SKYSCANNER

IN COLLABORATION WITH

UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)

FINAL DEGREE PROJECT

Skyscanner offer and demand comparison

Skyscanner's data comparison

Author:
Fèlix Arribas

Director:
Javier Arias
University supervisor:
Maria José Casany

A Project for the Computer Engineering Degree *in the*
Software Engineering and Information Systems department
Facultat d'Informàtica de Barcelona (FIB)
working with
DeLorean squad *from* Marketplace Engine tribe

Saturday 2nd June, 2018

Universitat Politècnica de Catalunya (UPC)

Abstract

Facultat d'Informàtica de Barcelona (FIB)
Software Engineering and Information Systems department

Computer Engineering Degree

Skyscanner offer and demand comparison

by Fèlix Arribas

In the last century, the world has became smaller. Communications are easier and faster than fifty years ago. Back then, you could talk through a fix phone, but you were not able to send any kind of media, like photos, videos, etc. Only the latest technology of that moment was able to do that. Since the smart phone revolution in 2007 almost everyone can text messages, sending images, share live videos or almost whatever you can imagine in less than a second.

But the internet, phones and communications are not the only thing that made the world smaller. Ways of traveling helped to this earth flattering too. In 1918 visiting another place was very difficult. If you wanted to go through the sea, you had to do it by boat. The fastest way to travel very far in a continent was by train, but not all places were connected with rails. Nowadays, all along with the internet revolution, anyone can travel to the other side of the world in less than a day by plane. Even for traveling inside the same country people use planes.

But, is the air industry as efficient enough? Are all airlines users satisfied with their purchases and possibilities? Skyscanner provides an easy to use tool to search cheap flights from any airport to another. Sadly, sometimes is difficult for users to find what they really want.

This project wants to help solving this problem, providing a HeatMap to explore differences and similarities between what users search and what airlines provides. Being able to compare between specific dates to guess user behavior.

Contents

Chapter 1

Context

This is a project developed in *Skyscanner* and evaluated by the *Universitat Politècnica de Catalunya (UPC)* as a Final Degree Project.

This project's purpose is to compare **user demand** and **flights provided** by airlines for routes. This comparison could improve flights advertisement according to user demand. The company could also develop complex software using the huge amount of data it will compare through an Application Programming Interface[1].

Skyscanner have more than 75 million flights information and about 13 million users queries every day. In order to compare all the data available and get significant results, the software should solve **Big Data**.

1.1 Skyscanner

Skyscanner[2] is a travel fare aggregate website. It was formed in 2004 when a group of people was frustrated by the difficulties of finding cheap flights.

In 12 years has evolved from a little office in the suburbs of Edinburgh to a world wide company with ten offices in seven different countries. In the next 5 years, Skyscanner wants to become the travel experience that people prefer to the myriad confusing and unconnected travel apps.

Now, is one of the top travel fare aggregate website. It has more than 4 million visitors every day and, more or less, a revenue of half a million pounds per day.

Before joining Skyscanner I wondered how they growth that fast and how they did this amount of money. Usually, *if you do not pay for the product, you are the product*, so my first thoughts were that Skyscanner sell user searches, and travel tendencies to airline companies. **Companies know what travelers buy, but now what they have searched before their final purchase.**

Once inside the company, I realized that it is not the way Skyscanner make money. Knowing that, when I joined DeLorean squad in Barcelona and had the opportunity to make the final degree project there, I proposed a tool to get traveler tendencies and compare them to DeLorean squad's data, timetables and flights.

1.2 DeLorean squad

DeLorean[3] is a squad of Marketplace Engine tribe[1], its mission is to provide the best data and services around the routes, timetables and modes of transportation to go from one point on Earth to another.

1.3 Marketplace Engine tribe

This tribe[4] is one of the most important tribes in Skyscanner, its mission is to provide the most comprehensive and accurate flight inventory for Skyscanner and her partners with minimum latency.

Its main goal is to evolve the search, pricing, routes and browse services to be horizontally scalable and set us up to build a lightning fast, super accurate and fully comprehensive flight search engine, enabling the traveler to instantly find the best flight at the best price with minimum effort.

[1] Learn more about *Skyscanner structure* in Appendix A on page 40

Chapter 2

State-of-the-art

Since this project is not oriented for Skyscanner users but the company itself, the *State-of-the-art* relates to services inside Skyscanner. Even so, a brief explanation about other metasearch engines would help to find the gap this project is developed.

2.1 Fare aggregators and metasearch engines

2.1.1 Google Flights

In the last years Google Flights has became the main competitor of Skyscanner. The new version is very fast and has a complete new interface, following Material Design guidelines.

Google is one of the top tech companies worldwide and has a lot of different platforms. It is a competitor to be aware of, the integration with Gmail, Google Calendar and Android OS makes Google Flight a part of Google's ecosystem. The traveler may feel comfortable.

2.1.2 Kayak

Kayak has always been the main competitor, both companies started in 2004. Unlike Skyscanner, Kayak started with Flights, Hotels and Car hiring. Skyscanner added those two extra search engines between 2013 and 2014.

2.1.3 Expedia

Launched in November 1998, is one of the oldest fare aggregator and metasearch engine. Apart of its own website, is also a Skyscanner provider. Some of the prices are taken from Expedia and sometimes the user is redirected to their website to finish their purchase.

2.2 Skyscanner services

In Skyscanner the user has never been a product, in fact, one of the statements of Skyscanner's culture says *Traveler != Product*[5].

There has never been a project getting value from user information because it does not follows the company culture, so the definition of the problem and the scope of the project must be very accurate to ensure it is fulfilling with Skyscanner's strategy[2].

2.2.1 Marketplace Engine

This tribe is formed by five squads, those constantly work to improve the routes and pricing service all along with an efficient search.

Marketplace Engine works with data *from the provider to the user*. In other words, it just serves **information to the user** but does not get any from him/her. All five squads take all the **data from providers**.

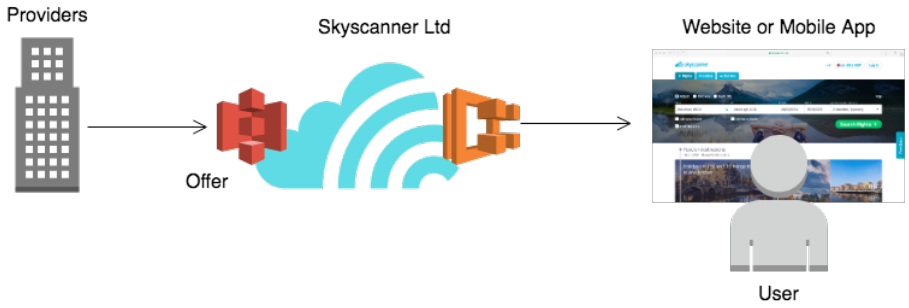

FIGURE 2.1: Simple explanation of Marketplace Engine data flow.

2.2.2 Data tribe

In the other hand, Data tribe has a lot of squads with services used to collect **data from user's activity**. The flow of the information is *from the user to Skyscanner*.

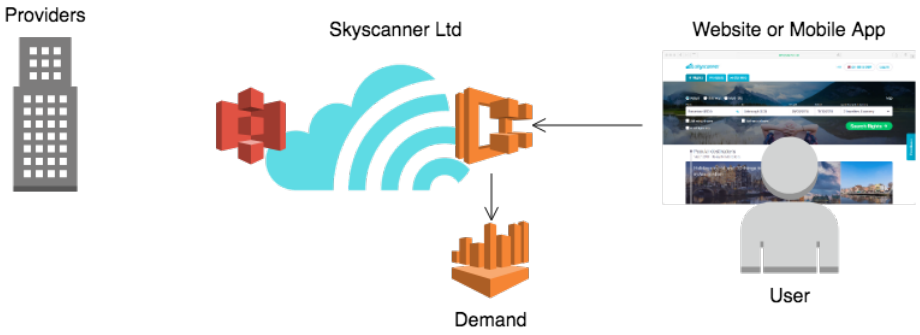

FIGURE 2.2: Simple explanation of Data Tribe data flow.

2.2.3 The gap

There is no tribe or squad that works with both **data sources**: Providers and Users. And here is where the will be.

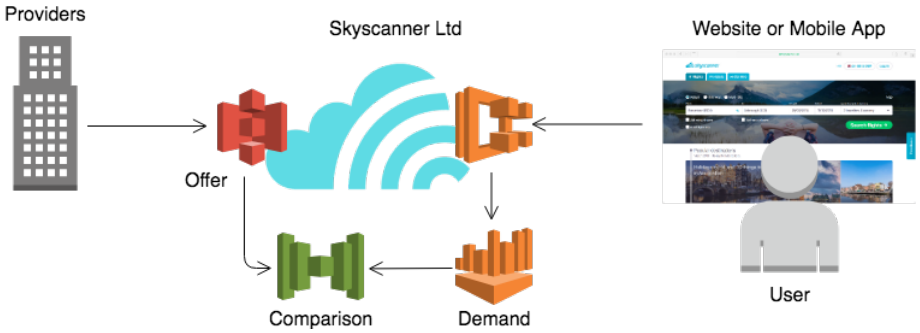

FIGURE 2.3: First approach of the Skyscanner offer and demand comparisondata flow.

2.3 Resources

Skyscanner is already in the air, it is working and has been working for more than ten years. In the past two years it has been migrating all their services to Amazon Web Services[6].

Amazon Web Services have a lot of available different services in it for different purposes. The services that will be used or important for this projects are[1]:

- Compute
  - Lambda
  - Batch
  - ECS: Elastic Container Service
- Storage and Database
  - S3: Simple Storage Service
  - DynamoDB: Dynamic Database
  - RDS: Relational Database Service
- Analytics
  - Athena
  - EMR: Elastic MapReduce
  - Data Pipeline
- Others
  - CloudFormation, Management Tools
  - API Gateway, Networking & Content Delivery
  - Simple Notification Service, Application Integration

[1] All the resources are well defined in chapter Design on page 27

Chapter 3

Offer and demand comparison

Knowing the context inside Skyscanner and all the state of the art of this project, the definition and scope is well defined.

3.1 Formulation of the problem

The main goal of this project is creating a tool for *Skyscanner* to ease the routes comparison with different parameters, taking into account values like **user searches** and **flights available** by airlines.

