Design Report

SENG2021 Requirements and Design Workshop
D2 Deliverable: Friday 27th March 2020

Group: We Push To Master

Chinmay Manchanda	z5216191
Marina Reshetnikova	z5285381
Michelle Seeto	z5061204
Xifei Ni	z5173159
Ye Wo	z5215628

Background

Currently, we as users need to browse different websites to find events or activities that intrigue us. Unfortunately, users often succumb to the easy option of staying at home, either not wanting to put in the labour to find an appropriate activity or overwhelmed by analysis paralysis.

Our web application, ‘Hello World’ aims to simplify this process of choosing an activity for the day, and in doing so, encourage a more active and engaged approach to one’s daily life. ‘Hello World’ aims to fetch event details from across the internet and provide users with a curated list of events and activities they can enjoy.

Solution

Objective

Our aim is to automate the uninteresting and often unfruitful process of choosing an activity for the day. Traditionally, this is made up of the following steps:

- Researching through a number of websites (Facebook events, Eventbrite, Timeout etc.)
- Filtering activities based on one's mood and interests
- Choosing the most appropriate activity for the current time of day and weather

Our application aims to take the decision out of the user's hand by executing the research and filtering on their behalf. Ultimately, we are seeking to provide an instant response to the question, "What should I do today?", saving users time that would otherwise be spent trawling through the web while taking into consideration weather, time of day and users' preferences and mood for optimal results.

As well as a curated selection of events, we will also suggest activities at random, a feature designed to save the indecisive user, or simply for those seeking surprise and spontaneity.

Visual Design

Figure 1. Visual prototype of the user interface

Part 1: Software Architecture

1. External data sources

OpenWeather API

Documentation: https://openweathermap.org/api

Weather and climate are major factors in one's choice of activity. For example, people are more likely to choose an outdoor sports activity when the weather is sunny or fine, whereas they may opt to stay indoors during the rain. Weather thus plays a major role in our activity-filtering algorithm.

To obtain current weather data, we will be using the free, public Weather API provided by OpenWeather. We initially sought to use a Google weather API, however upon further investigation, we found that the only weather API provided by Google was for Android devices and had been deprecated since August 2019.

Fortunately, the OpenWeather API has proven to be highly appropriate for our project. As it is a public API, registering for an OpenWeather account allowed us to instantly obtain an API key. The free pricing tier gives us access to up-to-date weather information for over 200 000 cities. We will be retrieving the following information: current "feels-like" temperature, weather condition, humidity, visibility and sunrise/sunset times, which will allow us to determine the best activities to recommend to a user based on their actual weather conditions and time of day. Figure x contains some examples of information extracted for different cities, and basic computations made with the extracted information such as whether it would be sunrise/sunset within the next hour and which types of activities would be suitable given the current weather.

The free pricing tier has a rate limit of 60 calls per minute, which we believe is sufficient for the purposes of this project.

```
Location: Tokyo, Japan
Time now: 2020-03-15 13:52:22.253621
Weather condition: Clouds - broken clouds
Weather icon: http://openweathermap.org/img/wn/04d@2x.png
Feels like: 4.74 degrees celcius
Humidity: 37%
Visibility: 10000m
Activity tags: ['indoors', 'outdoors', 'sport']
Sunrise: 2020-03-15 05:52:12.000013
Sunset: 2020-03-15 17:48:16.000013
```

```
Location: Sydney, Australia
Time now: 2020-03-15 15:52:09.504121
Weather condition: Rain - shower rain
Weather icon: http://openweathermap.org/img/wn/09d@2x.png
Feels like: 13.14 degrees celcius
Humidity: 77%
Visibility: 10000m
Activity tags: ['indoors']
Sunrise: 2020-03-15 06:54:27.000015
Sunset: 2020-03-15 19:13:51.000015
```

```
Location: Rome, Italy
Time now: 2020-03-15 05:52:17.258610
Weather condition: Clear - clear sky
Weather icon: http://openweathermap.org/img/wn/01n@2x.png
Feels like: 5.4 degrees celcius
Humidity: 76%
Visibility: 10000m
Activity tags: ['outdoors', 'sport']
Sunrise: 2020-03-15 06:21:34.000014
Sunset: 2020-03-15 18:16:22.000014
Sunrise is within an hour from now
```

Figure 2. Examples of information extracted from the OpenWeather API.

Ticketmaster API

Documentation: https://developer.ticketmaster.com/products-and-docs/apis/discovery-api/v2/

The Ticketmaster Discovery API gives our application a wider access to not only public events but also to family and attraction activities and makes sure we have something on our recommendation all the time. This is a public access API which made it fairly easy for us to obtain the API key and start right away. The default quota allows us to make 5000 API calls per day and rate limitation of 5 requests per second which we believe is enough for our project.

The Ticketmaster API covers countries like the US, Canada, Mexico, Australia, New Zealand, United Kingdom, Ireland, other European countries and more. Allows us to grab a list of events based on various filters like category, date, time, location etc. Most of our API calls will be to retrieve information like event name, description, time, location, date and a few pictures.w

The only challenge we're going to face is that because of the current situation of COVID-19 we won't have any events active as most of them are being shown as cancelled or postponed.

FourSquare API

Documentation: https://developer.foursquare.com/docs

One of the new features that our application offers is "activities", a curated selection of things to do that are not necessarily tied to official events. To find what one can do if they're not considering activities, we need places. For our places we'll be using Foursquare's API, as Google Places API requires partner access which takes around 4-6 week to approve.

Foursquare's public API gives us access to their database of 105 million places which includes places to eat/drink, parks, libraries and many more. The API calls will also retrieve more than just the name and location, they'll grab user-generated content as well which includes (photos, ratings, reviews and money(how expensive)).The API also Allows us to sort, filter and manipulate places data according to our needs. With this API we get access to over 30 filters and 900 venue categories to sort out places for our clients.

We'll be using a free personal account which gives us an API call quota of 100,000 calls per day (including 500 of the premium one's).

