Group 5 Project Documentation

Journey Genie

Group members - Jessica Brown, Joana Grafton, Oliwia Polakiewicz and Nadia Rehman

Honourable mention - Karen Gonzalez Reginato

Introduction

Have you ever felt like your ideal holiday was far from reach? Hours of searching to no avail? Or perhaps you've booked your stay, but then realised you have no clue which attractions to visit? What activities should you do? Was the weather even going to be good? You could wing it when you get there, but every decision takes mental effort and can feel draining. Holidays aren't forever, so, wouldn't you want to maximise every second when you're there?

This entire problem will soon be in the past. Journey Genie saves you hours of searching by helping you find the perfect hotel tailored to your needs, while also enhancing your experience with nearby activities and real-time temperature indicators—all in one convenient service. No more scouring the web for hours or feeling unsure about any part of your holiday.

Or maybe you want a break but don't even know where to go? Well, let Journey Genie take you to a magical new place and grant you a getaway you didn't even know you needed.

This project document will outline exactly how we intend to execute this web app, from the user journey to our methodology using MOSCOW and architectural diagrams, and details regarding the intended implementation and execution.

Background

The project aims to develop a console application designed to assist users in planning their holidays with ease and convenience. The initial phase will focus on building a robust console-based interface, with the potential to expand into a fully integrated frontend application, should time allow.

Users can either input a specific destination or opt for a "Take me anywhere" feature, where they'll be presented with random destinations. They can then enter their preferred travel dates. Based on the selected location and dates, the application will display a list of available hotels, along with their corresponding prices. Users will have the ability to refine their search results through various filters. Additional details such as the average temperature for the selected dates will be provided (based on historical data), helping users make informed decisions. Users can favourite hotels that catch their interest, triggering the application to suggest nearby activities and attractions, allowing them to curate a complete holiday experience. At the end of the search process, users can save their selected hotels and activities, and have the details sent directly to their email address.

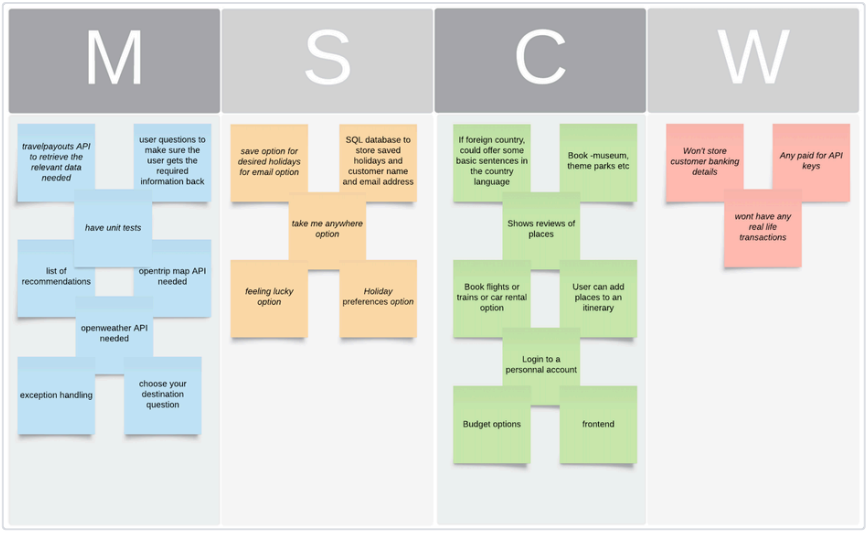
This project is designed with the user's convenience in mind, providing a seamless and comprehensive holiday planning experience.

