

# MyAlmanack

*Justin Oakley*

*Mark Bures*

*Mike Resnik*

*Pooya Motee*

*Hao Zhang*

# Table of Contents

## 1. Project Definition

- Why (it is needed)
- What (is the goal of the project)
- How (how will it be achieved)

## 2. Project Requirements

- Functional
- Usability
  - User interface
  - Performance
- System
  - Hardware
  - Software
  - Database
- Security

## 3. Project Specification

- Focus / Domain / Area
- Libraries / Frameworks / Development Environment
- Platform
- Genre

## 4. System – Design Perspective

- Identify subsystems – design point of view
  - Illustrate with class, use-case, UML, sequence ..... diagrams
  - Design choices (Optional)
- Sub-System Communication (Diagram and Description)
  - Controls
  - I/O
  - DataFlow
- Entity Relationship Model
- Overall operation - System Model
  - Simplified Sub-system to System interaction

## 5. System – Analysis Perspective

- Identify subsystems – analysis point of view
- System (Tables and Description)
  - Data analysis
    - Data dictionary (Table - Name, Data Type, Description)
  - Process models
- Algorithm Analysis
  - Big - O analysis of overall System and Sub-Systems

## 6. Project Scrum Report

- Product Backlog (Table / Diagram)
- Sprint Backlog (Table / Diagram)
- Burndown Chart

### 7.1 Subsystem: User Interface – Mike & Mark

- Initial design and model
  - Illustrate with class, use-case, UML, sequence ..... diagrams
  - Design choices
- Data dictionary
- If refined (changed over the course of project)
  - Reason for refinement (Pro versus Con)
  - Changes from initial model
  - Refined model analysis
  - Refined design (Diagram and Description)
- Scrum Backlog (Product and Sprint - Link to Section 6)
- Coding
  - Approach (Functional, OOP)
  - Language
- User training
  - Training / User manual (needed for final report)
- Testing

### 7.2 Subsystem: Database – Justin

- Initial design and model
  - Illustrate with class, use-case, UML, sequence ..... diagrams

- Design choices
- Data dictionary
- If refined (changed over the course of project)
  - Reason for refinement (Pro versus Con)
  - Changes from initial model
  - Refined model analysis
  - Refined design (Diagram and Description)
- Scrum Backlog (Product and Sprint - [Link to Section 6](#))
- Coding
  - Approach (Functional, OOP)
  - Language
- User training
  - Training / User manual (needed for final report)
- Testing

### 7.3 [Subsystem: Calendar Data](#) – Pooya

- Initial design and model
  - Illustrate with class, use-case, UML, sequence ..... diagrams
  - Design choices
- Data dictionary
- If refined (changed over the course of project)
  - Reason for refinement (Pro versus Con)
  - Changes from initial model
  - Refined model analysis
  - Refined design (Diagram and Description)
- Scrum Backlog (Product and Sprint - [Link to Section 6](#))
- Coding
  - Approach (Functional, OOP)
  - Language
- User training
  - Training / User manual (needed for final report)
- Testing

### 7.4 [Subsystem: Authentication](#) – Hao

- Initial design and model
  - Illustrate with class, use-case, UML, sequence ..... diagrams
  - Design choices
- Data dictionary

- If refined (changed over the course of project)
  - Reason for refinement (Pro versus Con)
  - Changes from initial model
  - Refined model analysis
  - Refined design (Diagram and Description)
- Scrum Backlog (Product and Sprint - Link to Section 6)
- Coding
  - Approach (Functional, OOP)
  - Language
- User training
  - Training / User manual (needed for final report)
- Testing

### 7.5 Subsystem: Authorization – Hao

- Initial design and model
  - Illustrate with class, use-case, UML, sequence ..... diagrams
  - Design choices
- Data dictionary
- If refined (changed over the course of project)
  - Reason for refinement (Pro versus Con)
  - Changes from initial model
  - Refined model analysis
  - Refined design (Diagram and Description)
- Scrum Backlog (Product and Sprint - Link to Section 6)
- Coding
  - Approach (Functional, OOP)
  - Language
- User training
  - Training / User manual (needed for final report)
- Testing

### 8. Complete System

- Final software/hardware product
- Source code and user manual – screenshots as needed - Technical report
  - Github Link
- Evaluation by client and instructor
- Team Member Descriptions

# 1. Project Definition

MyAlmanack is a productivity-focused web application that reduces the complexity of schedule management. This application was designed to enhance: personal agenda management, user-controlled shared schedule management, group schedule management, free-time calculation for one or many users, and planning special occasions. Social elements provide users the ability to connect their schedules and view shared freetime on their respective calendars.

Users possess a private online schedule system and can selectively share schedule information with friends and groups. Schedule information can be displayed in different calendar views which allows for multi-user schedule management between contacts. Groups can be formed to allow for greater multi-user schedule management and event planning. Schedule management using the MyAlmanack system involves automated free-time calculation for one or many users. The planning of events and special occasions is simplified with this application due to the manual control of event-times, the option for repeating certain events, and the ability to blacklist or whitelist specific users from viewing events.

## **2. Project Requirements**

### **Functional**

Our project needs a calendar which allows the user to add / edit events, and view free time between a select group of members. This will be achieved through a standard profile system and general social network capabilities (friends / groups). An invite/event system on top of selective viewing provides additional event functionality. The data for the users need to be stored in a database system, which will be displayed in a responsive user interface.

### **Usability**

Our project needs to have high usability, which will be increased by the responsiveness of our user interface. High usability meaning: an intuitive calendar graphic, linkability to different pages, and profile customization. This will be achieved by dynamically displaying flexible boxes on the user's screen, which will also support the use of MyAlmanack on varying displays.

### **System**

The hardware in MyAlmanack has both a server and clients. The server is any computer which is capable of running the Python runtime and has external network access. A client is any device that has a browser, thus widening the audience to all who have a device with internet capabilities. The environment for MyAlmanack's back-end is in Django, Python, and PostgreSQL. The environment for MyAlmanack's front-end is in HTML, Bootstrap, CSS, Javascript, and jQuery.

### **Security**

Only users can edit their own calendar, others can add to their invite calendar and will ask them to confirm the addition to their personal calendar. Only friends can view each others calendars, and friends can only see events if they are allowed to. There is a secure login system which maps to the user's profile information using a unique key. Policy checking will occur in the back-end (as a middleware) to ensure that users are authorized to perform certain actions before reading from and writing to the database.

## 3. Project Specifications

### Domain

The application is targeted at individuals who are both over the age of eighteen and willing to share their schedules with their acquaintances through a website. MyAlmanack is designed to accommodate groups, but individuals can also use the site for personal use and schedule management. Some of the target audiences include: student groups (for scheduling group meetings), professors / students (for scheduling exams and class meetings), and managers / workers (for scheduling shifts and work meetings).

### Development

#### Front-end

The structure for the user interface is written in HTML and CSS, and is made flexible to resizing with Bootstrap 4. The dynamic data population for the user interface is written in a combination of native JavaScript and jQuery. The calendar is generated in pure JavaScript by the user interface subsystem, which enables it to be responsive to event creation, viewing, and resizing. HTTP POST requests with Django forms and AJAX requests are used to send data from the front-end to the back-end.

#### Back-end

The structure for the back-end of MyAlmanack is written in Python 3 in coordination with the web framework Django 2. The database server runs on PostgreSQL through Heroku, which also hosts MyAlmanack. The back-end sends information to the front-end in the form of both JSON strings and query sets. The authentication process uses Google Firebase, which allows users to sign in with many different account types. Firebase allows MyAlmanack to both extend its range in audience and allows us to forego storing usernames and passwords. The authorization subsystem is written as an attribute-based access control authority and middleware to allow the user interface subsystem to check a user's access before performing reads or writes on the database from the user interface.

#### Development Environment

For version control, we use Git in conjunction with GitHub, and for web and database server hosting we use Heroku. Most coding for MyAlmanack is done in the Sublime Text as it supports several different programming languages and has a low memory usage compared to large integrated development environments.



## **Platform**

MyAlmanack will work on any device (desktop and mobile) with an active internet connection, a modern web browser, and a sufficient screen resolution. JavaScript must be enabled in the web browser for the application to work correctly.