Meetup API

Documentation: https://www.meetup.com/meetup_api/docs/

To obtain data for local meet ups and hobby-based events, we will be using the API provided by Meetup.com. Although the current version of this API requires a paid Meetup Pro account as of December 2019, we will be using a deprecated version that provides free access to authenticated users. The information that we will retrieve from the Meetup API includes name, description, start and end time, location, whether it is an online or live event, price and the event's category.

Eventbrite API with scraper

Documentation: https://www.eventbrite.com/platform/api

As a popular global platform for event listing, Eventbrite offers a rich source of data on current and upcoming public events. Furthermore, much of their API is free and publicly available through a registered Eventbrite account; it was thus an easy and instant process to obtain an API key.

However, we faced a major challenge in our initial attempts to use the Eventbrite API, as their Event Search endpoint, which returned a list of public events given some search queries had been deprecated since December 2019. Currently, it is only possible to retrieve individual event details through the event ID. To overcome this, we built a web scraper that extracts a list of event IDs from a search result webpage on the Eventbrite website. Given these scraped event IDs, we can then make individual API calls to retrieve each event's details.

This clearly presents performance issues as we must now make one API call per event, however we will minimise this by narrowing our search query on the Eventbrite site. Thus, our application will only scrape and retrieve the event IDs from filtered results, eliminating unnecessary API calls from being made. Furthermore, Eventbrite's API offers the ability to make batched requests, thus saving us the latency and cost of several request round trips. Given that our application is likely to debounce rather than make API calls in real-time, we believe these performance considerations are reasonable.

The information that we will retrieve for each event includes name, summary, start and end time, location, whether it is an online or live event and the event's category/subcategories. Using a free, community Eventbrite account, we can access information on public events only, which does not present an issue as we do not anticipate requiring information on non-public events. Furthermore, the rate limit of 2 000 calls per hour is sufficient for this project.

2. Software components

Our application is composed of several software components, illustrated in Figure 3.

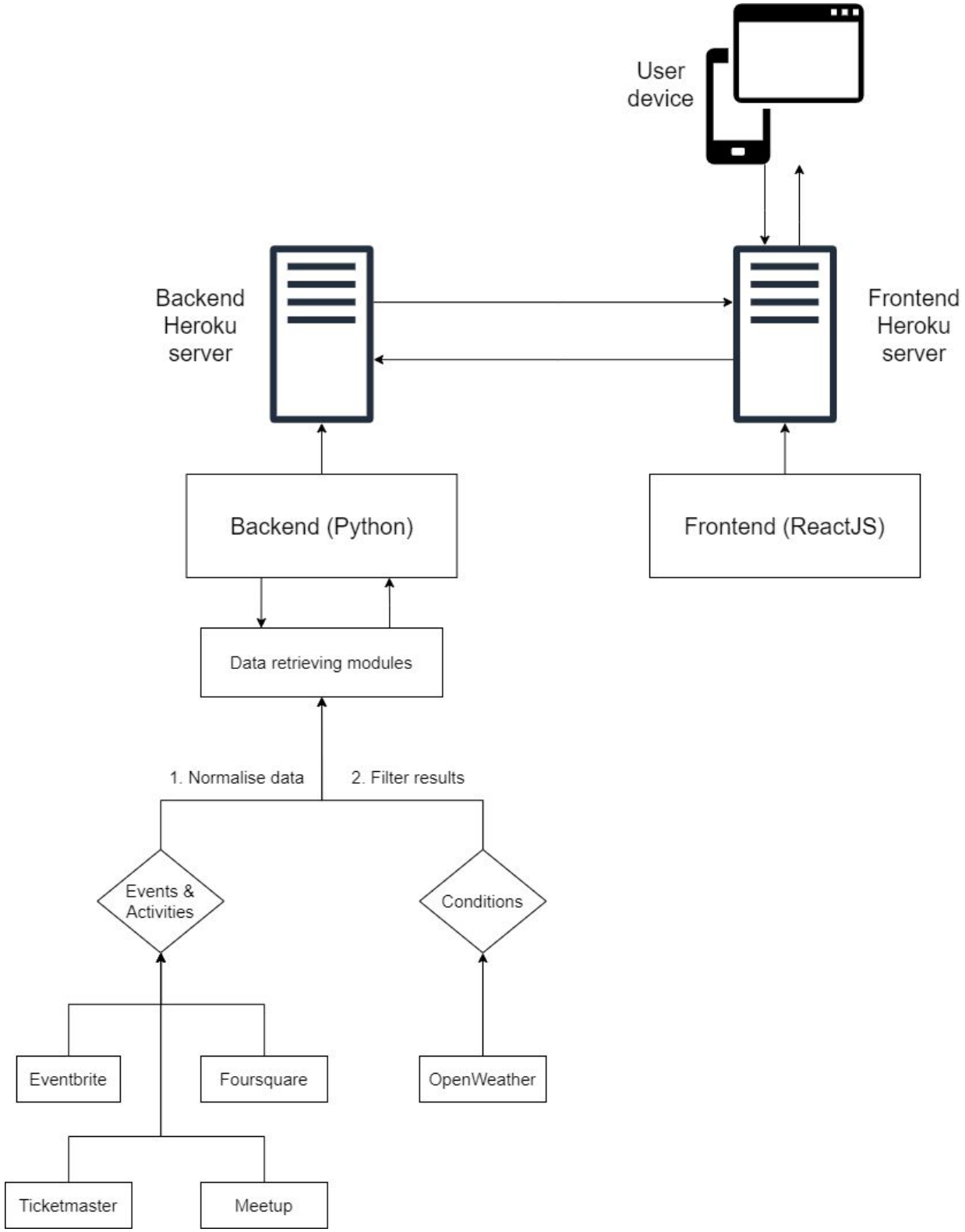


Figure 3. Software architecture

2.1 Frontend & user interface

Main Tech Stack

Our frontend will mainly be using React framework with JavaScript.