Specifications and Design

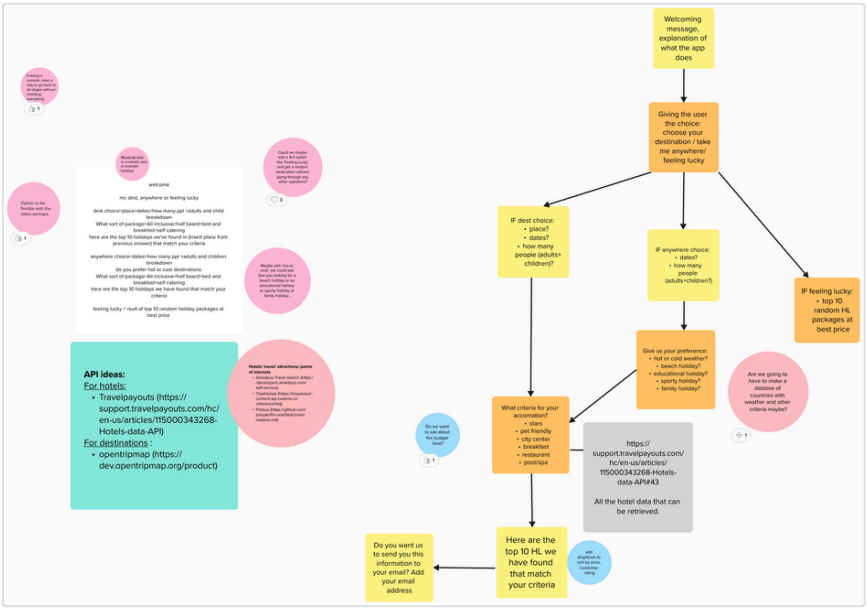
We have used many different processes to get a very precise overview of our project.

The first step involved determining what our project must, should, could and won't include followed by agreeing on the key features to implement.

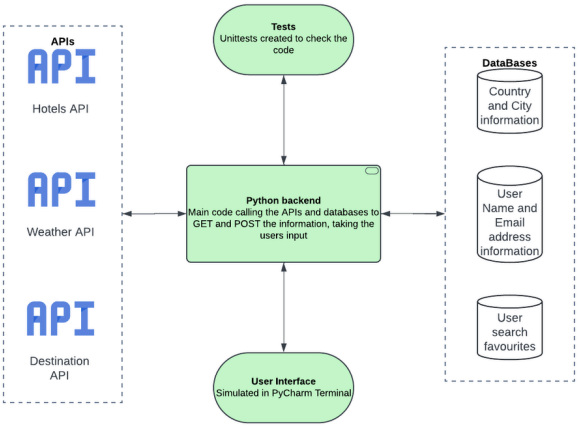
Here is our MOSCOW as shown below



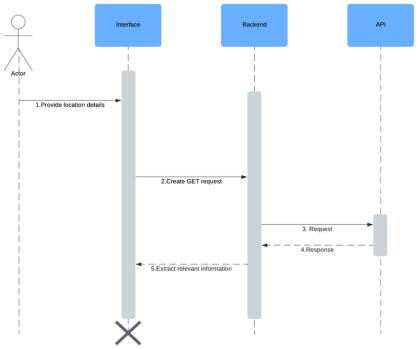
We then created a System flow (shown below) to begin to understand the flow of our project and for any additional ideas to be discussed after visually seeing exactly how the application would flow through from start to end.



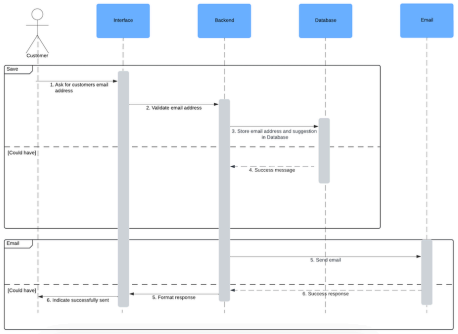
We then created the architecture diagram (shown below) to show an outline of how things should work and what will interact with the Python backend.



