EventHype

Project Report

Artur Grigoryan
Thomas Halstead
Russell Fenenga
Jonnathan Petote

CS125

Abstract

Our project will help people search for events around them from a mobile application as well as the ability to create events. We believe that this is a major area that still has not implemented a system that can be searched through and that is what our team and project aims to solve.

Introduction and Problem Definition

Currently to find events on college campuses or in any community people rely on seeing flyers or being invited to a Facebook page by someone they know. This system is highly inefficient and there is no real way to search for specific types of events on the fly. These events could range from small events like attempting to get a pickup game of soccer together at a local park to huge events like concerts. Currently it is very hard to find a list of all these possible events going on around a user's location and that is what we aim to solve.

Our system will require multiple parts that will be outlined in this report. The first of which is our backend which will house our SQL database as well as our API for interacting with our database. This API will connect to our iOS app which will provide the users the functionality to view all the events currently active around and search for specific events. This document aims to outline the technical aspects of this project and how we are going to structure all the components of this system.

Current State of Art

Currently there are no systems that operate in the same area that we are planning on targeting. Facebook has events but searching for events isn't easy and they aren't sorted by location around you so there isn't any reason to use it for searching for events.

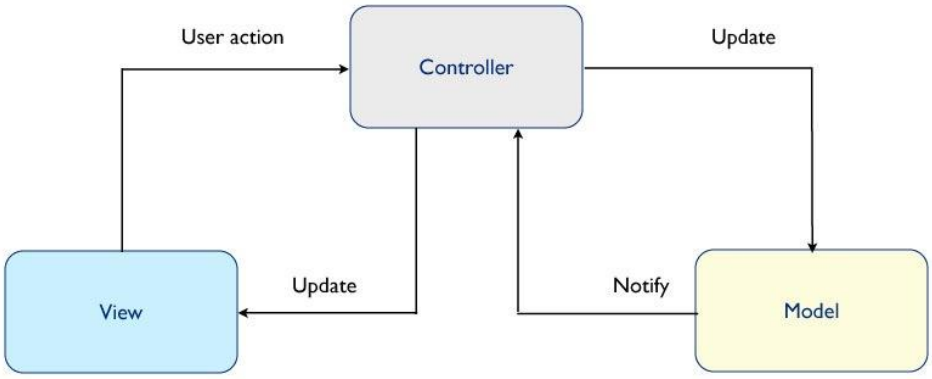
iOS App Functionality

The functionality that is available inside the iOS app is fairly robust and covers every area of functionality that our system provides. It allows users to be able to find events on a map relative to their current location then go into a detail page proving more information about that event. It also allows a user to input a new event into the system which will show up on the map. Along with the map view the app has a search screen where the user can search for an event by event name or event tag.