- React is a very popular framework, and experience we gain from here can be transplanted into the future.
- We use JavaScript because our group members have a reasonable level of familiarity with JavaScript.

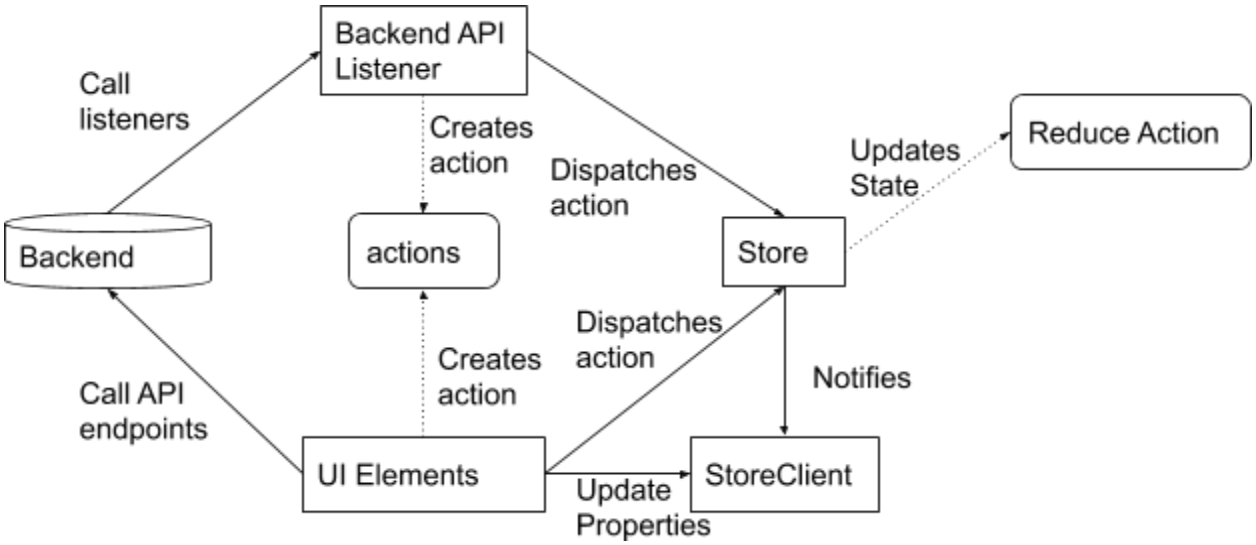
One Way Data Flow Model

One-way data flow is a model which has been recently popularised by React, Flux/Redux and friends. In particular, Redux is a tiny library focused around three principles:

- The page state should have a single source of truth
- UI elements should have read-only access to the page state
- Changes to the page state are made with pure functions

We think that sticking to these principles has the potential to simplify our frontend.

Data Flowchart



Store

The Store is the object that brings state together. The store has the following responsibilities:

- Holds application state;
- Allows access to state via getState();
- Allows state to be updated via dispatch(action);
- Registers listeners via subscribe(listener);

- Handles unregistering of listeners via the function returned by subscribe(listener).

Notably:

- `getState()` is the state tree for the entire application, including events, selected items, search terms, etc.
- `state` is publicly readable but not writable. The only way to modify it is through `dispatch(Action)`.
- `dispatch` is called with an `Action`, which is an object with a name field. `dispatch` will use the `Action` object to produce a new data object with the state of the world after that action. `Store` then notifies observers that the page state has changed. Actions are the only way for page state to be modified.

Handling actions with reducers

- Actions are handled using *pure functions* called *Reducers*. A reducer (in the same sense as the functional programming primitive reduce) takes a state and an action and produces a new state.
-
- Reducer functions *must* be pure. They should not mutate existing objects: instead, they create new objects with any required changes implemented. They must not make any API calls, or touch the DOM, or use Math.random, or anything else which could cause the same input to produce a different output.

- Reducer functions *may* implement business logic, but they do not have to. It is acceptable to require the action to include extra details to remove complexity from the reducer.

- Reducer functions *should* avoid creating new objects for things which do not change. Unnecessary copies waste memory and can cause extra work at the UI layer.

Initialization

The store should be initialized in one go, using `CreateStore in store.js`. This should be called with the entire initial state of the world, including:

- The id of all events
- The search term from the URL, if there is one
- The filter status from users' selection, if there is one.

These initialization tasks can be kicked off as the page is loading. No actions are allowed before `CreateStore` has been called, so we should hide the UI before this.

Writing UI Elements

First we need to make the `store` available to our app. To do this, we wrap our app with the `<Provider />` API provided by React Redux.

```
import { Provider } from 'react-redux'

const rootElement = document.getElementById('root')
ReactDOM.render(
  <Provider store={store}>
   ...
  </Provider>,
  rootElement
)
```

With store, UI elements will be able to update themselves whenever the state we are observing changes.

```
// ...other imports
import { connect } from "react-redux";

const EventList = // ... UI component implementation

const mapStateToProps = state => {
  const { allIds, byIds } = state.going || {};
  const going =
    allIds && allIds.length
      ? allIds.map(id => (byIds ? { ...byIds[id], id } : null))
      : null;
  return { goings };
};

export default connect(mapStateToProps)(EventList);
```

That's it! As before, `event` can be bound to UI and will update whenever the store changes.

Writing Actions

Actions should be created by functions which live in a shared `actions.js` file. These can be very simple, or if necessary, they can include logic based on the current state of the store:

```
export const toggleGoing = id => ({
  type: TOGGLE_GOING,
  payload: { id }
});
```

Creating actions like this has the useful side-effect of documenting all possible actions and their parameters.