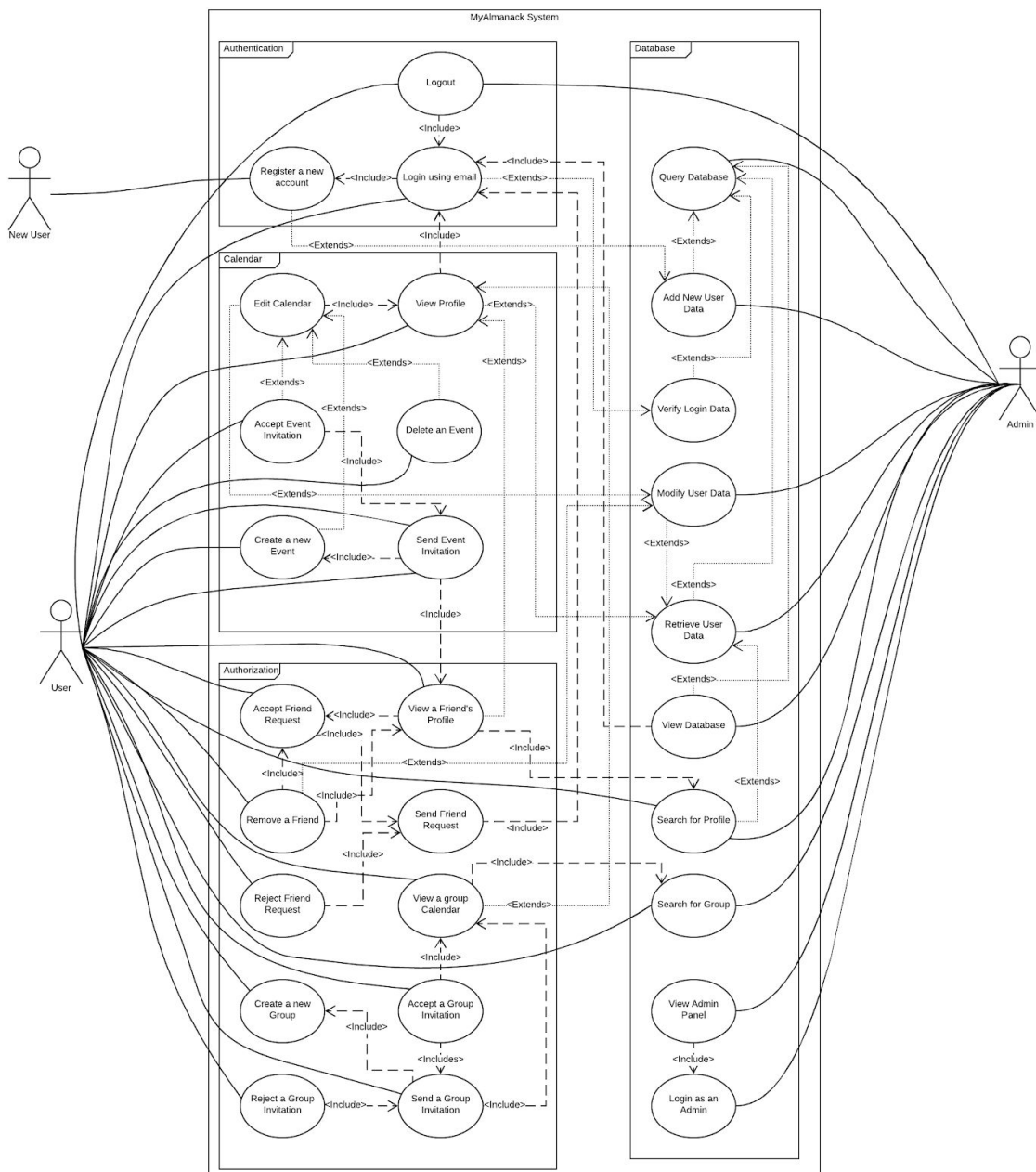
## **Genre**

MyAlmanack is a social media application with a focus on productivity. Both elements can be used independently (social media / productivity) as individuals can use this program for networking, personal schedule management, or both.

## 4. System – Design Perspective

### Subsystems

**Use Case Description:** There are three possible actors in our system: new users, users, and admins. New users only have access to registering with our system. Users have access to most database operations with restrictions caused by the authorization subsystem. The Admin has direct access to the database and can search, add, remove, and modify all data in a separate user-interface.



## User Interface

All use cases must be visualized using a user interface. Some use cases require the user interface to make new windows or views to facilitate the inputting of data. All pages required by the user interface are the **Default View, Profile View, Edit Profile View, Search View, and Group View**. All windows required by the user interface are: **Create Event, Create Group, Calendar Month, Week, Day and List Views, and View Event**. Some of these pages and windows will change based on the information in the database.

## Database

All use cases require reading from and writing to the database which are facilitated by a combination of raw SQL queries and Django model methods. All information needed by the user interface calls queries made by the database subsystem.

## Authentication

### Use Case 1. Register a new account

**Actor:** New User

**Precondition:** The user has logged in using a Firebase option and has been redirected to the edit profile screen.

**Primary Path:**

1. New User enters his / her first and last name.
2. New User enters a valid alias.
3. New User enters a valid birthday.

**Postcondition:** User is logged in, his / her profile page is displayed.

**Subsystems:** Authentication, User Interface

### Use Case 2. Login using email

**Actor:** User

**Precondition:** The user has registered an account using an email address and is currently logged out.

**Primary Path:**

1. User clicks the login button at the top right of the screen, loading a login popup.
2. User enters a valid email which takes them to an external authentication site.
3. User clicks the login button.

**Postcondition:** User is redirected to our website. User is logged in and his / her profile page is displayed.

**Subsystem:** Authentication

**Use Case 3. Logout**

**Actor:** User

**Precondition:** The user is logged in.

**Primary Path:**

1. User clicks on the “Logout” button on the navigation bar.
2. User is redirected to the default page.

**Postcondition:** User is logged out of the system.

**Subsystem:** Authentication

**Use Case 4. View Profile**

**Actor:** User

**Precondition:** The user is logged in.

**Primary Path:**

1. User clicks the “MyAlmanack” button on the navigation bar.

**Postcondition:** The user’s profile page is displayed.

**Subsystems:** Authentication, User Interface

**Authorization**

**Use Case 5. Send Friend Request**

**Actor:** User

**Precondition:** User is logged in, both users are registered.

**Primary Path:**

1. User searches for the friend to add
2. User clicks on his / her profile link from the list
3. User clicks on the “Add Friend” button on his / her profile

**Postcondition:** User was notified of a friend request.

**Subsystem:** Authorization

**Use Case 6. Accept Friend Request**

**Actor:** User

**Precondition:** User is logged in and has been sent a friend request.

**Primary Path:**

1. User clicks the notification icon on his / her dashboard.
2. User clicks on the friend request.
3. User clicks the “Accept” button

**Postcondition:** Users are notified the request was accepted.

**Subsystem:** Authorization

**Use Case 7. Remove a Friend**

**Actor:** User

**Precondition:** User is logged in and is friends with the person he / she is trying to remove.

**Primary Path:**

1. User searches for the friend to remove.
2. User clicks on the friend in the search.
3. User clicks "Remove Friend" on his / her profile.

**Alternate Path:**

1. User navigates to his / her profile.
2. User clicks the friend's alias from his / her friends list.
3. User clicks "Remove Friend" on his / her profile.

**Postcondition:** The user and the friend are removed from each other's friends lists and view access has been removed for each.

**Subsystem:** Authorization

**Use Case 8. Reject Friend Request**

**Actor:** User

**Precondition:** User is logged in and has been sent a friend request.

**Primary Path:**

1. User clicks the notification icon on his / her dashboard.
2. User clicks on the friend request.
3. User clicks the "Reject" button

**Postcondition:** User is not friends with the other user.

**Subsystem:** Authorization

**Use Case 9. View a friend's profile**

**Actor:** User

**Precondition:** User is logged in.

**Primary Path:**

1. User clicks the search bar on top of the screen.
2. User enters the name or username of the friend to search.
3. User clicks the "Users" tab.
4. User clicks on the friend's alias.

**Alternate Path:**

1. User navigates to his / her profile.
2. User clicks the friend's alias from his / her friends list.

**Postcondition:** Friend's profile is loaded.

**Subsystem:** Authorization

**Use Case 10.** View a group calendar.

**Actor:** User

**Precondition:** User is logged in.

**Primary Path:**

1. User clicks the search bar on top of the screen.
2. User enters the name of the group to search.
3. User clicks the "Group" tab.
4. User clicks on the correct group name.

