SKYSCANNER
IN COLLABORATION WITH
UNIVERSITAT POLITÈCNICA DE CATALUNYA (UPC)

FINAL DEGREE PROJECT

---

# Skyscanner offer and demand comparison

Skyscanner available flights and user searches 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

Friday 15th 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, send images, share live videos and much more 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 that earth flattering too. In 1918 visiting another place was very difficult. If you wanted to cross the sea, you had to do it by boat, and those trips could last days or weeks. 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 a user friendly 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 to solve this problem, providing a visual comparison to explore differences and similarities between what the users search and what the airlines provide. Making easier 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 searches** and **flights provided** by airline companies. This comparison could improve flights advertisement according to user demand. The company could also develop more complex software using the huge amount of data it will compare through an Application Programming Interface.

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 will be playing with **big data**.

## 1.1 Skyscanner

Skyscanner[1] is a travel fare aggregate website. It was founded in 2004 when a group of people was frustrated by the difficulties of finding cheap flights.

In 12 years it has evolved from a little office in the suburbs of Edinburgh (Scotland) 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 not 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 available flights.

## 1.2 DeLorean squad

DeLorean[2] is a squad[1] of Marketplace Engine tribe, 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[3] 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 78

# 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[4] has became the main competitor of Skyscanner. The new version is very fast and has a complete new interface, following Material Design[5] 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 and preffer Google over Skyscanner.

### 2.1.2 Kayak

Kayak[6] 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[7] 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* $\neq$ *Product*[8].

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[1].

### 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 for the user*. In other words, it just serves **information to the user** but does not get any from him or her. All five squads take the **data from providers**.

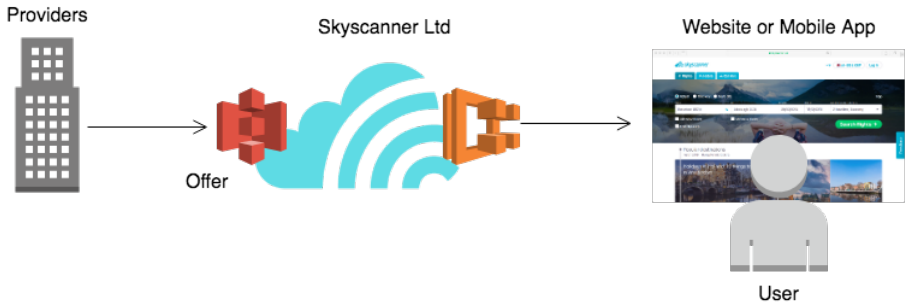



FIGURE 2.1: Simple explanation of Marketplace Engine data flow.

### 2.2.2 Data tribe

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

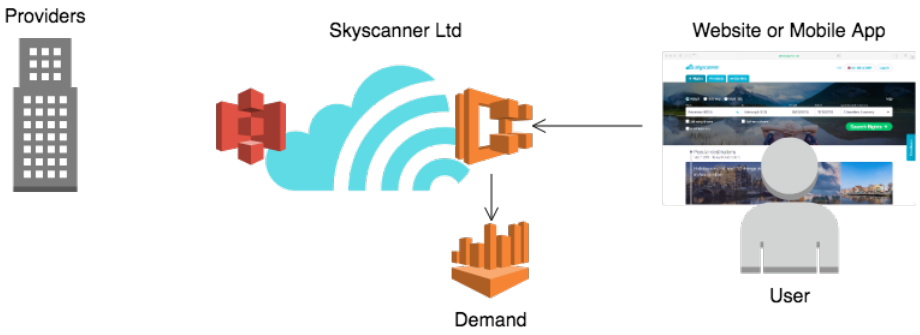



FIGURE 2.2: Simple explanation of Data Tribe data flow.

### 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 Skyscanner offer and demand comparison will be.

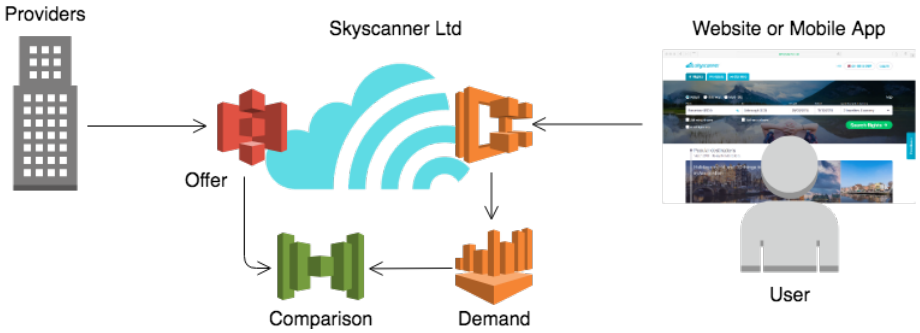



FIGURE 2.3: First approach of the Skyscanner offer and demand comparison data flow.

# 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 in this chapter.

## 3.1 Formulation of the problem

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

As explained before, in any team of Skyscanner, user queries and providers data are 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; so the client can find *over-requested* routes, those routes with more user demand than availability by the providers.

DeLorean squad manages a huge amount of data: All flights planned for the next year, 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 stories for the processing of those 75 million routes and all user session's queries to get some significant results, then we set a mission for this ptoject:

> Provide a **visual tool** to find routes with much **more demand** (user searches) **than offer** (available flights) 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 uses: 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 data pipeline 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 basically by two components:

#### 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; and a drop down list to choose the source information: Available flights, user searches of comparison of flights and searches.

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

#### Chart visualization

Simple chart with the comparison between available flights and user searches of the selected entity through time.

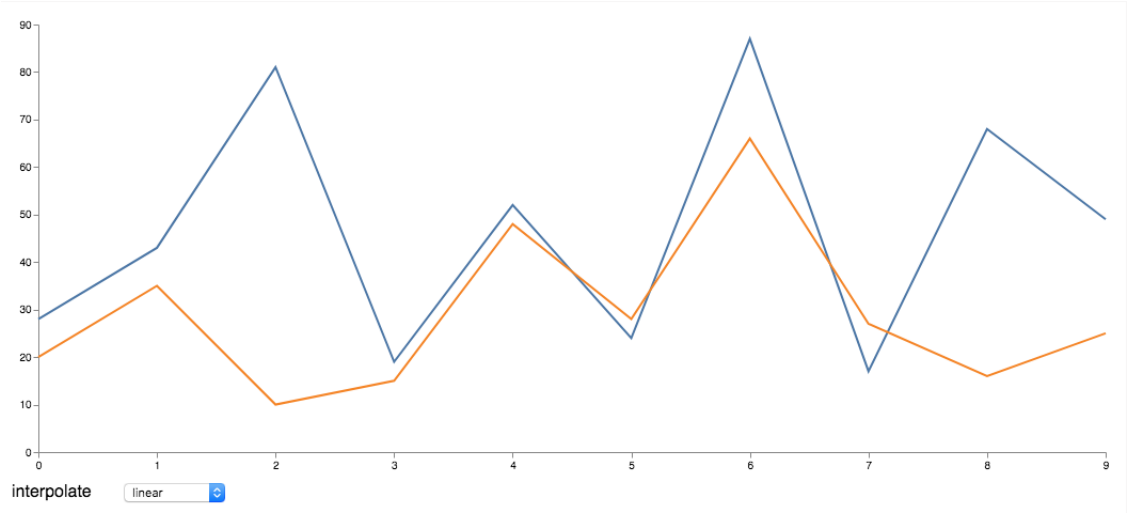



FIGURE 3.1: Chart mock-up. One color goes for available flights and the other one for user searches.

### 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 are not that critical 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[9], Country ID, City ID, etc.

Timetables are served, basically, in three different models: Unified Routes, Single Flight Number Routes and Summary Routes[1]. Both Single Flight Number and Summary Routes

[1]Explanation of different models in section Routes model on page 20

stores timetables by days of week. The current **Unified Model** is supposed to be the final one, it has been studied and refined by experts on the domain and DeLorean squad 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. In those cases, the search result is a list of routes or a list of dates. The user is not searching flights by route and date, it sets the period of time he/she can travel.

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 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 have good results. For London you can set six different airports (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, or more, amount of 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 the developer can ask for help to DeLorean squad members or data scientist of Skyscanner.

### 3.3.4 Kind of data

User searches and flights offer have very similar parameters: origin, destination, date, etc. But their model can be very different. There are some facts that can give very different results 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.

[2]Read more in section Resources

### 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 to use reliable libraries, like Vega[10]. Combining it with React.js and Skyscanner components, the Web UI development will be fast and easy to develop.

## 3.4 Resources

Skyscanner is already in the air, it is running and has been running for more than ten years. In the past two years it has been migrating all their services to Amazon Web Services[11].

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[3]:

- Compute
  - Lambda
  - 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
  - API Gateway, Networking & Content Delivery
  - Cloud Formation
  - Simple Notification Service

## 3.5 Methodology and rigor

### 3.5.1 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[12] 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.

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

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.

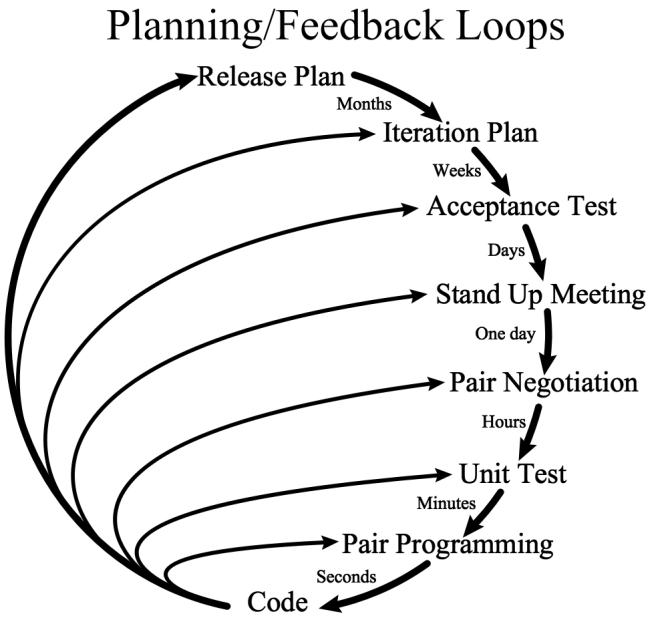



FIGURE 3.2: Extreme programming planning loops.

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.

### 3.5.2 `git`

GitLab will be the main tool for version control of the source code and issue tracking of different tasks. GitLab[13] is very similar than the well known Bitbucket[14] and Github[15], 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[4]. Using `git`, the versions will be forked

[4]Heatmap was the original name of this project

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 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.

### 3.5.3 Continuous delivery

In order to deploy pipelines and services automatically when merging to `master`, the projects have integration with Drone[16]. Every time it commit to a branch, the application is built, tested and, if everything goes well, it deploys to Sandbox. When merging to master, Drone automatically deploys to Prod.

### 3.5.4 Tasks and issues

The **issue tracker** provided by GitLab[13] 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.

### 3.5.5 Environments

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

#### 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 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 a Skyscanner offer and demand comparison stakeholder because it will be using its 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 server provide is very useful for constructed routes[1].

[1] combination of two SFN

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 searches and queries) will be obtained from these squads.

### 4.1.4 Other Skyscanner developers

Last (in Skyscanner) but not least, a new service will appear in the company, all developers will be able to use it and build software using the Skyscanner offer and demand comparison's data. For instance, it can be used as a training for a complex Machine Learning[17].

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[18].

### 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* routes.

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

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

3. **Search multiple availability values**: Be able to search and show multiple availability values for different flights 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 searches values**: Be able to query and display multiple user searches values for different routes in the same chart, easing the comparison between both queries. For example: *Route A-B in August 2017 shows more searches than route A-B in December 2017 from January to June.*

5. **Search multiple mixed values**: Enable comparison between availability and searches 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 (available flights and user 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.

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[19].

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 availability 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 chart 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 searches 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 availability and searches of one route in a specific month so that I can see if it is over-requested or 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 availability 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 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 needs.

### 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.

| 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 | 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

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, but it uses **Airport object** instead of an integer and a set of strings in date format `YYYY-MM-DD` in date sets' availability:

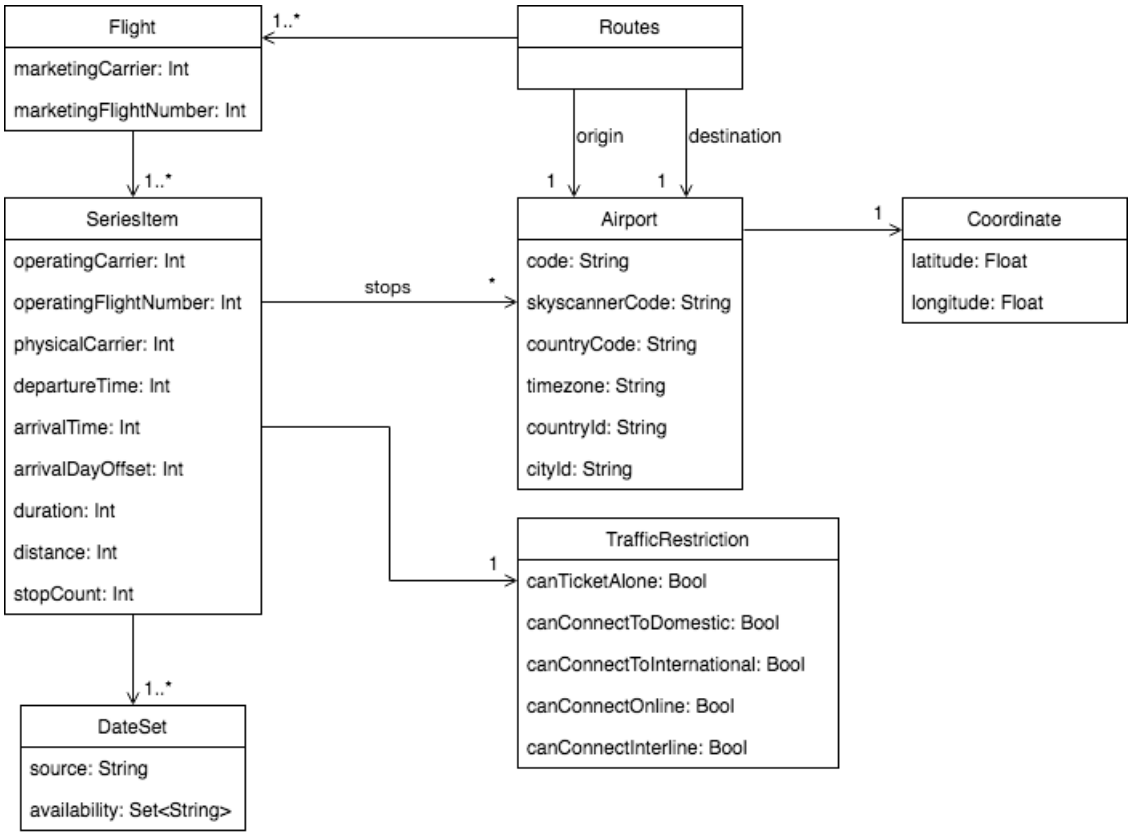



FIGURE 5.1: Unified Timetables UML class diagram

[1]The model changes and the pipelines groups by month. Learn why here: S3 and dates at month level on page 35

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[9].

