COMP30220 Assignment 2

REST Assured Travel Planner

Quaranteam

Team Members:

Ee En Goh (17202691)
Tanmay Joshi (17204430)
Seán McLoughlin (10360547)
Barry Murphy (17726901)
Olanipekun Akintola (17734755)
Ciarán Conlon (15725581)

UCD School of Computer Science
University College Dublin

January 5, 2021

Table of Contents

Chapter 1: About this project

REST Assured Travel Planner is a distributed system program that uses RESTful services to allow a user to plan all aspects of a trip or holiday including flights, hotels and activities/sightseeing at the destination.

Here is a complete list of modules implemented for different services :

Module	Brief description
activities	Generate a list of activities available in a given location
attraction	Generate a list of tourist attractions available in a given location
clients	Connects to the travel agent
core	Reusable java bean object definitions for each system component
flights	Generate a list of flight quotes available for a destination within a specified time period
hotels	Finds available hotels in the destination city
travel agent	Processes client requests and creates travel packages by communicating with each service.

Chapter 2: API and Technology used

2.1 APIs

- Amadeus Activities API
- Amadeus Points of Interest API
- Amadeus Hotel Search API
- Nominatim Search API
- Skyscanner API

2.2 Technology

- Java Spring Boot Framework
- MongoDB
- Netflix Eureka
- Kubernetes

Chapter 3: Module details

Each module demonstrates a certain service available in this system.

3.1 Activities

Activities[1] is a service module in this program that integrates with Nominatim Search API and Amadeus API to get a list of activities available from a given geocode. The use of Nominatim Search API is to translate a location based on 2 parameters, city and country full name in string, into a Geocode object that consists both the latitude and longitude of the location found. The Amadeus API uses this geocode to find the activities available in that location. Most cities are supported by this activity search service, except cities in China and North Korea (proven with unit tests) The process of how this module works can be divided into 5 stages :

No.	Process	Description
1	Get city search query from user through request	Location queries, ie. the full name of both city and country in String are accepted through program UI, compiled in an ActivityRequest object and sent over to this service component through the travel agent. Done in client-side service.
2	Validate received query	Done with regular expression, to reject queries with unwanted characters or blank queries
3	Search for the geocode of the location queried	Finding the geocode of the location queries with Nominatim Search API
4	Use the geocode to find activities	Retrieve list of activities available with the geocode found in the previous step through Amadeus API
5	Object translation to make it transferable	Translate each Amadeus Java SDK Activity object into corresponding Java bean ActivityItem object (core)

