Distributed Security Management

Shane Cooke	Daniel Houlihan	Jack Crowley
17400206	18339866	17484414

Synopsis:

System description

We intend to develop a software system that allows a client and microservices to interact with 'brokers' of different sorts via RESTful API using GET and POST calls. The actual services being provided are somewhat arbitrary, the architecture will be the same where a user can get quotations, make an order, and track an order. We of course will not actually be delivering and tracking items so we will have to come up with a way to mimic this functionality.

Application domain

This application is for users who want to order security services. Different urgencies of services could be needed for different users and we cater for this. However, the more urgently you need your service, the more it will cost you. Not all security services will be available at every location. For example, our sea service will not be available to those on land (or in space).

Functionality

The Distributed Security Management application will allow users to create a user account, get quotations from multiple different service providers, place an order with one of these providers, and track the order location and arrival time. The user will be prompted to create an account within our service, and provide information that will allow us to provide them quotations from our service providers that will fit their needs. They will then be shown multiple quotations from the service providers, and will be prompted to choose whichever quotation that they desire to order. When a quotation has been chosen and a service is ordered, the user will be provided with a tracking reference with which they will be able to track the location and delivery time of their order. Once the service has been delivered, the user will be notified through our tracking service.

It will be possible to create multiple different user accounts and multiple different orders within a single instance of our application. It will also be possible to track multiple different orders in one instance of our application.

Technology Stack

The main distribution technologies that we used in our project are:

- *REST*

Used for the communication between the brokers and all of our security services, each of which are microservices, and the client.

- *Docker*

Used to package up our application into a container. Using docker we can run all the microservies within a container environment and with just two commands. Docker makes the deployment and management of the project much easier.

- *Spring Boot*

Spring boot is an open source Java based framework which we will be using to create our microservices. It provides a very easy to use platform on which we can develop our project.

- *Maven*

Maven is an open-source build tool which allows developers to build, publish and deploy several projects at once. Maven focuses on the simplification and standardization of the building process and takes care of builds, dependencies and releases.

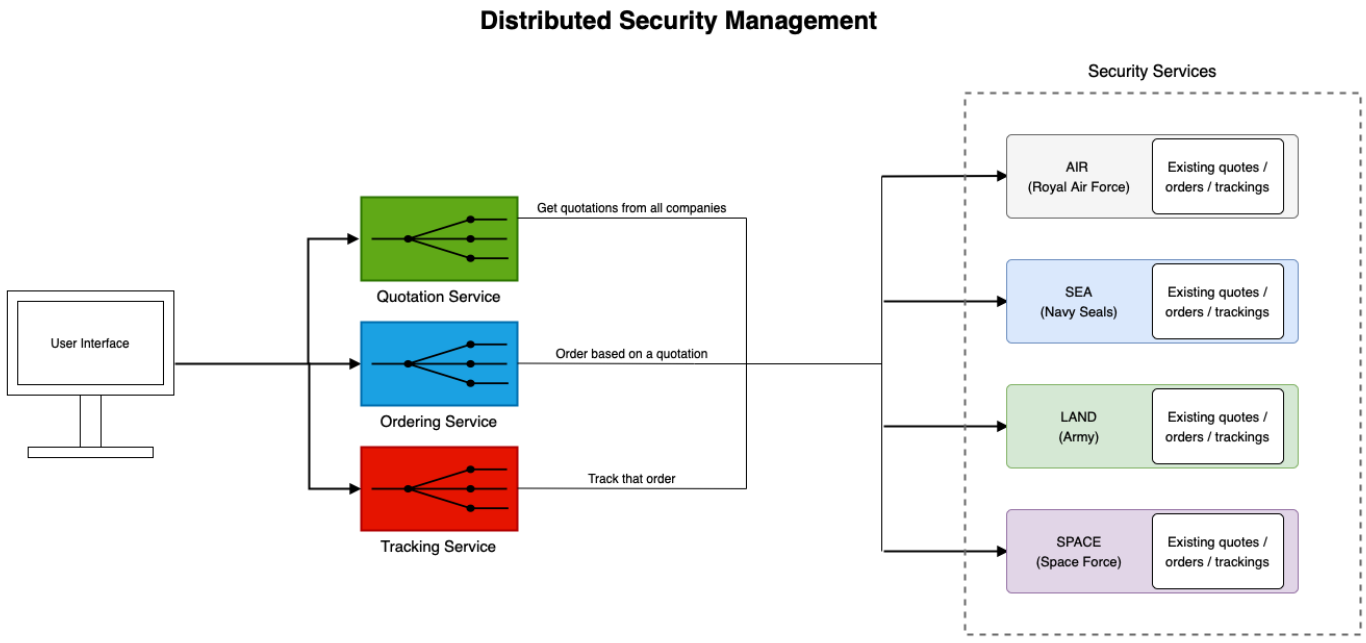
System Overview

Components

The main components of our system involve the three service providers (air, sea and ground), the quotation service, the ordering service, the tracking service and the client. The three service providers each provide a price dependent on the urgency and location of the required job. The quotation service will query each of these three microservices, and generate quotations where a quotation is possible. Once the user has decided upon the quotation they would like to order, the ordering service is used to create and store this new order. The tracking service can then be used to track the location, and time of delivery of each order.

