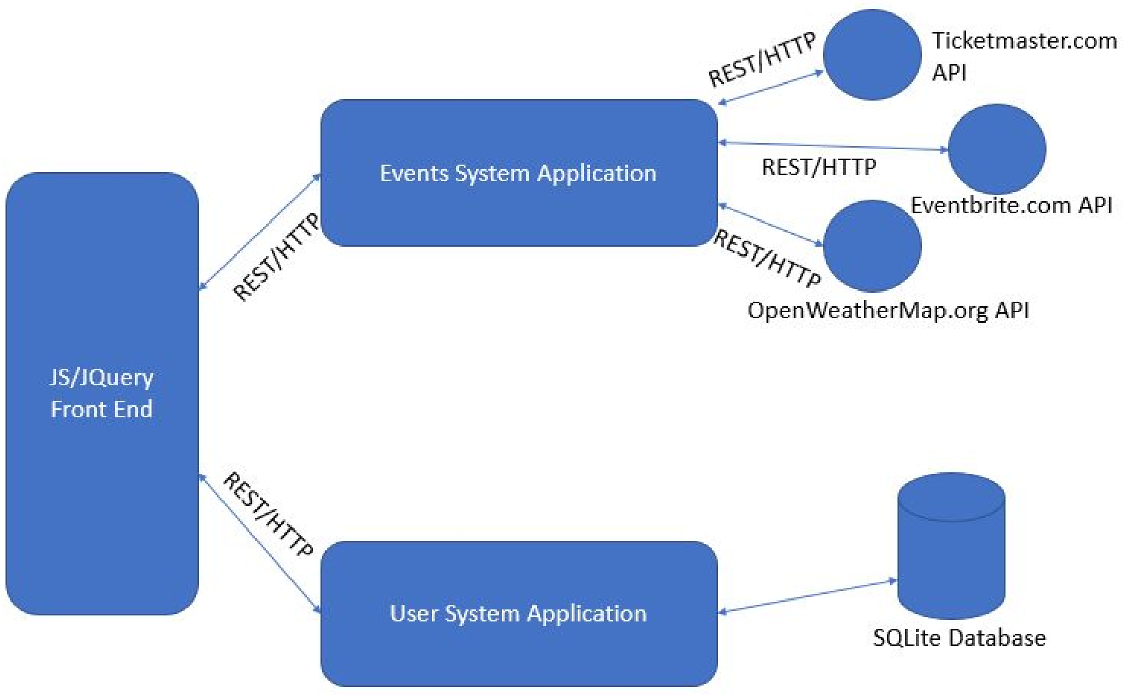
Frodo's Amazing Weather Machine

Team:
Declan Atkins - 14388146
Feargal Kavanagh - 14722459
Ross Gahan Suttle - 14340246

Overview of Problem
The problem faced was to develop a service that allowed users to obtain information (i.e. location, time, etc) with regard to upcoming events (e.g. music concerts), and provide them with the information about the expected forecast at the time the event was taking place. Also, we wanted to give the user the option of searching for an event based on what they had viewed before.

For a number of reasons a distributed system was necessary. Firstly without a distribution, each user would need their own API keys for each of the external services (i.e. Weather API and Event Host APIs) which would not make sense. On top of this by splitting the client and services we could provide the user with the web browser based interface that they would be used to. It also made sense to split the service that provides information about the events from the service that stored user data, as they would both be large services that accomplished very different tasks.

System Architecture



The above diagram shows the architecture of the system that we have implemented. The front end is connected to two separate REST applications. The first of these applications, the User System Application, has a static reference to a local SQLite database that contains all of the user information, from usernames and passwords, to the events that each user has clicked on. This information is passed to the front end in order to log the user in, and further data can be requested in order to be passed to the the second application, so that events can be recommended.

The front end also has the ability to get the user's location through the geolocation service provided through the HTML5 geolocation API. This, along with some of the information returned from the User System Application and some client inputted data, is passed to the Events System Application. This application requests a list of events from both Ticketmaster and Eventbrite, and then tries to retrieve the forecast information for the events from the Open Weather Map API. These results are then returned to the client browser for display.