**Alternate Path:**

1. User navigates to his / her profile.
2. User clicks the group's name from his / her groups list.

**Postcondition:** Group's page and calendar are loaded.

**Subsystem:** Authorization

**Use Case 11.** Create a new Group

**Actor:** User

**Precondition:** User is logged in.

**Primary Path:**

1. User clicks the "Create Group" button on the navigation bar
2. User enters the name for the group.
3. (Optional) User enters the description for the group.
4. User selects the friends to add to the group.
5. User clicks the "Create Group" button.

**Postcondition(s):** The group is created, group invitations have been sent to the friends that the user selected. The group page and calendar are opened.

**Subsystem:** Authorization

**Use Case 12.** Send a Group Invitation

**Actor:** User

**Precondition:** User is on the group page and is a group admin.

**Primary Path:**

1. User clicks the Plus button next to members.
2. User selects the friends to add to the group.
3. User clicks the "Invite" button.

**Postcondition:** Group invitations have been sent to the friends the user selected.

**Subsystem:** Authorization

**Use Case 13. Accept Group Invitation**

**Actor:** User

**Precondition:** User is logged in and has been invited to a group by another user.

**Primary Path:**

1. User clicks the “Groups” icon on the navigation bar.
2. User clicks on the group request.
3. User clicks the “Confirm” button.

**Postcondition:** User is added to the group. The notification is marked as read. The group calendar is opened.

**Subsystem:** Authorization

**Use Case 14. Reject Group Invitation**

**Actor:** User

**Precondition:** User is logged in and has been invited to a group by another user.

**Primary Path:**

1. User clicks the “Groups” icon on the navigation bar.
2. User clicks on the group request.
3. User clicks the “Reject” button.

**Postcondition:** User is not added to the group. The notification is marked as read.

**Subsystem:** Authorization

**Use Case 15. Create a new event**

**Actor:** User

**Precondition:** The user is logged in to the system and a calendar view is loaded.

**Primary Path:**

1. User clicks the “Create New Event” button, which loads a smaller window for entering event data.
2. User names the event to be created.
3. User selects valid future start and end times for the event.
4. User selects friends to invite to the event.
5. (Optional) User selects which friends to whitelist or blacklist from the event.
6. (Optional) User selects the repeat pattern for the event.
7. User clicks the “Confirm Event” button.

**Postcondition:** Event is created and those invited have been sent a group invitation request for the event.

**Subsystem:** Authorization

**Use Case 16.** Accept an event invitation

**Actor:** User

**Precondition:** User is logged in and has been invited to an event by another user.

**Primary Path:**

1. User clicks the notification icon on the navigation bar.
2. User clicks on the event request.
3. User clicks the “Confirm” button.

**Postcondition:** Event is added to the User’s calendar. The notification is marked as read.

**Subsystem:** Authorization

**Use Case 17.** Reject an event invitation

**Actor:** User

**Precondition:** User is logged in and has been invited to an event by another user.

**Primary Path:**

1. User clicks the notification icon on the navigation bar.
2. User clicks on the event request.
3. User clicks the “Reject” button.

**Postcondition:** Event is added to the User’s calendar. The notification is marked as read.

**Subsystem:** Authorization

## **Entity-Relationship Diagram**

(Entity-Relationship Diagram can be found in [Section 7.2](#))



## 5. System – Analysis Perspective

### Subsystem Analysis

#### Mark & Mike – User Interface

The purpose of the User Interface is to provide a graphical interface for the other subsystems. The space complexity of the user interface depends on how many resources can be reused when making the user interface. The time complexity depends on the external utilities we use, our own scripts, and django's built-in functionality. The highest complexity script is either loading or updating the calendar. The user interface is also responsible for displaying the authorization restrictions on users in friends and groups. This means friend / group relationships and event display are also responsibilities of the user interface.

By representing the data and relationships of the other subsystems, the user interface creates an intuitive environment for someone to store social and personal information, while protecting user data through requests and authorization.

#### Justin – Database System

The system utilizes ten different data tables in a PostgreSQL database for the main application's functionality. These tables can be categorized based on their role in the application's components, which are user-profiles, groups, the contact-list system, the event-system, and the invite-system.

In the profile table, it stores individual user information and user events. The contact-list table contains inboxes for various sent and received invites/requests for a specific user, a list of current "friendships" with other users, and a list of current group memberships. The event table contains information about the events themselves, start and end dates, participating users, whitelists and blacklists, and event creators and administrators; as for inheritance, the repeat-event table is the child of the event table and contains information that assists the application in repeating events. Finally the invite table is solely used for containing basic invite information and being the parent table needed for creating various types of invites and requests. It should be noted that there is a major difference between invites and requests: invites can be sent to multiple users and allows said users to join either an event or a group while requests can only be sent to one person and allow for users to either gain "friends" or memberships to a group.

## Hao – Authentication & Authorization

### Authentication (Login / Registration)

The system will leverage Google Firebase's authentication service. In doing so, Firebase will return an authenticated user with a unique identifier associated with the user in the Firebase application and an identification token to verify the user with MyAlmanack's back-end. The database will not need to handle or store passwords, as Firebase will take care of it. The front-end should serve the internal/external login/registration page, then retrieve the authenticated user object once login/registration is successful.

### Authorization (Permission)

The system will use an attribute-based access control (ABAC) model for its dynamism and flexibility. The use of an ABAC model allows for fine-tuned permission authorization based on static attributes (i.e. whether a user is in another's contact list) and dynamic attributes (i.e. whether an event can be displayed based on a set start time). The system will take in authorization requests in the tuple of the following:

**Subject:** The user making the request

**Action:** What the subject is doing with respect to the object

**Resource:** The resource the request is acting on

**Context (optional):** Additional information that might affect the system's authorization evaluation

This makes converting authorization requests from plain English into code simple. For example, if a user named John would like to view Mary's profile:

**Subject:** [User] John

**Action:** [Action] VIEW\_USER\_PROFILE

**Resource:** [User] Mary

**Context:** Blocked = False

The system will then evaluate the requests based on dynamic policies and return whether the request is permitted, denied, or not applicable.

## Pooya – Calendar Logistics

**Event Repeat Handler:** This method takes in a start\_date and end\_date and generates repeated events based on the days of the week requested.

**Generate Days of the Month:** This method takes in a month and a year value and returns the whole month in a 2d array of 6x7 with the number of days in each index for the UI subsystem to be able to fill in the month view grid with the correct values.

**Freetime Calculator:** This method takes in a list of current events from the database and a threshold value and sorts the list by start times and then compares each index's end\_date with the next index's start\_date to see if there's any free time between them. It then returns an array of the free time found if they're greater than the threshold.

**Events Per Day:** This method takes in a list of current events from the database and a start\_date and end\_date value. It returns an array of [end - start] indexes and sorts the events in the list and puts them in the correct index of the array.

**Freetime Per Day:** This method takes in a list of current events from the database and calls the freetime class and takes the list generated and it creates an array of [end-start] indexes and sorts the free time list and puts them in the correct index in the array and returns the array.

**Conflict Finder:** Takes in a new event and checks the list of current events from the database and returns true if it conflicts with any current events and false if it doesn't.

## Data Dictionary

(Data dictionary can be found in [Section 7.2](#))

## Algorithm Analysis

### User Interface

#### Viewing a Profile (Personal or Friend's):

Time Complexity:  $O(\log(n) + p_c + L*m)$

Where  $n$  is the number of users in the database,  $p_c$  is the complexity of loading the calendar,  $L$  is the number of models to load, and  $m$  is the complexity of loading a model in Django.

The user ID for the profile to view is referenced, which means viewing a profile would query the database using ID. Viewing a profile also depends on the efficiency of the Calendar utility and how many models are loaded by the user-interface.

#### Searching for a User / Group:

Time Complexity:  $O(n + L*m)$

Where  $n$  is the number of users in the database,  $L$  is the number of models to load, and  $m$  is the complexity of loading our models in Django.

Because this search uses a database query that doesn't rely on ID's, the worst time complexity for searching by a non-sorted field (like username, name, group) is  $O(n)$ . For the search we will restrict the model to only loading the first 25 closest matching strings based on the user's search.