Below is a diagram of the system architecture of our application:

System architecture diagram



There are four security services, each providing the same methods but different implementations. These methods are for making quotations, ordering a service, and tracking a service. These are all connected to each broker service via RESTful API. These broker services act as a middle man between the user and the security services. The brokers interact with the user interface (client) via RESTful API as well.

Scalability and fault tolerance

Each security service keeps track of any quotations / orders / tracking that is taking place with their service. This makes the architecture more fault tolerant in that the data is not lost in the event of a disconnection between services and brokers. By design our system allows massive scalability in that unlimited orders and user accounts can be created in a single instance of our application, given an unlimited amount of memory. We used Java storage structures such as HashMaps, TreeMaps and ArrayLists in order to allow a virtually unlimited amount of user account

creation, order creation and order tracking in a single instance of the application. New security services can be added easily. After adding a new security service, only one new line of code needs to be added to the broker services to allow the program to function with it. One could instead use Netflix Eureka and Netflix Zuul as service discovery so that no new code would be needed. After a few failed attempts to implement this, we decided to focus on making the program more robust and fault tolerant as we were running low on time left to complete the project.

Contributions

Shane Cooke:

To contribute to this project, I firstly worked with Daniel to design and create the four services (air, ground, sea and space). This involved creating the quotations that each service would return, and ensuring that a connection was open between each service and our other classes. Daniel mainly took care of setting up and connecting the ordering service and quotations service, while I mainly focused on the setting up and connection of the tracking service. This involved ensuring that the tracking service was connected to the quotations and ordering services, and also ensuring that correct information was being communicated. I also contributed to the creation of the user interface, and the fault tolerance associated with the UI. We both debugged and tested the application vigorously many times, and ensured that no user input or connection errors would crash the application.

Daniel Houlihan:

Shane and I worked together to come up with an idea that was both interesting and realistic in the time frame given. I set up the architecture and bare bones of the software, making the RESTful connections where needed with arbitrary methods returning arbitrary results. Once this was completed it was much easier for Shane and I to implement proper methods to give functionality to both the broker services and the security services. We both worked on the client to add any features which we thought would be beneficial to the system. Both of us working on it ensured that the UI was intuitive and nicely designed. I implemented the quotations, tracking, and ordering modules and Shane helped to develop the methods within. Testing the client together was very beneficial as there were things both of us had implemented which had obscure bugs that we did not see initially. I attempted to set up Eureka and Zuul and got all the services to register with Eureka. However, I did not get much further than that and decided to spend the remainder of the time making sure the project was implemented properly and using best practices.

Jack Crowley:

Reflections

The Distributed Security Management project was overall a great success. We started off with a vision of how the project would eventually turn out and we feel as though we have achieved said vision. Although some aspects of the project were much more difficult and time consuming than others, we ultimately achieved all of the goals which we set out to achieve, and thoroughly enjoyed the process of doing so. Overall, we felt as though this project went very smoothly, however there were a few key challenges that we faced in the process of creating our application.

Challenges

The first key challenge that we faced in completing our project was the successful connection of all of our microservices. The system that we set up requires a vast amount of connectivity and communication, so connecting all of these micro-services and setting up communication between each was a major challenge. To overcome this challenge, we first had to study and learn how ports are used by RESTful applications. We had some understanding of this from our Distributed Systems studies during the semester, however in order to create our own system from the

ground up, we found that we needed a deeper understanding on the topic. We then carried out a massive amount of experimenting and debugging in order to produce successful communication between all components of our system. This process took a vast amount of troubleshooting and debugging, however eventually we successfully connected all of the micro-services and allowed communication between each of them.

Another key challenge that we faced in completing our project was allowing the user to create multiple different accounts and multiple different orders in one instance of our application. Allowing the user to create a single account and a single order proved relatively easy, however allowing multiple accounts and orders caused us some trouble, and took some time to solve. In order to overcome this challenge, we decided to use a multitude of Java storage methods such as TreeMaps, HashMaps and ArrayLists in order to store each order and user along with an index. Once this was successfully set up, we were able to store and access whatever user or order which we desired, by referencing the index at which each was stored. Now, multiple users and multiple orders can be created, placed and tracked in a single instance of our application, which we saw as a vital aspect of our system from the outset of this project.