Technology Choices

Technology	**Motivation for Use**
Swagger (SwaggerHub)	Provides effective services in terms of API design/editing, as well as allowing end users to easily interact with the resulting API
REST	A powerful tool allowing us to pass objects and other data in JSON format for use in the front end.
SQLite	Provides a simple to implement SQL database tool
JavaScript/JQuery	Provides a method of implementing a web based front end for the project, with JQuery providing the tools for making the necessary requests to the services

Team

Student Number	**Name**	**Assigned Task(s)**
14340246	Ross Gahan Suttle	- Development of Swagger API (SwaggerHub) and integrating it into project
14722459	Feargal Kavanagh	- Development of JS/JQuery front end for the system
14388146	Declan Atkins	- Development of Events Service accessing the Ticketmaster and Eventbrite APIs, as well as the Open Weather Map API. - Development of Users Service API, with accompanying SQLite Database

Task List
In order to achieve the end goals of this project, the completion of several task was required. These tasks included the development of the users system application, the events system application, the development of the web based front end, as well as the development of a Swagger API

Task 1 (Declan) Development of User System Application
The aim of this task was to create the database that would the user information as well the API that would be used to access it. The decision was made to use SQLite for the database due to the ease of its implementation and its ease of use. The database contains two simple tables, one for the usernames and passwords, and one for the events that each user has clicked on.

An API was created to access this database, which has endpoints used to both input and retrieve data from both of the tables. A static map was created in the application that mapped each user name to a UserInfo object. This was done so that when making future calls to the API, the client could pass the user name as a URL parameter instead of passing an entire user info object.

It was in the development of this application that we encountered our first distribution issue. When connecting to the front end we discovered that despite the fact that our testing using a Java client had worked fine, when we tried to make the requests from the browser, all of them errored due to no Access Control Allow Origin headers being present. As a result we had to add a CORS service to the application, and manually specify the headers to allow all origins.

Task 2 (Declan) Development of the Events System Application
We then proceeded to develop the Events Service, the aim of which was to provide an aggregation of events retrieved from Ticketmaster and Eventbrite and combine the results with data from the Open Weather Map API. This API has just two endpoints, one for taking in the data from the client and performing the search, and one for the client to use to retrieve the results of the search.

The Ticketmaster and Eventbrite APIs are quite different so two separate classes were created to implement their requests. Ticketmaster requests the location of the user in geohash format, so an external library was imported in order to create the geohash from a latitude-longitude specified location. Eventbrite returns the results in pages, with 50 events per page, so the decision was made to request that the results be sorted by date, and then only look at the first page of results. This was due to the fact that the free version of the Open Weather Map API only allowed us to get the forecast for the next 5 days.

A static map was created that mapped a location data object to a Weather Info object. This was done in order to avoid being rate limited by Open Weather Map, as for each of the events retrieved we wanted to get the weather information. We decided to use a fairly arbitrary threshold of 25km to specify a distance large enough to require different weather information. As a result, for most cases, if the search was limited to Dublin, for example, only one request would need to be made to Weather API.

Task 3 (Feargal) Development of the Front End
The aim of this task was to develop the frontend. This frontend is what the user uses to interact with the distributed system. The frontend consists of 2 main parts, the visual part which the user interacts with, and the part which interacts with the event and user system applications. The visual part was built upon a webpage using html, javascript, and css. The functionality of the frontend to interact with the distributed system was implemented using a combination of javascript, AJAX, and jQuery. AJAX requests were created using jQuery and javascript to query both the user system application, as well as the event system application. In total, 7 ajax requests were required to communicate with all of the system's functionality. Three were required to access the user application (obtain user's info from the database, login, and register). Two requests accessed the event system, these requests submitted a search and retrieved the results. The final two requests accessed both the event and the user system. The first associates a user with an event they have accessed in a search. This allows a search to be recommended to a specific user based on their previous search interest. The second requests the search to recommend to the user upon request. Each of these requests have typical AJAX form, with the url for the requests being the associated endpoint for that request. Most of the requests use JSON, this caused obtaining and manipulating the data to be simplified, as working with JSON in javascript is quite straightforward. This was especially true for the search results request as the results of a search is held in a single JSON object.