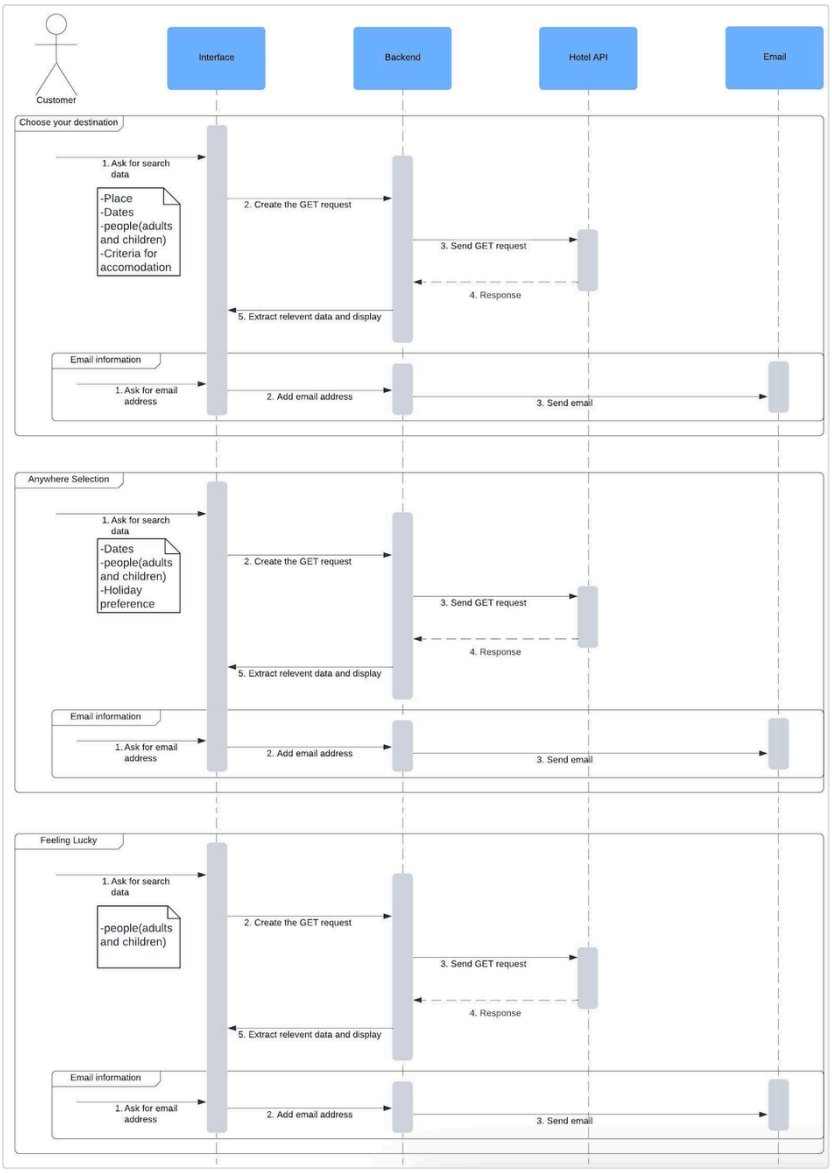
To fully show the sequence of events for our project we created sequence diagrams to show the API calls, the email sequence and user journey from start to finish. These show exactly the several interactions throughout the entire journey for the user and also help us visually see what will happen at each stage of the sequence.