**Viewing a Group Calendar:**

Time Complexity:  $O(n + p_c + L \cdot m)$

Where  $n$  is the number of users in the database,  $p_c$  is the complexity of using FullCalendar, an external Calendar viewing utility,  $L$  is the number of models to load, and  $m$  is the complexity of loading a model in Django.

The time complexity relies on querying the database without using an ID, the efficiency of the FullCalendar utility and the models we make for the group calendar. This can be done in one pass as the database can add all entries matching a specific entry into a list when querying. The number of models is slightly larger than on a user profile because this page requires a group menu as well.

**Note on Loading Models on Different Pages:**

The documentation for Django's time / space complexity of its model and view objects aren't publicly available. This means we're assuming that the complexity of loading a view is composed of the models it contains and the scripts we write for the specific page. Some pages require the same model, like the dashboard. We reuse a view template for viewing a personal calendar and a friend's calendar to reduce redundancy and improve uniformity. **All other use cases and functionality accessed by a user interface** will have an added time-complexity of  $L \cdot m$  where  $L$  is the number of models on the page and  $m$  is the complexity of loading a model.

**Database****Querying the Database using an ID:**

Time Complexity:  $O(\log(n))$

Where  $n$  is the number of users in the database.

Because the database is indexed by ID's, the time complexity is  $O(\log(n))$  for a query in the PostgreSQL database. This means that the time complexity of anything that depends on querying the database using an ID will have a time complexity of  $O(p) > O(\log(n))$ .

**Querying the database (General):**

Time Complexity:  $O(n)$

Where  $n$  is the number of users in the database.

The user will perform a search query using a specified field which will be compared to users taken out in batches in the database for similarity comparison (for now using a constant-time algorithm such as Soundex). This is in  $O(n)$  instead of  $O(\log(n))$  because the

**Authentication & Authorization****Logging in / Registering a New Account:**

Time Complexity:  $O(\log(n) + p_f)$

Where  $p_f$  is the time complexity of Firebase verification, and  $n$  is the number of users in the database.

The user will submit their login information which will be hashed in constant time and sent to Firebase for verification. Firebase will return a reference key which will be used to look up the user in the database in constant time for low numbers of the database, but grows to  $O(\log(n))$  because the database is indexed by id. Thus logging in depends on the complexity of querying the database and Firebase's verification.

**Sending (Friend, Event, Group) Requests / Invitations:**

Time Complexity:  $O(I)$

Where  $I$  is the number of invitations sent, less than or equal to the number of friends a user has.

The user will send a request or invitation which will generate a unique request in the back-end, and insert a new row in the invites table in the database with no other look-up needed. Because this row is inserted at one end of the database, the complexity relies on the number of invitations to be added.

## Calendar Logistics

### Finding Free Time Between One or More Friends:

Time Complexity:

$O(r[])$  : Time-Complexity of finding free time on that range (in days).

$O(\text{Day}) = O(86,400 * f)$

$O(\text{Week}) = 7 * O(\text{Day}) = O(604,800 * f)$

$O(\text{Month}) = 31 * O(\text{Day}) = O(2,678,400 * f)$

$O(\text{Year}) = 365 * O(\text{Day}) = O(31,536,000 * f)$

$O(r[]) = |r| * O(\text{Day}) = O(86400 * |r| * f)$

The two free variables in the time complexities listed above are the range ( $r$ ) and the number of friends that can be invited at once ( $f$ ). Because our system only uses minutes as our base time unit for event scheduling, it suffices to write the number of minutes per day, week, month, year, and in a range and operate using array intersections.

These algorithms can be optimized by skipping large ranges of empty time among the event list. This can be achieved by performing a one-dimensional intersection of “ $f$ ” different sets at points with the largest number of collisions. Reducing a large search space to several smaller ones is a common practice in data complexity optimization and data analysis.

Common examples of spaces with low collisions are in people’s sleep patterns. There is a correlation between a user’s sleep patterns and the earliest and latest times available for that user. Daily optimizations are great for weekly, monthly, and yearly optimizations, because they can reuse the code for day intersections.

Repeating events can also be optimized as they are a guaranteed spot for no-free time for the range of the event. Written above is the worst-case for complexity, but they do not closely bound the average time-complexity. Viewing events as an itemized list is a more efficient, but must be implemented using a generalized collision algorithm for “ $f$ ” objects. The complement of this is then intersected with the range specified (Universe set).

**Adding / Modifying / Removing an Event:**

Time Complexity:  $O(p_c + m \cdot \log(n))$

A user's calendar contains event names and ID's attached to the event, this means that adding or removing an event relies on the time-complexity of the FullCalendar utility (denoted by  $p_c$ ), the time-complexity of querying a user's ID in the database ( $\log(n)$ ), and the number of connections that are in the database 'm'.

## 6. Project Scrum Report

### Product Backlog

Product backlog can be viewed from this link:

<https://app.zenhub.com/workspaces/capstone-product-backlog-5c6ee83bbfd2a674d5c47df5/board?repos=167442195>

### Sprint Backlog

Sprint backlog can be viewed from the same link used above.

### Burndown Chart

Start of Individual Subsystems to Early Testing

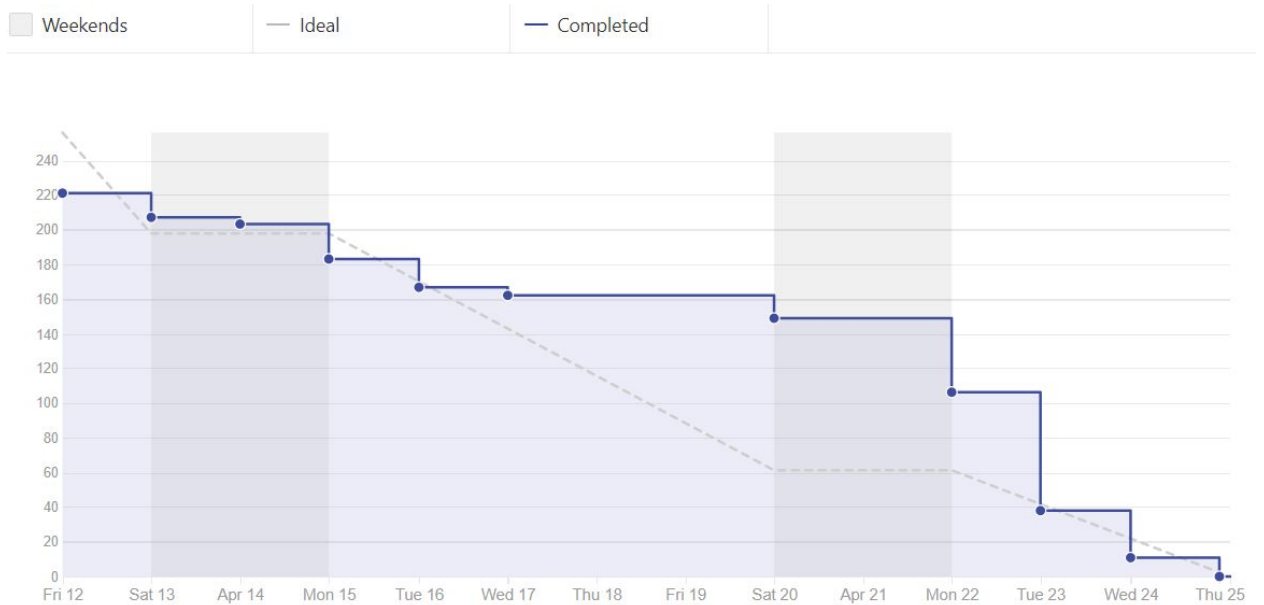




## Implementing and Merging Subsystems for Working Application



## Final testing, implementation, and Completion of MyAlmanack Application

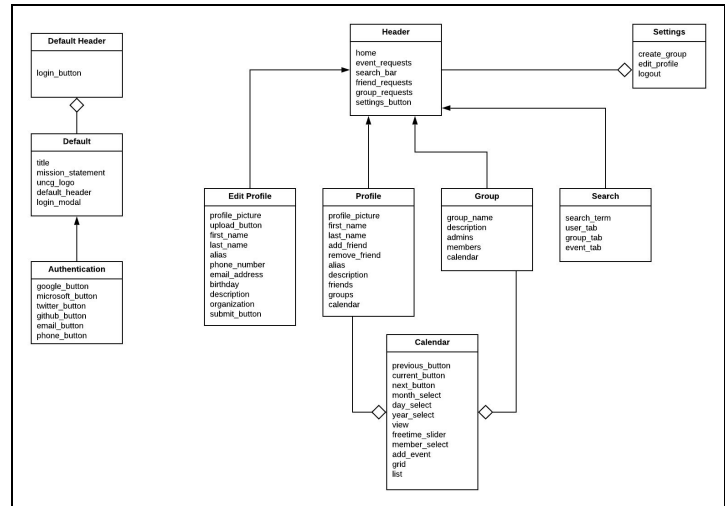


## 7.1 Subsystem: User Interface – Michael & Mark

### Initial Design / Model

The diagram on the right describes the different view inheritance and composition. This is a graphical representation of different pages in the system and their respective page elements.

