerData (attributes which can be queried as parameters) and PlayerInfo (attributes which cannot be queried as parameters) key-value pairs.

The endpoint performs filtering by first retrieving the entire dataset from either the scouts or cache, gradually reducing down the dataset through a series of private methods which take in a list of players and a parameter to filter by, returning a list of filtered players. The sequence of these private methods allows for both single and multiple parameters to be executed on the endpoint. For example, the age parameter allows the user to restrict the result to players of that age and below (e.g. /players?age=25 returns only those players that are 25 or younger). Multiple parameters can also be combined together (E.g. players?age=25&position=Forward returns forwards that are 25 or younger).

Two pathways are available for the collection of player data from the network. If the cache is unavailable or expired, the broker collects data directly from the network of scouts. It utilises the Eureka Server to collect the sockets associated with the scouts to do so. As a requirement of this, the Spring Boot application is itself a registered Eureka Client. When running the data collection, it first retrieves all applications registered on the server, with socket information extracted from those applications with the name “SCOUT” only (all scouts register under the same name). It then queries the instances of scout in turn, returning a list of player data back to the endpoint call. It then logs the time at which the scout call occurred for caching purposes, as well as saving down all data to Redis for later retrieval. Although it would have been preferable to save down the player data in the nested PlayerReport format, the data is cached as a flat map as this proved to be more compatible with the map APIs available in Jedis.

If cache is available and fresh (retrieved in the last 60 seconds for demonstration purposes), then the Redis server is queried by the broker. All entries to the Redis server are saved with a common key prefix (playerReport:), allowing all player data entries to be collected together in a single query. These are then reconstituted from the flat format into the PlayerReport class using its associated helper classes (PlayerInfo and PlayerData).

Scouts and Field Scout

Each of the four scouts contrast each other slightly implementing different databases, however, use the same concepts and technologies to create rest API's to query these SQL databases using spring boot as the foundation of this work. The data for each of these is taken from the 17/18 premier league season as we wanted realistic information without needing to create fake data. The teams in the league were divided into four groups and each group was assigned a region to simulate scouts from different parts of the world. The intention of the data was to showcase the application and therefor some detail was removed from the initial data to simplify the process of creation and to focus on only the key metrics associated with each player such as goals scored with some slightly more detailed stats such as goals per match (per 90 minutes). Players from a certain region can be accessed with different URL paths for example Europe can be found using 8081 or all can be found using 8080. The key operations that can be deployed are to find all players, find a particular player, delete a player, or update a player (player names are used as primary keys and so cannot be changed). Europe , all of the Americas , Africa and Asia + Oceania were chosen to represent a fictional version of every league in the world , however in a real world context a players club and league would also be available in the database so that more localised searches can be preformed for example an Irish club can search just players playing in the league of Ireland or a specific club rather than just searching all of Europe however adding this detail was deemed unnecessary due to the relatively small scale of having approximately 500 players compared to a real world application which would store thousands.

The application properties file in each scout package is used to configure information such as where to find the desired database, username and password for example. It also allowed for some other vital information to be given to the system such as allowing the pathmatch matching strategy to be defined (ANT_PATH_MATCHER being used in this case). Another initial step needed to start the creation of the API's were to initialise spring-boot using the scoutAppApplication file. This used annotation such as @springbootapplication to inform spring to scan for spring components found in the package.

Once the first steps were completed and the correct dependencies added the scout packages where the API's were found were then split into several different packages – data, payloads and web alongside an exception package used to handle exceptions. The data package allows the information from the database to be modelled with additional information such as which feature was the primary key and @GenericGenerator allowing the primary key to be a string(name) rather than needing to be an id number solving the issue of the same id being associated with two or more players across the different databases by accident using the "uuid" system. The data package also contains files to deal with requests and responses allowing information such as each feature called form the database being for example not null by again using annotation.

The service package is where the business logic of the application is stored and is split into an interface and a class which implements this interface to ensure more separation of concerns. This file allows all four crud operations to be performed on the database. This service is called by the controller and can access the database instead of the controller accessing it directly. The player service class also makes use of several classifiers such as @service which makes the class assessable during classpath scanning and @autowired which is used to preform field dependency injection. The final main section of the rest API is the web package which houses the controller class. This provides a path for each of the operations to be preformed using each of get, post, put and delete while also allowing the actual path itself to be defined such as player/find/<playername> being used to find a single player. There is also some rudimentary swagger implementation which is used to document and test the API and can be found using localhost:8081/swagger-ui.