In any team of Skyscanner, user queries and providers data is compared in order to guess valuable trends.

Found that gap, a bunch of new ideas appeared. After some talks with product owners of different squads and some senior engineers, a promising idea showed up:

Comparison of **user searches** and **flights available** by airlines, enabling finding *over-requested* routes or airports. Those routes or airports with more user demand than availability by the providers.

DeLorean squad manages a huge amount of data: All flights planned for the next year and a half, these are more than 75 million records. The database of all user queries in the website or mobile application is even bigger, there are 4 million visitors per day and a total of 13 million queries per day. Not much more information needed to say that this is **Big Data** problem.

With DeLorean squad's product owner help, we found some use cases for the processing of those 75 million routes and all user session's queries to get some significant results:

Provide a **visual tool** to find routes with much **more demand than offer** and be able to observe the **evolution** of it through time:

- A route with a lot of demand, but not enough offer to cover it, will be **over-requested**.
- A route with much more offer, but not that amount of demand, will be **non-profitable**.

3.2 Scope

Merging both data sources (providers and users) generates a lot of new valuable data with a lot of different application: From simply selling it to providers, to complex deep learning systems.

The final goal of this project is displaying the comparison in a simple Web UI for Marketing squads or tribes. This can be split in three smaller goals or components:

3.2.1 Pipeline

Distributed application that maps and merge all the data from both sources in its given format to the required data model.

The pipeline reads from Marketplace Engine and Data tribe services. Then, it maps the provider and user data to the desired data model. The new entities are stored in a database where the service will read from.

The application will be split in two sub applications, one for providers data and other for users. So both of them can vary independently without depending on each others' sources and changes may have in the future.

3.2.2 Service

Simple HTTP Service with a basic Application Programming Interface to serve Pipeline's results. The service will have an internal endpoint only available for other Skyscanner applications or developers.

3.2.3 Visual representation

Website with a visual representation of the data. There are plenty of ways to draw charts and maps visualizations. The Web UI will be composed by two pages:

Browser

Simple browser with four input text fields, one for the origin airport, the second for the destination and the last two for the date, month and year.

Once the inputs are set, the user will be able to click the *Search* button and move to the next page.

Chart visualization

Simple chart with the comparison between providers offer and user demand of the selected entity through time.

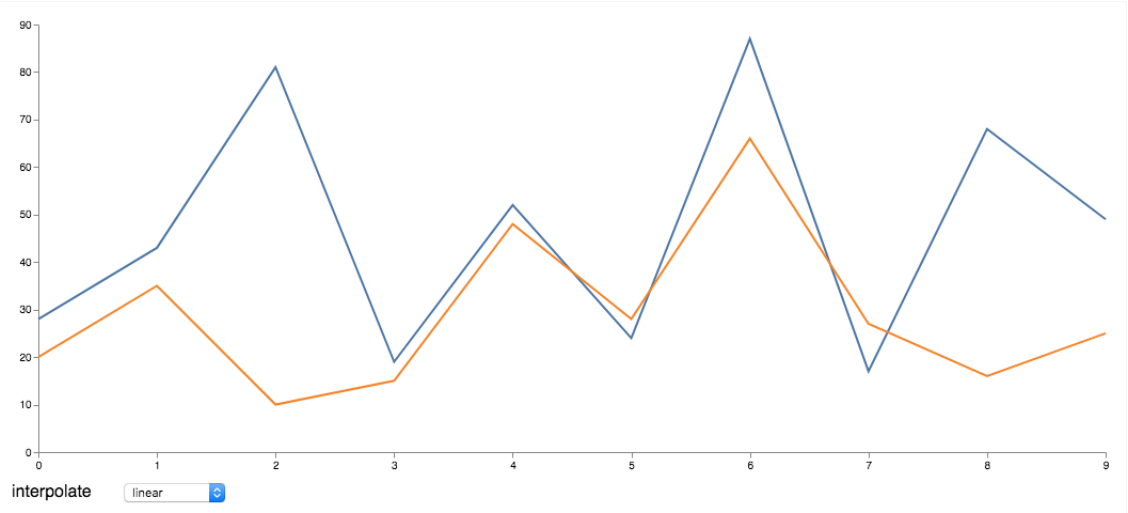

FIGURE 3.1: Chart mock-up. One color goes for Providers Offer and the other one for User Demand.

3.2.4 Not list

It is also important to define what this project will **not** be.

- **Prices** or **quotes**: In any moment will check for flight prices or quotes.
- **Carriers**, **cities** and **countries**: The comparison will be only available between routes and airports, not airlines (carriers), cities nor countries.
- **Create**, **update** or **delete** data through the **Server**: The only input will come from the pipeline. Entities are never deleted or modified in order to keep historical data.
- **Create**, **update** or **delete** data through the **Web UI**: The only input will come from the pipeline. Entities are never deleted or modified in order to keep historical data.

3.3 Risks

There are several risks can appear while developing the project. Most risks appear because of the dependencies with other tribes and squads and dependencies with other services. In the other hand, all performance risks of the Pipeline can be ignored because Skyscanner's hardware is enough for big applications, like this one.

3.3.1 Routes contract

DeLorean squad's routes service is under development and during the Skyscanner offer and demand comparison development the routes data model may change a little bit. For example, the origin and destination recently changed: In December 2017 the service was giving an *Airport ID*, but now is giving an Airport object with more parameters like IATA Code[7], Country ID, City ID, etc.
Timetables are served, basically, in three different models: Unified Routes, Single Flight Number Routes and Summary Routes. Both Single Flight Number and Summary Routes stores timetables by days of week[1]. The current **Unified Model** is supposed to be the final one, has been studied and refined by experts on the domain and DeLorean squad

[1] Explanation of different models in section Routes model on page 19

has been adapting the output until reaching this final model. Even so, it is impossible to guess if there will be some changes.

3.3.2 Users information

In the website and mobile application, the user have plenty of different ways to search the perfect flight. The most common one is by origin, destination and date, but he/she can also search by month or by destination. Then, the user is not searching flights by route and date. It sets the period of time he/she can travel and Skyscanner offers cheap destinations.

Even searching by origin, destination and date, there can be invalid search queries. There are a lot of **cities with more than one airport** and usually when travelers search for traveling from there or to that city, they set the origin or destination **city**, but no the airport. For example: If you want to travel from New York to London, you have a lot of airports in New York (JFK, LGA, BUF, ROK, ALB, etc.), which provably only three will be useful for that search. And a total of six airports in London (LCY, LHR, LGW, LTN, STN, SEN). The final purchase, if there is, will be from two **airports**, but the Skyscanner offer and demand comparison looks for user intentions, not for final purchases.

3.3.3 Amount of data

As explained before in the Formulation of the problem, there is a very big amount of data that need to be mapped. Luckily, Skyscanner have great cloud machines and *unlimited* space[2]. Anyway, still an issue to be aware of.

The data processing will not be an actual issue, other teams already process the same amount or more data than the Skyscanner offer and demand comparison will do. This project's data source is already processed data from its original sources. DeLorean squad gets much more data from OAG, filters it, aggregates it with other inside company sources and writes the whole dump. Data tribe processes all user queries, flights, hotels and car hires.

If in some point this became an issue I can ask for help to DeLorean squad members or data scientist.

3.3.4 Kind of data

User searches and flights offer have very similar parameters: origin, destination, date, etc. But can be very different. There are some facts that can give very different numbers for the same route and date, for example, seats in a plane. The flight offer pipeline counts number of flights, not of available seats. Is not the same one flight from Barcelona to Sydney that the aircraft will provably have 500 seats, than one flight Barcelona Mallorca that the aircraft will be much smaller and will barely reach 100 seats.

3.3.5 Web UI

Creating the interactive map and plots for the proposed website from zero, is a whole project itself. In order to avoid failing in the *Visual representation* goal, the best option is

[2]Read more in section Resources

to use reliable libraries, like Vega[8]. Combining it with React.js Skyscanner components, the Web UI development will be fast and easy.

3.4 Methodology and rigor

3.4.1 Extreme Programming

This project will be developed along with DeLorean squad's work. This squad is following Scrum, an agile methodology. After some research and discussions with the rest of the team, Extreme Programming (XP)[9] showed up as the best option.

Extreme Programming is a style of software development focused in excellent applications, programming techniques and clear communication. To accomplish that, XP includes a philosophy of software development, body of practices, complementary principles and a community that shares these values.

This methodology works with short development cycles, resulting in early, concrete and continuing feedback. Has an incremental approach, making way to a flexible schedule for the implementation. It relies in oral communication and tests to reach the goal of the project.

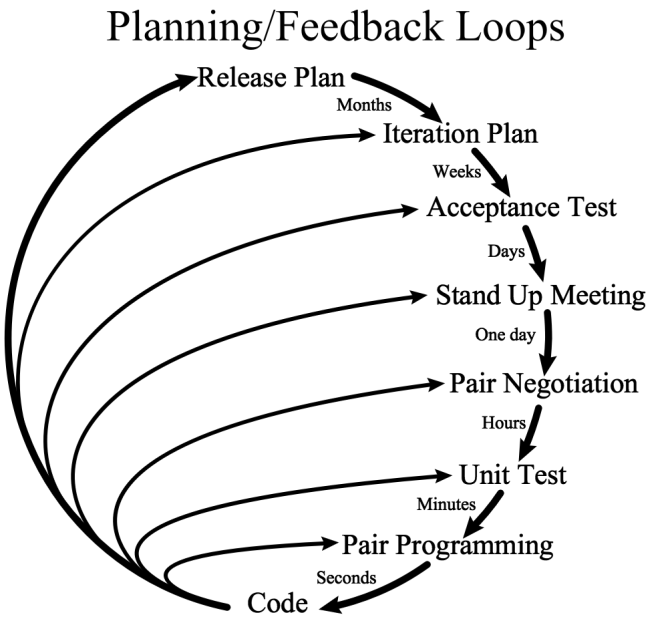

FIGURE 3.2: Extreme programming planning loops.

The original Extreme Programming methodology is for teams of developers, but this project will be only developed by myself, so the original idea has been modified a little bit. The pair programming and pair negotiation has been removed because I have nobody to pair with. In order to have some feedback and update the requirements properly with the supervisor approval, I will be attending all DeLorean squad stand up meetings and explaining my progress and, if necessary, create meetings with the product owner and the rest of the squad to take the project on the right track.