The design shows the calendar, with different pages such as the month, week, and daily view. This comes with a list of the events that a user has during the selected time period.



## Design

### Default Page


**MyAlmanack**LOG IN

# MyAlmanack

Justin Oakley, Hao Zhang, Pooya Motee, Mark Bures, Michael Resnik

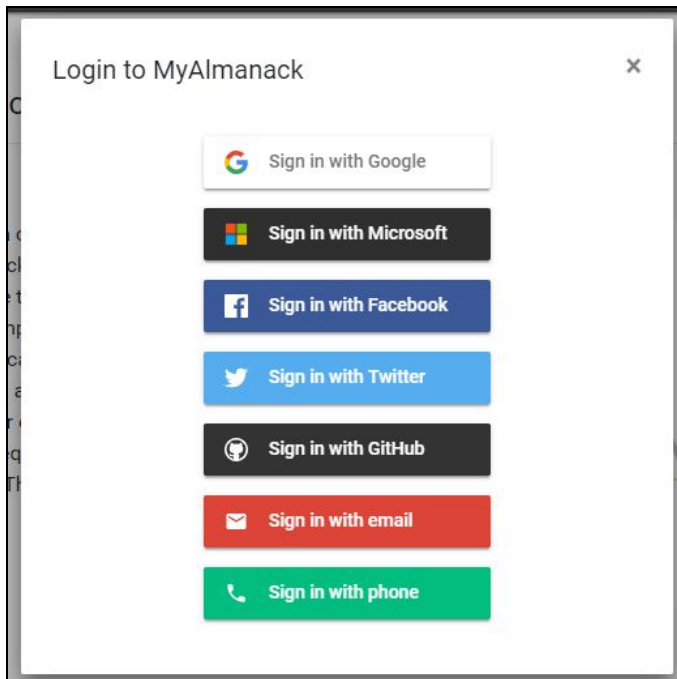
## Mission Statement

Finding the time that people can meet is a constant struggle, especially for those with busy work lives. Our solution to this problem is MyAlmanack, a productivity focused social media website where users can manage their schedules and compare them with others. Users can achieve this by inputting their schedule into our online calendar and comparing their schedule with users that they are friends with or groups that they are a member of. A user can maintain their schedule in their profile, where they are the only person capable of creating, updating, and deleting any information on their calendar. Only people who have been given permission by a user can view that user's schedule, and users can only modify another user's schedule after sending a request. Some events won't be shown if the user trying to view the event isn't allowed on event creation. These Hidden Events are still used in free time calculations but provides no information to the user.



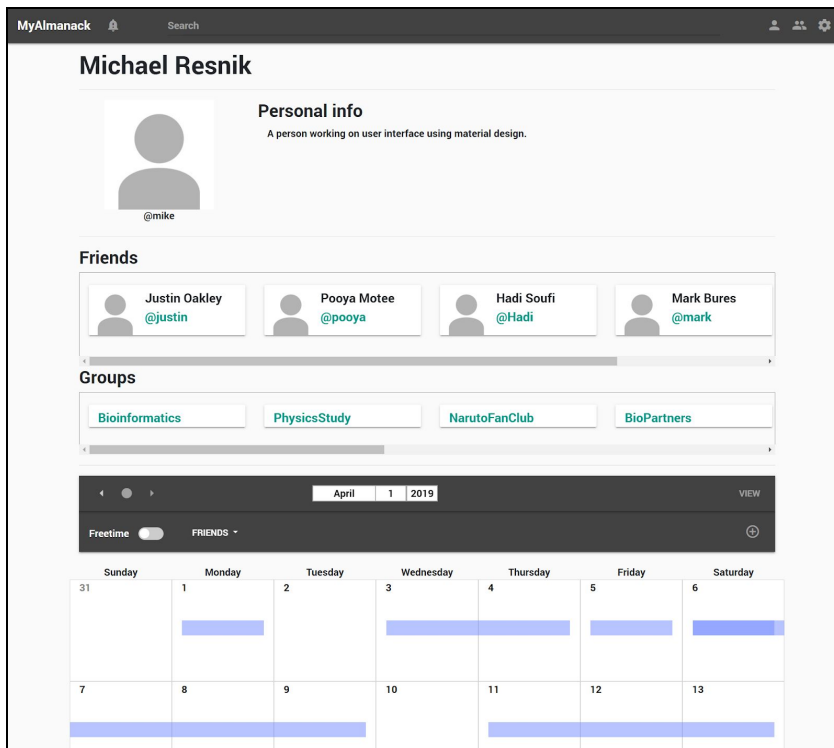
**UNC GREENSBORO**  
Department of  
Computer Science

## Authentication View



## Profile View

The profile view allows users to see necessary data about the user selected, such as their current **Friends**, **Groups**, and **Calendar Data** (if they are allowed access).




## Edit Profile View

The edit profile view allows users to upload custom profile pictures and modify their profile data. Profile modification is important as users are ever-changing.

MyAlmanack

Search

### Edit Profile



Upload a different photo...

Choose File

No file chosen

#### Personal info

\* Denotes a Required Field

\*First name:

Michael

\*Last name:

Resnik

\*Alias:

mike

✓

Email:

\*Birthday:

05/24/1997

Phone Number:

Organization:

Description:

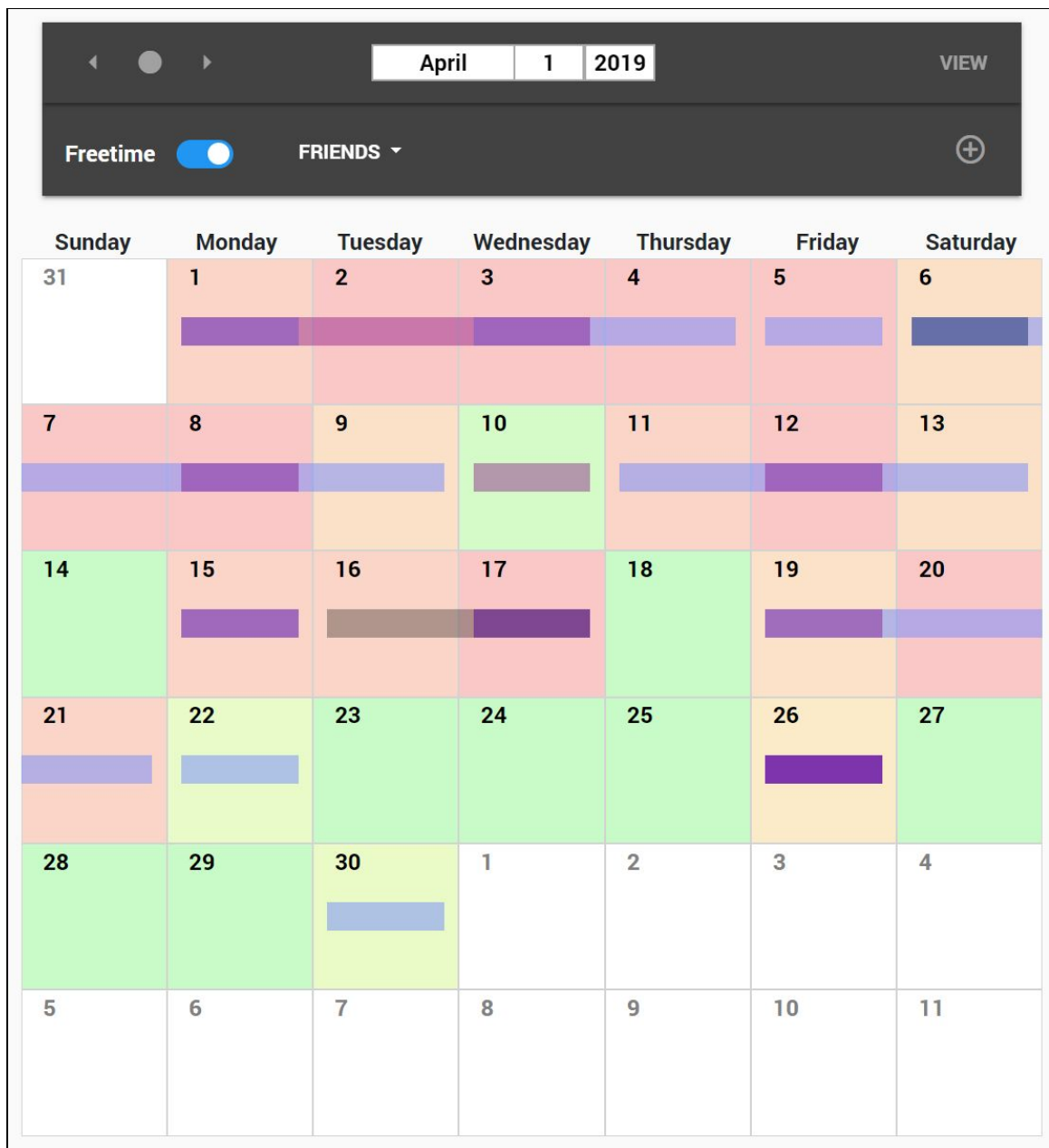
A person working on user interface using material design.