The field scout service is used as an example of how local scouts may be able to access and update player information as would be the case in the real world. This is implemented in a basic manner however is intended to demonstrate the potential that allowing local scouts to alter each database can have making the application more realistic while also having the benefit of allowing the database to be updated allowing for broker caching to be displayed. This package runs in a loop of ten seconds to demonstrate its functionality in a quick and appropriate manner for the context of this assignment.

Scalability and Fault Tolerance

Scalability

The initial offering of the service is based on the premise of each continent having a single scout. Under this circumstance, the hard-coding of the sockets for each scout could be justified. However, this only represents the starting point for the system. As the service grows, multiple scouts could be employed on each continent, with each scout being responsible for their own group of teams and players. The system could even move away from continents altogether and offer a more granular service, with scouts being assigned to countries, leagues or teams. Moreover, there could be shifting needs in terms of scouts being assigned to different locations. Having sockets hardcoded at the broker therefore represents a bottleneck to the future growth of the service.

To prevent this bottleneck, Netflix Eureka was utilised for its ability to register services. Both the broker and the scouts register themselves with the Eureka server on launch. The broker can then make use of the registry to access both the host address and port of all scouts. All scouts are registered under the same name (“SCOUT”), with the read endpoint of these instances being queried in turn by the broker. Using service discovery in this fashion allows the number of scouts available to the broker scale up and down as required, while also allowing a great degree of flexibility in changing network addresses and ports (though this isn’t taken advantage of in the scope of this demonstration).

Fault Tolerance

Caching is used to provision fault tolerance within the system. While more involved methods of fault tolerance were also considered (e.g. failover at the broker and the scouts), caching suits the use case well as the player statistics aren’t foreseen as being updated with great frequency. These will change from week-to-week after a given match weekend, with occasional updates occurring during the week as well. After all, the quality of a player is something which emerges over time and seldom becomes immediately apparent from one week to the next.

As such, the preference is for making data available, as opposed to making it consistent. Caching thus works effectively as a fault tolerance solution. By having caching data from the scouts in the broker for an extended period of time (a real-life use case would be a number of days), the client can be assured of access to data even if scouts drop off the network. Should a scout deregister itself from the system due to being taken offline, clients can continue to access data while the developer team seeks to bring a particular scout back online. A monitoring service would further assist in this, though this wasn’t explored in the scope of this project.

Beyond fault tolerance, caching also helps to improve performance. With scouts foreseen as being located around the globe, fetching data directly from the scouts will be quite time consuming. Having this data stored at the broker allows for shorter lines of communication and quicker response times.

Contributions

Michael

- Created mysql table structure and original cleaning and inserting of data
- Created the regional scout packages(excluding netflix eureka part)
- Worked collaboratively on field scout

Bryan

- React Frontend
- Django Backend (User Authentication service)
- Broker requests from client
- UI/UX
- Dockerisation of client

Jack

- Spring Boot Application for Broker.
- Caching service built using Redis.
- Discovery service built using Netflix Eureka (incl. introducing Eureka to the Scouts).
- Worked collaboratively on the field scout.
- Dockerisation of all parts of the system except Client and Authentication service.

Reflections

Key Challenges

Some of the difficulties that we overcame that will be spoken upon are communication, scale and new technologies. While we communicated well as a group meeting times and locations were at occasionally difficult to organise due to different timetables , commitments and locations. We overcame this by allowing as much flexibility for each other member as possible. We also alternated between in person meetings which we aimed to have at least once a week while also utilising Microsoft teams and zoom to communicate online to discuss our progress. Whatsapp was also used to provide small updates and organise meetings. As the course of the project progressed, we became more and more willing to meet each other at unsocial hours when needed and the effort of all three members ensured that the potential communication issues between members were by the completion of the project very minimal.

The scale of the project and the use of new technologies at times were daunting for each group member. There were many elements and ideas we wished to try and add to our project and this was done successfully by incrementally building up our application opposed to trying to tackle major issues all in one go. The ability to rely on our other team members was also incredibly helpful in solving problems. Patience was the final tactic used to combat these issues as we reassured ourselves that we would be able to fix all our problems if enough time and effort was applied.