3.4.2 `git`

GitLab will be the main tool for version control of the source code and issue tracking of different tasks. GitLab[10] is very similar than the well known Bitbucket[11] and Github[12], all these tools are suitable for the development of the project, but Skyscanner uses only GitLab.

All code projects (pipelines, service and website) will be stored in a project space in Skyscanner's GitLab domain called Heatmap[3]. Using `git`, the versions will be forked in branches. Branches allow fork code versions and then merge them all together in a master single branch with a better conflict control. Each branch will stand for an specific issue, `master` will be the main branch where the latest production version will be.

- **`master`** branch will contain the accepted version of the project, this version must work in production with no error and must be tested in its development of task branch
- **Development branches** (named following the style `ISSUE_NUMBER-ISSUE_TITLE`) contains all development commits and code for a given task planned previously in the iteration plan. Those branches are tested in Sandbox environment
- **Bug fixing branches** (named `HOTFIX-BUG_NAME`) are branches created for fixing bug in production. The idea is to apply a fast solution in the branch and merge it directly to `master`, so the bug is fixed as fast as possible, then create an issue to find a better solution with more time.

3.4.3 Continuous delivery

In order to deploy pipelines and services automatically when merging to `master`, the projects have integration with Drone[13]. Every time it commit to a branch, the application is build, tested and, if everything goes well, it deploys to Sandbox. When merging to master, Drone automatically deploys to Prod.

3.4.4 Tasks and issues

The **issue tracker** provided by GitLab[10] will be very helpful in order to monitor the evolution of the project. Issues will be composed by a title, description, milestone, labels (if needed), due date and weight, and represents a new functionality. In order to know the status, issues will be listed in three columns:

- **Backlog:** Known tasks that haven been started yet. Could be a well defined task, with a very clear description, a due date and weight, or just a draft with empty fields.
- **WIP:** Work in Progress. The task is being considered, developed or tested.
- **Done:** Tasks finished, tested and working fine. Ready for merging into `master` and deploy to production.

3.4.5 Environments

In Skyscanner, for almost all Amazon Web Services resources, there are two different environments: Sandbox and Production, also known as *prod*.

[3] Heatmap was the original name of this project, the name remains the same since then

Sandbox

In this environment, every developer is allowed to do whatever they want, they can create, edit and remove every resource, execute whatever program, query, etc. For this project, all resources but user demand data source is available in sandbox.

Every three months, Cloud Operations squad cleans all Sandbox resources.

Prod

In prod things work little different, first of all, in order to get a profile you need a project, without project you cannot create anything. Once you have the project's profile, it have no permission to do anything, this is different than Sandbox as well. In order to get permissions you have to raise a ticket to Cloud Operations squad.

Chapter 4

Requirements analysis

4.1 Stakeholders

Initially it seemed difficult to find stakeholders and actors in these project apart from the providers. It is not a tool for the user of Skyscanner.

After talking with the squad Lead and then the Product Owner of DeLorean squad a lot of stakeholders appeared: DeLorean squad, Marketing Automation squad, Data tribe, etc. Each of these stakeholders has different use cases and the project became very interesting for a considerable part of Skyscanner.

4.1.1 DeLorean squad

DeLorean's Single Flight Number service, also known as *Timetable SFN Service*, provides all the **current** flights. This is a little bit of a problem when trying to get historical data because Timetable SFN Service does not provide past flights information, it is always **up-to-date**. In order to get this data it is needed to go one step back in the whole DeLorean data processing: *Timetable Pipeline*.

The Skyscanner offer and demand comparison must look old versions of the file created by the *Timetable Pipeline* to get older routes. Then, DeLorean squad is interested in the Skyscanner offer and demand comparison because it will be using Pipeline's data.

Product Owner

Jen Agerton is the Product Owner of DeLorean squad. She realized that the Skyscanner offer and demand comparison is very useful for other squads like Marketing Automation squad and providers (air carrier companies). Apart from being the DeLorean squad product owner, she is also in charge of the negotiation with providers and airlines. The comparison of offer and demand is very useful to find user trends and can help airlines when scheduling flights.

DeLorean's squad Lead

Francisco López and I had the initial idea for this project. He saw an opportunity for the future (after project's delivery) orienting the Skyscanner offer and demand comparison for Machine Learning purpose: The information that the comparison stores is very useful for constructed routes.

Looking at the evolution of timetables and user demand, DeLorean squad could make some assumption when combining routes to create constructed routes.

4.1.2 Marketing Automation squad

Marketing Automation squad enables scalable growth by automating workflows, and the collection of insightful data. They have three main goals:

- Provide data to support decision making
- Automated, data driven campaign management
- Budget process automation

The Skyscanner offer and demand comparison will be very useful for the first goal. The data provided by the comparison has high value in marketing decisions. Looking at historical data, Marketing Automation squad could post an advertisement about trend routes in a specific time of the year.

4.1.3 Data tribe

In Data tribe, State Machine squad captures the user actions, so they know where the user gets stuck or if they finally reach the provider of the flight. Other squads like Clan A and Clan B just gets users queries in flights, hotels and car hiring. The second data source of the Skyscanner offer and demand comparison (user requests and queries) will be obtained from these squads.

4.1.4 Other Skyscanner developers

Last but not least, a new service will appear in Skyscanner, all developers will be able to use it and build software using the Skyscanner offer and demand comparison's compared data. For instance, it can be used as a training for a complex Machine Learning[14].

The server Application Program Interface, used by the Web UI to visualize all data, will be public inside Skyscanner. This and all the documentation will be very helpful for developers.

4.1.5 OAG

OAG is a company that collects all logistic flights information. DeLorean squad reads data from them, it is the main provider of information regarding routes. They are the world's largest network of air travel data to provide accurate, timely, actionable digital information and applications to the world's airlines, airports, government agencies and travel-related service companies[15].

4.1.6 Providers

In the future, providers could take profit from Skyscanner offer and demand comparison. Companies will be able to know which of their routes or airports work better with user tendencies, they will be able to improve the flights service and make it more efficient, reducing number of flights in *non-profitable* routes. They will also know which are the best places to invest looking at *over-requested* airports.

4.1.7 Traveler

Skyscannerusers are one of the main sources of information. Without them, the comparison cannot be made. The results of the comparison can also help them, not directly, but if providers somehow manage flights and routes following the Skyscanner offer and demand comparison results, the traveler experience will improve.

4.2 Functional requirements

Skyscanner offer and demand comparison has a lot of functional requirements, features or functionalities that define this software usage.

1. **Search offer values by origin, destination, month and year**: Let the user of the Skyscanner offer and demand comparison search flights offer or availability evolution by date, for a given route (origin and destination), month and year of the flight.

2. **Search demand values by origin, destination, month and year**: Same as feature #1, but for user demand instead of flights offer or availability.

3. **Search multiple offer values**: Be able to search and show multiple offer values for different flight in the same chart, easing the comparison between both queries. For example: *Route A-B in August 2018 shows more availability than route A-C in August 2018 from January to March, but A-C has more availability than A-B from April until today.*

4. **Search multiple demand values**: Be able to query and display multiple user demand values for different routes in the same chart, easing the comparison between both queries. For example: *Route A-B in August 2017 shows more demand than route A-B in December 2017 from January to June, but A-B in December has more availability than in August from July until December.*

5. **Search multiple mixed values**: Enable comparison between offer and demand as well. Search and display the comparison in the chart.

6. **Add new query to chart**: Search for offer or demand (features #1 or #2) and display the result in the chart.

7. **Remove query from chart**: Remove data from the chart, stop displaying an specific query's data.

8. **Historical data API**: Enable an endpoint where developers can get historical data for a given route, month, year and source (Flights offer/availability and User demand/searches).

4.3 Non-functional requirements

Apart from the features, the Skyscanner offer and demand comparison also need to satisfy other requirements in terms of usability, latency, precision and more.

1. **Usability**: The final user will be focused on the data that the Skyscanner offer and demand comparison will serve, so the user cannot be distracted or annoyed by the website usage.

2. **Time availability**: The data should be available at any time, 24 hours per day, 7 days a week. The user can check it at anytime.

3. **Usage availability**: This tool must be only available for Skyscanner employees, for now; and providers that are interested and Skyscanner agreed to give them access. That is why the endpoint will only be accessible from Skyscanner VPN[16].

4. **Speed and latency**: The application will have a lot of data to show, but the the charts to load cannot take much time or the comparisons will be slow, maybe annoying and could confuse the user.

5. **Reliability**: A loss in historical data can be catastrophic, users could make wrong decisions because of not reliable data.

6. **Scalability**: Every day, the amount of data of the application increases by one million records at least. The system must be able to process all this data, store it and serve it with no problem and with any notable difference from the first day, until the last one.

7. **Maintainability**: The code can adapt to changes easily, user searches and flights models may change in the future, the code must adapt to these changes.

4.4 User stories

Instead of use cases, the requirements will be defined by user stories. Usually use cases are more specific, but user stories often work better for agile development.

Since a use case is a description of interactions between one or more actors and the application, a user story focus in what the customer will actually do with the system.

User stories will follow the following format: *As an [actor] I want [action] so that [achievement]*. Stories will also have an acceptance criteria, a list of conditions that must be fully satisfied in order to the story be completed.

Story #1

> As a Marketing Automation squad member I want to compare offer of two or more routes in a specific month so that I can guess which is the most common route to travel with airlines.

Acceptance criteria

1. User can execute at least two queries in a Web UI
2. The website displays a line chart with the amount of flights available by date
3. The chars shows a clear difference between different queries
4. The user can remove or add new routes

Story #2

> As a Marketing Automation squad member I want to compare demand of two or more routes in a specific month so that I can guess which is the most desired route by travelers.

Acceptance criteria

1. User can execute at least two queries in a Web UI
2. The website displays a line chart with the amount of searches by date
3. The chars shows a clear difference between different queries
4. The user can remove or add new routes

Story #3

> As a Marketing Automation squad member I want to compare offer and demand of one route in a specific month so that I can see if it is over-requested or is non-profitable.