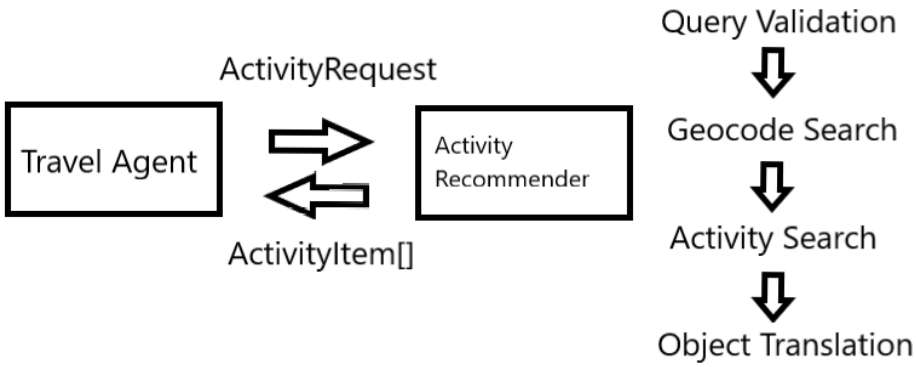

Figure 3.1: Activity request process

3.2 Attractions

Attractions[2] is a service module that works on the same principles as the Activities module. Instead of using the Amadeus Activities API, it uses the Amadeus PointsOfInterest API. The query sent by the travel agent contains geographic coordinates and a (default) radius to search from the coordinates. What is returned is a list of places such as monuments, historical sites, parks, beaches and pretty much anywhere that is free to enter. Each listing includes the type of attraction as well as a general category. The categories provided by the API to distinguish attractions are: Sights, Beach/Park, Historical, Nightlife, Restaurants and Shopping. There is a limitation with this API, that the free (testing) version only returns results for a small number of cities. Although this seriously limits the scope of usefulness for this module as a (hypothetical) commercial product, it was used because of the similarity to the other Amadeus API already in use and for the simple interface. The module still sufficiently serves to demonstrate relevant knowledge of implementation and API usage, despite the limitation. For the purposes of testing, all of the following cities have been verified to return results:

- Bangalore, India
- Barcelona, Spain
- Berlin, Germany
- Dallas, United States
- London, United Kingdom
- New York, United States
- Paris, France
- San Francisco, United States

As with the Activities API, the search results are converted to java bean objects, but of type "Attraction".

3.3 Clients

The client module reflects the interaction between a user and the application. A ClientRequest object is created in this module after the user enters all specific requirements in the User Interface (UI). This object is then sent to the Travel Package module. The returned Travel Package object is used in the ClientController to display the results that meet the user's requirements in the UI.

Knowledge acquired from last year's Web Development module was used in implementing the way that the user inputs details to the web pages and the result returned to the user. Most of the web pages were created using bootstrap, thymeleaf etc. Some of the web pages had to be designed to reflect the fact that this app was a product of distributed systems. The data entered in the UI were used in the communication between the client and travel agent through HTTP. The web pages that displayed the returned results from the various services also reflected the fact that the travel agent has pulled the requested information from various services.

The team implemented the Model – View – Controller design pattern. Initial work on the Controller ensured that whatever results were being returned by the main method in another version of Client class could be replicated using the Controller with another Client class (a version that had no main method) that worked with a UI. Some 505 errors were encountered which were resolved when it was traced back mostly to the services. Every UI page has input validation so that if a user enters invalid input, they are prompted to enter again. For example, in the Flights page, if a user enters a city/country that does not exist, they are shown an alert box stating that the city/country is invalid.

In the pages where the user is shown a list of flights/hotels/activities/attractions, if the user enters a choice that does not exist, or any invalid input, they are prompted to enter a valid input.

3.4 Core

The core module contains all the core components in this system. It includes the definition of Request object for individual services (eg. ActivityRequest, AttractionRequest and ClientRequest), the implementation of service result objects (such as Booking, Flight, Hotel) and other classes that are required throughout the entire system process. All of them are implemented by following the Java Bean conventions, which means they are reusable by every component in this distributed system.

3.5 Flights

Flights[3] is a service module that works on the same principle as the Activities module. This module uses the SkyScanner API. The travel agent sends a flight request object specifying the user requirements to the flight service. The use of the SkyScanner API is to find flights based on 8 parameters namely, departure city and country, destination city and country, and outbound and return dates, type of currency and whether it is one-way or return trip. A list of flight quotes is returned from which the Client must choose preferred flight. Each quote would consist of an airline, price, outbound/return date; and departing and destination cities and countries. In the case where no flights are returned by the API for the selected dates, the client is informed and the

booking is cancelled.

3.6 Hotels

Hotels[4] is a service module which uses the Amadeus API. The hotels service requires 5 parameters which are - city code (the user enters the name of the city and we convert it to the city code), arrival date, departure date, number of guests and minimum number of stars. The client gives us these parameters which are then sent to the travel agent, which directly communicates with Amadeus API to return a list of hotels found according to the user requirements. Each hotel quote returned has the following parameters - name, address, rating, price, description, phone number, amenities, room type and bed type. The client is then sent this list of hotels so that they can choose their preferred hotel and the choice is sent back to the travel agent which adds it to the booking. In the case where there are no hotels found for the dates requested, the client is informed about it and they are allowed to continue creating their travel package.