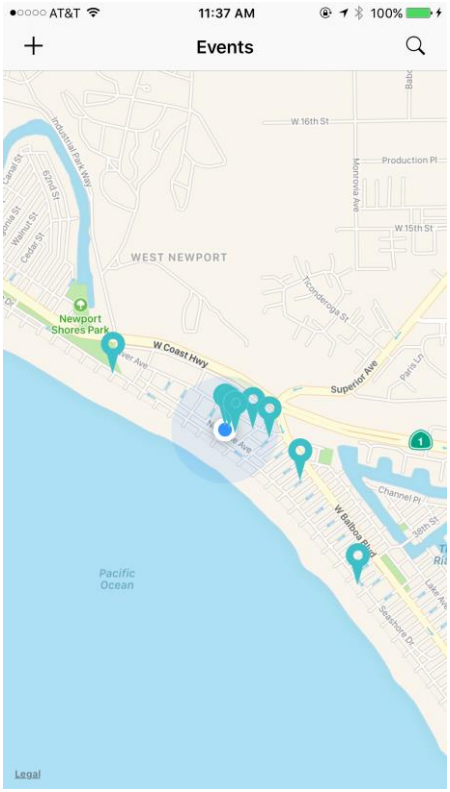
iOS App Architecture

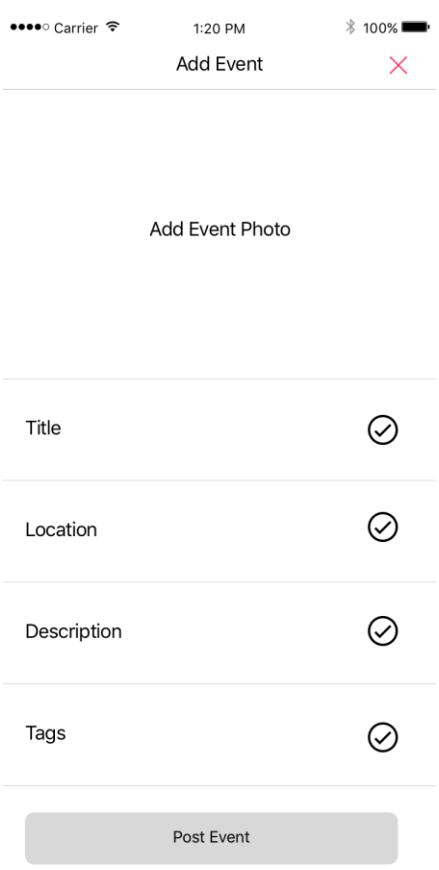
The general architecture of the iOS app is incredibly standard to fit within Apple's guidelines. The project is built using the Model View Controller design pattern. We decided to go with this choice because it is the standard for iOS apps and fits the needs of our project perfectly. The first thing we will discuss is how each part of the app works and how it is built. To start it is important to undress the MVC pattern which is

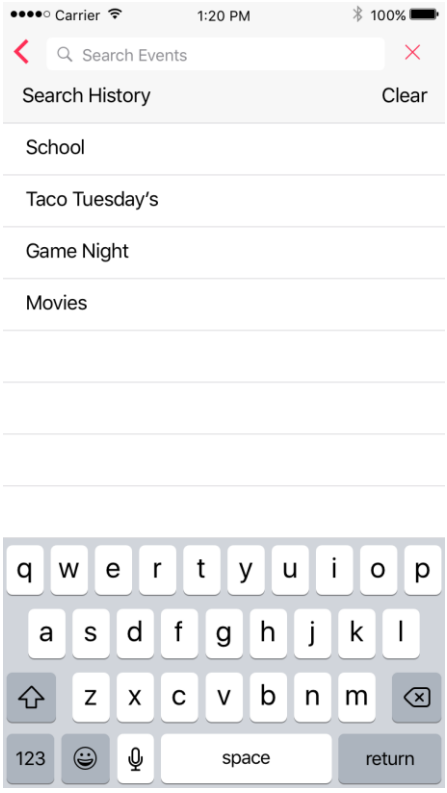
illustrated in the pictures below.



iOS Views







Mobile Model

The model of our app is the most imperative section of the app because without it functioning nothing in the app would work. Our model consists of 2 classes which handle loading from our API, as well as a class to represent our data. The first class I'm going to go over is our **Event**. The purpose of this class is to provide a structure for defining events inside of our app. It has 18 properties which represent every column in our database and then a dictionary that we keep updated with key value pairs. The class is used as follows:

```
Event *event = [[Event alloc] initWithParameters:eventDict];
```

where the eventDict parameter is the dictionary representing the event. By building this class it allows us to access event parameters much easier around the app.

The next portion of the model and the most important piece is the EventDataLoader. To first understand how this class functions it is necessary to understand how delegation works. Delegation allows us to "give objects a chance to coordinate their appearance and state with changes occurring elsewhere in a program, changes usually brought about by user actions. More importantly, delegation makes it possible for one object to alter the behavior of another object without the need to inherit from it. The delegate is almost always one of your custom objects, and by definition it incorporates application-specific logic that the generic and delegating object cannot possibly know itself." We use this pattern in our DataLoader by declaring two delegate methods EventsLoaded and EventCreated that then can be implemented in a class that conforms to the DataLoading Delegate. Our DataLoading delegate for loading all events is called as such from EventDataLoader:

```
[self.delegate sendEventData: arrayOfEventObjects];
```

then in our class the conforms to the delegate we provide an implementation as such:

```
-(void)sendEventData:(NSArray *)array {
    NSMutableArray *allPins = [[NSMutableArray alloc] init];
    [self.allEvents removeAllObjects]; //avoids duplicate events
    for(Event *event in array){

        EventAnnotation *eventAnnotation = [[EventAnnotation alloc]initWithEvent:event];
        [allPins addObject:eventAnnotation];
        [self.allEvents addObject:event];
    }
    [self.eventMapView addAnnotations:allPins];
}
```

This code snippet is from our EventMapViewController and is how we add the events to the app which we would not want to be inside of the DataLoader because then the model would have direct knowledge of the view which goes against the principles of Model View Controller. This way our model has no coupling to the view so we can safely update our loading methods without worrying about breaking our views. The next portion of the model that is important is that this class is a **Singleton** which means that only one copy of it can be made. We do this because we don't want multiple EventDataLoading objects all calling our API and updating our list of events as it can lead to synchronization issues. Another reason for making it a singleton is that it allows multiple different classes to conform to the delegate. The last piece of information on how this class is built is the individual functions for saving an event and loading all the events. For saving an event we use a GET request and serialize the Event's dictionary as a body of the request. For getting all the events we just make another GET request then serialize the array of dictionaries that represent events into our Event class.