## 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`.

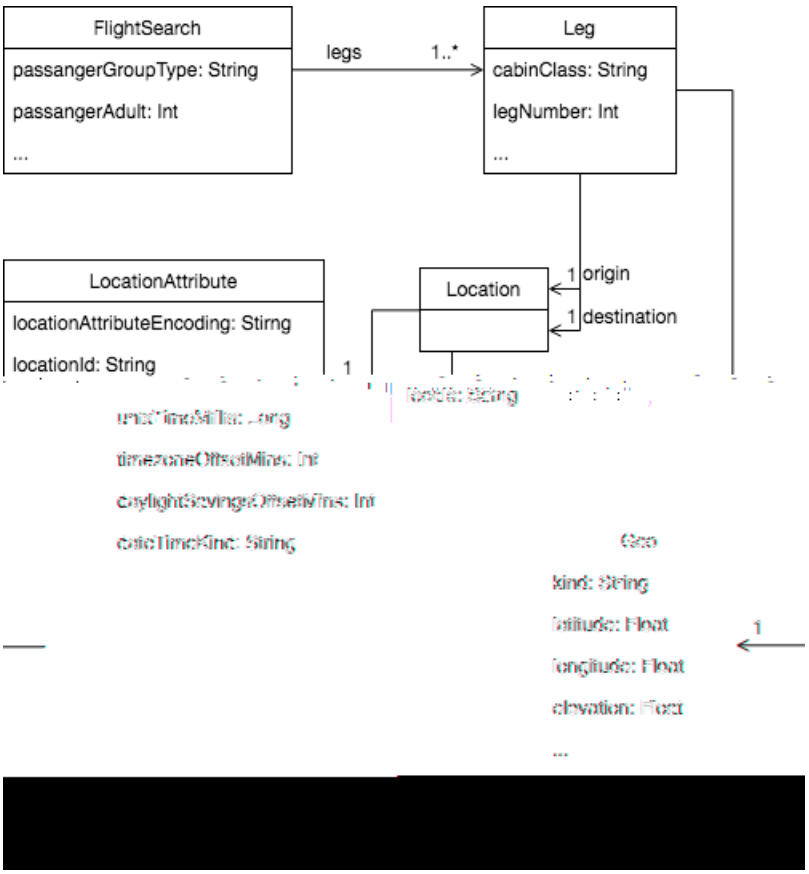



FIGURE 5.2: Flight search UML class diagram

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.

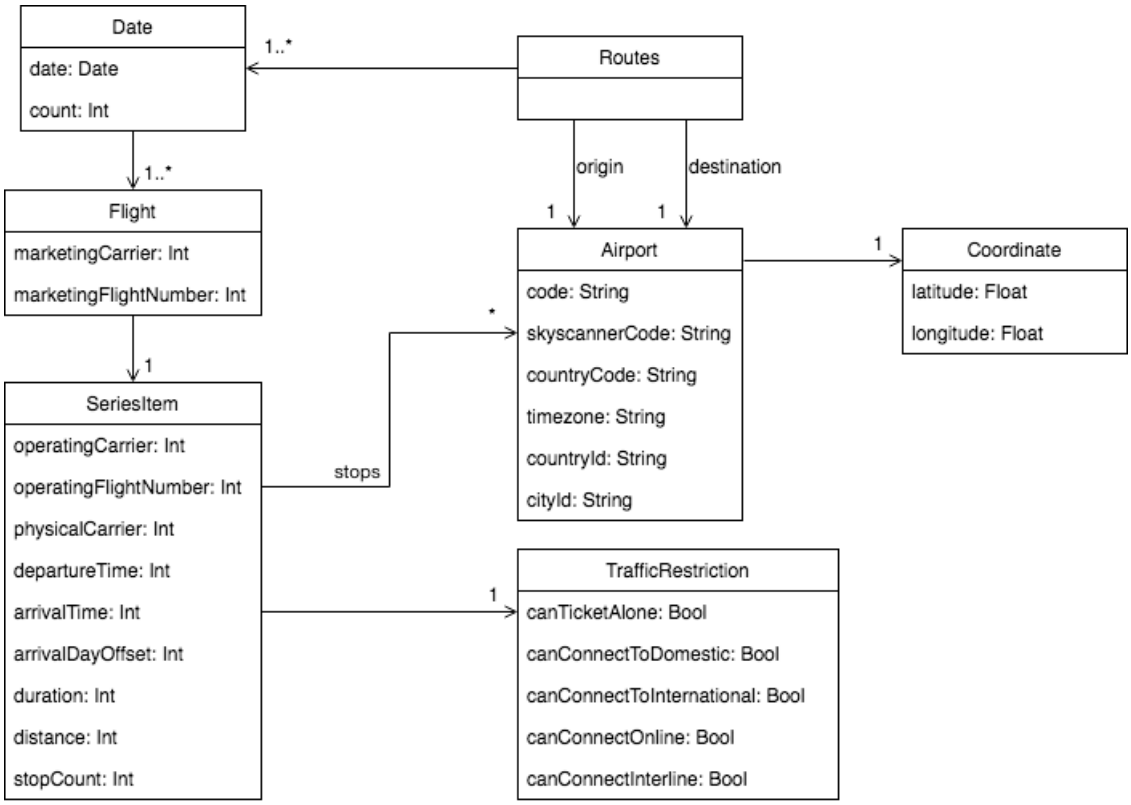



FIGURE 5.3: Flights Availability UML class diagram

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.

## 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 needed fields.

**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 are 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*.

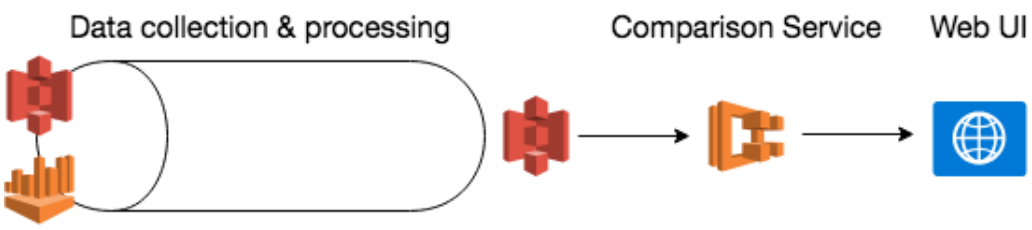



FIGURE 6.1: General view of the Skyscanner offer and demand comparison architecture.

### 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 ScalaScala[41]. 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 amazon web services (Amazon Athena and Amazon Simple Storage Service, S3). The whole data processing or data pipeline ran in Apache Spark, in an Amazon Elastic MapReduce 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

FIGURE 6.3: Amazon Athena logo

Amazon Athena[20], is an Amazon Web Service that lets query data stored in Amazon Simple Storage Service (S3) using standard SQL[21]. 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 is now the easiest way to access user searches data.

#### Amazon Simple Storage Service

FIGURE 6.4: Amazon Simple Storage Service logo

Amazon Simple Storage Service[22], 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 permission to buckets can be controlled its configuration.
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[23] 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™

FIGURE 6.5: Apache Spark™ logo

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[24].

Is one of the most popular fast and general-purpose cluster computing systems. It can run in a lot of different environments, Apache™ Hadoop®[25], Kubernetes[26], 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** are functions that creates a new RDD form an 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 are edges in the Apache Spark™'s Directed Acyclic Graph.

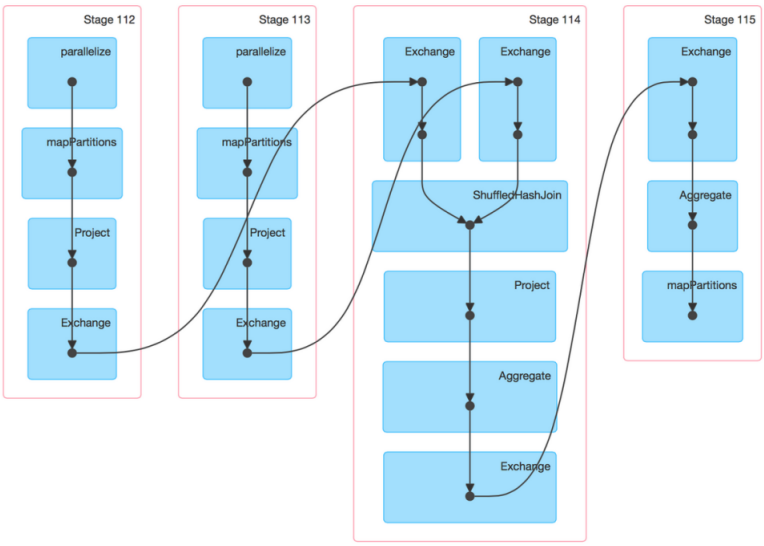



FIGURE 6.6: Example of a Directed Acyclic Graph (DAG).

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

FIGURE 6.7: Amazon Data Pipeline logo

Amazon Data Pipeline[27] 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[28] 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

FIGURE 6.8: Amazon Elastic MapReduce logo

Amazon Elastic MapReduce[29] (EMR) provides a Hadoop[25] 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 | `m3.xlarge` |
| **Cluster core instance type** | Instance type of the slaves | `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 ($2 \times 40$ GiB) |

TABLE 6.2: `m3.xlarge` specs

### 6.1.2 Service

The service is the middle layer of the Skyscanner offer and demand comparison. A simple Python[30] service 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 the Python[30] service is deployed.

#### Amazon Elastic Container Service

FIGURE 6.9: Amazon Elastic Container Service logo

Amazon Elastic Container Service, also known as ECS[31], is a container management service that supports Docker. It ease the container orchestration with a simple API.

#### Docker


FIGURE 6.10: Docker logo

Docker[32] is 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 is set, 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[23] 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 Strings, one record per line; so when reading DeLorean squad's file, it will get an RDD of Unified Routes in JSON[23] format.

For all data processing, the pipeline is split in well-defined stages. Each stage is a Singleton[1] that 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 Gson[48].
- **Flights Flattening stage**: Flats all routes' series at date level, duplicating origins and destinations of flights that 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 routes to 75,000,000 flights. The amount of data increases a lot in this point.
- **Group Routes stage**: After flattening at date level, flights are grouped by date. There is a lot of shuffling[2] which makes it a slow process.

Apart from these four stages, it has other to parse grouped routes by date objects to JSON[23] and another to write it to Amazon Simple Storage Service, but those are not necessary, both stages are disabled.

[1] Design pattern explained in Design Patterns on page 80.
[2] Sharing data between different nodes or machines

**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 Scala Amazon Web Service's Software Development Kit[47].

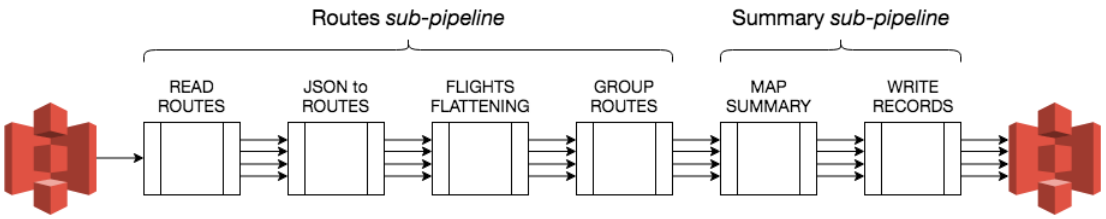



FIGURE 6.11: Available flights pipeline architecture and data flow diagram

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

**Amazon Relational Database Service**

FIGURE 6.12: Amazon Relational Database Service logo

In the beginning, the idea was to write all data to Amazon RDS, Relational Database Service[33]. 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**

FIGURE 6.13: Amazon DynamoDB logo

Another idea about the database where the pipeline could write was Amazon RDS, Relational Database Service[34]. 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: The **Cost**, it is very expensive to have an DynamoDB, even more expensive

than Amazon Relational Database Service.

Having this in mind, after some brainstorming, the final solution was found:

**S3 and dates at month level**

The main problem of this pipeline, that happens in the User Searches Pipeline as well, is that **versions are written every day** and does not delete anything. So it satisfies the 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 in a **single file**, but how about the reading from the service? Boto[3] 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, in the worst case scenario, there are 365 files to open and read (days in one year). 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.