3.7 Travel Agent

The travel agent (TravelAgentService.java) is responsible for taking requests from the client (ClientRequest.java) and sending these requests to the respective services. The travel agent searches for flight services, hotel services, activity services and attractions services and then passes the requests on to each service. The services then send the relevant data back to the travel agent which then sends the available flights, hotels, activities and attractions for the location selected back to the client. Once the user has made their choices, they will be sent back to the travel agent who will store them via MongoDB.

Chapter 4: **Instructions to run this program**

Full instructions can be found in the root README.md file.

Chapter 5: Extra Technologies Used

5.1 Eureka

Eureka[5] allows for client-side discovery, allowing for services to find and discover each other without hard-coding ports and hostnames. Each service registers with the Eureka registry upon creation, and deregisters upon deletion. A service can query the Eureka registry for the address of a service by name, and if there are multiple instances of a service, Eureka chooses one to return to the client. In this way, multiple instances of a service can be registered simultaneously to a Eureka registry, allowing for fault tolerance should one or several fail or be unavailable. Eureka was chosen as it can be integrated with the Spring Boot Environment and - after successful development on a local machine - Kubernetes.

5.2 Kubernetes

Kubernetes[6][7] allows for the orchestration of containerised applications, in our case Docker images. In conjunction with Eureka, this allows for the creation of multiple instances of our flights, hotels, activities and attraction services. Combined with service discovery provided by Eureka, this allows a travel agent to find working services to query to build a travel package. Multiple Eureka registries offer fault tolerance should one become unavailable, and automatic redeployment means that container failure is a temporary problem that the other replicas can fill in for while the container is redeployed.

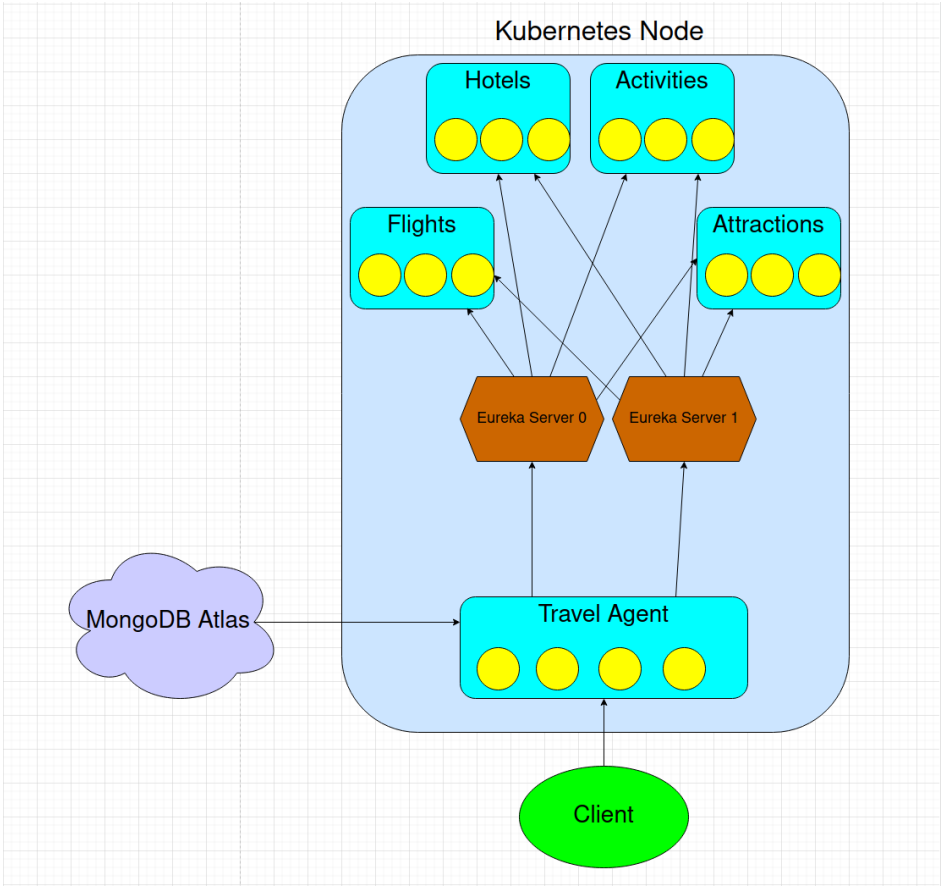

Figure 5.1: Kubernetes Process Diagram

5.3 MongoDB Atlas

Storage for the client booking is provided by the Mongo Atlas[8] cloud database. A NoSQL database was considered a good fit for the type of data being stored in a client booking, and by being cloud hosted it allowed us to more easily maintain a persistent database which the entire team could test their local implementations on and compare with other members when debugging.