What the Team Would Have Done Differently

The main change that we would make as a group would be to increase the amount of planning before moving to the following step of the project. There were at times instances were the next step seemed clear and obvious and we deemed it appropriate to move straight into an idea whereas a few days to consider other options from time to time would have been beneficial. One example of this was trying to connect multiple different databases to the one application. While this can technically be achieved it turned out to be very difficult to implement while also being a poor idea in terms of performance and would have added little benefit to the project. The final decision was to have multiple tables in one database and this was both much easier to achieve, while also helpful to reduce the amount of resources needed to run the application making it faster. This was a good lesson to learn and demonstrated how complexity for the sake of complexity is not a good idea.

In terms of system architecture, that only a single instance of the broker is provisioned represents a single-point of failure within the system. To resolve this, a replica of the broker should have been created, with the replica serving as a passive backup. A load balancer would then sit in front of the broker and its replica, ensuring that at least one broker is always available to the client. Furthermore, a monitoring service would also serve the system well, particularly for monitoring the health status of the Eureka server. Should this fail, the broker will no longer be able to contact the scouts. A monitoring service would therefore help to quickly resolve the failure of the Eureka server. Socket information from the scouts could also have been cached down in redis for this purpose.

Takeaways about Technologies Used

Netflix Eureka
In terms of the developer experience, getting started with Netflix Eureka required some groundwork, as the documentation is somewhat sparse. This is understandable as it’s an project that’s been generously made open-source by Netflix. Once the fundamentals

of the package were understood from existing projects, it was impressive how easy it was to both register and discover services using Eureka. In newer versions of the package, only the dependencies are required for registration, with no annotations required. Sadly, due to some compatibility issues with the javax.persistence engine, using Eureka required Java to be downgraded to v1.8 and Spring Boot to be downgraded to v2.5. A potential limitation to the package is how well it could be used in the wild, particularly if applications were distributed across the globe. Although docker compose does provide each service with its own network address, the components still ultimately exist on the same machine in this demonstration. Registration and discovery speeds may well suffer if used in a real-world scenario.

Redis
Despite learning during the research stage of the project that redis was known for its blazingly fast speeds, the team was impressed by how quickly redis could both write to and read from cache. Query times were significantly improved at the client once caching was introduced. However, it must be noted that beyond achieving a separation-of-concerns for caching data, the use of redis in this system doesn't offer a significant value-add over simply saving down the data in-memory within the Spring Boot application itself. Using redis does however lower the costs in performing broker replication at a later stage due its support for out-of-memory storage. A replica of the the broker could be easily maintained by having a master-slave relationship, with the slave acting as passive failover should the master broker fail.

Docker
Docker proved highly effective at ensuring that a system composing of many different kinds of software functioned well across different machines. This eased the coordination difficulties in the development phase, as well offering assurances that the system could be used without difficulty by others. However, timing the building of containers proved to be a major issue in using docker-compose. Although solutions such as wait-for-it are available, they proved to have compatibility issues in terms of querying certain parts of the system, in particular the MySQL server. Better research into using Docker would have prevented this and potentially led to a different choice of RDBMS. The workaround of using fixed delays to launch different containers is brittle but proved workable through being generous with delay time, although these launch delays will prove frustrating for users on newer and/or faster machines. The use of launch delays could also lead to compatibility issues with older systems, with this being impractical to test.

MySQL
MySQL proved to be an effective solution for use with our scout's CRUD API. However, the resources required to run several mysql-server containers simultaneously were

underestimated, leading to the demonstration using a single container to back-up all 4 scout applications. This led to the use of a single MySQL server within the scope of the demonstration project. A more lightweight database solution (E.g. SQLite) may have helped to alleviate this. As previously noted, better research into container orchestration may also have led to a different choice in RDBMS solution.

REST

REST proved to be a more than sufficient architectural style for this use case. With a CRUD API required at each scout, as well as an endpoint returning JSON data. Netflix Eureka also works on the basis of REST, with the REST API integrating well with this.

Conclusion

This project proved to be successful and allowed us as a team to build a distributed, scalable and fault tolerant project that addresses a real world problem. It also allowed us to utilise new distributed technologies and gain a better understanding of the technologies explored and thought within the module itself. Distributed systems offers a number of benefits, including improved scalability, reliability, and availability. They are also a key aspect of modern computing. With the proliferation of cloud computing and other distributed technologies, a strong understanding of distributed systems is becoming increasingly important in the field of computer science and also for working as a software engineer.