The 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, Boto3[49] could run the `list_bucket` function, that list all objects under the given key. This key, must contain the basic model's information: `$ORIGIN-$DESTINATION/ $DATE/ $VERSION_DATE-$COUNT.json`

[3]Boto is the Amazon Web Services (AWS) SDK for Python, explained in Boto3[49] on page 47

```
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 this implementation, 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 slow and expensive because of the EMR cluster running time.

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

Finally, 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 **grouping flights by months** instead of having them by day. The final S3 directory structure would be:

```
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[35] in S3, each row had a record with a format similar to JSON[23]. Those records could not be parsed to an object using any library. Using RegEx[36] applied to rows treated as a String, the result were obtained.

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. 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** in total.

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. Four executions, one per hour range.
- **S3 Reader stage**: Gets the file name of the query result and reads it. CSV[35] to String RDD. Four executions too, one per file.
- **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 Scala Amazon Web Service's Software Development Kit[47].

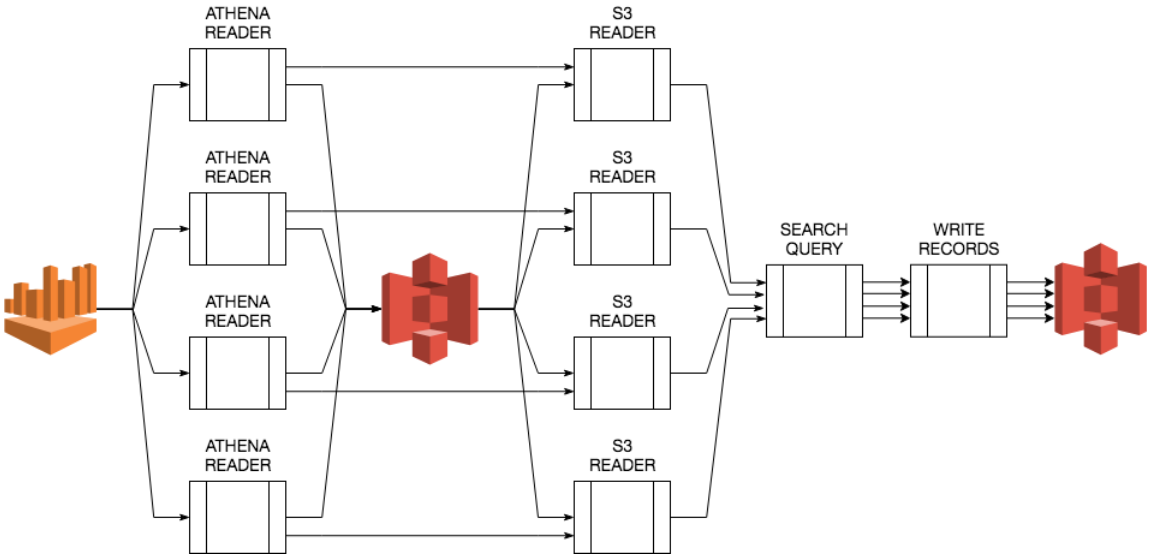



FIGURE 6.14: User searches pipeline architecture and data flow diagram.

### 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[37] API.

The Comparison Servie's API provides two endpoints, one for the comparison and another one for a single source request (flights or searches)[4]. The main class, API Handler, execute a function when the endpoint is queried and uses an AWS Adapter[5] that is directly connected to Boto3[49]. The API Handler returns a set of records in JSON format.

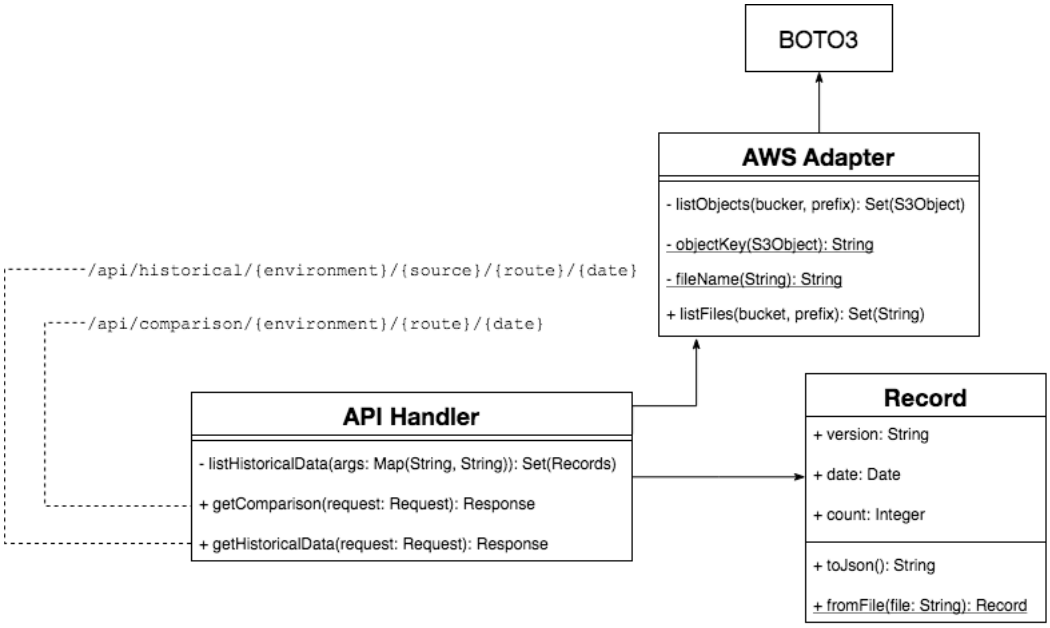



FIGURE 6.15: Comparison service class diagram.

**`/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[6]: All existent versions. The eight functional requirement of the Skyscanner offer and demand comparison is to provide an endpoint to get historical data, this endpoint satisfies the requirement.

Here is an example of the usage:

```
GET /api/historical/prod/searches/BCN-EDI/2018-10

RESPONSE
{
    queries: [
        {
            id: 1529050761587,
            name: "Searches BCN-EDI 10/2018",
```

[4]The *environment* field can be `sandbox` or `prod`. Learn more about those environments in section Environments on page 12.

[5]Design pattern explained in Design Patterns on page 80.

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

```
            source: "Searches",
            origin: "BCN",
            destination: "EDI",
            month: "10",
            year: "2018"
        }
    ],
    results: [
        {
            date: 1523404800000,
            version: "2018-04-11",
            count: 122,
            key: 1529050761587
        },
        {
            date: 1523491200000,
            version: "2018-04-12",
            count: 197,
            key: 1529050761587
        },
        {
            date: 1523577600000,
            version: "2018-04-13",
            count: 143,
            key: 1529050761587
        },
        ...
    ]
}
```

In this response, the results object contains an array of records, each one with the date in milliseconds, the date in human format (version), amount of searches or available flights (count), and a key that is the same for all the results and the same as the query. This data has enough detail for the developer start making some analysis.

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**

FIGURE 6.16: Amazon Lambda logo

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[38] 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[30] or Node.js[39] 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

FIGURE 6.17: Amazon API Gateway logo

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

Skyscanner reached the limit of API Gateways[40]. 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 comparison 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[7] becomes a responsive application with a high code scalability and maintainability.

[7] Design pattern explained in Design Patterns on page 80.

### Architecture

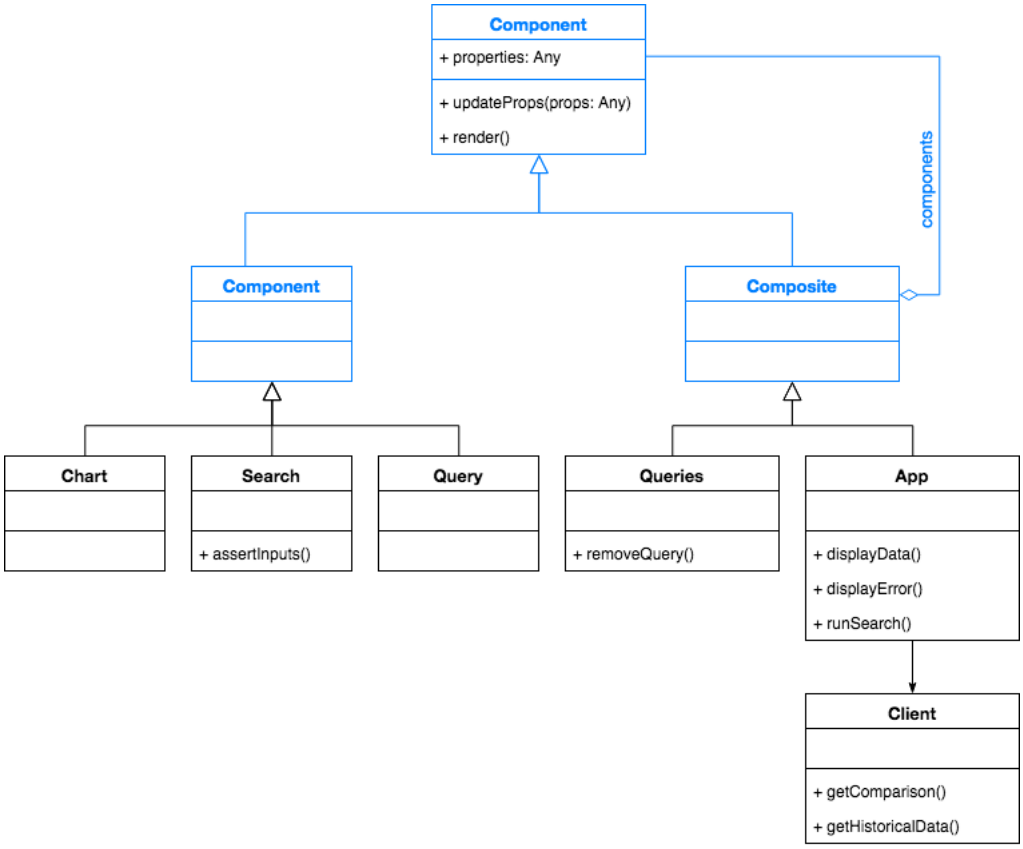



FIGURE 6.18: Web UI class architecture. In blue we can appreciate the Composite Pattern base classes and relations.

The main composite class of the website is `App`, it renders the header and give arguments and properties to the `Search`, `Queries` and `Chart` components. The `App` component also is connected to the `Client` class, it runs requests to the API provided by the Comparison Service.

The `Search` component display input fields and calls the `runSearch` function when the user clicks the *Search* button.

All the results will be shown in the `Chart` component and the queries in the `Queries` component, composed by `Query`; this will be used as a legend and will let the user remove queries from the chart.

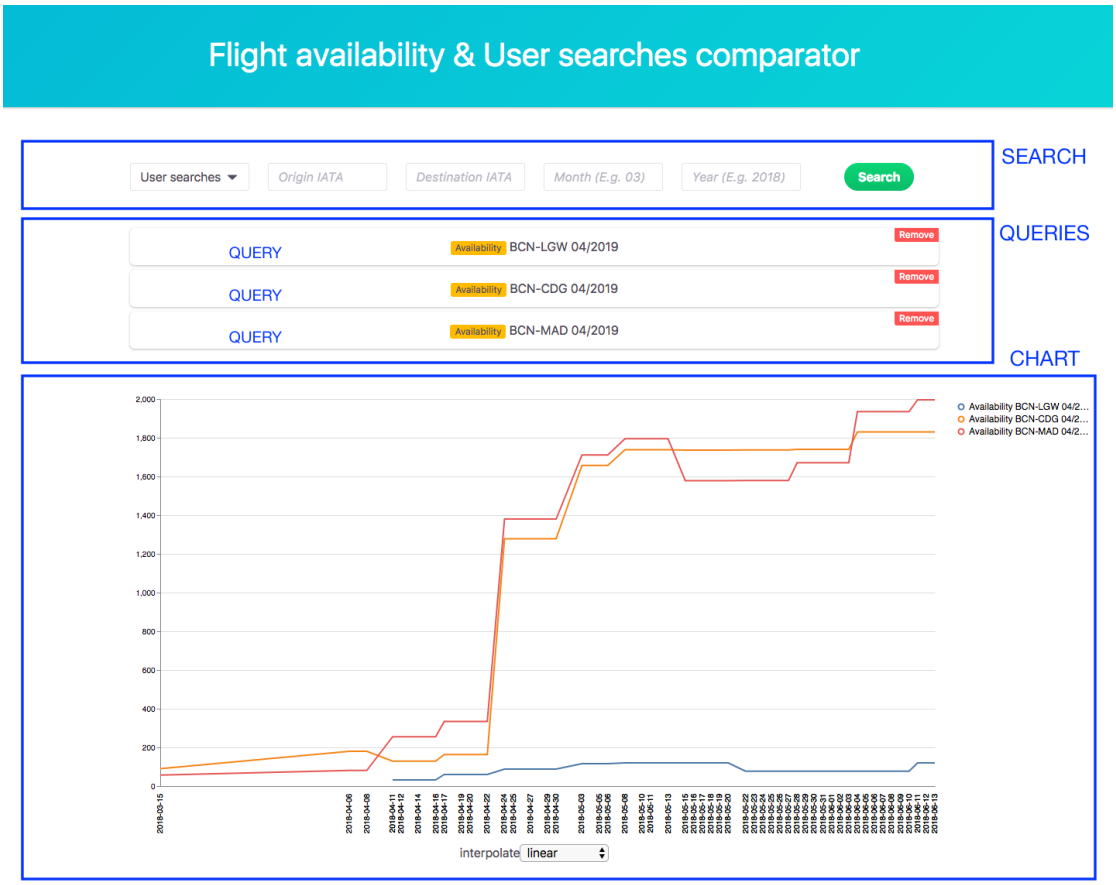



FIGURE 6.19: Web UI design.

### Requirements satisfaction

All functional requirements except eight must be satisfied in the Web UI. Let's review them and explain how they get satisfied:

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

   In the search component, the user can select available flights, set an origin, destination, month and year.