Mobile Controller

The controller acts as the middleman between the Model and the View and in our case is going to performs 2 main tasks, handling user input the then relaying data from the model back to the view. Our class has multiple Controllers each of which corresponds with the view that we use. The first controller we will discuss is the EventMapViewController. This controller is responsible for handling user input on the map and displaying the events to user. This controller conforms to our model's

delegate so that when the view begins to load it makes a request to our model for the events then triggers our delegate method. The delegate method then creates new pins for each event and adds them to our map. This controller also handles transitioning views to the search screen, add event screen, and also the detail view for an event. For transitioning to the event detail screen we use a method called **segue** which allows us to pass data forward to the new screen. We do that so that we can pass the event to be

```
EventAnnotation *eventAnnotation = (EventAnnotation *)view.annotation;
Event *event = eventAnnotation.event;

// Get the storyboard named secondStoryBoard from the main bundle:
UIStoryboard *secondStoryBoard = [UIStoryboard storyboardWithName:@"EventContent" bundle:nil];

// Load the initial view controller from the storyboard.
// Set this by selecting 'Is Initial View Controller' on the appropriate view controller in the storyboard.
EventContentTableViewController *eventContent = [secondStoryBoard instantiateInitialViewController];

eventContent.myEvent = event;

UIStoryboardSegue *segue = [UIStoryboardSegue segueWithIdentifier:@"ToContent" source:self destination:eventContent
    performHandler:^(void) {
    //view transition/animation
    [self.navigationController pushViewController:eventContent animated:YES];
}];
[self prepareForSegue:segue sender:self];
[segue perform];
```

displayed forward so that the view can access the event information to display.

The next ViewController to discuss is the AddEventViewController which is used for creating new events inside of our app. The controller works by aggregating the user's input in all of the different fields then when the post event button is pressed it creates a new event object and sends it to our EventDataLoader class to save to our database. It then dismisses itself to present the map view again, where the new event with appear once it saves with the database.

The last view controller to discuss is the SearchViewController which handles the logic for searching inside of our app. The search controller works at a high level by keeping a copy of the list of events we want to search in then also another list that

contains the events that match the current search query. The searching happens locally within the app and doesn't have to make any calls to our API. This decision was made because when we load all the events onto the map we have already have a list of all the events and their related information. We then just give this list to the search controller to filter through. The actual logic for matching is relatively simple.

```
-(void)filterContentForSearchText:(NSString *)searchText{
    self.filteredEvents = self.allEventsBeingSearched;
    self.filteredEvents = [self.allEventsBeingSearched filteredArrayUsingPredicate:[NSPredicate predicateWithBlock:^BOOL
        (Event *event, NSDictionary *bindings) {
        NSRange matchOnEventName = [event.eventName rangeOfString:searchText options:NSCaseInsensitiveSearch];
        NSRange matchOnTag = [event.category rangeOfString:searchText options:NSCaseInsensitiveSearch];
        return ((matchOnEventName.location != NSNotFound) || (matchOnTag.location != NSNotFound));
    }]];
}
```

We use a custom defined predicate that looks to see if what the user is typing is found somewhere inside any of the event's names or tags. If it is we will add that into our filtered list which is the list that gets displayed. If a user taps one of the shown events, it will take them to the EventDetailController using the same logic discussed above.