Save Changes

## Calendar View

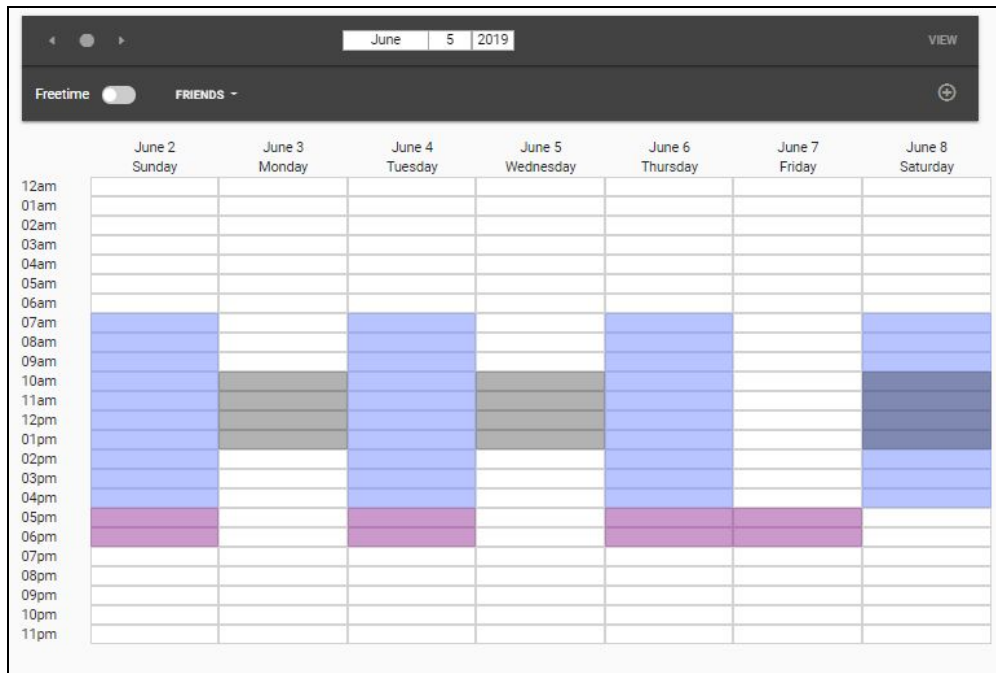
### Month

The month calendar view allows users to selectively add group members to the calendar and view days that have higher free time than others. Events that are marked in *blue* are user-created events while those marked in *purple* are events created by the users selected. The events marked in *grey* are hidden events which have blacklisted the current user from seeing them.

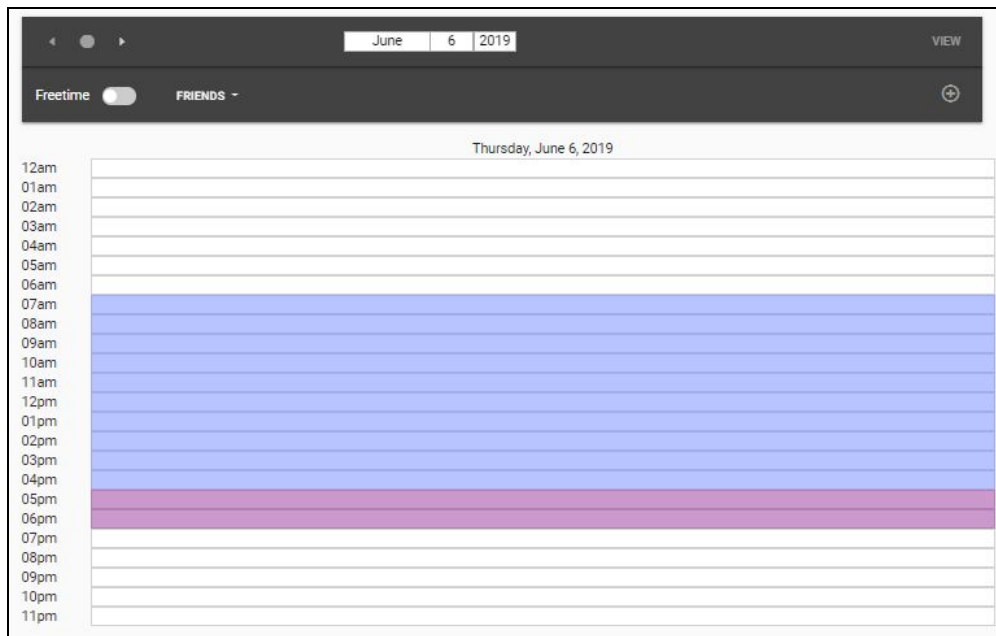


## Week

The week and day calendar views follow the same design conventions as the [month calendar view](#), but provide a different format for event display.

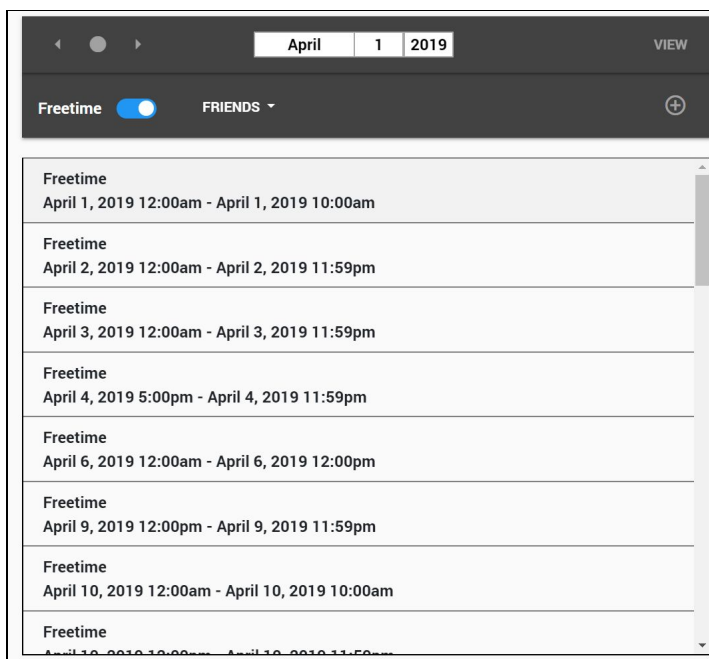
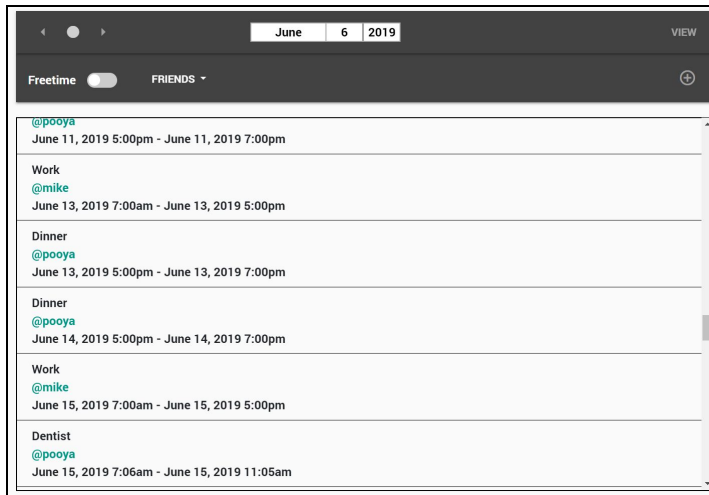