Writing reducers

Reducers live in a shared `reducers` directory.

```
const initialState = VISIBILITY_FILTERS.ALL;

const visibilityFilter = (state = initialState, action) => {
 switch (action.type) {
   case SET_FILTER: {
     return action.payload.filter;
   }
   default: {
     return state;
   }
 }
};

export default visibilityFilter;
```

Proposed UI Component

The React UI Components

We have implemented our React UI components as follows:

- `App` is the entry component for our app. It renders the header, the `AddGoing`, `EventList`, and `VisibilityFilters` components.
- `AddGoing` is the component that allows a user to input a event item and add to the GOING list upon clicking its “Going” button:
    - It uses a controlled input that sets state upon `onChange`.
    - When the user clicks on the “Going” button, it dispatches the action (that we will provide using React Redux) to add the event to the store.
- `EventList` is the component that renders the list of events:
    - It renders the filtered list of events when one of the `VisibilityFilters` is selected.
- `Event` is the component that renders a single event item:
    - It renders the event content, and shows that a event is ended by crossing it out.
    - It dispatches the action to toggle the event's going status upon `onClick`.
- `VisibilityFilters` renders a simple set of filters:
    - Clicking on each one of them filters the todos:
    - It accepts an `activeFilter` prop from the parent that indicates which filter is currently selected by the user. An active filter is rendered with an underscore.

    - It dispatches the `setFilter` action to update the selected filter.
- `constants` holds the constants data for our app.
- And finally `index` renders our app to the DOM

Test Plan for Frontend

Unit tests:

- Existing store tests are replaced with unit tests of everything in `actions.js` and `reducers` These are easy to test, since everything in them is a pure function.
- We should migrate existing tests of UI elements to use a TestStore, which would allow a state to be specified and to check that the correct actions are dispatched.

Integration tests:

- We should add an integration test for Store/StoreClient
- And add the UI/backend integration tests which we've been talking about for a while

Caveats/Alternative Approaches

We see this model as having several advantages:

- We rely less on react specifics: We no longer need react router
- Many operations are conceptually simpler, since we can either operate on a list of IDs or on nodes directly.
- Search can be treated the same as displaying a regular list of results, since both are just a list of IDs.
- Page UI state is no longer coupled to UI itself(eg, we separate out event going/not going state), which makes bulk-updating the state easier
- It is easier to test how the page state changes, since reducers are pure functions. There's no more uncertainty about whether our path notifications are *actually* correct.

However, it has significant caveats:

- We are reimplementing part of the data-binding system for ourselves. Our system will undoubtedly run into problems that either React or Redux have already solved.
- Writing correct reducers involves being very careful about copying/mutating data
- We are preventing React from performing certain performance optimisations

We think the tradeoffs make sense in this case, due to the type of data that we have and the type of operations that we want to do. This approach may not make sense for other projects with different data structures/actions.

Alternatives considered

Two-way data binding

Two-way data-binding is the well-lit path for React applications. However, due to the nature of the data structure and API, we would not benefit greatly from this: we would still need a

centralised place to perform updates in response to API listeners and would run into the same problems with path notifications.

2.2 Backend

Choice of Language

For all of our backend software components, we have decided to use Python as our programming language of choice. This is primarily because of the extensive number of libraries and most importantly frameworks that it offers, such as Flask/Django. Furthermore, all team members are perfectly familiar with Python , which means that the project won't experience any growing pains associated with learning a new programming language

Choice of Framework

Flask is the framework we will be using for 'Hello World'. It's extremely simple to implement and has all the desired functionalities that we are after. As this is a relatively small scale project, we decided that we simply had no need for the extra functionality offered by a framework such as Django, as we won't have a database or require an admin panel. Additionally, Flask is far less complex and hence easier to implement, helping the team fastrack the development process.

Database

Having analysed the functions of our web application and how we will be handling data, we have concluded that no database will be necessary for this particular project at this time, as we will not be storing any user input or any information about the user.

Caching

A cache will be implemented in order to store the responses from the Foursquare API as the API only allows for 500 premium calls a day, which is not sufficient for our web application. The cache will be configured to store a number of events for only a few hours, then it will be required to make another request to the API.

The MVC Architecture Model

For our backend, we will be adhering to the MVC (Model View Controller) architectural framework.

Figure 4.0

Model -

Summary:

The *model* component of our application's structure will solely consist of our Python backend. The model will be responsible for handling the user-inputted data that's been passed to it by the controller, calling available APIs, normalising the data obtained through the APIs, filtering it and then finally sending an appropriate response back to the controller as seen in Figure 5.0 below.