FIGURE 6.20: Demo of the search available flights functionality.

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

   In the search component, the user can select user searches, set an origin, destination, month and year.


FIGURE 6.21: Demo of the search user searches functionality.

3. **Search multiple availability values**: *Be able to search and show multiple availability values for different flights 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.*

   Once the user has searched for flight availability, it can search again and the results will appear in the chart.

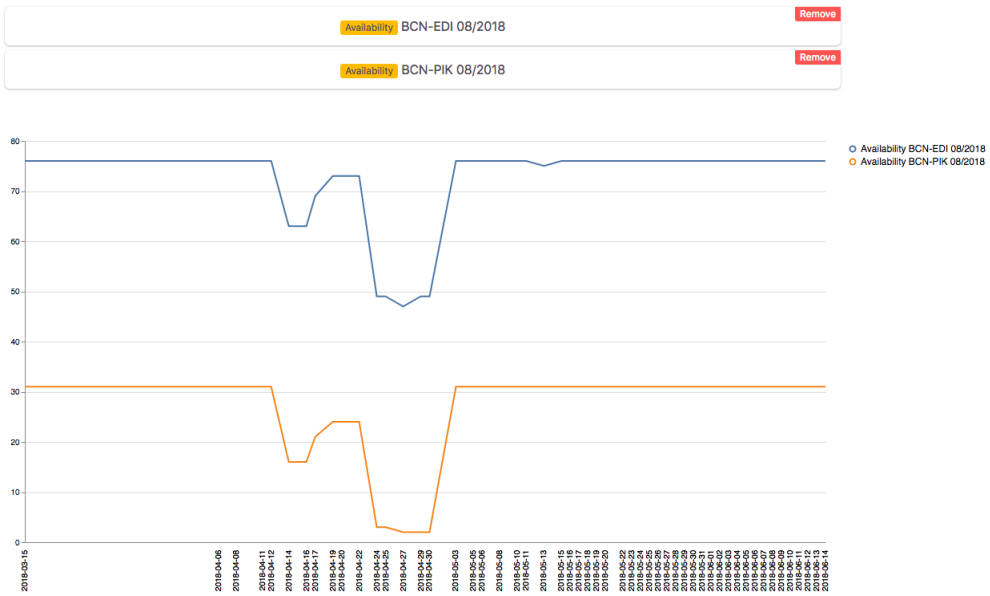



FIGURE 6.22: Demo of the search multiple available flights functionality.

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

   Once the user has searched for user searches, it can search again and the results will appear in the chart.

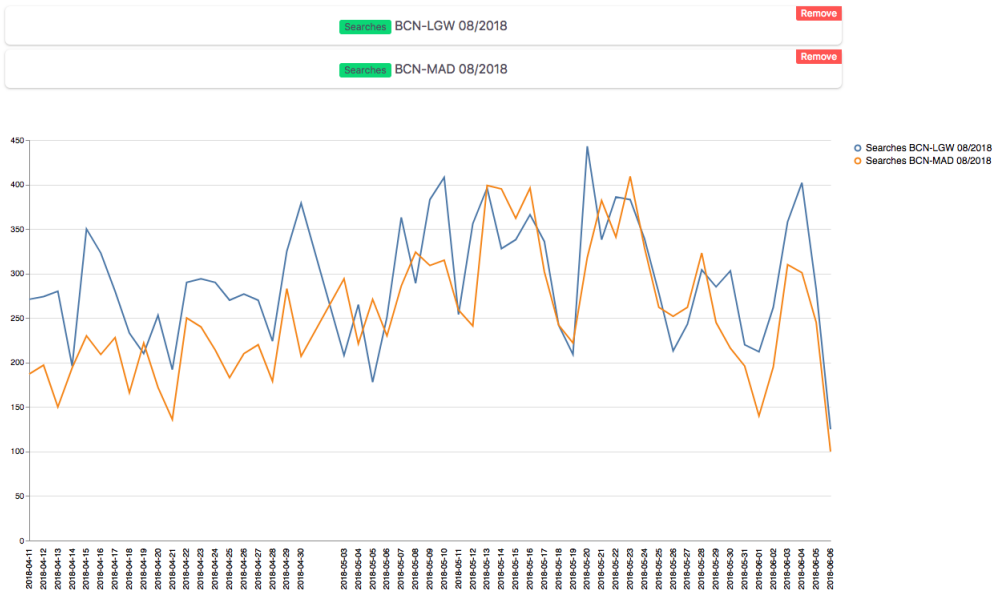



FIGURE 6.23: Demo of the search multiple user searches functionality.

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

   In the search component, the user can select the comparison field, set an origin, destination, month and year. It is the same as search first for available flights and then for user searches with the same parameters.


FIGURE 6.24: Demo of the search comparison functionality.

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

   Every time the user searches for data, the query is **added** to the chart.


FIGURE 6.25: Demo of the queries functionality.

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

   Each query has a remove button that removes it from the chart.


FIGURE 6.26: Demo of the remove queries functionality.

# Chapter 7

# Development

Once the system is designed and its architecture defined, the implementation took an important part of the time. The Skyscanner offer and demand comparison used a lot of frameworks and libraries that eased the development. Some of them has been already explained, like Apache Spark™ that offers an API to process big data dumps and it parallelizes the process automatically.

First of all, lets check the main programming languages used in this project and then all libraries and frameworks component by component, finally the tools used for the development and deployment.

## 7.1 Programming languages

### Scala[41]

FIGURE 7.1: Scala logo

Scala (`.scala` as file extension) is multi-paradigm programming language compiled by the Java Virtual Machine[42]. Its main paradigms are **functional** (very useful for Apache Spark™, Java[43] also provides this since version 8, but in Scala it is much easier to write and debug) object-oriented, imperative and concurrent.

Scala is the preferred language by the Apache Software Foundation for Apache Spark™, that is why it is the main language used in both pipelines, Available Flights Pipeline and User Searches Pipeline. The latest Apache Spark™ API is for Scala, then it usually comes out for Java and finally for Python.

I have never programmed in Scala before this project, but it was very fast and easy to learn. Knowing Java, the syntax is not much different and, usually, more intuitive.

### Python[30]

FIGURE 7.2: Python logo

Python (with file extension `.py`) is the language used for the service. Python 3.6 is the version used instead of 2.7, which is very different. Python2.7 will not have support anymore in a few months.

The syntax of python is different than C based languages and allows a lot of *tricks* that the developer cannot do in most programming languages, for instance: `a[-1]` gets the last element of the array `a`. It is the only language used in the Comparison Service, thanks to Python's flexibility; imperative, functional, object-oriented, procedure and reflective paradigms; duck, string and dynamic typing and compatibility with lost of technologies makes the service easy to be implemented in few lines.

### JavaScript[44]

Often abbreviated as JS, JavaScript is an interpreted programming language core of the World Wide Web technologies all along with HTML[45] and CSS[46]. It is used in the Web UI and, like Python, it has compatibility with a lot of technologies, axios[51], React.js[52] and Vega[10].


FIGURE 7.3: HTML, JS and CSS logos

## 7.2 External libraries and frameworks

### Scala Amazon Web Service's Software Development Kit[47]

Amazon provides a lot of APIs for different programming languages. The AWS SDK for Java enable Java developers to easily work with Amazon Web Services and build scalable solutions with Amazon S3, Amazon Athena, Amazon DynamoDB, and more. Since Scala is assembled by the Java Virtual Machine, Amazon created the Scala Amazon Web Services SDK, it's like the AWS SDK for Java, but more Scala-y.

Thanks to that Software Development Kit both pipelines can write to the Amazon Simple Storage Service and query Amazon Athena in the case of the User Searches Pipeline.

### Gson[48]

Most data is usually serialized in JSON[23] format, to deserialize it and get an Scala object, the Available Flights Pipeline uses Google's library Gson.

### Boto3[49]

This Python library is also very important for the project. It is the Amazon Web Service's Software Development Kit for Python. It is not as complete as the Java AWS SDK or the Scala AWS SDK but the Amazon Simple Storage Service API it provides is more than enough for the Comparison Service.

### aiohttp[50]

In order to have an Asynchronous HTTP Client and Service for Python, the Skyscanner offer and demand comparison is using AIOHTTP. One of its key features is the support to Server WebSockets.

### axios[51]

For the Web UI to call the Comparison Service, axios is a JavaScript library that provides a promise based HTTP client for the browser.

### React.js[52]

React is a set of libraries combined in a single framework for building user interfaces in JavaScript. Allows the developer create a Declarative and Component-Based interface. React is build a way that when a change happens it does not render all the page again, it only renders the component that has changed, making the website much faster than typical static pages.

### Vega[10]

Interactive Data Lab created Vega and Vega Lite, a library that allows create visualization designs using a declarative format. All visualizations are described in JSON[23] and Vega does all the work for you. The Web UI uses the Line Chart provided by Vega Lite.

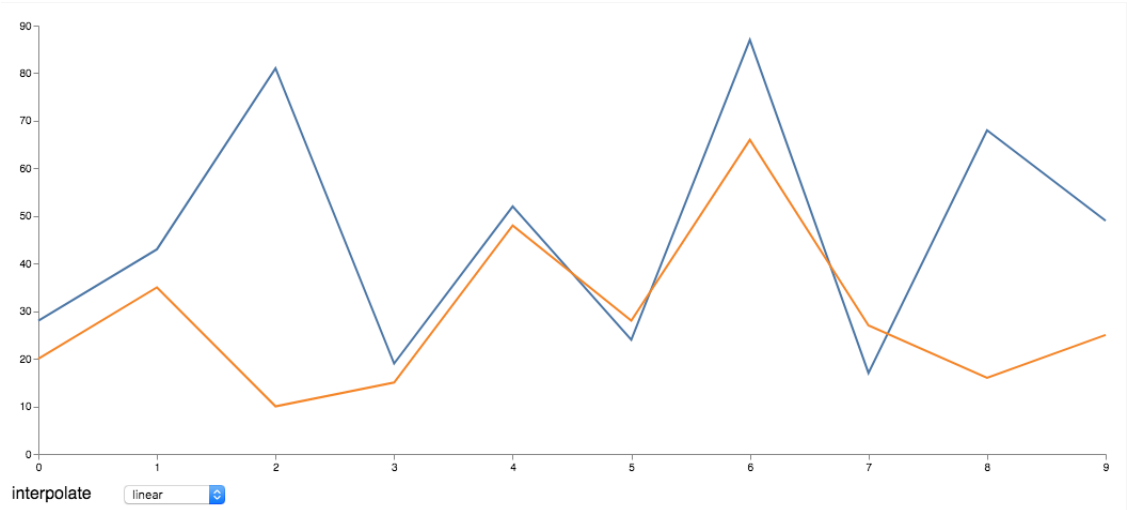



FIGURE 7.4: Vega Lite line chart example.

## 7.3 Developing tools

The two main tools used for the development of the Skyscanner offer and demand comparison are IntelliJ IDEA by JetBrains and Sublime Text 3.

### IntelliJ IDEA by JetBrains[53]

FIGURE 7.5: IntelliJ IDEA logo

*Capable and Ergonomic IDE for JVM*. IntelliJ is the preferred IDE[54] in Skyscanner. Ease the development of Java projects and, with a Scala Building Tool plugin IntelliJ eases the development of Scala projects. This Integrated Development Environment has been used for writing, testing, executing and assembling Available Flights Pipeline and User Searches Pipeline projects.

### Sublime Text 3[55]

FIGURE 7.6: Sublime Text 3 logo

Sublime Text is not an IDE, is just a text editor for code, markup and prose. Sophisticated and full of features for the developers, makes easier the development of software, systems and applications that do not have an specific IDE or that the existing development environment complicates the project. That is why Sublime Text 3 has been used for the Comparison Service and the Web UI.

## 7.4 Deployment

The deployment of the software is automatic when the new code is pushed to its `git` branch. Deploying to sandbox if the branch is a feature branch, deploying to prod if the branch is `master`.

This process is automatic thanks to Drone. This tool can deploy to Amazon Web Service if there is an stack available created by the Amazon CloudFormation. In order to track the deployment and scheduled executions of the pipelines, the project have Amazon Simple Notification Service configured.

### Drone

FIGURE 7.7: Drone logo

Drone[16] is an open source Continuous Delivery platform that automates your testing and release workflows. Every repository has a `.drone.yml` file where the release workflow is described. The GitLab repository is linked to Drone and when the code is pushed, Drone automatically runs the commands set for that specific `git` action.

### Amazon CloudFormation

FIGURE 7.8: Amazon CloudFormation logo

Cloud Formation[56] allows the developer create a AWS stack. Posting a configuration file describing all the resources it is going to use, Cloud Formation creates and link them, so it is easier to work with multiple resources at the same time.

Pipelines' cloud formation have similar parameters:

| **Field** | **Description** | **Value** |
|---|---|---|
| **Contact** | Main contact for this Stack | felix.arribas@skyscanner.net |
| **Project** | Project's name | heatmap |
| **Field** | **Description** | **Value** |
| **EMRClusterCore InstanceType** | Instance type of the master | `m3.xlarge` |
| **EMRClusterMaster InstanceType** | Instance type of the slaves | `m3.xlarge` |
| **EMRClusterSize** | Number of slaves clusters | 6 |
| **EMRRelease** | EMR version | `emr-5.12.0` |
| **PipelineKickOff** | Starting date of the data pipeline | 2018-04-12T16:00:00 |
| **Schedule** | Scheduling of the date pipeline | 1 day |

TABLE 7.1: Cloud Formation common parameters of both pipelines

Apart from these parameters, the Cloud Formation file also sets an Amazon Simple Notification Service that mail *Contact* when the pipeline succeeds or fails.