## Day



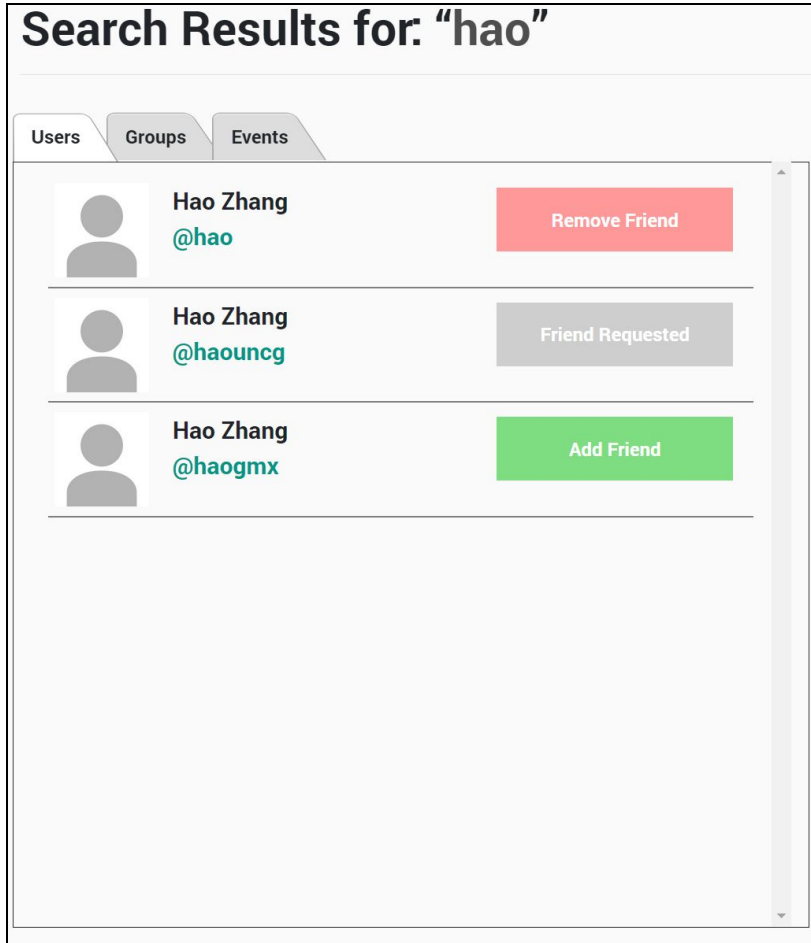
## List

The list view allows the user to view all events created by both the user and the selected members. Freetime can also be visualized in this way. Clicking on an event without freetime selected will either let the user [view](#) or [edit](#) the event. Clicking on a freetime element will allow the user to [create a new event](#) on the time selected.



## Search View

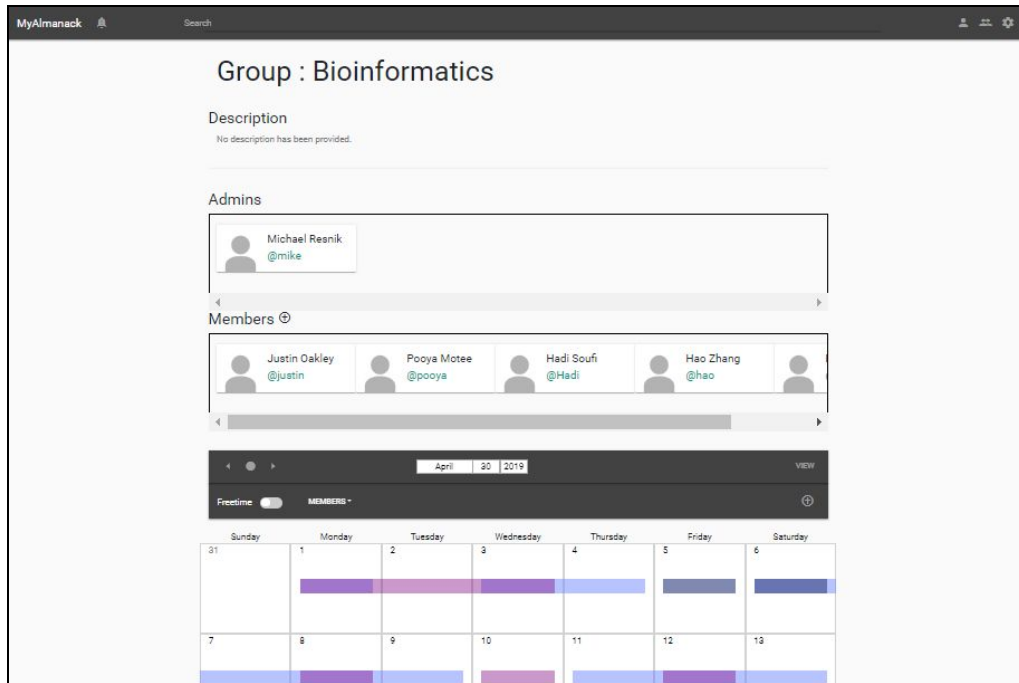
The search view allows users to search for a specific user, group, or event, and are filtered by both how relevant the search term is to the elements displayed and how relevant the elements are may be for the user who is searching.





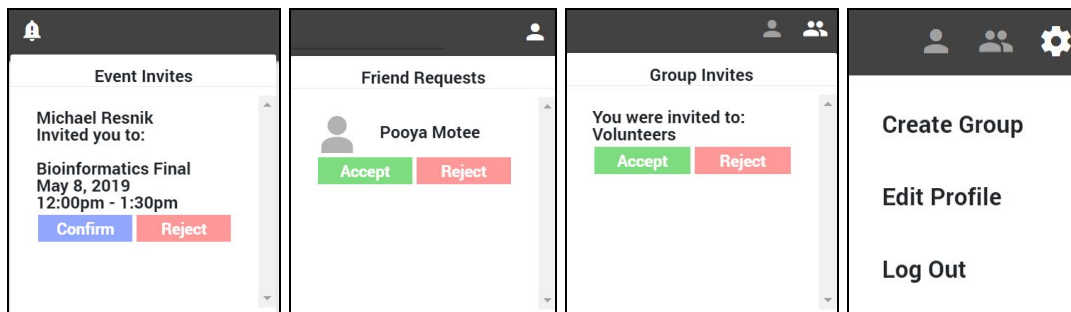
## Group View

The group view allows users to view their calendar with those of other members in the group, and handle specific selection of user events. A group admin has the ability to invite other members to join the group, while users have the ability to join the group and leave the group at any time.



## Dropdowns

Dropdowns allow the user to interact with the navigation bar in an intuitive manner. They allow the user to respond to invites and open various settings pages.



## Create Dialogs

Creation dialogs are important for inputting new data into the database. They allow the user to create new instances of either events or groups, as well as filtering read access.

The image shows two side-by-side dialog boxes for creating new items. The left dialog is titled "Create a New Event" and the right is titled "Create Group".

**Create a New Event Dialog:**

- \*Event Name:** Picnic
- Description:**
- \*Start Date:** 05/04/2019
- \*End Date:** 05/11/2019
- \*Start Time:** 10:00 AM
- \*End Time:** 12:00 PM
- Invite:** @JUSTIN, @POOYA
- Blacklist:** @HADI
- Repeat:** ☒
- \*Repeat Pattern:** Su Mo Tu We Th Fr Sa (Sa is checked)
- Create Event** (button)

**Create Group Dialog:**

- \*Group Name:** DanceTeam
- Description:**
- Invites:** @JUSTIN, @HADI, @MARK
- Create Group** (button)

## Edit Event Dialogs

Editing dialogs are important for modifying pre existing data in the database. This allows the event design to be flexible / ever-changing.