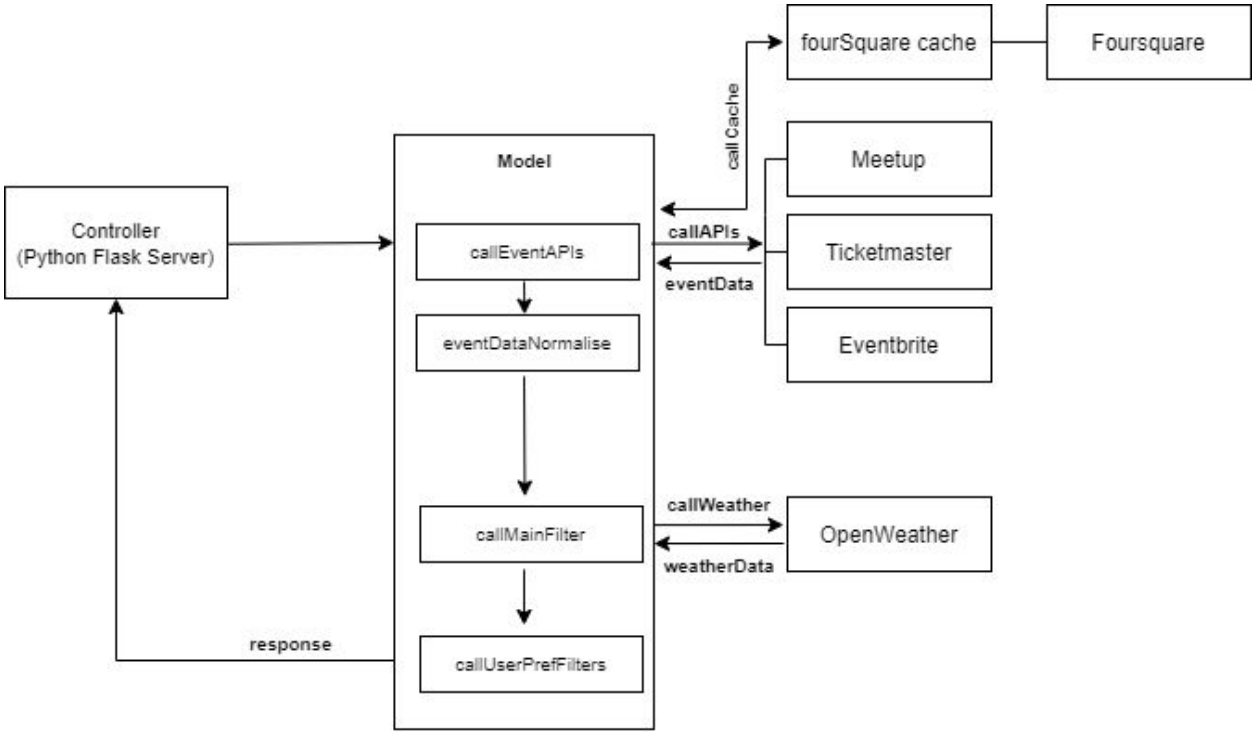


Figure 5.0

Description :

A call to the model by the controller will trigger the method `callEventAPIs.` This method is responsible for calling all available APIs and gathering raw data about any events and activities. Then, `eventDataNormalise` is called in order to format the data obtained and prepare it for filtering. An instance of the following class will be created and populated with the data obtained through the API calls :

```
Class Event:
    def __init__(self, event_id, name, start_time, end_time, location,
organiser, is_free, is_online summary, description_html, tags, reviews,
url):
    self.event_id = event_id
    self.name = name
    self.start_time = start_time
    self.end_time = end_time
    self.location = location
    self.organiser = organiser
    self.is_free = is_free
    self.is_online = is_online
```

```
    self.summary = summary
    self.description_html = description_html
    self.tags = tags
    self.reviews = reviews
    self.url = url
```

The data in the Event class is of the following format:

Property	Type
event_id	string in format
name	string
start_time	integer (unix timestamp)
end_time	integer (unix timestamp)
location	dictionary {longitude, latitude}
organiser	string
is_free	boolean
is_online	boolean
summary	string
description_html	string containing HTML
tags	array of enum(indoors, outdoors, romantic, artsy, geeky, history, hungry, family-friendly, sporty)
reviews	dictionary {rating, review}
url	string

Following the data's normalisation, it is now ready to be processed by the model's filtering methods. First, it is passed to the `callMainFilter` method. At this stage, only 3 basic, or otherwise, built-in filters are applied - time, weather and location. Time is determined using the datetime python library, location is made available to the web app by the user and sourced using geolocation API and finally, the weather is obtained using the OpenWeather API. All data is normalised within the function and then superimposed with the `Event` objects previously obtained by `callEventAPIs`, allowing the backend to seed out a number of events that do not meet the 'primary' criteria.

The remaining objects are passed to the final method, `callUserPrefFilters,` which uses the tags originally passed to the model by the controller, to filter out any events that don't meet the user's personal criteria. The user tags include the following:

- `Indoors`
- `Outdoors`
- `Sporty`
- `Romantic`
- `Family-friendly`
- `Artsy`
- `Geeky`
- `Historic`
- `Hungry`

Once `callUserPrefFilters` has applied the final layer of filtering, the model returns a response to the controller in the form `{[event_id : id, event : Event]}.` Where the data is then broken down and distributed to the view.

Controller -

Summary:
The *controller* component of our web application's backend will be done in Python using the in-built Flask framework. The controller will not be handling or manipulating any data. It will only be responsible for passing user input gathered by the view to the model and vice versa.

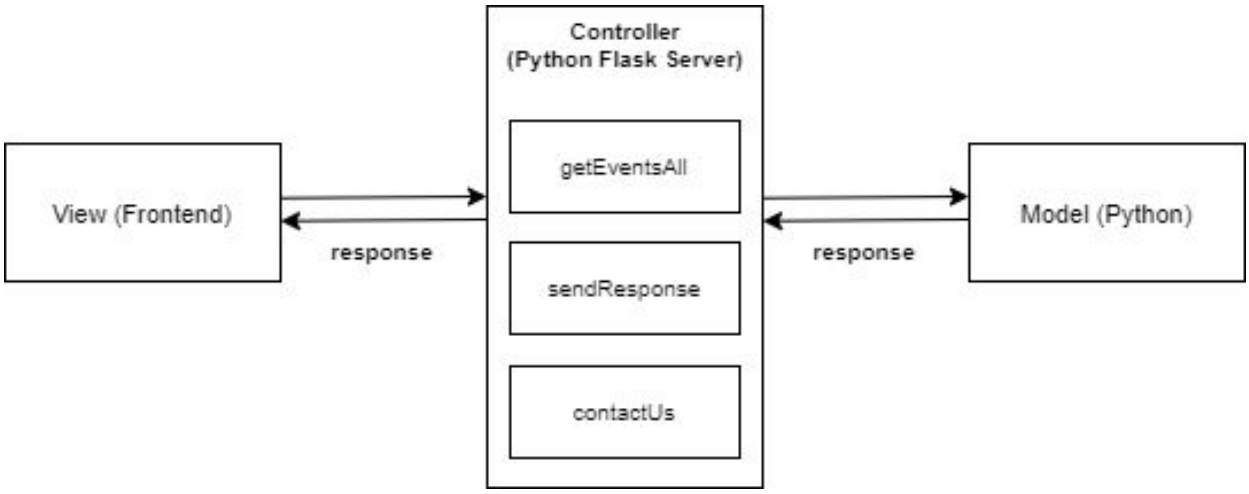


Figure 6.0

Description:

The controller is listening to the fronted on a specified port, which will be configured closer to the deployment. At the moment, it's expected to deal with only 2 types of requests, obtaining a list

of events and sending an inquiry to the team via the ‘Contact Us’ page. The following is the team’s most basic implementation of the server:

```
from json import dumps
from flask import Flask, request, jsonify

APP = Flask(__name__)

def sendResponse(response):
    return dumps(data)

@APP.route(‘events/get_all’, methods = [‘GET’])
def getEventsAll():
    tags = request.args.get(‘tag’)
    try:
        call_model(tags)
        return sendResponse(data)
    except ValueError as e:
        return sendResponse(e.args)

if __name__ == ‘__main__’:
    APP.run(port = (sys.argv[1] if len(sys.argv) > 1 else 5000))
```

The server is configured for specific routes. Depending on which route is called, the appropriate model method will be called.

View -

Summary:

The view component of the MVC architecture consists entirely of the frontend. It is responsible for all the graphic elements of the design, data gathering and data displaying. Upon certain actions by the user, such as manually choosing which filters they’d like to apply to their list of displayed events, the view passes this information to the controller, which based on the specified route, calls the model to apply data manipulation if required and then pass the results back. The response is then visually communicated to the user via the view. For further detail, please see section 2.1 Frontend.

Key Benefits and Achievements of the MVC

Having decided to implement the MVC (Model View Controller) architecture for this project, we will be achieving both high cohesion and low coupling at the same time. This is due to the very

nature of the structure. This architectural choice allows for logical separation of all elements of the design into three main groups, which in turn promotes low coupling between the respective elements but high cohesion within the respective groups. Further benefits of this choice include the following:

- Simultaneous development: team members are able to work on different elements of the web stack at the same time without disturbing each other's work and possibly causing serious merge conflicts.
- Easy maintenance: since the model, view and controller don't necessarily depend on one another, applying changes is extremely easy and in most cases won't require any adjustments to the other elements.

2.3 User devices/Third-party components

Clearly, a major component of our system are the devices through which users cess our application. This is clearly a third-party component.

At this stage, we will be deploying 'Hello World' as a response web application that can be accessed through a modern browser (e.g. Chrome, Firefox, Safari). We will not be implementing native mobile versions of our application. Choosing to build a responsive web application means that we can provide both web and mobile users access to our website while only maintaining one codebase.

2.4 Deployment platform

Although it is not a requirement of this project, we will be deploying our application through cloud application platform Heroku. Heroku was chosen because it supports both Python and NodeJS applications and thus can be used to host both our frontend and backend. It can also be used for automated CI/CD pipelines from Github, allowing fast deployment as we build and enhance our application. Finally, we have prior experience deploying web applications on Heroku, thus avoiding a steep learning curve in this non-crucial aspect of the project.

3. Choice of development platform

The team's primary choice of development platform is Windows. This is simply because all the team members have Windows running on their machines and already have all previously mentioned programming languages installed and running. And while we could remote access our university machines, which run Linux, this method is unreliable at times due to an unstable internet connection and slow responses.

Additionally, Windows supports design applications, such as the Adobe suite, which the team are using to create all the design elements of the project. Which means, upon creating or editing

an element it can be included into the web application with minimal latency, and hence we are able to speed up the production process.

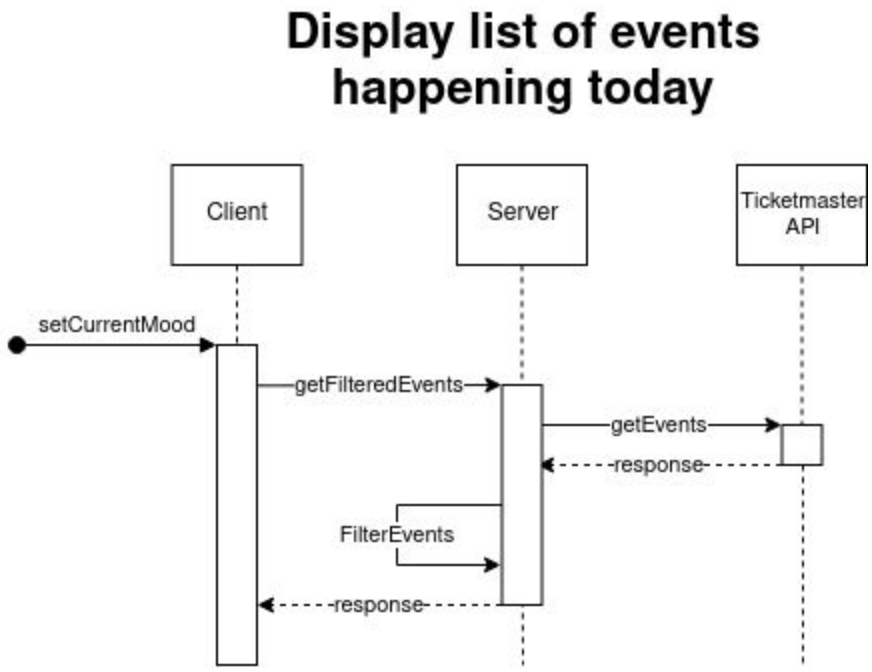
Part 2: Initial Software Design

User stories & sequence/interaction diagrams

Feature: Display list of events happening today

As a: Event attendee
So that: I can attend the events in my spare time
I want to: View a list of events happening today

Given I am on the main page
When I fill in “mood”
Then I should see a list of events happening today