Acceptance criteria

1. User can execute two queries in a Web UI
2. The website displays a line chart with different results
3. The chars shows a clear difference between offer query and demand query

Story #4

> As a Skyscanner developer I want to get a big amount of historical data from flights availability so that I can develop complex Deep Learning software.

Acceptance criteria

1. The developer has an API endpoint
2. The data structure is simple
3. The data has some well-known format

Story #5

> As a Skyscanner developer I want to get a big amount of historical data from user searches so that I can develop complex Deep Learning software.

Acceptance criteria

1. The developer has an API endpoint
2. The data structure is simple
3. The data has some well-known format

Story #6

> As a airline provider of Skyscanner I want to see the evolution of demand so that I can schedule flights properly depending on user demand.

Acceptance criteria

1. The website shows data readable for humans
2. The data is displayed by date
3. The user knows what he/she is seeing

Chapter 5

Specification

As explained before, the Skyscanner offer and demand comparison is a big data problem, but in order to this amount of data be useful and make actual sense, the data model must be correct and very well defined. If there is some conceptual mistake the data displayed in the charts will not be reliable, so it will not be satisfying non-functional requirement of reliability (#5).

The data model must be also subject to change and pretty clear, to fulfill non-functional requirement of Maintainability (#7).

5.1 Routes model

DeLorean squad's pipeline, writes the data in **Single Flight Number** model. This model is not useful for this project because of the Single Flight Number Timetable Date set fields schema. The timetable pipeline can also write following the **unified timetable** schema, this model presents the final master set of single flight number timetables that have been merged and de-duplicated from multiple sources. Dates schema are compatible with the Skyscanner offer and demand comparison purpose.

5.1.1 Single Flight Number Object Model

The **Single Flight Number** (SFN) timetables data schema is designed with the following object model.

SFN Route

Represents a route between 2 airports and the SFNs timetable that are available on that route. It is composed by three fields:

Field	Description	Data Type	Required	Key
Origin	The origin airport or station for the timetable	Integer	Yes	Yes
Destination	The destination airport or station for the timetable	Integer	Yes	Yes
Series	The SFN timetables associated with the route	List of SFN Timetable Series	Yes	No

TABLE 5.1: Single Flight Number Route fields

SFN Timetable Series

Represents a SFN timetable for the year ahead.

Field	Description	Data Type	Required	Key
Marketing Carrier	The ID of the carrier that is marketing the flight	Integer	Yes	Yes
Marketing Carrier Flight Number	The flight code assigned by the marketing carrier	Integer	Yes	Yes
Series Items	1..n Series Items	List of SFN Series Item	Yes	No

TABLE 5.2: Single Flight Number Timetable Series fields

SFN Timetable Series Item

A Timetable Series Item represents a specific configurations of operating carrier, stops and times of day for a given SFN timetable. For example for a given Marketing Carrier / Flight Number, the Operating Carrier or times of day may change through the year.

Field	Description	Data Type	Required	Key
Operating Carrier	The ID of the administrating operating Carrier	Integer	Yes	Yes
Operating Carrier Flight Number	The flight code assigned by the administrating operating carrier	Integer	No	No
Physical Operating Carrier	The ID of the 'physical' operating Carrier	Integer	Yes	Yes
Stop Count	The total number of stops (can be derived from the list of stops)	Integer	Yes	No
Stops	A list of via airports	List of Integer	No	Yes
Departure time	The time of departure in the current time zone of the origin station (minutes after 00:00)	Integer	Yes	Yes
Arrival time	The time of arrival in the current time zone of the destination station (minutes after 00:00)	Integer	Yes	Yes
Arrival day offset	The number of days after the departure date that the flight arrives	Integer	Yes	Yes
Duration	The total flight duration (can be derived from the departure and arrival times and day offset)	Integer	Yes	No
Traffic Restrictions	Restrictions relating to how a flight can be sold	Traffic Restrictions instance	Yes	Yes
DateSets	The dates that this Timetable Series Item is flown	Array of DateSet	Yes	No

TABLE 5.3: Single Flight Number Timetable Series Item fields

Traffic restrictions

A set of flags indicating restrictions relating to how an timetabled flight might be sold. Not relevant for this project.

SFN Date Set

A Date Set represents a start date and day of week pattern for a given Timetable Series Item.

Field	Description	Data Type	Required
Start Date	The start date that this set is applicable from	String (date format YYYY-MM-DD)	Yes
Data Sources	The source(s) of the timetable data	Array of sources (Enun)	Yes
Availability	A set of offsets indicating the dates that the associated series is available (up to 13 months ahead)	Array of week days (Enum)	Yes

TABLE 5.4: Single Flight Number Timetable Date set fields

Since the dates are represented by week days as an enumeration (*mon*, *tue*, *wed*, ...), the grouping of flights by **day** or **month**[1] becomes very difficult and non trivial. That is why the Skyscanner offer and demand comparison uses the **Unified** model.

5.1.2 Unified Timetable Object Model

In the timetable pipeline owned by DeLorean squad first maps all data to Unified model then it is mapped to the final Single Flight Number Timetable model. Unified model is very similar than Single Flight Number Object Model, it uses **Airport object** instead of an integer and a set of strings in date format `YYYY-MM-DD` in date sets, availability:

[1]The model changes and the pipelines groups by month. Learn why here: S3 and dates at month level on page 34

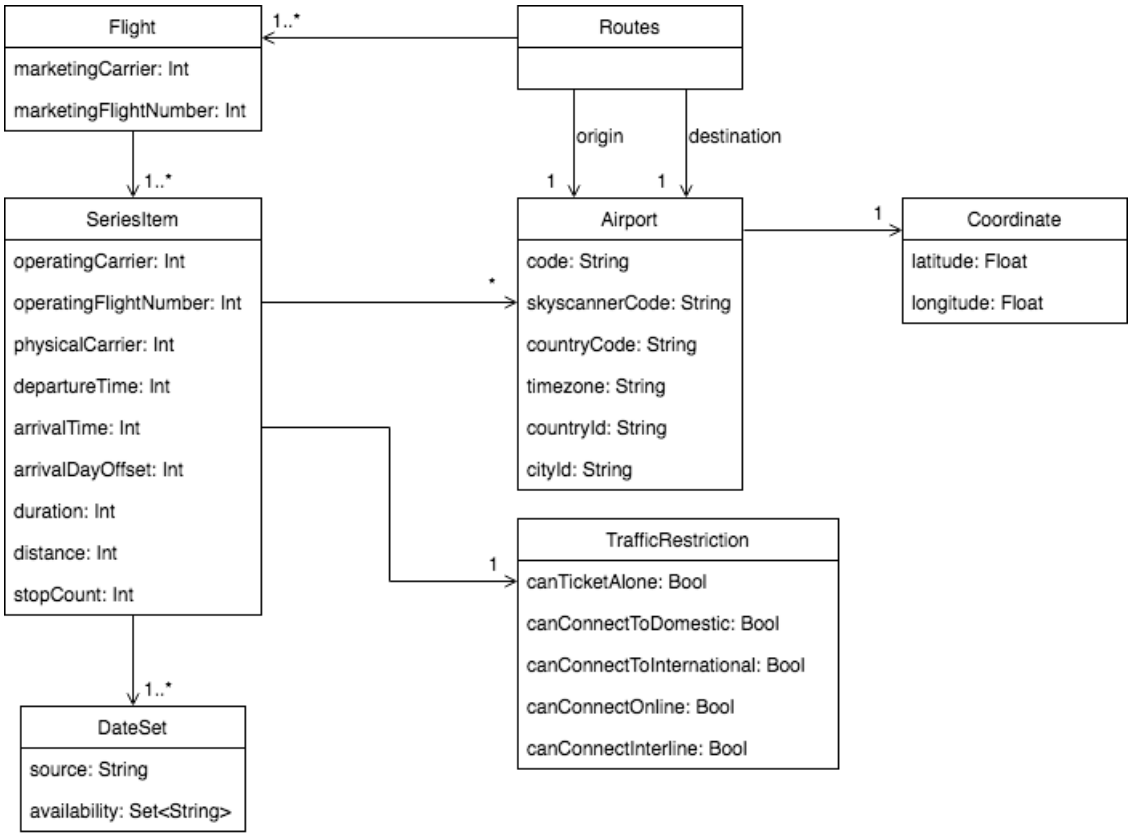

FIGURE 5.1: Unified Timetables UML class diagram

With this model, the Available Flights Pipeline can apply flattening and grouping easily.

Airport's code will be used to identify them and pairs of codes will represent a route. This code is also know as IATA Code[7].

5.2 Searches model

The user searches are stored in a huge database that contains flight search, car hire search and hotel search, one has value an the other two are `null`.

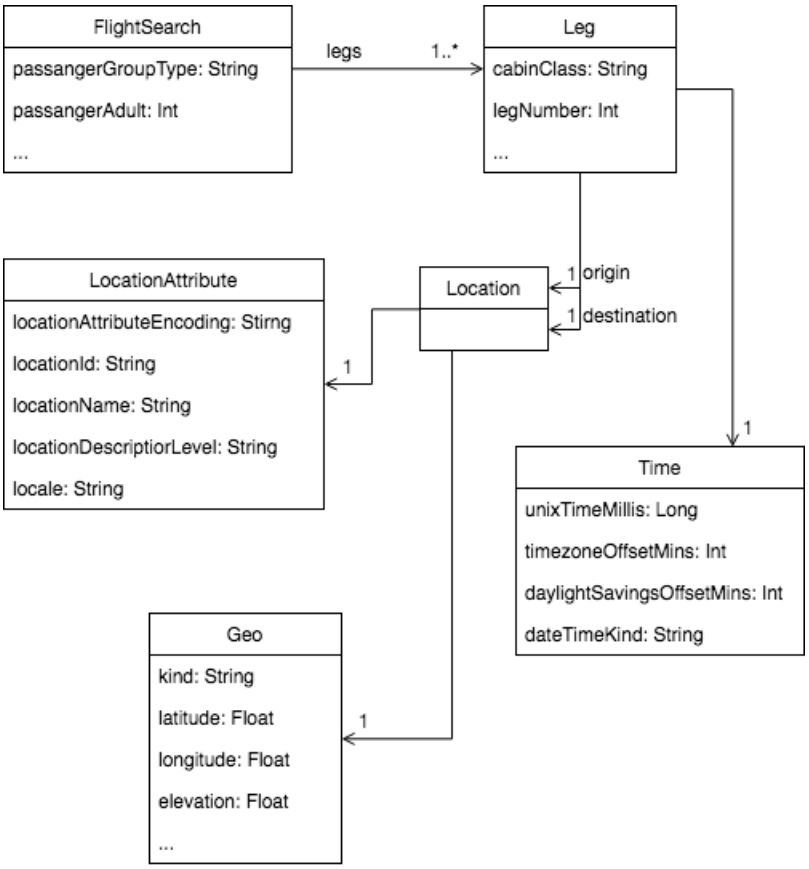

FIGURE 5.2: Flight search UML class diagram

A user query in Skyscanner contains a lot of information, but most of it is not useful for the purpose of this project.