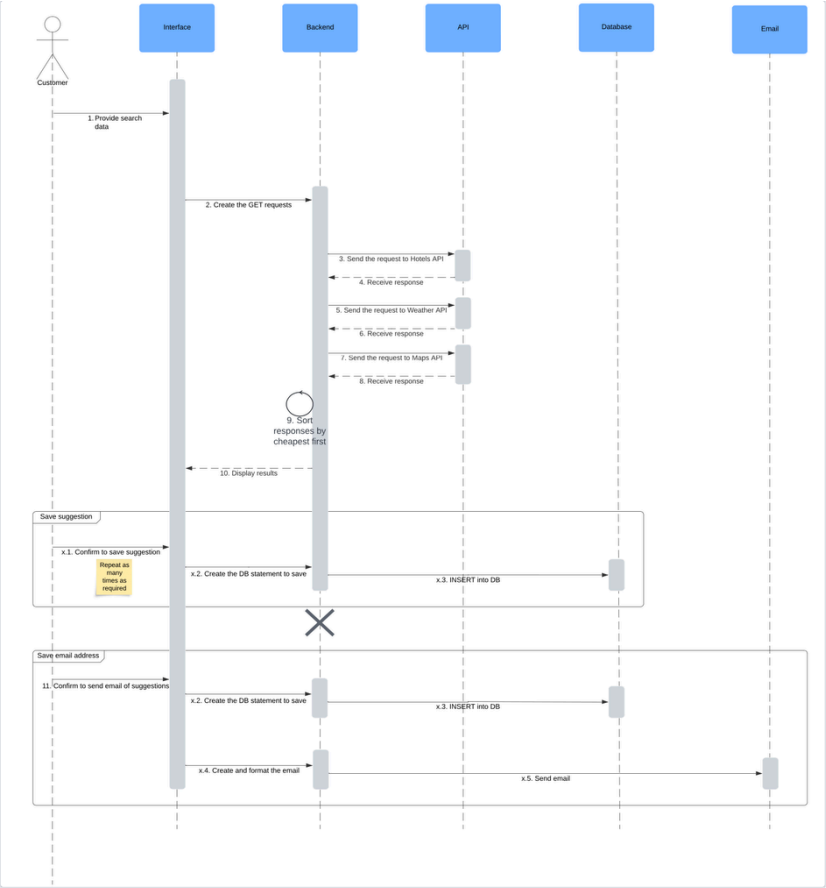
Map and Weather API



Sending an email Sequence Diagram



Hotel API Sequence Diagram



User Journey Sequence Diagram

Implementation and Execution

We implemented several different processes to ensure a smooth project flow and transparency throughout the team to make sure we are all working as effectively as we can.

- For team members' roles, we are a multidisciplinary team performing various tasks while also leveraging each team member's strengths as needed.
- Tools that will be used are: Postman for REST calls and the libraries we will be using are unittest, requests, random, collections, pprint, flask, and mysql.connector, datetime, re, emoji, urllib.
- The implementation process was to storyboard how we all wanted it to work, create user stories to understand how it could then work and finally develop and test each story at a unit level with the stretch goal of having user journey level testing (integration testing).
- We used an Agile approach (Jira - using 2-week sprints. All our tickets have been split into epics with stories and tasks to cover all aspects and allow all team members to contribute to any aspect of the ticket). All code is refactored during development and code reviews are mandatory for each Pull Request.
- GitHub was used and all team members were using their branches so then any code going to the main branch was reviewed and all appropriate testing was performed before pulling to the main.
- Meaningful documentation and comments throughout allow any person not currently working on the project to easily follow instructions as would be expected within the IT industry.
- Slack was our main form of communication alongside regular video calls which will be recorded for any member unable to attend and for us to rewatch information as needed.
- Many parts of the project were coded at the same time by different people to be able to get all the coding done within our deadlines set for each sprint.
- Pair programming was used on some sections of code to speed up some of the more complicated coding challenges we will face during the project.
- Challenges we faced have been limits to single APIs, whereby we require two or more to perform a single feature of the project (for example: needing to use an API to pass the name of a city to retrieve the latitude and longitude of that city to be used in a different API for a more specific use).

Testing and evaluation

Our testing strategy followed the Agile testing pyramid model focusing on preventing bugs rather than finding them. Our UI has a small number of tests in comparison to the unit tests as the unit tests show working functionality as intended through extensive testing - this is derived from a clear and confident understanding of what each function must accomplish via the user story tasks (acceptance criteria) at a functional(story) level. With the foundations of the code being tested correctly more time can be spent developing newer pieces of functionality rather than bug fixing later on in the project's lifecycle, disturbing the goal of a finished project.

As we have set our acceptance levels the tests will be more complex and less complicated when it comes to implementing them. As we are following the pyramid model this means the majority of low-level functionality is already tested to a high degree of confidence and UI/AC level tests will have more steps to pass successfully (complexity) while relying on previously created functional tests (less complexity as the ‘hard work’ has already been done).

Regression testing will be easy to maintain with each new function added to the unit test collection, meaning any breaking changes will be found very early on in the development lifecycle, we are also attempting the Test Driven Development (TDD) approach later on in the project as we have been shown how powerful it is in Agile development.

The APIs we are using are free tier-level access to what we need, as such more complicated functionality that could give a better product is either very complicated (and not worth it after researching how much time it will take for us all to learn what is needed for a very small gain) piece of work that might not work out (we don’t know what we don’t know yet) or requires a subscription to a paid service. This is a limitation for us currently.