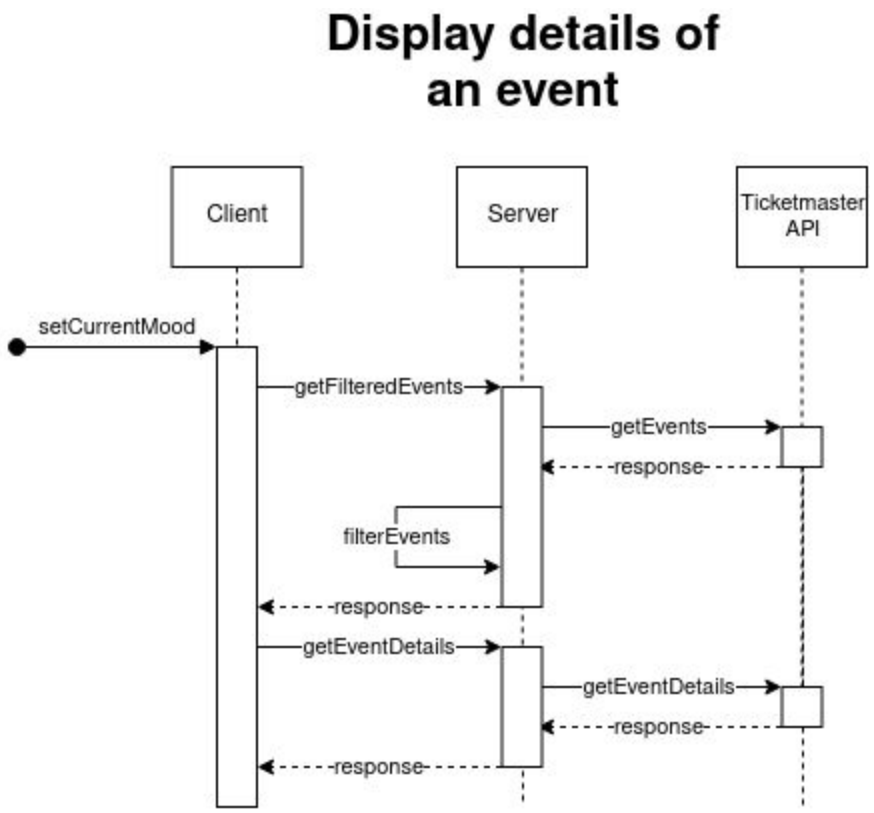


Feature: Display details of an event

As a: Event attendee

So that: I can understand more about the event
I want to: View the details of an event

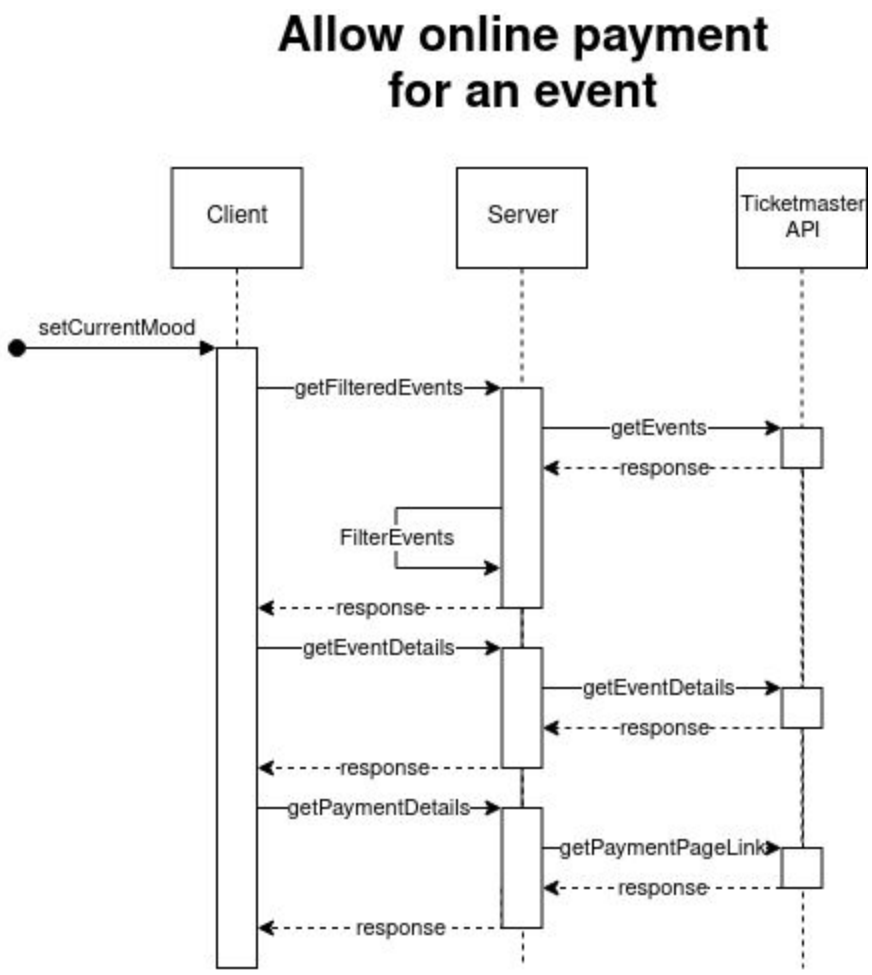
Given I am on the events page
When I click on one of the events
Then I should see the details about that event



Feature: Allow online payment for an event

As a: Event attendee
So that: I can attend a paid event
I want to: Pay online for a paid event