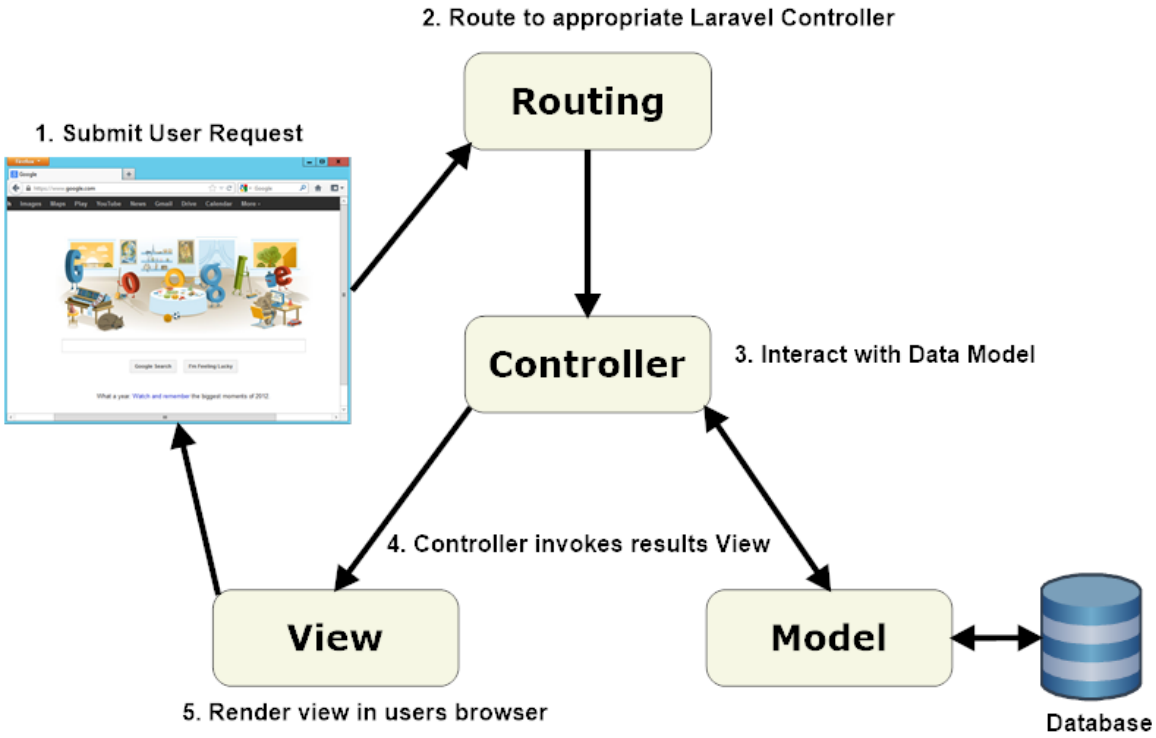
Mobile View

The last component of the architecture for the iOS app are the views. Each controller class corresponds with a view that it controls so we have an EventMap View, EventSearch View, an AddEvent View, and an EventDetail View. The views are all made inside of Xcode using their interface building tools then further customized inside of their respective view controllers to get the exact display we want. When building the views, we need to consider the different phone screen sizes and how this will affect how data is displayed. To solve this, we use a system called AutoLayout which allows us to

layout elements relative to other elements so that when screen sizes change we can change the layout or size of the elements in the view so that it still functions properly.

Desktop App Architecture

The API for the entire project requires a MySQL Database with the ability to make POST and GET API requests. To accommodate the API requests we are using a Laravel MVC framework. Laravel has an ORM that will make the queries against the MySQL Database, and the framework Models allow us to make modifications to the data before serving it via API. The modifications include evaluation algorithms, adjustments dependent on the client location, and other requirements that the project might have. The Desktop App will use the MVC model mentioned previously, and an additional routing server that formats incoming user requests and parses the URI.



Desktop Model

The Laravel Model will use the Eloquent ORM to make the queries and evaluate the data. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table.

It will also be creating the index, based on the query data returned. We will be using the "Faker" OSS repo to create the initial Event data for the application

Desktop Controller

Laravel Controllers can group related request handling logic into a single class. The Controller for the Desktop App will serve two functions. First, it will format the Model data in JSON to be returned to the Mobile or Desktop client. Second, since we're also creating a Desktop version of the app, the Controller of the Desktop App will create the objects that are required to be rendered on the Desktop Client view.

Desktop View

The Laravel View – called Blade – allows the controller data to be formatted using HTML/CSS/JS and be displayed on the user's device. We will use the Laravel Blade to render the client pages on the screen. In conjunction with using the Blade to perform server-side logic in the browser client, Google's Map API will be used on the client side to display information about events matching the user's query, overlaid on a map and centered at the user's location. The client will automatically acquire the user's location through HTML 5 geolocation, with the user's permission, to be used as part of the event query. By default, the desktop view will display events for the current

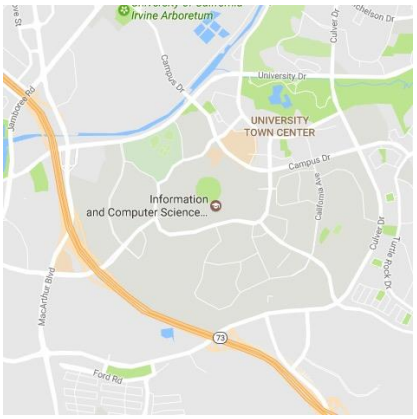
day. The user may then modify the parameters for events of interest and the page will send a request to the server to update the map with new query results.