The information from some JSON objects had to be heavily manipulated to improve the user experience. For example the time of an event is simply given as a timestamp of the form ”2018-04-21 19:30:00”. In this case it would be far better for the user’s experience to have the date as a month and the time in a normal format.

The visual part of the frontend was implemented using html, javascript, and css. A minimalistic design was chosen to represent the frontend of the distributed system. It was deemed appropriate to have a visually appealing frontend to the system, as if a barebones frontend was portrayed, it would cause the user to view the system in a poorer light. The main part of the visual frontend was showing the search results in a visually appealing way. This was time consuming as each iteration required a search to be requested, and the speed of a search varied dramatically. The implementation of visualising the search results was achieved by using javascript to append html dynamically to the webpage. Styling using CSS was then used to format the html divs into a more visually appealing manner.

Task 4 (Ross) Swagger API Implementation
The aim of this task was to implement Swagger API and incorporate it into the project by ensuring that it effectively interacted with the corresponding parts of the other tasks of the project. The Swagger API was designed and developed through the use of SwaggerHub, an online API development tool, which provides several useful tools for API development, testing and debugging. The ease of use provided by SwaggerHub allowed the components of this task to be broken up into four simple headings i.e. path (addition to URL caused by a certain action, moving the end user closer to their desired outcome), definitions (commonly used elements), parameters (inputs associated with a given path) and responses (the possible outcomes with regard to each path). At first the task proved somewhat difficult, mainly due to having not used SwaggerHub before, but quickly became easier due to the gentle learning curve of SwaggerHub, and as such, as the project went on the integration of the API with corresponding elements of different tasks became much easier.

Reflection

Appropriateness of Technologies Used
Both REST and Swagger were well suited to the task that we faced. The simplistic nature of the SwaggerHub tools allows for easy implementation of the API as well as effective integration with other parts of the project. Similarly, REST provides a powerful tool that allowed us to send entire objects from our Java based APIs to the JavaScript front end, and from there to the other API. The REST applications are also quite easy to implement, making them even more appropriate.

Unexpected Problem Caused by Distribution
The requirement of a CORS service was something that we hadn’t anticipated, however the issue was very easy to resolve once we had researched and understood the problem fully. We

also had to be more strict with JSON syntax than maybe we would have been if we weren't working with two different languages, as when passing the list of results from the events service to the client browser for display we had to add keys for it to parsable from the JS.

Technology Limitations
As we were using free versions of the external APIs, we were limited in what we could do. Particularly with Open Weather Map, the limitation to a five day forecast meant that for a large number of the events we were unable to retrieve the weather information, which was a bit disappointing. As well as this, REST isn't really suited to serving a website, rather as a tool to be accessed from one. As a result we could only have a single webpage that accomplished everything, which is not that elegant. A separate web server to deal with the pages would probably have made more sense.

Technology Benefits
We have briefly discussed the benefits of using REST and Swagger, however they cannot be overstated. The gentle learning curve and (somewhat) simplistic nature of SwaggerHub allows for effective integration of Swagger API with the other elements of the project. REST was also a powerful tool, without which parts of the project would not have been possible, or at least would have been made much more difficult. Similarly JQuery was invaluable for simplifying the requests that we had to make to the APIs.

Lessons Learned
Over the course of the project we learned a great deal about the practical application of Distributed Systems, and how powerful they can be when used in the correct context. We learned about Swagger, and the excellent tool it provides for declaring what each part of the API is used for. We learned how to generate html dynamically as well as style it in a visually appealing manner. We also learned about development across multiple languages, through the linking of our Java APIs and our JS front end, and importantly about how much easier distribution makes that process.

Discovering the limitations of the tools we were working with was also an invaluable learning experience, and taught us a bit about why things are done in specific ways.