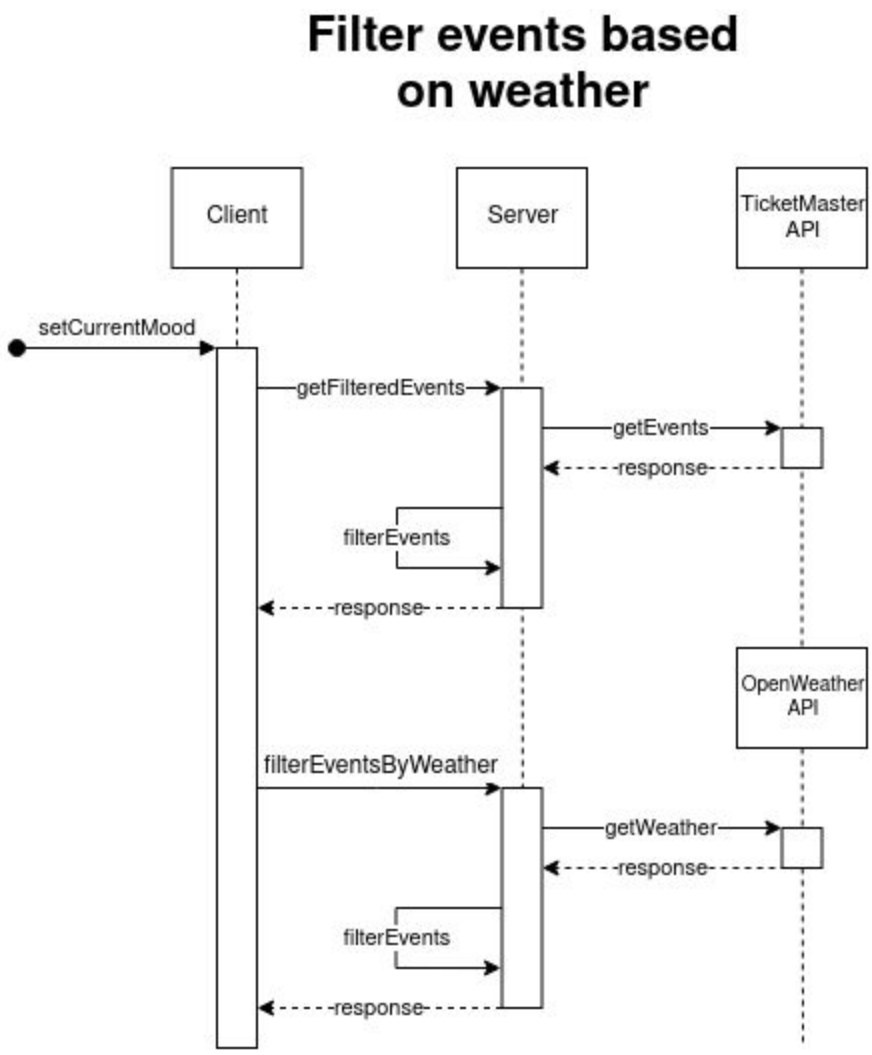
Given I am on the event details page
When I click on “Buy Ticket”
Then I should be on a newly-opened tab in a third-party event ticket page
Then I should be able to pay online for the event in the third-party event ticket page



Feature: Filter events based on weather

As a: Event attendee
So that: I don't have to see events that are not suitable for the weather at that time
I want to: Attend events without having to worry about the weather

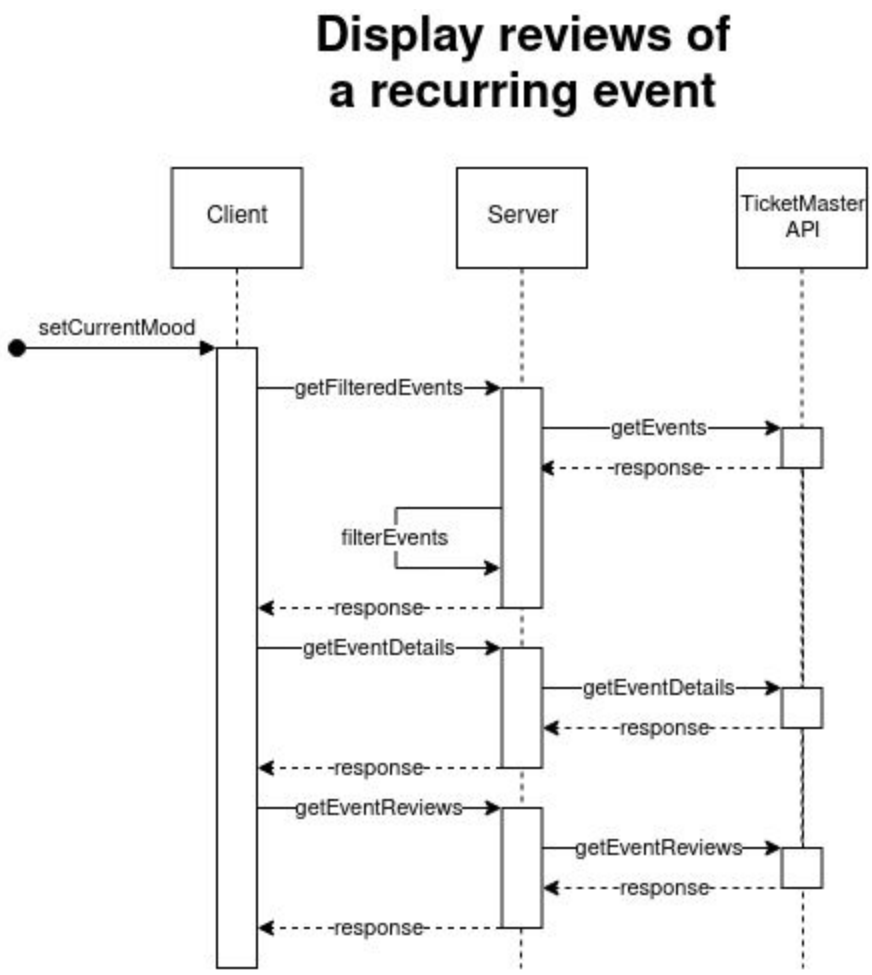
Given I am on the events page
When I click on "filter by weather"
Then I should see a list of events which are suitable for the weather at that time



Feature: Display reviews of a recurring event

As a: Event attendee
So that: I can make a more informed decision about whether to attend
I want to: See reviews of a recurring event

Given I am on the event details page
When I click on “View reviews”
Then I should see a list of reviews for the event



Feature: Display ratings of a recurring event

As a: Event attendee
So that: I can make a more informed decision about whether to attend
I want to: See ratings of a recurring event

Given I am on the event details page
Then I should see the ratings for the event

Display ratings of a recurring event