5.3 Available Fligths Model

As we can see above in the Unified Timetable Object Model, flights are grouped by routes, then by flights, series and finally we reach the date. The Skyscanner offer and demand comparison will be queried by route and date, so it is important to have the flights stored and count its availability at date level.

The flights availability records will be grouped by route and then by date.

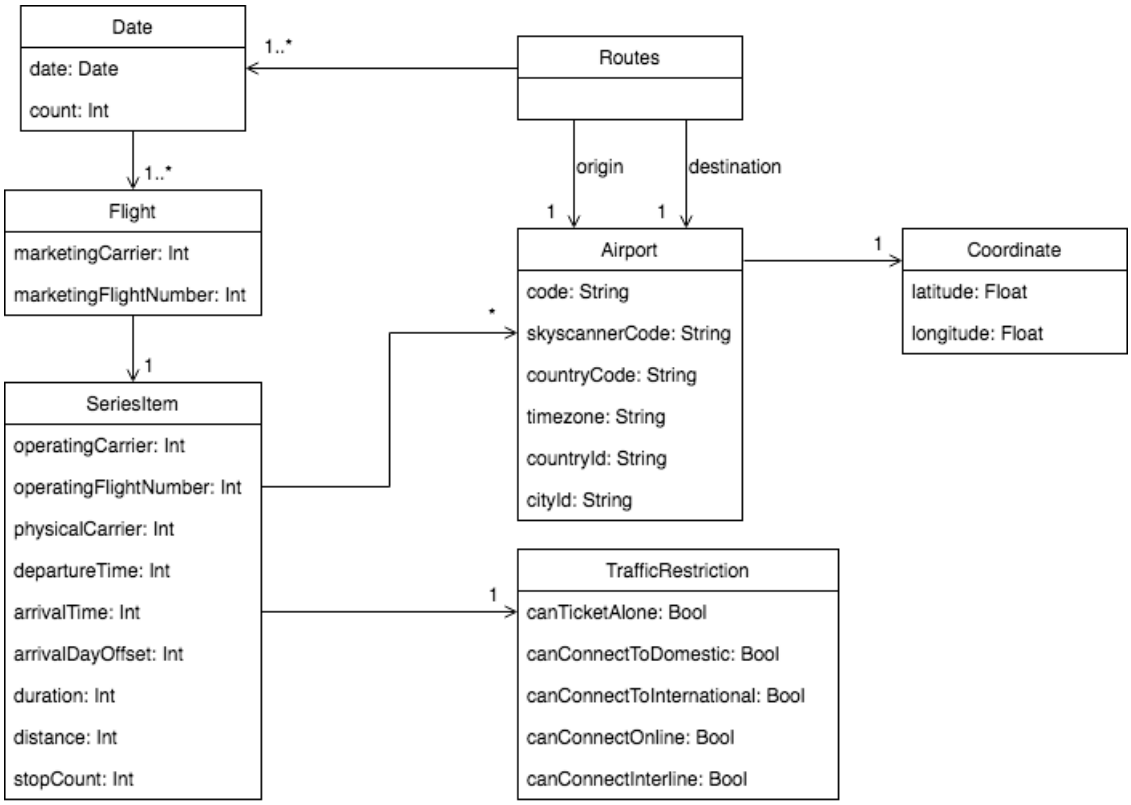

FIGURE 5.3: Flights Availability UML class diagram

Flights Availability and Unified Timetables UML class diagram looks very similar. The main difference is that a route in Unified model contains a set of flights but a set of dates in Flight Availability.

Date class contains a strings in date format `YYYY-MM-DD` and the number of available flights on that date. This last field is the same as the size of the set of flights it is composed by. The flight availability model has **a lot of duplicated values**, one flight operates a lot of dates and it is grouped by date[2].

5.4 User Searches Model

The user searches model has been simplified and a lot of fields from the original model has been removed. Since the amount of data of user searches increases on the order of millions per day, the model contains the basic data.

[2]All flights data transformations explained in **??**

User Searches Route

Field	Description	Data Type	Required	Key
Origin	The origin airport or station IATA code	String	Yes	Yes
Destination	The destination airport or station IATA code	String	Yes	Yes
Dates	The SFN timetables associated with the route	List of Searches Dates	Yes	No

TABLE 5.5: User Searches Route fields

Searches Date

Field	Description	Data Type	Required	Key
Date	Date for the leg that was searched	String	Yes	Yes
Count	Number of searches for that route in that date	Integer	Yes	No

TABLE 5.6: Searches Date fields

Chapter 6

Design

With all the specification of the data the Skyscanner offer and demand comparison will work with, a good architecture for the system is needed. The whole software project is split in four different parts: Available Flights Pipeline, User Searches Pipeline, Comparison Service and Web UI.

In this chapter all four architectures for the four different parts will be defined and analyzed. All alternatives will also be defined and explain why it is not in the final architecture.

Even so, first it is good to understand the general architecture of the whole Skyscanner offer and demand comparison.

6.1 Architecture

The Skyscanner offer and demand comparison is split in very notable three layers: Data collection and processing, Comparison Server and Web UI. Two of this layer, server and user interface, suit exactly two of the four remarkable in the whole system, the Comparison Service and Web UI. Named the same to avoid confusion.

An important characteristic of the Skyscanner offer and demand comparison is that the only way to feed the database is from the data collection and processing layer. This is different of most of projects and examples I have worked before in the University.

The most common architecture used is a three layers (data layer, domain layer and presentation layer) where in the presentation layer the user could usually put data into the data layer. The Skyscanner offer and demand comparison work very different: The user interface (that is the presentation layer) does not put or modifies any data from the database. It only reads. The only way the database can be filled with valid data is from the *data collection and processing layer*.

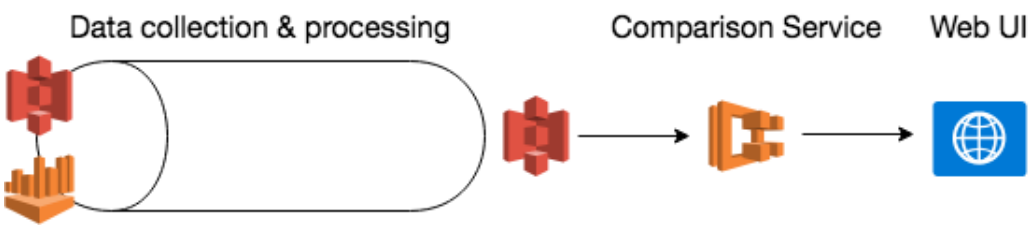

FIGURE 6.1: General view of the Skyscanner offer and demand comparison architecture.

6.1.1 Data collection and processing

This layer is composed by two components, one for each data processing: Available Flights Pipeline and User Searches Pipeline. The purpose of this layer is to collect all data necessary to do the comparison and filter and group according the final data model needed.

Both components in the this layer uses very similar technologies and are written in Scala[17]. The data is read from different sources that already exist in Skyscanner and have a well defined design (Data tribe and DeLorean squad's pipeline) and services (Amazon Athena and Amazon Simple Storage Service, S3). The whole data processing or data pipeline ran in Apache Spark, in an EMR cluster. In the end of the process both pipelines write into the same Storage service.

FIGURE 6.2: General view of the data collection and processing layer's architecture.

Amazon Athena

Amazon Athena[18], is an Amazon Web Service that lets query data stored in Amazon Simple Storage Service (S3) using standard SQL[19]. It have no server running, so it have no infrastructure to manage.

To access the data, Athena simply points to an S3 bucket with a defined schema and it lets you query its data with SQL queries. The cost of the service is based on the queries you run.

Data tribe is responsible of creating schemas for Athena to read S3 buckets. They have a huge bucket called `grappler_master_archive` that contains all the user searches. Athena becomes the easiest way to access user searches data.

Amazon Simple Storage Service

Amazon Simple Storage Service[20], also known as *S3*, stores data as objects within resources called buckets. It allows store unlimited objects in a bucket, also write, read and delete those objects. The maximum size of these objects is 5 terabytes. The access to buckets can be controlled.

In Skyscanner, most of the results of Amazon Data Pipelines are stored as a single or multiple files in S3, then those are processed by Amazon Lambda functions or other services. DeLorean squad stores timetables as a JSON[21] file in its S3 bucket. Data tribe has also a bucket for user searches (and much more) data, but, as explained before, it provides access from Amazon Athena.

Apache Spark™

Apache Spark™ is a unified analytics engine for large-scale data processing. It was created and it is currently maintained by the Apache Software Foundation[22].

Is one of the most popular fast and general-purpose cluster computing systems. It can run in a lot of different environments, Apache™ Hadoop®[23], Kubernetes[24], Amazon Elastic MapReduce and much more. Spark provides four APIs: Java, Scala, Python, R and SQL. Over 80 high-level operators can help building parallel processes and can be called from most of its APIs. Both data pipelines are written using the Scala API.

Apache Spark™'s architecture is based in *Resilient Distributed Datasets*, RDD, a read-only multiset distributed in multiple machines. Those machines are based in the MapReduce paradigm, a programming model for big data dumps processing. Since the data is distributed, this process runs in parallel.

To an RDD, you can apply two kind of operations: transformations and actions:

A **Spark Transformation** is a functions that creates a new RDD form the existing one. Transformations are lazy operations and only are executed when an action is called. When calling a transformation, a link is created between two RDDs, this happens successively until the action is executed. Then, all transformations are applied until the action. These links between transformations creates a directed acyclic graph.