Chapter 6: Reflections/Experiences with Extra Technologies

6.1 Eureka and Kubernetes

Eureka proved to be straightforward to implement on top of an existing Spring Boot framework. There was some initial difficulty getting Maven dependencies correctly configured and properly implementing the RestTemplates, but the lack of changes to the actual code of the classes that made use of it made it easy to implement across multiple services after the first one was correctly configured.

Running Eureka on Kubernetes was considerably more difficult however. The combination does not have much documentation online, as the preferred practice seems to be to use Kubernete's native DNS resolution offering instead. Combined with the learning curve of using Kubernetes for the first time, this caused a few headaches. In the end the creation of a Eureka stateful set solved the main problem of having Eureka registries in a Pod interfacing through a service. Through the Pod/Service route, only one server would be updated by any one client state change, which led to inconsistent registries. By giving the Eureka servers defined identities through a Stateful Set, updates were sent to both simultaneously, as well as allowing the registries to register each other too.

If I were to start again, I would consider developing with Kubernetes in mind immediately and skip Eureka altogether, but as it stands Eureka was probably the best way to go to allow for the learning curve of orchestration to be more spread out. I would also implement a better alternative to NodePorts - such as an Ingress - which the client uses to access the travel agent service, as in production it would be considered a security risk.

6.2 Travel Agent

The current implementation only uses the first POST request in each service (there are two). The POST request in FlightService returns the available flights and the POST request in HotelService returns the available hotels etc. This data is then stored in the travel agent so that when the user makes their choices (via the second POST request in TravelAgentService), the data is retrieved and stored in MongoDB. However, the implementation that we were aiming for was a more sophisticated solution. Once the user has made their choice, then another POST request would be sent to each service and the service itself would then store the data in their own database. Ultimately, we did not have the time to complete this version.

We were also planning on making use of the PUT request so that the user could alter their booking if they wished to do so. The implementation for PUT is completed for the travel agent and all of the services, and this implementation also makes use of the GET requests that can be found in each service. Unfortunately, we did not have the time to implement this functionality on the client side. If you take a look at our solution, you will see that the code is all there but is regrettably

not being made use of.

6.3 Attractions API

For the Attractions module, an alternative API was considered, OpenTripMap. It had no limitation to the number of cities it could return results from and had 5000 free queries per day as opposed to the 1000 provided by Amadeus. Ultimately, it was decided that the simpler interface and similarity to the Activities implementation would serve better in our timeframe. If this was an actual commercial product being launched, it would have then been more advantageous to use OpenTripMap.

Bibliography

1. Amadeus. Amadeus Activities API: https://developers.amadeus.com/self-service/category/destination-content/api-doc/tours-and-activities/api-reference.
2. Amadeus. Amadeus Points of Interest API: https://developers.amadeus.com/self-service/category/destination-content/api-doc/points-of-interest/api-reference.
3. Skyscanner. Skyscanner API: https://www.partners.skyscanner.net/affiliates/travel-apis.
4. Amadeus. Amadeus Hotel Search API: https://developers.amadeus.com/self-service/category/hotel/api-doc/hotel-search/api-reference.
5. Netflix. Eureka API: https://github.com/Netflix/eureka.
6. Kubernetes. Install and Set Up kubectl: https://kubernetes.io/docs/tasks/tools/install-kubectl/.
7. Kubernetes. Install and Set Up minikube: https://minikube.sigs.k8s.io/docs/.
8. MongoDB. MongoDB Atlas: https://www.mongodb.com/cloud/atlas.