### Amazon Simple Notification Service

FIGURE 7.9: Amazon Simple Notification Service logo

Amazon Simple Notification Service[57], also known as SNS, is a flexible, fully managed pub/sub messaging and mobile notifications service for coordinating the delivery of messages to subscribing endpoints and clients. Not only for mobile notifications, it can notify other AWS resources, like Amazon Lambda, or send other kind of notifications, like **e-mails**.

# Chapter 8

# Results

How the system keeps running every day? After every day execution a notification is sent to the contact mail (felix.arribas@skyscanner.net). If something goes wring, is easy to find the logs in Amazon Web Services and create a ticket in the related project.

Right now the database does not have much data. Is very difficult to extract tendencies, there are only records from the last two months. Even so, there is enough to make some conclusions from the different comparisons.

There are some examples of usage of the Skyscanner offer and demand comparison.

### Experiment 1

What London airport people prefer when traveling from Barcelona?

There are six airports in London, searching user demand from Barcelona to those airports for traveling in October 2018 we get the following chart:

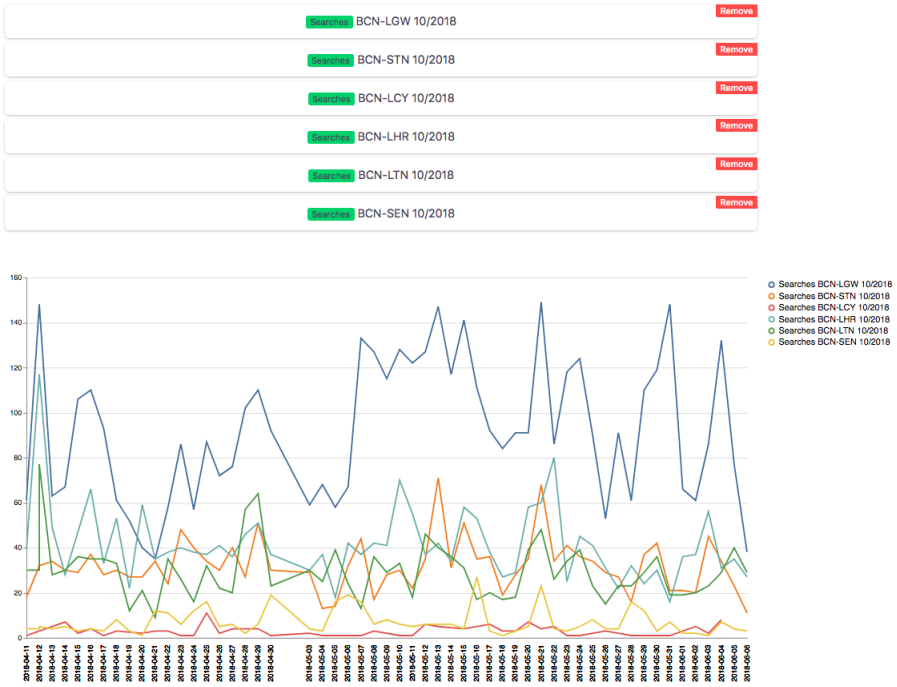



FIGURE 8.1: Searches from Barcelona to London Airports October 2018

From its six airports: City, Heathrow, Gatwick, Luton, Stansted and Southend; Heathrow is the London Airport with more passengers (78 million in 2017[58]) but Gatwick (45 million passengers in 2017) is the preferred airport when traveling from Barcelona.

### Experiment 2

When traveling from New York (JFK), the results are different:

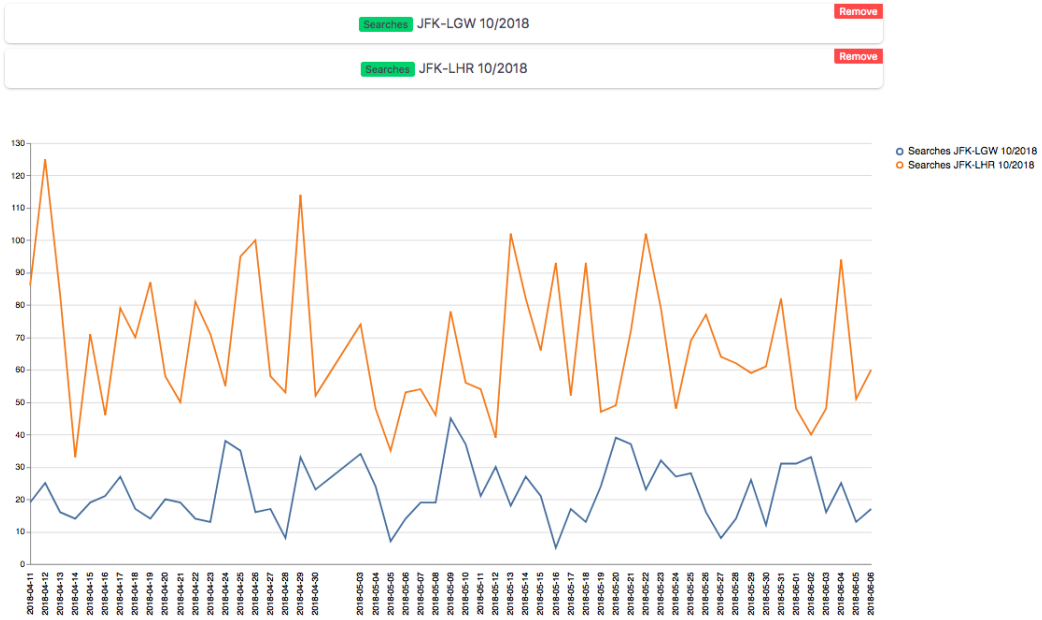



FIGURE 8.2: Searches from New York JFK to LHR and LGW October 2018

London Heathrow is searched more than London Gatwick. Provably because Heathrow usually operates longer flights than Gatwick airport.

# Chapter 9

# Project planning

Before defining tasks and time distribution of those, it is important to remember that this project is developed under an Agile Methodology, Extreme Programming. XP attempts to reduce costs of changes in requirements and long term plans, this means that this *time planning* is just an initial aproximation and has changed during the development. Every week a small iteration plan has been made in order to adjust the tasks with the scope and time of the project.

Even so, here is a general plan. Tasks has been **grouped regardless the technology**, for instance[59]: Task 27, App Component from the Web UI. It will have an HyperText Markup Language part only for the view and a Java Script part for getting and displaying server's data, but there is only one task for it.

Tasks are grouped in **Milestone**. Used to mark specific points or goals along of a project timeline. Milestone are very useful in order to do not losing time perception in long software projects when using Agile Methodologies.

Another important thing to take into account is the **tasks overlapping**. Is a good practice[60] not to work to much time in the same thing, task or goal. Swapping between different tasks and milestone can help the developer to **not obfuscate**.

## 9.1 Tasks

All development tasks (from number 14 to 30), are composed by small cycles where the developer designs, codes, tests, refactors and deploys the software. In every task the developer loops through these cycles, as explained in Methodology and rigor section.

Development tasks duration is specified in days. Weeks are composed by five working days, so 5 days equals to a week. In a day, the developer is supposed to work five hours. From week seven to week twenty-three, there are fifteen working weeks (not counting holidays[1]). Doing simple maths, the development will take:

$$15 \text{ weeks} \times 5 \frac{\text{days}}{\text{week}} \times 5 \frac{\text{hours}}{\text{day}} = 375 \text{ hours}$$

**375 hours** of development. Adding the small inception of the beginning and the project report in the end, 8 weeks, The whole project will take a total of **575 hours**.

[1]There are two holiday weeks in March, one is *Holy week* (13th), and the other is extra. That is why the project starts earlier.

It is important to understand that similar tasks like 16th and 20th or 17th and 21st have different times, that is because the second time it is supposed to be very similar than the first, so there will be some previous knowledge when executing them for second time.

### 9.1.1 Inception

Regard this an agile project, there has been an small inception part where the project was predefined and explained to Skyscanner product owners to see if it was viable.

| | |
|---|---|
| **Milestone name** | Inception |
| **Number** | 1 |
| **Description** | Identify the initial scope of the project, stakeholders, context and environment where the project will be developed. |
| **End date** | 9th of February, 2018 |

| | |
|---|---|
| **Name** | Definition of the problem |
| **Number** | 2 |
| **Description** | Defining, at a high level, what the system will do. |
| **Duration** | 10 days |
| **Milestone** | Inception |
| **Dependencies** | – |

| | |
|---|---|
| **Name** | Scope |
| **Number** | 3 |
| **Description** | What the software project will do and will not. Not list. |
| **Duration** | 5 days |
| **Milestone** | Inception |
| **Dependencies** | 2: Definition of the problem |

| | |
|---|---|
| **Name** | Risks |
| **Number** | 4 |
| **Description** | Find possible problems and obstacles may appear in the future development. |
| **Duration** | 8 days |
| **Milestone** | Inception |
| **Dependencies** | 3: Scope |

| | |
|---|---|
| **Name** | Scope refinement |
| **Number** | 5 |
| **Description** | After finding the risks, review the scope. Make it possible. |
| **Duration** | 4 days |
| **Milestone** | Inception |
| **Dependencies** | 4: Risks |

| | |
|---|---|
| **Name** | Environment |
| **Number** | 6 |
| **Description** | Find the correct environment in order to build the project. |
| **Duration** | 2 days |
| **Milestone** | Inception |
| **Dependencies** | 5: Scope refinement |

### 9.1.2 Project management

| | |
|---|---|
| **Milestone name** | Project management (GEP)[61] |
| **Number** | 7 |
| **Description** | First stage in the TFG. Get thesis started. |
| **End date** | 20th of April, 2018 |

| | |
|---|---|
| **Name** | Context and scope |
| **Number** | 8 |
| **Description** | Indicate general objective of the TFG and context. |
| **Duration** | 12 days |
| **Milestone** | Project management |
| **Dependencies** | – |

| | |
|---|---|
| **Name** | Project planning |
| **Number** | 9 |
| **Description** | Planning of the entire execution of the TFG. |
| **Duration** | 4 days |
| **Milestone** | Project management |
| **Dependencies** | 8: Context and scope |

| | |
|---|---|
| **Name** | Budget and sustainability |
| **Number** | 10 |
| **Description** | Explanation of the sustainability of the project. Economical, social and environmental. |
| **Duration** | 5 days |
| **Milestone** | Project management |
| **Dependencies** | 9: Project planning |

| | |
|---|---|
| **Name** | First oral presentation |
| **Number** | 11 |
| **Description** | Three minute oral presentation on video. |
| **Duration** | 10 days |
| **Milestone** | Project management |
| **Dependencies** | 10: Budget and sustainability |

| | |
|---|---|
| **Name** | Competences review |
| **Number** | 12 |
| **Description** | Review of the competences of the bachelor's thesis. |
| **Duration** | 5 days |
| **Milestone** | Project management |
| **Dependencies** | – |

| | |
|---|---|
| **Name** | Final document |
| **Number** | 13 |
| **Description** | Project management, Project management and Project management. Reviewed. |
| **Duration** | 5 days |
| **Milestone** | Project management |
| **Dependencies** | 11: First oral presentation |

### 9.1.3 Available flights pipeline

| | |
|---|---|
| **Milestone name** | Available flight pipeline |
| **Number** | 14 |
| **Description** | Application that maps the provider data to the desired data model. |
| **End date** | 16th of March, 2018 |

| | |
|---|---|
| **Name** | Reading from DeLorean's S3 |
| **Number** | 15 |
| **Description** | Connect to DeLorean's application that gets all routes and read from it |
| **Duration** | 5 days |
| **Milestone** | Available flights pipeline |
| **Dependencies** | – |

| | |
|---|---|
| **Name** | Data processing |
| **Number** | 16 |
| **Description** | Filter and map all data in DeLorean's model to desired for the |
| **Duration** | 10 days |
| **Milestone** | Available flights pipeline |
| **Dependencies** | 15: Reading from DeLorean's S3 |

| | |
|---|---|
| **Name** | Writing into S3 |
| **Number** | 17 |
| **Description** | Once the data is processed, write into the S3 bucket. |
| **Duration** | 10 days |
| **Milestone** | Available flights pipeline |
| **Dependencies** | 16: Data processing |

### 9.1.4 User searches pipeline

| | |
|---|---|
| **Milestone name** | User searches pipeline |
| **Number** | 18 |
| **Description** | Application that maps the user data to the desired data model. |
| **End date** | 27th of April, 2018 |

| | |
|---|---|
| **Name** | Reading from Athena |
| **Number** | 19 |
| **Description** | Find correct table and read from Athena service. |
| **Duration** | 10 days |
| **Milestone** | User searches pipeline |
| **Dependencies** | – |

| | |
|---|---|
| **Name** | Data processing |
| **Number** | 20 |
| **Description** | Filter and map all data in Data Tribe's model to desired for the Skyscanner offer and demand comparison |
| **Duration** | 5 days |
| **Milestone** | User searches pipeline |
| **Dependencies** | 19: Reading from Athena |

| | |
|---|---|
| **Name** | Writing into S3 |
| **Number** | 21 |
| **Description** | Once the data is processed, write into the S3 bucket |
| **Duration** | 5 days |
| **Milestone** | User searches pipeline |
| **Dependencies** | 20: Data processing |

### 9.1.5 Comparison server

| | |
|---|---|
| **Milestone name** | Comparison service |
| **Number** | 22 |
| **Description** | Web server that provides all data processed by pipelines |
| **End date** | 19th of May, 2018 |

| | |
|---|---|
| **Name** | Reading historical data |
| **Number** | 23 |
| **Description** | Read from S3 |
| **Duration** | 5 days |
| **Milestone** | Comparison server |
| **Dependencies** | 17: Writing into S3 |

**Name** Comparison
**Number** 24
**Description** Read from both sources
**Duration** 5 days
**Milestone** Comparison server
**Dependencies** 17: Writing into S3, 20: Writing into S3

**Name** Deployment and Error control
**Number** 25
**Description** S3 can generate a lot of errors, have a control over them.
**Duration** 10 days
**Milestone** Comparison server
**Dependencies** 23: Reading historical data, 24: Comparison

### 9.1.6 Web UI

**Milestone name** Web UI
**Number** 26
**Description** Website with a visual representation of the data.
**End date** 8th of June, 2018

**Name** App component
**Number** 27
**Description** Main component with API calls to the Service.
**Duration** 5 days
**Milestone** Web UI
**Dependencies** –

**Name** Search component
**Number** 28
**Description** Search inputs for querying the service.
**Duration** 5 days
**Milestone** Web UI
**Dependencies** 27: App component

**Name** Chart component
**Number** 29
**Description** Get data from service and display it in the chart.
**Duration** 10 days
**Milestone** Web UI
**Dependencies** 23: Reading historical data, 24: Comparison, 28: Search component

| | |
|---|---|
| **Name** | Queries view |
| **Number** | 30 |
| **Description** | List of queries . |
| **Duration** | 5 days |
| **Milestone** | Web UI |
| **Dependencies** | 28: Search component |

### 9.1.7 Final presentation

| | |
|---|---|
| **Milestone name** | Final presentation |
| **Number** | 31 |
| **Description** | TFG whole report document and prepare the final presentation. |
| **End date** | 22nd of June, 2018 |

## 9.2 Current plan and alternatives

### 9.2.1 Current plan

In the current plan, the whole system is developed with two data layers, a presentation layer and a domain layer in the middle. Data layer composed by Available flights pipeline and User searches pipeline, domain layer by Comparison server and presentation by Web UI.