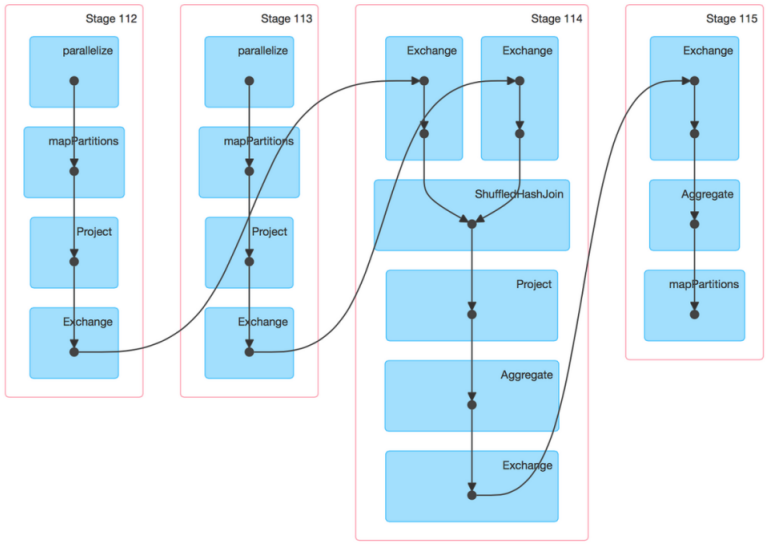

FIGURE 6.3: Example of a Directed Acyclic Graph (DAG).

Most common transformations used in the project are:

`def map[R](f: Function[T, R]): RDD[R]`
Return a new RDD by applying a function to all elements of this RDD.

`def flatMap[U](f: FlatMapFunction[T, U]): RDD[U]`
Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

`def filter(f: Function[T, Boolean]): RDD[T]`
Return a new RDD containing only the elements that satisfy a predicate.

`def groupBy[U](f: Function[T, U]): PairRDD[U, Iterable[T]]`
Return an RDD of grouped elements. Each group consists of a key and a sequence of elements mapping to that key.

A **Spark Action** are RDD operations that do not return an RDD-like value. Actions are executed when called, first running all transformation in the directed acyclic graph of transformations.

Most common actions used in pipelines are:

`def count(): Long`
Return the number of elements in the RDD.

`def fold(zeroValue: T)(f: Function2[T, T, T]): T`
Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral "zero value".

`def foreach(f: VoidFunction[T]): Unit`
Applies a function f for all elements of this RDD.

Amazon Data Pipeline

Amazon Data Pipeline[25] service helps running processes reliably and move data between Amazon Web Services Storage services. In the Skyscanner offer and demand comparison it is used to move data From Amazon Athena or Amazon Simple Storage Service to Amazon Elastic MapReduce and from Amazon Elastic MapReduce back to Amazon Simple Storage Service.

If, for any reason, the pipeline fails, this service retries the execution, if the execution fails constantly, the data pipeline stops and sends a failure notification to the owner.

Data Pipelines can also be scheduled to run every hour, every two hours, every day, week, etc. Both pipelines are scheduled to run once a day every day. In every executions, first builds the environment (Amazon Elastic MapReduce and Apache Spark™) and then runs the JAR[26] file generated from available flights pipeline and user searches pipeline. To avoid unnecessary dependencies, there are two Amazon Data Pipelines, one for the flights and another for the user searches.

Amazon Elastic MapReduce

Amazon Elastic MapReduce[27] (EMR) provides a Hadoop[23] framework that makes easy, fast, and cost-effective to process vast amounts of data. As explained before, Apache Spark™ runs in Hadoop. These clusters are prefect for running big data applications and pipelines.

EMRs use master/slave communication model. This, applied to Apache Spark™, means that one cluster divides the data and does the partition of the RDDs while the rest of the instances, the slaves, execute the actions and transformations.

The EMR configuration for each pipeline will be:

Field	Description	Value
Release	EMR version	`emr-5.12.0`
Cluster master instance type	Instance type of the master cluster	`m3.xlarge`
Cluster core instance type	Instance type of the slaves clusters	`m3.xlarge`
Cluster size	Number of slaves clusters	6
Timeout	Insurance to not run forever	5 hours

TABLE 6.1: EMR configuration

EMR 5.12.x was the latest version when the project started. Newer versions does not have any relevant upgrade.

`m3.xlarge` has the following hardware specs:

Name	API Name	Memory	vCPUs	Instance Storage
M3 General Purpose Extra Large	`m3.xlarge`	15.0 GiB	4	80 GiB (2×40 GiB)

TABLE 6.2: `m3.xlarge` specs

6.1.2 Service

The service is the middle layer of the Skyscanner offer and demand comparison. Simple Python[28] Reads data written by the components in the Data collection and processing layer and serves the data through an Amazon Elastic Container Service.

A Docker container is created and managed by Amazon Elastic Container Service. In this container a Python[28] service is deployed.

Amazon Elastic Container Service

Amazon Elastic Container Service, also known as ECS[29], is a container management service that supports DockerDocker. It ease the container orchestration with a simple API.

Docker

Docker[30] tool that creates a container where you can run whichever application you want with its environment. A Docker container can be created in a lot of machines, for example in an Amazon Elastic Container Service.

6.1.3 Website

Finally, the presentation layer gets data from the Service and displays. The architecture of this layer is very simple, the website does an HTTP request to the service and gets a JSON back.

It is also ran in a Docker container deployed in Amazon Elastic Container Service.

6.2 Components

Once the technology each layer will use, lets define the code architecture of each component.

6.2.1 Available Flights Pipeline

The flight offer data pipeline reads timetables from DeLorean squad's ingest pipeline, stored in Unified Timetable Object Model and group, map and filter to Available Fligths Model.

DeLorean squad's Ingest Pipeline writes all the timetables in its Amazon Simple Storage Service bucket in very JSON[21] alike format. Apache Spark™ and the components of DeLorean squad that read routes from the result file line by line, each of those lines is a Unified Route. Luckily when Apache Spark reads a file i creates an RDD of String, one record per line; so when reading DeLorean squad's file, it will get an RDD of Unified Routes in JSON[21] format.

For all data processing, the pipeline is split in well-defined stages. Each stage has a name to be identified and a function `run` that executes the Spark transformations and actions needed to complete its purpose. The Available Flights Pipeline is composed by six stages grouped in two *sub-pipelines*:

Routes *sub-pipeline*

This *sub-pipeline* basically regroup Unified Routes data to a friendly model for the Routes Availability. It does not discards any field just in case those want to be used for future features, improvements or applications.

- **Read Routes stage**: It reads the `version.txt` file from DeLorean squad's bucket, there it get the latest version of the timetables and read them, returning an Strings RDD.
- **JSON to Route stage**: Maps from a String RDD to a Unified Route Class using **??**.
- **Flights Flattening stage**: Flats all routes' series at date level, duplicating origins and destinations since each flight have the same route. There are three flattening, from series to series item, from series item to date sets and finally from date sets to single dates. Going from 64,000 routs to 75,000,000 flights. The amount of data increases a lot here.
- **Group Routes stage**: After flattening at date level, flights are grouped by date. There is a lot of shuffling[1] which makes it a slow process.

Apart from these four stages, it has other to parse grouped routes by date objects to JSON[21] and another to write it to Amazon Simple Storage Service, but since in the end those are not necessary, these two stages are disabled.

Summary *sub-pipeline*

- **Map Summary stage**: Discards lots of fields from the model resulting from the routes *sub-pipeline* and maps the data to the final records that will be written in the database.
- **Write Records stage**: Writes into Amazon Simple Storage Service using **??**.

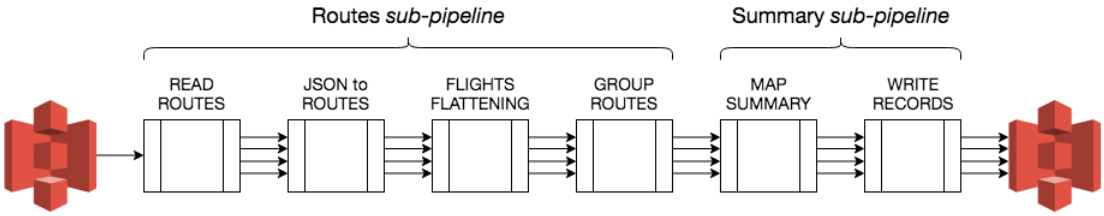

FIGURE 6.4: Available flights pipeline architecture and data flow diagram

To end up doing applying this architecture, there have been a lot of alternatives during all the development of the Available Flights Pipeline.

Amazon Relational Database Service

[1] Sharing data between different nodes or machines

In the beginning, the idea was to write all data to Amazon RDS, Relational Database Service[31]. This service offers a relational database in the cloud that can be accessed very fast. But there are two important reasons thy this option has been discarded:

- **Cost**: It is very expensive to have an RDS active all the time. Some squads in Skyscanner use Amazon Relational Database Service but only active some time a week for very specific purposes.
- **Knowledge**: I know about some squads that used that in some occasion, but not based in Barcelona. Reading some documentations and examples, it looked very difficult to set up and use, there were alternatives

Amazon DynamoDB

Another idea about the database where the pipeline could write was Amazon RDS, Relational Database Service[**dynamodb**]. This service offers a non-relational database in the cloud that can be accessed very fast. But there was one important reasons thy this alternative was discarded:

- **Cost**: It is very expensive to have an DynamoDB, even more expensive than Amazon Relational Database Service.

S3 and dates at month level

The main problem of these pipeline that happens in the User Searches Pipeline as well, is that **versions are written every day** and does not delete anything to satisfy functional requirement of Historical data (#8)

DeLorean squad provides flights for one year ahead and every day the data is uploaded in a new file to S3, the Available Flights Pipeline reads this file and gets all flights for all days in the next year and writes them to the Database. It would be very easy for the service then read with an SQL table like:

`origin`	`destination`	`version_date`	`flight_date`	`count`
BCN	LGW	2018-06-13	2018-08-20	14
BCN	LGW	2018-06-13	2018-08-21	12
BCN	MAD	2018-06-13	2018-08-20	34
BCN	EDI	2018-06-13	2018-08-12	20
BCN	EDI	2018-06-14	2018-08-12	20
BCN	EDI	2018-06-15	2018-08-12	30
BCN	EDI	2018-06-15	2018-09-14	25
...	...	...	...	...