The limitations for APIs and databases are;

Weather API

- This API has two endpoints we are utilising, the /future and /history are the most relevant for our use case. The /future has a few limitations that have added complexity to our code, a minor one being only accepting a single date for query, as our product allows for multiple dates this means we have coded in such a way to have several requests made to retrieve the desired data.
- The major issue with this endpoint is its date range for retrieval of weather data, it can only accept dates 14 after the current date and up to 300 days from today's date meaning logic has been added to check if the dates required are over the 300 days limit and to use the previous years' data instead. This has also meant error handling has been added for the 'dead' 14 days after the current date as no data can be retrieved.
- Another major limitation is the 'plan' the API key is linked to, once it has expired on the the free "x days of premium for free" the API becomes useless for our needs as nothing we need can be retrieved with the 'free' plan.

OpenTripMap

- This API is well-structured, free and easy to use. It allows for a wide range of data retrieval for the 'activities' part of our project. However, some aspects of it slightly complicated our task.
- To be able to get data about activities according to specific filters per city, the name of the city is not an accepted parameter, but it required latitude and longitude which is why we had to create a function to retrieve the latitude and longitude of a given city to be able to get activities.
- The activity details retrieve a lot of data with some duplicates and unfortunately, the text description is in the language of the selected country with no possibility of translating it to English. Some of the returned links also seem to not be working (this occurrence is very rare).

Hotel API

- The API did not contain a country search, which meant that we had to create a database with cities within countries and ask the user to resort to selecting a city. We attempted to search for alternative APIs such as Amadeus, Booking.com: The largest selection of hotels, homes, and vacation rentals , Expedia.com and Search Flights, Hotels & Rental Cars | KAYAK however, Amadeus did not have a country search option and to use the regional search feature for the other 3 APIs required permissions and to be part of organisations
- Another limitation noted was that for some cities, no search results came up such as searching Havana, Cuba. After looking at the database on the API documentation which had the list of all possible entries, Havana was present. However, testing many other cities such as Paris, Tokyo, New York City, and London. Singapore and more, there were no issues with return results. After trying to debug on the internet. we concluded that the API has its form of ranking system that might not be representative and instead prioritises the more popular cities within its city search option, however as we are not advanced coders, we are still open to the possibility that this claim might be wrong. Whilst a potential option to solve this would be to extract the relevant information from the database, we felt as though if we did this, we would struggle with the time constraints and instead resorted to creating our own database with certain cities within countries as that was already a solution to the country availability problem.
- Another functionality we desired to implement was for the user to have the option to book the hotel directly. Within TravelPayouts, there exists the option however, it is on a request basis. Although we could perhaps find a way to search for the hotel name somehow not within the API itself, it was decided that the link to HotelLooks was enough as the price comparison site itself provides several websites from which you can select and book from there.

Database

- Scalability - to start with, the one database will be fine but if the app grows, and we end up having a lot of users, the databases might struggle with the number of data, especially with the favourites as each person can choose multiple items
- Rigid schema design - changing the schema, such as adding new columns or modifying data types, can be complex and disruptive, which might be an issue if we want to, for example, add additional columns with more information in any of our tables.

Conclusion

Overall, our goal with Journey Genie was to create a program capable of streamlining the lengthy holiday planning process by combining it into one service through the use of API calls and databases. This document has highlighted specifications such as the planned backend architecture, applying the MOSCOW method, and our implementation - from our preferred system of working to the many tools we will use to execute this.

Did we achieve what we set out to do?

We believe the goals we set have been met, our product delivers what we set out to achieve and our methods(or ways) of working have been adhered to.

From a technical perspective, we have a project that successfully takes user input, verifies the input is correct, accesses several APIs, transforms the retrieved data into information, accesses a database and then has the capability then store all of what is gathered into the database for later use via a Flask hosted API we have created. All of this has been developed using OOP and TDD.

From a team perspective, we set ourselves some personal stretch goals on what we believed we were capable of delivering and what we wanted to be able to deliver, these have been challenging but ultimately rewarding in what the finished product is capable of.

From an organisational perspective, we have been using Jira for time and workload management in an AGILE environment.

Working as a team has been a positive experience and we have achieved what we hoped for, everyone has helped one another to hit the personal milestones alongside group tickets we have set ourselves during the project to achieve a fully functional product with all the must-haves we hoped for.