In this project the important part is the source of the data. So the most important part is to obtain the data correctly, then we can worry about its visual representation.

Why the Available flights pipeline comes before the User searches pipeline? The answer is simple: Available flights pipeline gets data from DeLorean squad, my squad. I know how their system works and, if all initial problems are found in the beginning, the consequences will be softer.

Available flights and user searches pipelines $\longrightarrow$ Comparison server $\longrightarrow$ Web UI

### 9.2.2 Alternative: Overlap pipelines

In case the project takes too much time, both pipelines can be totally overlapped. It will be possible because the process of what both pipelines do are the same but with different sources. There are **no dependencies** between them.

The first stage of the pipelines, those are reading from their source. Then, both pipelines map the data to the desired model. Finally the pipelines write in to the service database.

This will reduce the number of weeks from fifteen to **eleven weeks**. All milestone User searches pipeline would be done at the same time as milestone Available flights pipeline.

### 9.2.3 Alternative: Service reading from Data Tribe

Another alternative, is to remove the second pipeline. It is known that in order to get all flights, data must be flattened and, then, processed. This cannot be done in the service, it would overflow its heap memory. This means that the Available flights pipeline cannot be removed.

User searches pipeline will read data from a database that has the data already flattened. It has a lot of not necessary information, but filtering from a single record is faster than exploding and consumes much **less memory**. The service could do the users search data processing.

One problem is that the server latency will increase a lot. It will be reading from a very slow database for single queries (Data Tribe's database work very well when querying big amounts of data, but not small ones). Also, Data Tribe people will provably complain, because their service is not created for to much requests per second and is exactly what will happen if the services maps their data without a pipeline in the middle.

This option reduces the development time to **ten weeks** instead of fifteen.

## 9.3 Gantt

| | Task Name | Duration (days) | Start | *End* |
|---|---|---|---|---|
| 1 | Inception | 29 | 02.01.18 | 09.02.18 |
| 2 | Definition of the problem | 10 | 02.01.18 | 15.01.18 |
| 3 | Scope | 5 | 16.01.18 | 22.01.18 |
| 4 | Risks | 8 | 23.01.18 | 01.02.18 |
| 5 | Scope refinement | 4 | 02.02.18 | 07.02.18 |
| 6 | Environment | 2 | 08.02.18 | 09.02.18 |
| 7 | Project Management | 45 | 19.02.18 | 20.04.18 |
| 8 | Context and scope of the project | 12 | 19.02.18 | 06.03.18 |
| 9 | Project planning | 4 | 07.03.18 | 12.03.18 |
| 10 | Budget and sustainability | 5 | 13.03.18 | 19.03.18 |
| 11 | First oral presentation | 10 | 20.03.18 | 02.04.18 |
| 12 | Competences reveiw | 5 | 03.04.18 | 09.04.18 |
| 13 | Final document | 5 | 03.04.18 | 09.04.18 |
| 14 | Available Flights pipeline | 25 | 12.02.18 | 16.03.18 |
| 15 | Reading from DeLorean's S3 | 5 | 12.02.18 | 16.02.18 |
| 16 | Data processing | 10 | 19.02.18 | 02.03.18 |
| 17 | Writing into S3 | 10 | 05.03.18 | 16.03.18 |

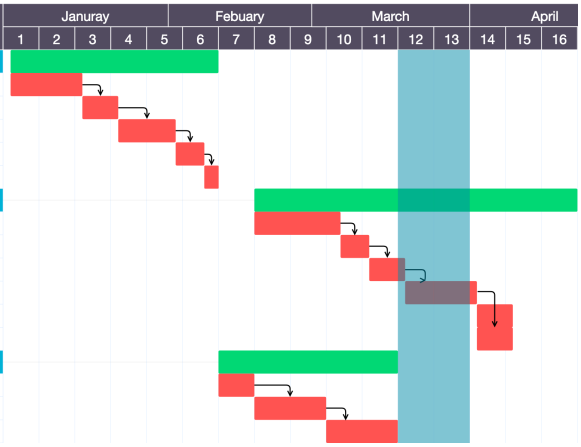



FIGURE 9.1: Gantt diagram from task 1 to 17

| | Task Name | Duration (days) | Start | *End* |
|---|---|---|---|---|
| 18 | User searches pipeline | 20 | 02.04.18 | 27.05.18 |
| 19 | Reading from Athena | 10 | 02.04.18 | 13.04.18 |
| 20 | Data processing | 5 | 16.04.18 | 20.04.18 |
| 21 | Writing into S3 | 5 | 23.04.18 | 27.04.18 |
| 22 | Comparison Service | 20 | 23.04.18 | 18.05.18 |
| 23 | Reading historical data | 5 | 23.04.18 | 27.04.18 |
| 24 | Comparison | 5 | 30.04.18 | 04.05.18 |
| 25 | Deployment & Error control | 10 | 07.05.18 | 18.05.18 |
| 26 | Web UI | 25 | 07.05.18 | 08.06.18 |
| 27 | App component | 5 | 07.05.18 | 11.05.18 |
| 28 | Search component | 5 | 14.05.18 | 18.05.18 |
| 29 | Chart component | 10 | 21.05.18 | 01.06.18 |
| 30 | Queries component | 5 | 04.06.18 | 08.06.18 |
| 31 | Final presentation | 10 | 11.06.18 | 22.06.18 |

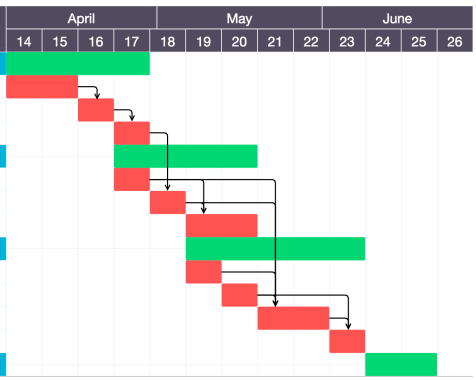



FIGURE 9.2: Gantt diagram from task 18 to 31

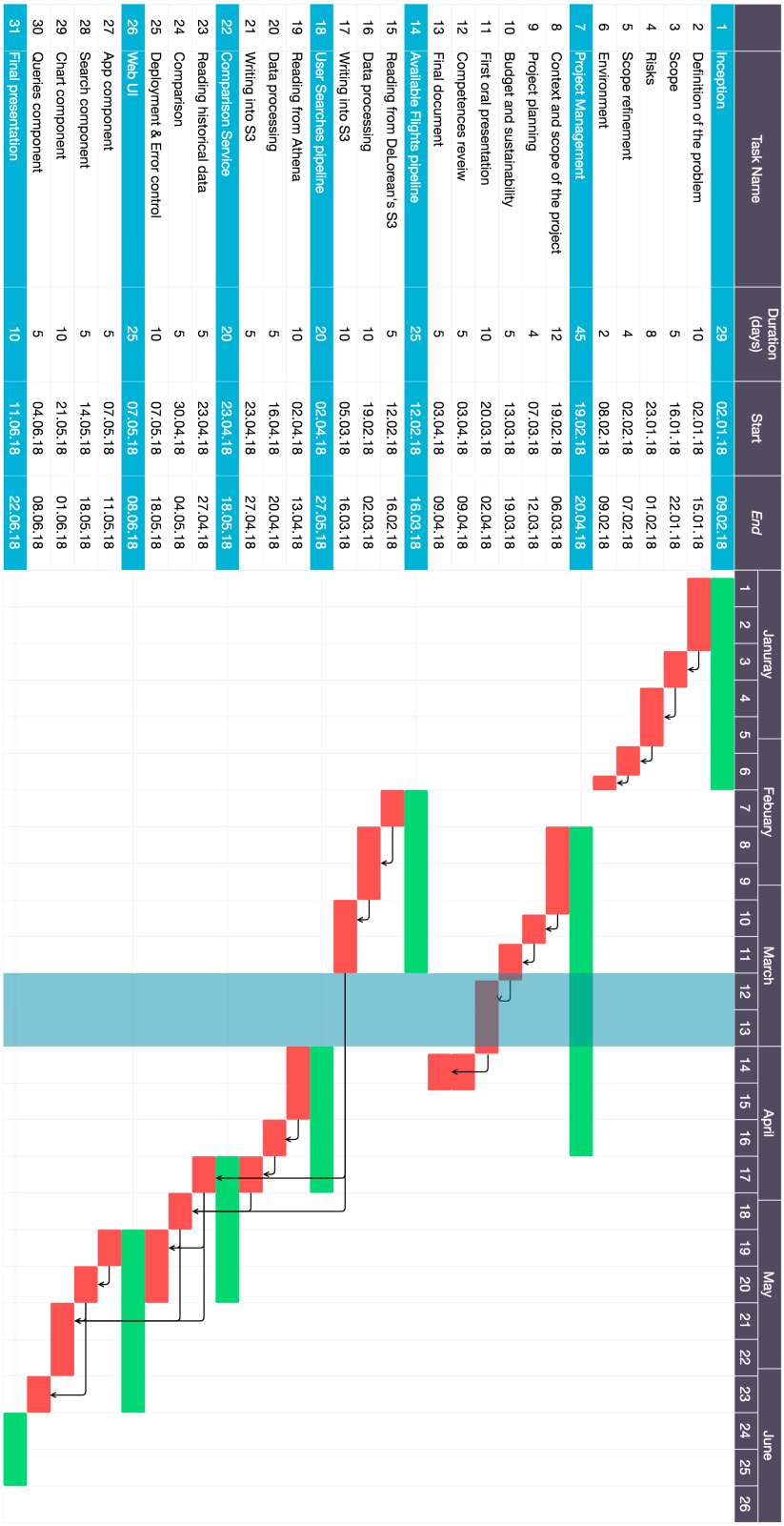



FIGURE 9.3: Complete Gantt diagram

# Chapter 10

# Budget and Sustainability

## 10.1 Budget

Once the project is planned in time and its technologies are drafted, we can calculate the project's budget. Skyscanner is not economically transparent, even for the employees. So, this whole calculation will be approximated.

First of all, we have to take into account the employees, which is only one and in Intern position, and also count the taxes.

Apart from that, all the hardware material and software licenses. AWS costs will continue increasing after the deployment, but since it is going to be used during the development for testing it is counted in the total budget.

| Concept | Price per unit | Units | Amortization | Total £ |
|---|---|---|---|---|
| **Salary** | 28 (taxes included) | 575 h | | 16100 |
| **MacBook Pro** | 1700 | 1 | life cycle 8 years | 210 |
| **JetBrains License** | 230 | 1 | 1 year per developer | 115 |
| **AWS S3** | 0.023 | 50 TB | | 1150 |
| **AWS** | EC2 | 575 h | | 213.325 |
| **Screen** | 200 | 2 | life cycle 4 years | 100 |
| **Office** | Uknown | 1 | | Uknown |
| **TOTAL** | | | | **17888.325** |

TABLE 10.1: Budget calculation

## 10.2 Sustainability

### 10.2.1 Economical

In economic terms, this project is initially unsustainable. It uses resources from Skyscanner for a comparison that might be useful in the future for other projects, improving some services or advertisement.

But, if this product is sell to providers, Skyscanner can take a lot of profit from them. It is a very valuable application for providers, since they could compare airlines offer with actual user demand. Letting them improve their flights distribution and make more money.

### 10.2.2 Social

The Skyscanner offer and demand comparisonis not directly involving society, but, as explained before, if providers have access to the comparison, flights will improve in terms of traveler experience. Travelers will have accurate routes depending on what they really want.

For example, imagine that X carrier have several flights from BCN to ORY, Paris, and a few from BCN to FCO, Rome. The Skyscanner offer and demand comparisonshows that the demand, compared with the offer is bigger in Rome than in Paris. Then, X airline could schedule more flights to FCO instead of ORY.

### 10.2.3 Environment

The environmental impact of the Skyscanner offer and demand comparisonis directly related with the social impact.

Right now, some airlines may have half full flights. This means that the airplane is not taking its most advantage of the fuel. It could be carrying more people.

If carriers know where flights are really needed those flight will be full of people, which means that the fuel a flight uses is profited at its most.

Otherwise, if an offer is under requested, the flight is not giving all the profit it could. In other words, fuel per person will decrease.