So, for instance, if the user queries for the evolution of the flights available from Barcelona to Edinburgh on August 12 of 2018, the service could simple run an SQL query to the Database like `SELECT * FROM flight_availability WHERE origin = "BCN" AND destination = "EDI" AND flight_date = "2018-08-12"`. Then the service could map the result to the desired schema and response it back to the client.

Looks easy and simple, but the problem is that, as explained before, Amazon Relational Database Service is very expensive. The only storage service left that does not take much time to develop is Amazon Simple Storage Service.

Is very fast to write to S3, the whole data dump with more than 75 million flights can be written in less than a minute to a single flight, but how about the reading from the service? Boto3[2] takes more or less an eighth of a second to access the file and more or less a quarter second to read the whole file. This is fine if the service just needs to read one file, but as explained before, the pipeline writes one file every day. So, for one query, the worst case scenario is 365 (one year) files to open and read. In other words, more than 4 minutes for the service to response a query. This does not satisfy the fourth non-functional requirement of Speed and Latency.

After analyzing the data, one solution was to only write changes, so there are no that big amount of files to read, but it was discarded because user searches change every day, not like flights, and the same problem would happen when reading from the other pipeline.

In the end, the final solution was to write one file per record, and inside it all the data about the flights for that route in that date. That way, **??** could run the `list_bucket` function, that list all objects under the given key.

```
BCN-LGW/
        2018-08-20/
                    2018-06-13-14.json
        2018-08-21/
                    2018-06-13-12.json

BCN-MAD/
        2018-08-20/
                    2018-06-13-34.json

BCN-EDI/
        2018-08-12/
                    2018-06-13-20.json
                    2018-06-13-20.json
                    2018-06-13-30.json
        2018-09-14/
                    2018-06-15-25.json
```

The problem with that is that there are 75 million different records to write. It would take more than one day for the pipeline to write the whole data, very expensive because of the Amazon Elastic MapReduce cluster running time and very slow.

After checking the user stories and the purpose of this tool, a final decision was made: Data was going to be store that way, one record per day, but the number of records will be reduced to much **grouping flights at month level** instead of at day level. The final S3 directory structure would be:

[2]Boto[32] is the Amazon Web Services (AWS) SDK for Python, explained in **??** on page **??**

```
BCN-LGW/
        2018-08/
                2018-06-13-26.json

BCN-MAD/
        2018-08/
                2018-06-13-34.json

BCN-EDI/
        2018-08/
                2018-06-13-20.json
                2018-06-13-20.json
                2018-06-13-30.json
        2018-09/
                2018-06-15-25.json
```

With this design, the service takes less than a second to get all the data and the pipeline takes one hour to write all the records.

6.2.2 User Searches Pipeline

The user demand is obtained from **user searches** table in Amazon Athena. Then the results of the Athena query are mapped to the same data model of the stored in S3 by the Available Flights Pipeline.

The first problem found in this pipeline was that the results from Athena were stored in a CSV[33] in S3 and each row had a record with a format very similar than JSON[21]. Those records could not be parsed to an object using any library[3].

Another problem found is that as more data you request, more time it gets to finish the query, and it is not a lineal progression. To solve that, the query was split in four subqueries. From one query that requested all the user flight searches for one day to four that request all user flight searches in hour ranges 00:00-05:59, 06:00-11:59, 12:00-17:59 and 18:00-23:59. Usually the resultant files take **30 gigabytes**.

User Searches Pipeline is not split in *sub-pipelines*. It has four stages in total:

- **Athena Reader stage**: Queries Athena for user searches. Athena writes automatically to S3 and returns the file name.
- **S3 Reader stage**: Gets the file name of the query result and reads it. CSV[33] to String RDD.
- **Search Query stage**: Gets the records from the Athena records, filters them by kind of search an other parameters, maps each record to the desired model and groups by route.
- **Write Records stage**: Writes into Amazon Simple Storage Service using **??**.

[3]Solution explained in **??** on page **??**

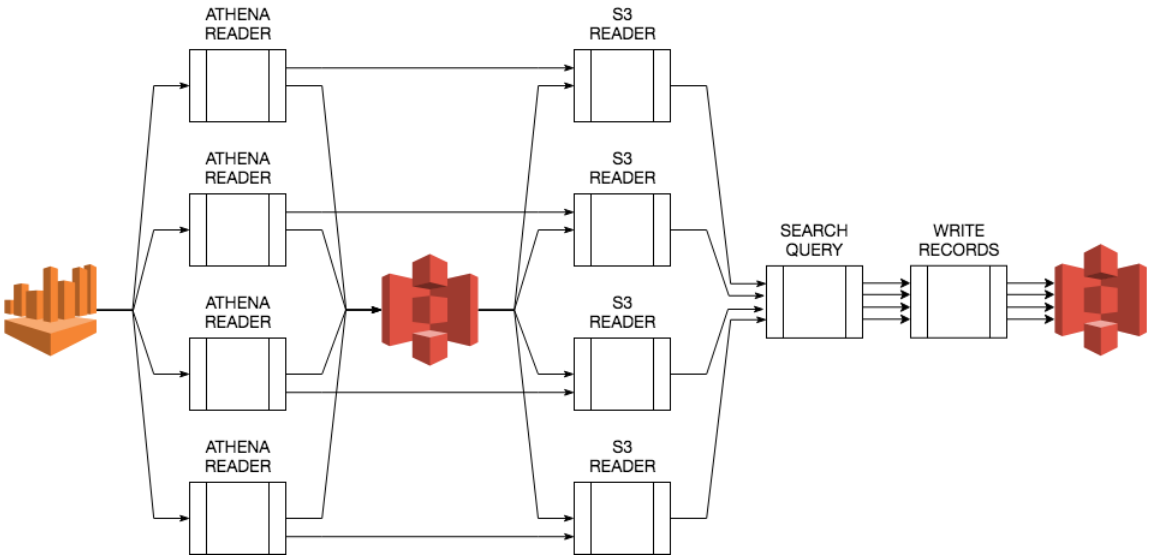

FIGURE 6.5: User searches pipeline architecture and data flow diagram.

6.2.3 Comparison Service

Once all the data is processed every day and stored into Amazon Simple Storage Service, it has to be served in an HTTP[34] API.

The Comparison Servie's API provides two endpoints, one for the comparison and another one for a single source (flights or searches) request[4]. The main class, API Handler, execute a function when the endpoint is queried and uses an AWS Adapter that is directly connected to **??**. The API Handler returns a set of records in JSON format.

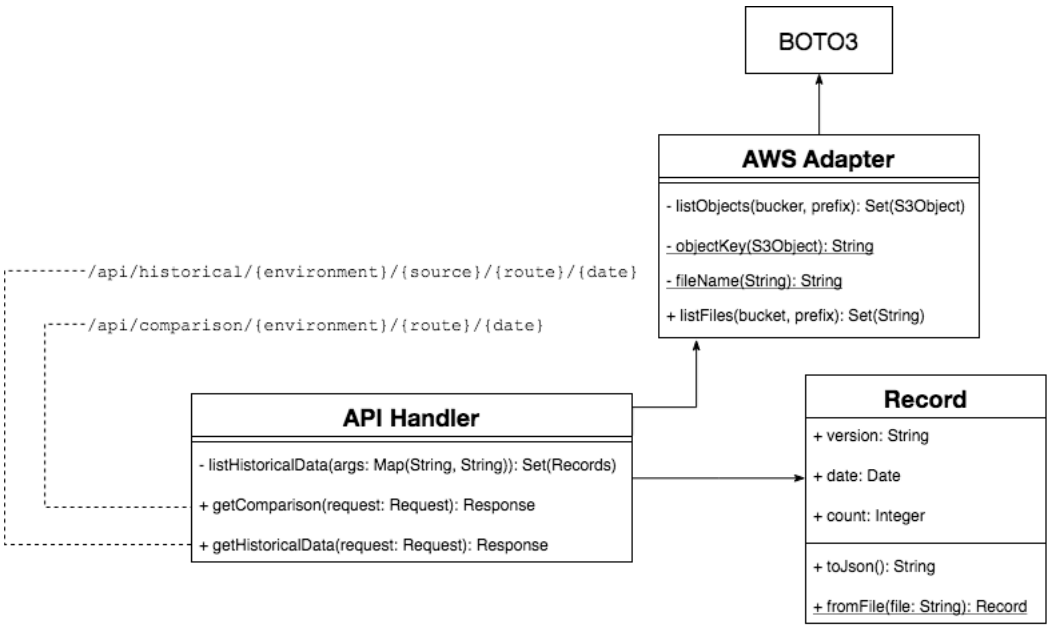

FIGURE 6.6: Comparison service class diagram.

[4]The *endpoint* tag can be `sandbox` or `prod`. Learn more about those environments in section Environments on page 11.

`/api/comparison/environment/route/date`

GET two sets of data for a given route and date: All existent versions values for available flights and user searches.

`/api/historical/environment/source/route/date`

GET one sets of data for a given route and date and source[5]: All existent versions.
The first idea for the comparison service, was using an AWS Lambda Function and exposing it through AWS API Gateway. But it ended up in Amazon Elastic Container Service with a Skyscanner gateway.

Amazon Lambda

Is one of the most uses service used bu Amazon Web Service. It is a *serverless* system that lets you run code easily.

> *Run code without thinking about servers. Pay only for the compute time you consume.*

Lambda functions can be executed from an S3 event (run a lambda when a file under a given prefix is putted in a bucket) or an API Gateway, it get an event and a context input, runs Python[28] or Node.js[**nodejs**] code and returns (or not) some value. By default Lambdas have a three minutes timeout, which is not a problem. Also, Lambdas **scale automatically**.

This service looked very promising but was not used because of API Gateways.

Amazon API Gateway

Simple service that provides a front door for whatever other Amazon's service.

Skyscanner reached the limit of API Gateways. Using a Lambda I would have open an special request in the company to get a new API Gateway, this request would have been provably reject because API Gateways are reserved for very specific uses. That is why Amazon Lambda and API Gateway was not used for the service.

6.2.4 Web UI

Finally, in the front end of the application, the Web UI. A simple website with only one page, based in the Composite Pattern becomes a responsive application with a high code scalability and maintainability.