Another key challenge that we faced towards the end of our project was the Dockerization of the system. Due to the amount of micro-services we use in our system, it was at first required to run the air, ground, sea, quotations, ordering, tracking and client services separately in many different terminals. This was far too much work in order to run our system, so we decided that Dockerization was essential. Setting up this Dockerization correctly was at first very difficult due to the amount of microservices, however after some trial-and-error and debugging, we successfully containerised our project so that all micro-services could be run as a single-entity.

If we could start our project again, I think the major thing we would have done differently is set out a better plan. Before starting our project, we had ideas and preferences as to how we would like our system to operate, and what technologies we would use, however we did not have a concrete and solid plan as to how we would accomplish some of these goals. We did end up achieving all of the goals and benchmarks which we set out to achieve, however if we had formed a more definite plan from the outset, we think we could have saved ourselves a lot of extra work and stress. For instance, we decided to add a tracking service quite late in our project, which meant that the integration and communication of our tracking service was causing us quite a lot of trouble at first. If we had set out from the beginning knowing that we would be implementing a tracking service and had planned for it, it would have been much easier to integrate the tracking functionalities, and would have caused much less interoperability issues with the rest of our system.

Another thing we would change if we started our project again is that we would set strict standards for Git commits from the outset. At times, our Git workflow was not as good or efficient as it could have been, however midway through our project we found our rhythm and used Git very effectively and cooperatively.

Technologies

We learned a lot about the technologies that we used throughout the process of this project, and below we will outline some of the major benefits and limitations that we found with each:

- REST:

REST proved to be a significantly valuable technology for our Distributed Security Management system. Once communication and coordination was successfully created between each microservice, REST allowed us to create a very complex and efficient communication infrastructure between each aspect of our distributed system.

The only limitation we found with REST was that the initial set-up and configuration of ports and URI's was quite tedious and relatively difficult, however once all systems were in place and all micro-services were successfully connected, REST proved to be a massively beneficial technology for our overall system.

- Docker:

Through this project, we learned that Dockerization and containerisation carries huge benefits when dealing with a system with many micro-services such as ours. Before we decided to use Docker, each microservice in our system had to be run separately, which led to us needing seven separate terminals in order to run our entire system. This was highly impractical, and made troubleshooting and debugging a much less time-efficient task than it could have been. Once we implemented containerisation using Docker, our system simply required the running of the container using a single terminal, and another terminal in which to run our client. This led to much faster troubleshooting and optimisation, and led to an overall much more useful and efficient application.

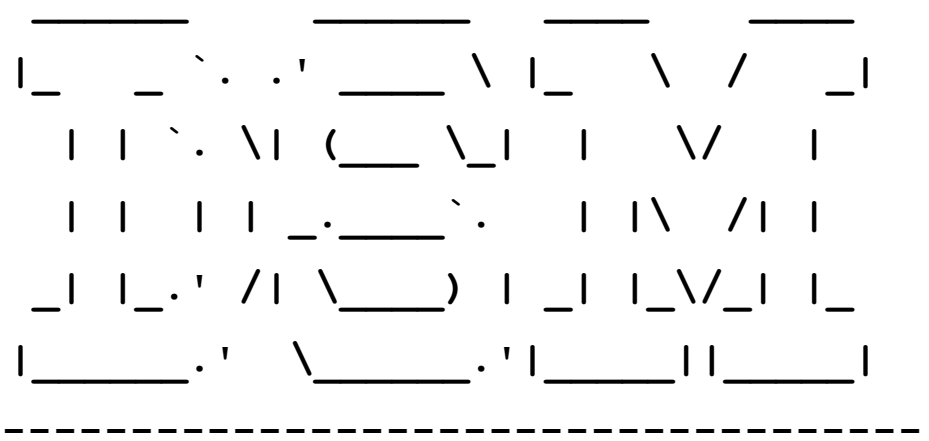
- Spring Boot:

Spring Boot is an extremely powerful and useful tool which is designed to allow developers to launch Java applications and microservices very easily. Spring Boot provided us with exactly that, and allowed us to create and launch each of our microservices in our application very easily and efficiently. It was extremely fast to set up, and provided us with all of the tools necessary to create our distributed system. Overall we did not find any limitations while using Spring Boot, and the technology provided exactly the services which we required.

- Maven:

Maven also proved extremely valuable for our project, and provided us with a simple project setup and easily defined dependencies. This allowed us to focus on some of the more vital aspects of our project, while Maven took care of the underlying build details. Overall Maven was extremely beneficial to our project and like Spring Boot, we found no major limitations worthy of reporting.

Overall, all of the technologies that we used for this project provided us with extremely valuable resources that allowed us to build Distributed Security Management. We feel as though we chose our technologies very well, and each was vital towards the end goal of a distributed, fault tolerant, scalable and reliable application.



Thank you for using Distributed Security Management!