### 10.2.4 Sustainability matrix

In order to understand the general impact of the project, the following general rating and evaluation is provided:

The economical impact will be rated in 7/10. It could be a 10/10 it is sell to providers. Air companies could pay a lot of money for the Skyscanner offer and demand comparisonbecause of the information it provides.

The social impact will get a 4/10. It does nothing good nor bad to the society, only if the application is sell to providers and they use it properly, it could make some good to the people. In the other hand, the software will not be free, it will be property of Skyscanner.

The environmental impact is a 4/10 as well. The environmental impact could be good if the application is sell to providers and they use it properly, but for now will not be sell to anyone. It gets a 4 because it will be using Amazon Web Service, and those machines are powered mainly by non-renewable energy[62].

| Economical | Social | Environmental |
|---|---|---|
| 7/10 | 4/10 | 4/10 |
| 15/30 | | |

TABLE 10.2: Sustainability matrix

# Chapter 11

# Conclusions

After almost six months of the development of a full stack project, and after the whole report of the Skyscanner offer and demand comparison, there are some conclusions.

## 11.1 Results

Sadly, results are not the expected, maybe in the beginning the idea was too optimistic thinking that it would be easy to make assumptions when comparing available flights and user searches.

From the beginning it was known that one value was for available flights without counting seats of the aircraft or actual **available seats**, and the other one were searches. Here is an example of why the comparison is not as good as expected:

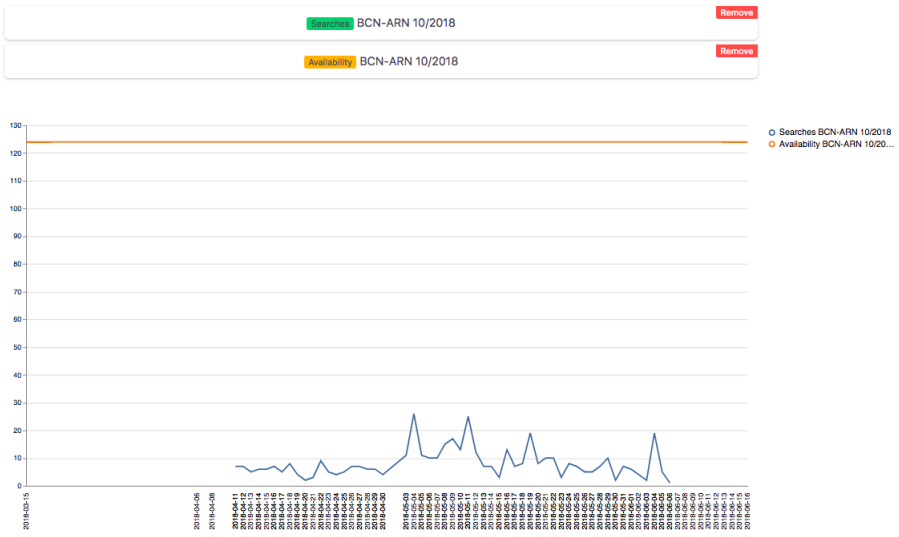



FIGURE 11.1: Comparison from Barcelona to Stockholm, October 2018

Sometimes, data is not what looks like.

Luckily, same source comparisons can give relevant results, see Experiment 1 on page 51 and Experiment 2 on page 52.

Apart from that issue, all functional and non-functional requirements are fully satisfied. **The project is completed and works fine**.

## 11.2 Development

The most valuable thing of this project is the knowledge earned during the making of. Not only as a programmer, but as a Software Engineer.

#### Inception

In the beginning looked impossible to find stakeholders and define the project. Being new in the company, was kind of intimidating start talking to product owners. Thanks to my Squad Lead I met them and defining something promising, he set some meetings with product owners he already knew and I end up doing **elevator pitch**es to them.

Now that I am in the end of the project and I know more people and how things work inside of Skyscanner, it would be much easier for me to do that process. I would not be afraid of talking to some Product Owners. I can say that **I learned how to sell a software idea** to anyone, in this case inside the same company.

#### Ping pong

When specifying and developing the Skyscanner offer and demand comparison, if I had some question, I have done ***ping pong*** between a lot of people. The process is usually the same: You ask something to someone you think they know the answer, but they do not. If you are lucky, they send you to someone that actually knows the answer. But what usually happens next is that they send you to someone they think they know the answer, but they do not neither. The process repeats and repeats until you find the answer.

#### Architecture

When designing the architecture, I learned a lot. There has been a lot of analysis to different technologies, a lot of alternatives has been studied and discarded, finding a final solution.

#### Methodology

The agile methodology has been great, but there has been some big blockers that was caused because of the methodology of the company: It took one month to get access to Athena, the ticket I opened to get access to it took a lot of time to be validated and processed.

Sometimes is good to work in a big company, because you have a lot of resources and in a start up you cannot get, but for cases like these, you have less *freedom*.

## 11.3 Final conclusion

It has been great to work in a project of these dimensions. From the beginning until the end.

I have learned a lot of great development practices, used different technologies and combined them, taken risks always into account and applied different strategies for them and delivered a satisfactory product. I brought something good for Skyscanner.

*The end.*

Special thanks to **Gary Fernie**, Javier Arias, Francisco Lopez, Esteve Julià, José Durães, Susana Carrera, Jen Agernton

# List of Figures

# List of Tables

# Bibliography


[1] 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).

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

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

[4] Google Inc. *Google Flights*. 2018. URL: https://www.google.com/flights/ (visited on 06/15/2018).

[5] Google Inc. *Material Design*. 2018. URL: https://material.io/design/ (visited on 06/16/2018).

[6] KAYAK Software Corporation. *Kayak*. 2018. URL: https://www.kayak.co.uk/ (visited on 06/15/2018).

[7] Expedia Group. *Expedia*. 2018. URL: https://www.expedia.es/ (visited on 06/15/2018).

[8] 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[34] Amazon Web Services Inc. *Amazon DybamoDB*. 2018. URL: https://aws.amazon.com/dynamodb/ (visited on 06/12/2018).

[35] 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).

[36] Multiple authors. *Regular expression*. 2018. URL: https://en.wikipedia.org/wiki/Regular_expression (visited on 06/16/2018).

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

[38] Amazon Web Services Inc. *Amazon Lambda*. 2018. URL: https://aws.amazon.com/lambda/ (visited on 06/12/2018).

[39] Node.js Foundation. *Node.js*. 2018. URL: https://nodejs.org/en/ (visited on 06/16/2018).

[40] Amazon Web Services Inc. *Amazon Lambda*. 2018. URL: https://aws.amazon.com/api-gateway/ (visited on 06/12/2018).

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

[42] Multiple authors. *Java Virtual Machine*. 2018. URL: https://en.wikipedia.org/wiki/Java_virtual_machine (visited on 06/14/2018).

[43] Multiple authors. *Java*. 2018. URL: https://en.wikipedia.org/wiki/Java_(programming_language) (visited on 06/14/2018).

[44] w3school. *JavaScript Tutorial*. 2018. URL: https://www.w3schools.com/js/ (visited on 06/14/2018).

[45] w3school. *HTML Tutorial*. 2018. URL: https://www.w3schools.com/html/ (visited on 06/14/2018).

[46] w3school. *CSS Tutorial*. 2018. URL: https://www.w3schools.com/css/ (visited on 06/14/2018).

[47] David Murray. *aws-scala-sdk*. 2018. URL: https://github.com/awslabs/aws-scala-sdk (visited on 06/14/2018).

[48] Google. *Serialization/deserialization, Java Objects - JSON*. 2018. URL: https://github.com/google/gson (visited on 06/14/2018).

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

[50] Aiohttp contributors. *Welcome to AIOHTTP*. 2018. URL: https://aiohttp.readthedocs.io/en/stable/ (visited on 06/12/2018).

[51] axios. *axios*. 2018. URL: https://github.com/axios (visited on 06/14/2018).

[52] Facebook Inc. *React*. 2018. URL: https://reactjs.org/ (visited on 06/14/2018).

[53] Jet Brains. *IntelliJ IDEA*. 2018. URL: https://www.jetbrains.com/idea/ (visited on 06/14/2018).

[54] Multiple authors. *Integrated development environment*. 2018. URL: https://en.wikipedia.org/wiki/Integrated_development_environment (visited on 06/14/2018).

[55] Sublime HQ Pty Ltd. *A sophisticated text editor for code, markup and prose*. 2018. URL: https://www.sublimetext.com/ (visited on 06/14/2018).

[56] Amazon Web Services Inc. *Amazon CloudFormation*. 2018. URL: https://aws.amazon.com/cloudformation/ (visited on 06/16/2018).

[57] Amazon Web Services Inc. *Amazon Simple Notification Service*. 2018. URL: https://aws.amazon.com/sns/ (visited on 06/16/2018).

[58] Civil Aviation Authority. *Airport data 2017*. 2018. URL: https://www.caa.co.uk/Data-and-analysis/UK-aviation-market/Airports/Datasets/UK-Airport-data/Airport-data-2017/ (visited on 06/17/2018).

[59] W3.CSS Website Templates. *Some responsive W3.CSS website templates to use.* 2018. URL: https://www.w3schools.com/w3css/w3css_templates.asp (visited on 03/12/2018).

[60] Robert C. Martin. *The Clean Coder. A Code of Conduct for Professional Programmers*. Prentice Hall, 2011.

[61] Rubrics in English. *GEP rubrics in English*. 2018. URL: http://atenea.upc.edu/mod/folder/view.php?id=1605178 (visited on 03/12/2018).

[62] Greenpeace. *Clickclean 2016*. 2016. URL: http://www.clickclean.org/international/en/ (visited on 04/09/2018).

[63] 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).

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

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

[66] Ralph Johnson John Vilssides Erich Gamma Richard Helm. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1975.

# Appendix A

# Skyscanner structure

Skyscanner has a very horizontal structure, based on Spotify's[63].

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[64].

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[65].

## A.1 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.

## A.2 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.

## A.3 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.

## A.4 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.

## A.5 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.

# Appendix B

# Design Patterns

Extract from the Design Patterns: Elements of Reusable Object-Oriented Software[66].

## B.1 Singleton

### Intent

Ensure a class only has one instance, and provide a global point of access to it.

### Motivation

It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

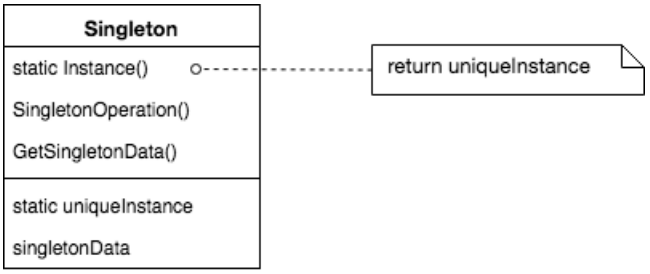
A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

### Applicability

Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

### Structure




### Participants

- Singleton
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++).
  - may be responsible for creating its own unique instance.

### Collaborations

- Clients access a Singleton instance solely through Singleton's Instance operation.

### Consequences

The Singleton pattern has several benefits:

1. Controlled access to sole instance. Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
2. Reduced name space. The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
3. Permits refinement of operations and representation. The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.
4. Permits a variable number of instances. The pattern makes it easy to change your mind and allow more than one instance of the Singleton class. Moreover, you can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
5. More flexible than class operations. Another way to package a singleton's functionality is to use class operations (that is, static member functions in C++ or class methods in Smalltalk). But both of these language techniques make it hard to change a design to allow more than one instance of a class. Moreover, static member functions in C++ are never virtual, so subclasses can't override them polymorphically.

## B.2 Adapter

Also known as Wrapper.

### Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

### Motivation

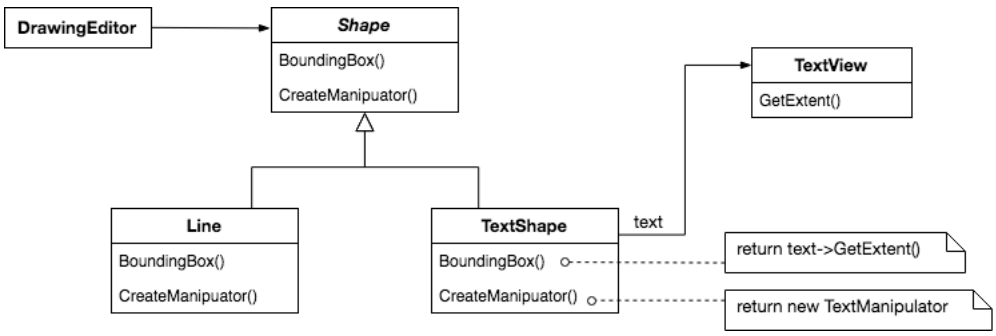
Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.

Classes for elementary geometric shapes like LineShape and PolygonShape are rather easy to implement, because their drawing and editing capabilities are inherently limited. But a TextShape subclass that can display and edit text is considerably more difficult to implement, since even basic text editing involves complicated screen update and buffer management. Meanwhile, an off-the-shelf user interface toolkit might already provide a sophisticated TextView class for displaying and editing text. Ideally we'd like to reuse TextView to implement TextShape, but the toolkit wasn't designed with Shape classes in mind. So we can't use TextView and Shape objects interchangeably.

How can existing and unrelated classes like TextView work in an application that expects classes with a different and incompatible interface? We could change the TextView class so that it conforms to the Shape interface, but that isn't an option unless we have the toolkit's source code. Even if we did, it wouldn't make sense to change TextView; the toolkit shouldn't have to adopt domain-specific interfaces just to make one application work.

Instead, we could define TextShape so that it adapts the TextView interface to Shape's. We can do this in one of two ways: (1) by inheriting Shape's interface and TextView's implementation or (2) by composing a TextView instance within a TextShape and implementing TextShape in terms of TextView's interface. These two approaches correspond to the class and object versions of the Adapter pattern. We call TextShape an adapter.