[5]The two available source the Service have are: Available Flights and User Searches

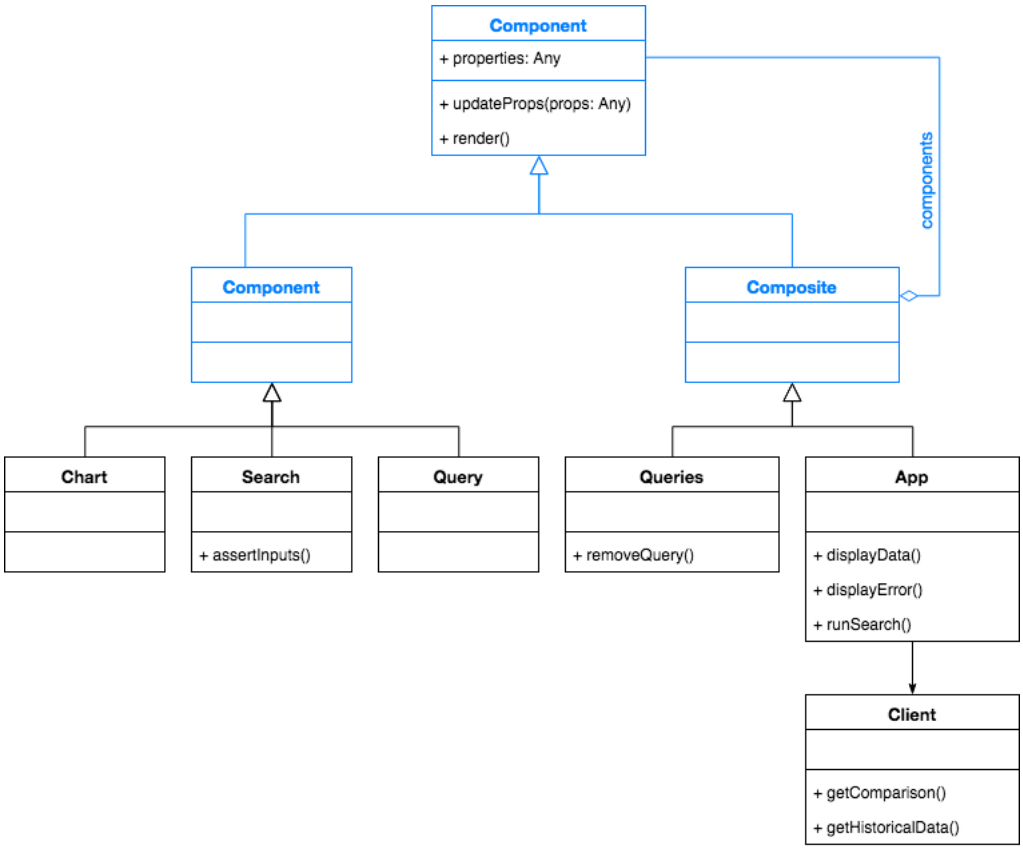

FIGURE 6.7: Web UI class architecture. In blue we can appreciate the Composite Pattern base classes.

Appendix A

Skyscanner structure

Skyscanner has a very horizontal structure, based on Spotify's[35].

In the top of the company hierarchy there is Gareth Williams (CEO and Co-founder). Bellow the rest of CxOs: CCO, CTO, CPO, CFO, CLO and the Senior Executive Assistant. Then vice presidents, senior managers, managers and then developers and interns[36].

Apart from this hierarchy structure, the whole team, except the CEO, CxOs and the Senior Executive Assistant, is mainly split in **Squads**, each of those belong to a **Tribes**. Apart from Squads and Tribes there are also Chapters, Guilds ans XBT'S[37].

Squad

Are independent teams of no more than 8 people that are focused on delivering a core mission. Each squad has the freedom to act and be accountable to its mission.

Tribe

Squads belong to a Tribe. The tribe will have an aligning mission linking to each squad's mission and is only achievable depending on the success of each squad. The Tribe lead is responsible for providing the right environment to deliver and providing direction.

Chapters

Are people who do similar work. This is a secondary home, and how people are line managed. Chapter leads are responsible for developing people and in tribe practices.

Guilds

Are communities of interest of people who do not necessarily do similar work. It is people from across the business that want to share knowledge, tools, and word practices.

XBT'S (cross business teams)

XBTS' provide a platform to help solve business problems or opportunities with no natural home while giving all employees the ability to make an impact across any area of Skyscanner.

Bibliography

[1] Multiple authors. *Aplication Programming Interface*. 2018. URL: https://en.wikipedia.org/wiki/Application_programming_interface (visited on 06/05/2018).

[2] Mark Logan. *Skyscanner's Strategy*. 2015. URL: https://skyspace.sharepoint.com/sites/CxOGMblogs/Marksblog/Lists/Posts/Post.aspx?ID=2 (visited on 02/27/2018).

[3] Francisco Lopez. *DeLorean Home*. 2017. URL: https://confluence.skyscannertools.net/display/DEL (visited on 01/28/2018).

[4] Francisco Lopez. *Marketplace Engine Tribe Home*. 2017. URL: https://confluence.skyscannertools.net/display/MET/Marketplace+Engine+Tribe+Home (visited on 02/27/2018).

[5] Skyscanner Ltd. *The Road Ahead*. 2016. URL: https://skyspace.sharepoint.com/docs/Internal%20Communications%20and%20Events%20Squad/The%20Road%20Ahead.pdf (visited on 02/28/2018).

[6] Amazon Web Services Inc. *Amazon Web Services (AWS) Documentation*. 2018. URL: https://aws.amazon.com/documentation/ (visited on 03/12/2018).

[7] Multiple authors. *IATA airport code*. 2018. URL: https://en.wikipedia.org/wiki/IATA_airport_code (visited on 03/01/2018).

[8] Interactive Data Lab. *Vega: A visualization grammars*. 2013. URL: https://vega.github.io/vega/ (visited on 03/01/2018).

[9] Kent Beck. *Extreme Programming Explained. Embrace Change*. Addison-Wesley Professional, 2005.

[10] GitLab. *GitLab*. 2018. URL: https://about.gitlab.com/product/ (visited on 06/03/2018).

[11] Atlassian. *Bitbucket*. 2018. URL: https://bitbucket.org/ (visited on 06/03/2018).

[12] Github. *Github*. 2018. URL: https://github.com/marketplace (visited on 06/03/2018).

[13] Drone. *Drone*. 2018. URL: https://drone.io/ (visited on 06/03/2018).

[14] Stanford University. *Machine Learning*. 2018. URL: https://www.coursera.org/learn/machine-learning (visited on 03/12/2018).

[15] OAG Aviation Worldwide Limited. *OAG: Connecting the World of Travel*. 2018. URL: https://www.oag.com/about-oag (visited on 03/12/2018).

[16] Palo Alto Networks Inc. *GlobalProtect VPN*. 2017. URL: https://www.paloaltonetworks.com/documentation/31/globalprotect/gp-agent-user-guide (visited on 06/05/2018).

[17] École Polytechnique Fédérale Lausanne (EPFL). *The Scala Programming Language*. 2018. URL: https://www.scala-lang.org/ (visited on 06/12/2018).

[18] Amazon Web Services Inc. *Amazon Athena*. 2018. URL: https://aws.amazon.com/athena/ (visited on 06/11/2018).

[19] w3school. *SQL Tutorial*. 2018. URL: https://www.w3schools.com/sql/ (visited on 06/11/2018).

[20] Amazon Web Services Inc. *Amazon S3*. 2018. URL: https://aws.amazon.com/s3/ (visited on 06/11/2018).

[21] JSON.org. *The JavaScript Object Notation (JSON) Data Interchange Format*. 2017. URL: https://tools.ietf.org/html/rfc8259 (visited on 01/30/2018).

[22] The Apache Software Foundation. *The Apache® Software Foundation*. 2018. URL: https://www.apache.org/ (visited on 06/11/2018).

[23] The Apache Software Foundation. *Welcome to Apache™ Hadoop®!* 2018. URL: http://hadoop.apache.org/ (visited on 06/11/2018).

[24] The Kubernetes Authors. *Kubernetes: Production-Grade Container Orchestration*. 2018. URL: https://kubernetes.io/ (visited on 06/11/2018).

[25] Amazon Web Services Inc. *Amazon Data Pipeline*. 2018. URL: https://aws.amazon.com/datapipeline/ (visited on 06/12/2018).

[26] Multiple authors. *JAR (file format)*. 2018. URL: https://en.wikipedia.org/wiki/JAR_(file_format) (visited on 06/12/2018).

[27] Amazon Web Services Inc. *Amazon EMR*. 2018. URL: https://aws.amazon.com/emr/ (visited on 06/12/2018).

[28] Python Software Foundation. *Python™*. 2018. URL: https://www.python.org/ (visited on 06/12/2018).

[29] Amazon Web Services Inc. *Amazon ECS*. 2018. URL: https://aws.amazon.com/ecs/ (visited on 06/12/2018).

[30] Docker Inc. *What is Docker*. 2018. URL: https://www.docker.com/what-docker (visited on 06/12/2018).

[31] Amazon Web Services Inc. *Amazon RDS*. 2018. URL: https://aws.amazon.com/rds/ (visited on 06/12/2018).

[32] Amazon.com Inc. *Boto 3*. 2018. URL: http://boto3.readthedocs.io/en/latest/ (visited on 06/12/2018).

[33] Y. Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. 2005. URL: https://tools.ietf.org/html/rfc4180 (visited on 01/30/2018).

[34] Multiple authors. *Hypertext Transfer Protocol*. 2018. URL: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol (visited on 06/13/2018).

[35] Skyscanner Engineering. *The Culture of Growth Squads in Skyscanner*. 2016. URL: https://www.youtube.com/watch?v=5_x-_EJNWPs (visited on 03/01/2018).

[36] Skyscanner Ltd. *Crew Chart hierarchy*. 2017. URL: https://flightdeck.skyscannertools.net/crewchart.html (visited on 06/26/2017).

[37] Skyscanner Ltd. *How Skyscanner Works*. 2017. URL: https://skyspace.sharepoint.com/sites/Information/Pages/How-Skyscanner-Works.aspx (visited on 03/01/2018).