Software

For the iOS portion of this project there are multiple pieces of software we are going to use. For designing the user interface of the app we are going to us Sketch which is a vector designing software and Paintcode which allows us to design interactive elements and it generates native code for them. The app will be written in Objective C as opposed to Swift. We decided to go with Objective C because of the stability that it has with Cocoa's Library APIs vs. Swift which is still evolving and breaks a lot if a new version were to release while we are working on this project. When it comes to building the actual app we are going to be using Xcode which is Apple's IDE for iOS Development.

One major component of this project is going to be making calculations based off of location and distances so we are going to need to write software that handles the math for calculating what events are inside of the area defined by the user.

Event Database and Index

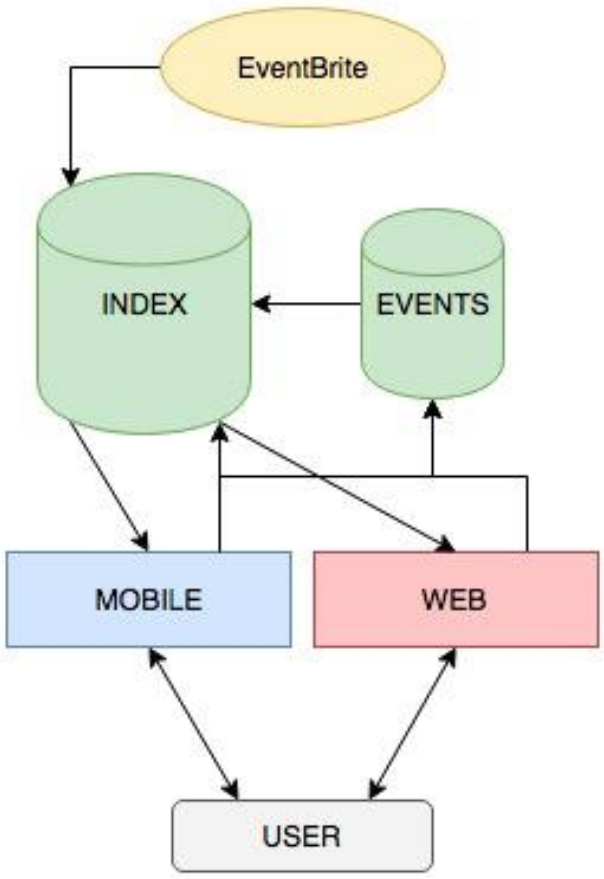


EventHype is implementing and using Eventbrite's API. We collect/update/remove events in the 10mile X 10mile radius of UCI. The database is updated on user request and user input, while the index is updated daily.

We found that Eventbrite has a 500 query/day limitation for free accounts. Initially, when we started implementing the Eventbrite API, our OAuth tokens got revoked due to "malicious behavior". This was because we were trying to implement a live event feed generation upon user request.

Due to Eventbrite's databases being in BCN Form, we had to make 4 API calls for each event; thus, for a radius of 10 miles, we ended up with over 1200 API calls (hence, the malicious behavior).

Our solution to the problem was creating an index that would maintain the mocked dataset generated by EventHype, and add a normalized datasets of events from Eventbrite. The index would have few key features: any user post would immediately get added to index (as well as the event database table), while Eventbrite events would be updated nightly, with a single API call to Eventbrite. This allowed us to minimize the bandwidth used, while allowing users to have access to a live data stream; while staying within Eventbrite API quotas.



User Studies and Evaluation

For us, adding an analytics library into the app is going to be the easiest way for us gather information on the how the users are using the app. This will gather info on how many of the users are creating events vs. looking at events. For testing the app we

will have to populate the app with demo data until people start actually using the application.

Conclusion

Our app aims to solve the problem of finding local events to get involved in. There are systems that exist for creating events and interacting with attendees but there are no systems for actively being able to find ones based off the user's location. Our filtering system will allow users to only search for events that they are interested in getting involved with. We hope that our app will provide a new avenue for people to get connected with people who they share similar interests with. Our next steps for the project would be to implement a chatting system that way users can interact with other event attendees. This would also require building an authentication system and saving events that a user has joined to their profile. Another interesting idea would be to let the user subscribe to a type of event an event is created with that tag it would notify the user that a new event with that type had been created. This is still search as we are giving the user what they want but much more implicitly.

References

https://developer.apple.com

https://www.raywenderlich.com/133121/testflight-tutorial-ios-beta-testing