This diagram illustrates the object adapter case. It shows how BoundingBox requests, declared in class Shape, are converted to GetExtent requests defined in TextView. Since

TextShape adapts TextView to the Shape interface, the drawing editor can reuse the otherwise incompatible TextView class.

Often the adapter is responsible for functionality the adapted class doesn't provide. The diagram shows how an adapter can fulfill such responsibilities. The user should be able to "drag" every Shape object to a new location interactively, but TextView isn't designed to do that. TextShape can add this missing functionality by implementing Shape's CreateManipulator operation, which returns an instance of the appropriate Manipulator subclass.

Manipulator is an abstract class for objects that know how to animate a Shape in response to user input, like dragging the shape to a new location. There are subclasses of Manipulator for different shapes; TextManipulator, for example, is the corresponding subclass for TextShape. By returning a TextManipulator instance, TextShape adds the functionality that TextView lacks but Shape requires.

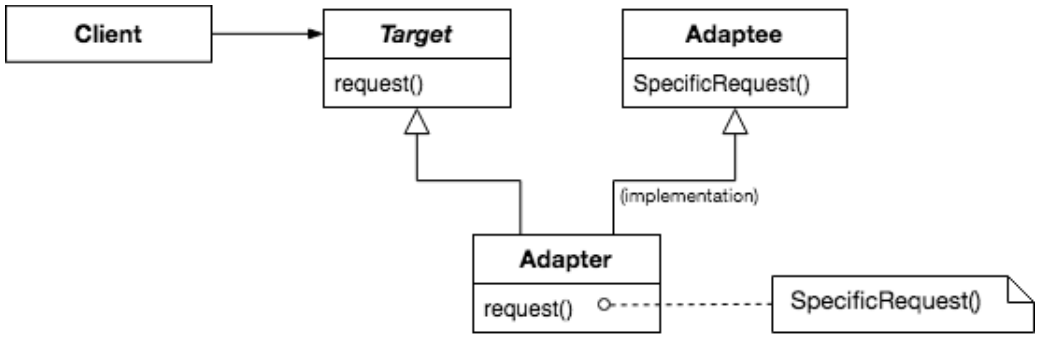
### Applicability

Use the Adapter pattern when

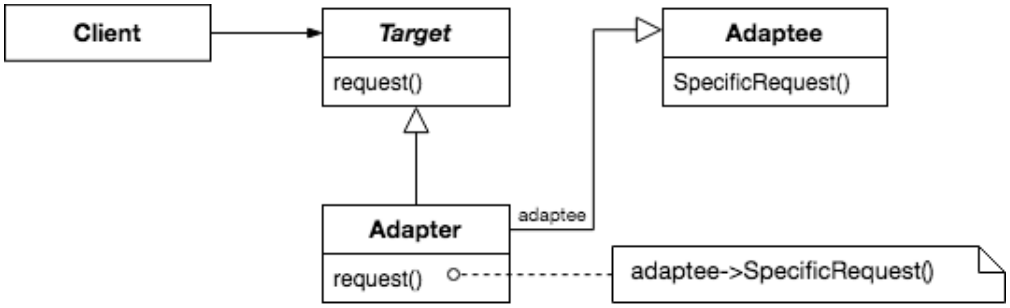
- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing

### Structure

A class adapter uses multiple inheritance to adapt one interface to another:




An object adapter relies on object composition:



### Participants

- Target (Shape)
  - defines the domain-specific interface that Client uses.
- Client (DrawingEditor)
  - collaborates with objects conforming to the Target interface.
- Adaptee (TextView)
  - defines an existing interface that needs adapting.
- Adapter (TextShape)
  - adapts the interface of Adaptee to the Target interface.

### Collaborations

- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

### Consequences

Class and object adapters have different trade-offs. A class adapter

- adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

- lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Here are other issues to consider when using the Adapter pattern:

1. How much adapting does Adapter do? Adapters vary in the amount of work they do to adapt Adaptee to the Target interface. There is a spectrum of possible work, from simple interface conversion—for example, changing the names of operations—to supporting an entirely different set of operations. The amount of work Adapter does depends on how similar the Target interface is to Adaptee's.

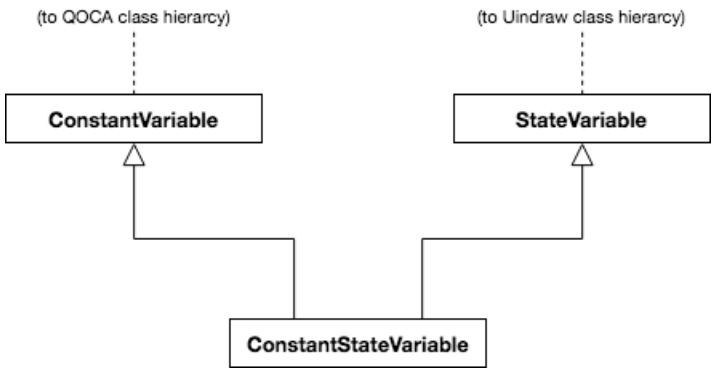
2. Pluggable adapters. A class is more reusable when you minimize the assumptions other classes must make to use it. By building interface adaptation into a class, you eliminate the assumption that other classes see the same interface. Put another way, interface adaptation lets us incorporate our class into existing systems that might expect different interfaces to the class. ObjectWorks/Smalltalk [Par90] uses the term pluggable adapter to describe classes with built-in interface adaptation.

   Consider a TreeDisplay widget that can display tree structures graphically. If this were a special-purpose widget for use in just one application, then we might require the objects that it displays to have a specific interface; that is, all must descend from a Tree abstract class. But if we wanted to make TreeDisplay more reusable (say we wanted to make it part of a toolkit of useful widgets), then that requirement would be unreasonable. Applications will define their own classes for tree structures. They shouldn't be forced to use our Tree abstract class. Different tree structures will have different interfaces.

   In a directory hierarchy, for example, children might be accessed with a GetSubdirectories operation, whereas in an inheritance hierarchy, the corresponding operation might be called GetSubclasses. A reusable TreeDisplay widget must be able to display both kinds of hierarchies even if they use different interfaces. In other words, the TreeDisplay should have interface adaptation built into it.

   We'll look at different ways to build interface adaptation into classes in the Implementation section.

3. Using two-way adapters to provide transparency. A potential problem with adapters is that they aren't transparent to all clients. An adapted object no longer conforms to the Adaptee interface, so it can't be used as is wherever an Adaptee object can. Two-way adapters can provide such transparency. Specifically, they're useful when two different clients need to view an object differently.
   Consider the two-way adapter that integrates Unidraw, a graphical editor framework [VL90], and QOCA, a constraint-solving toolkit [HHMV92]. Both systems have classes that represent variables explicitly: Unidraw has StateVariable, and QOCA has ConstraintVariable. To make Unidraw work with QOCA, ConstraintVariable must be adapted to StateVariable; to let QOCA propagate solutions to Unidraw, StateVariable must be adapted to ConstraintVariable.




   The solution involves a two-way class adapter ConstraintStateVariable, a subclass of both StateVariable and ConstraintVariable, that adapts the two interfaces to each other. Multiple inheritance is a viable solution in this case because the interfaces of

the adapted classes are substantially different. The two-way class adapter conforms to both of the adapted classes and can work in either system.

## B.3 Composite

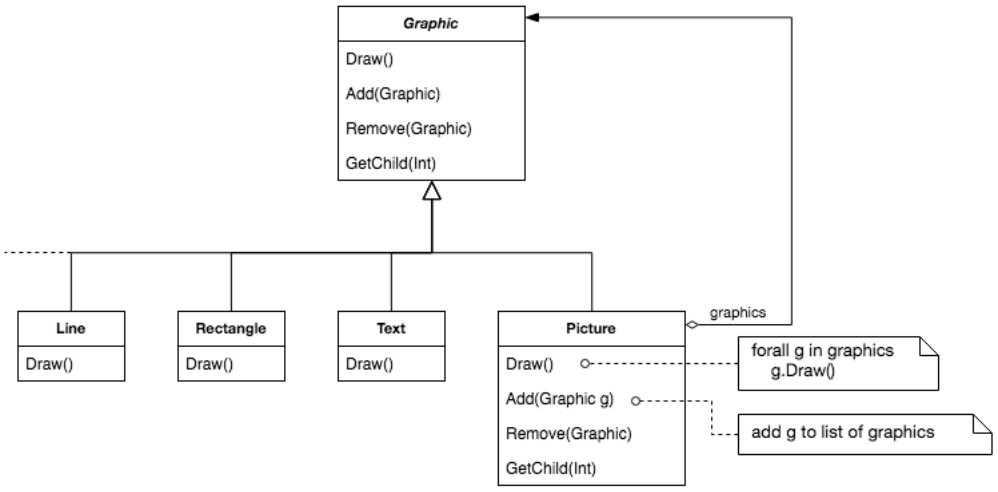
### Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

### Motivation

Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.

But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex. The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction.



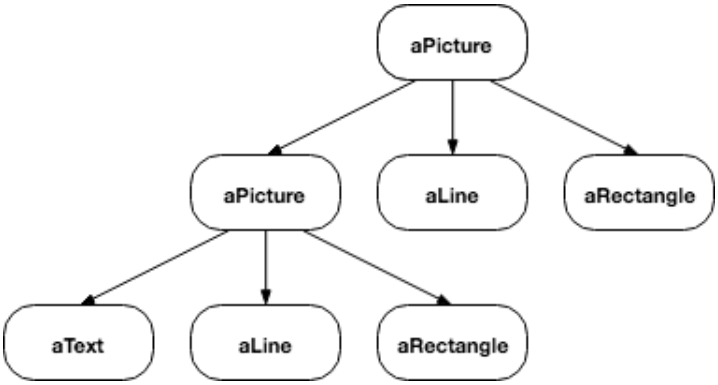

The key to the Composite pattern is an abstract class that represents both primitives and their containers. For the graphics system, this class is Graphic. Graphic declares operations like Draw that are specific to graphical objects. It also declares operations that all composite objects share, such as operations for accessing and managing its children.

The subclasses Line, Rectangle, and Text (see preceding class diagram) define primitive graphical objects. These classes implement Draw to draw lines, rectangles, and text, respectively. Since primitive graphics have no child graphics, none of these subclasses implements child-related operations.

The Picture class defines an aggregate of Graphic objects. Picture implements Draw to call Draw on its children, and it implements child-related operations accordingly.

Because the Picture interface conforms to the Graphic interface, Picture objects can compose other Pictures recursively.

The following diagram shows a typical composite object structure of recursively composed Graphic objects:

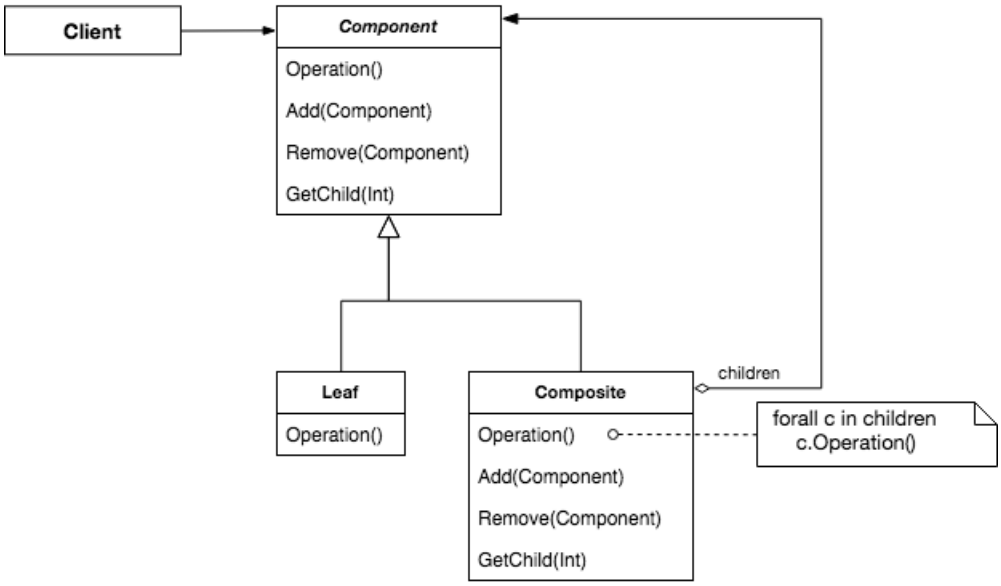



### Applicability

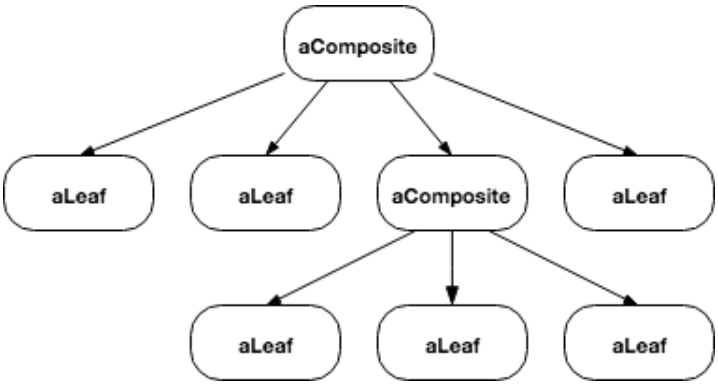
Use the Composite pattern when

- you want to represent part-whole hierarchies of objects.
- you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

### Structure




A typical Composite object structure might look like this:




### Participants

- Component (Graphic)
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- Leaf (Rectangle, Line, Text, etc.)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.
- Composite (Picture)
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- Client
  - manipulates objects in the composition through the Component interface.

### Collaborations

- Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

### Consequences

The Composite pattern

- defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.

- makes the client simple. Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.
- makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code. Clients don't have to be changed for new Component classes.
- can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.