The image shows two side-by-side dialog boxes for editing existing items. The left dialog is titled "Edit Repeat Event Selected" and the right is titled "Edit Event Selected".

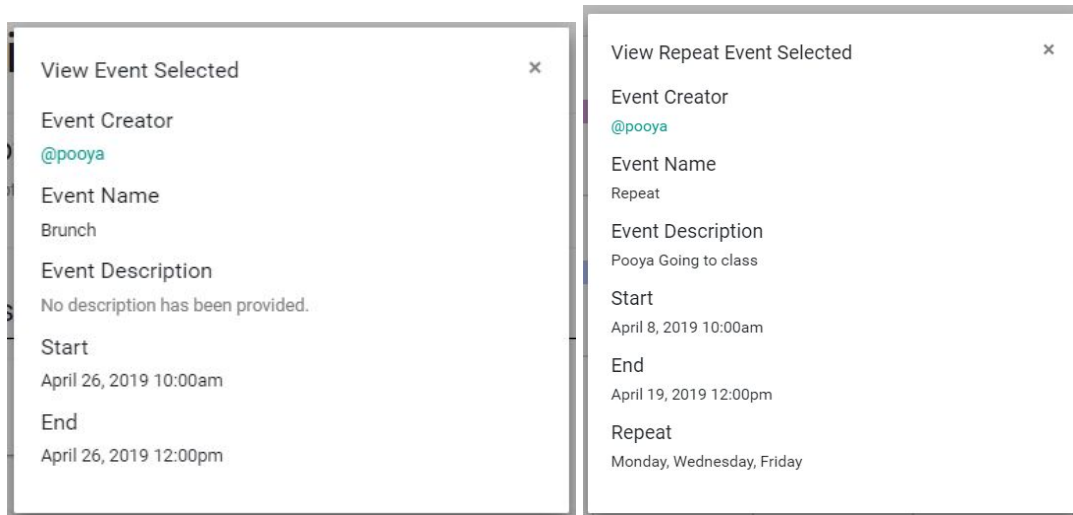
**Edit Repeat Event Selected Dialog:**

- \*Event Name:** Class Meeting
- Description:**
- \*Start Date:** 06/01/2019
- \*End Date:** 06/30/2019
- \*Start Time:** 10:00 AM
- \*End Time:** 03:00 PM
- Repeat Pattern:** Su Mo Tu We Th Fr Sa (Mo, We, Fr are checked)
- Submit Changes** (button)

**Edit Event Selected Dialog:**

- \*Event Name:** Trip to DC
- Description:**
- \*Start Date:** 04/11/2019
- \*End Date:** 04/13/2019
- \*Start Time:** 10:00 AM
- \*End Time:** 12:00 PM
- Submit Changes** (button)

## View Event Dialogs



## Refinement

The benefits to having a user interface are: it gives a visual representation of other subsystems, a way for the user to interact with the program, and it is an engaging life-improvement if done well. The disadvantages of a user interface are: the implementations require a lot more time to configure and it's harder to update graphical information without knowledge of every subsystem. Django minimizes the amount we need to change and is very scalable, but user interface still needs to know a lot about other subsystems for connectivity purposes. Another disadvantage of the user-interface is the battle between design and functionality. All issues and commits must keep both of these design choices in mind.

A depth first view of the user interface would be to refine one element until it looks *good enough*, yet a breadth first view would be establishing pure functionality while ignoring design. There were major design changes from the original to the current design, but the functionality remains the same from our initial thoughts. A usable interface is one that provides high functionality and high usability, so both need to be thought of when making changes.

In the future, user interface will provide higher functionality in order to connect with other subsystems, while maintaining a consistent level of usability.

## Scrum Backlog

The scrum backlog can be viewed using this link:

<https://app.zenhub.com/workspaces/capstone-product-backlog-5c6ee83bbfd2a674d5c47df5/board?repos=167442195>

## Coding

Functional programming was used for creating the User Interface to make displaying more dynamic than an object oriented approach. The languages used to create the User Interface were JavaScript, HTML, CSS, Python, jQuery, and Django.

## Data Dictionary

User interface uses [all data dictionaries](#) referenced by the database by either using custom queries or direct calls. All data stored in the user interface are arrays of structures. The parsing of data in queries is done by interpreting JSON strings or by Django query sets. Creation and modification to elements in the user interface are sent to the back-end in both HTML forms and AJAX requests. Wrapper functions were made in the back-end views file to interpret the data provided by custom queries.

## Testing

For testing, we consulted peers on both design and functionality choices before and during implementation. We also constantly checked other websites and applications implementations of the calendar system and the overall look of the website. We used standards for the navigation bar and profile views of other websites with profile functionality.

Every time we added something new in the user interface, we ran durability tests on it. This ensured that all parts of the user interface ran with minimal errors. We also tried actively using the website for a week to see our impressions of it. This eliminated the learnability aspect and because of that, we needed to consult peers on the usability and functionality of the website.

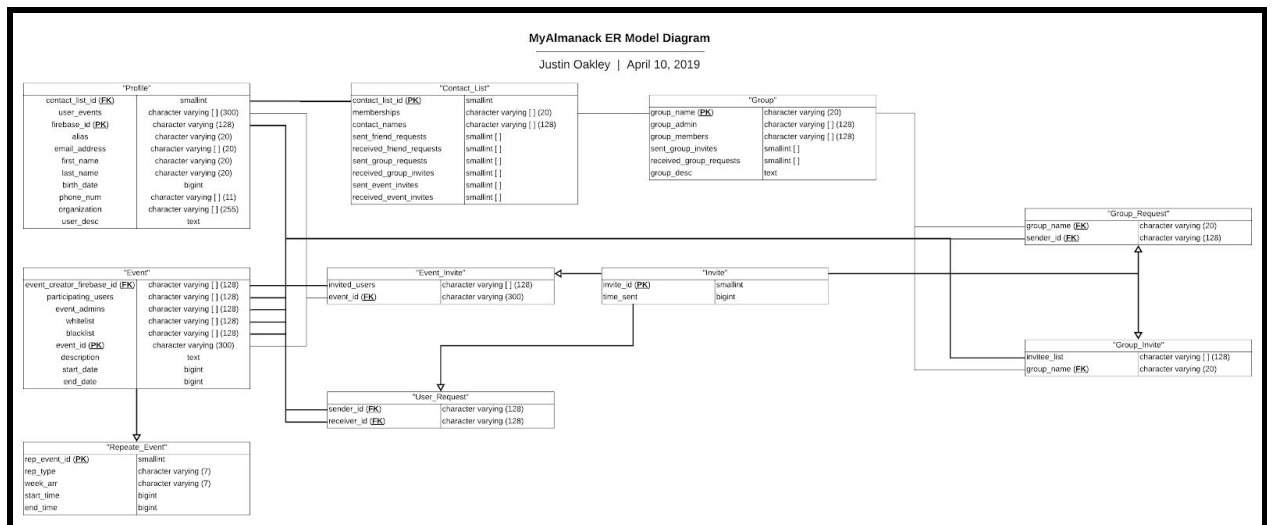
In the future we will use formal user interface testing documents to gauge the time taken to complete certain tasks and surveys for different elements on the website. Another possible method of training is to track the user's mouse on the web page and see what areas are visited more than others. We found that three months of development on this project was enough time to develop functions and stylistic choices, but didn't provide enough time for formal interface testing overall.

## 7.2 Subsystem: Database – Justin

### Initial Design

This subsystem was designed to handle back-end programming involving database creation and management. Tasks and issues that were a part of the subsystem involved finding a web server to host the application and database, creating a data dictionary to provide data guidelines for project members, initializing data tables, and maintaining the database while providing the necessary changes needed to maintain application functionality.

The database model and design can be illustrated in the Entity-Relationship diagram below:



### Data Dictionary

This subsystem heavily relied on an initial data dictionary conception to create a starting point for the database foundation. From there, any changes made to the database meant that the data dictionary must be updated to provide accurate information about the database for other project members and maintain comprehensibility and reliability of the subsystem.

The updated data dictionary can be seen below:

### Profile Database Table

This table contains important profile attributes and different identifiers that are connected to the many different features in MyAlmanack.

| Column Name     | Data Type                   | Null Value | Description                                      |
|-----------------|-----------------------------|------------|--|
| firebase_id     | character varying (128)     | Not Null   | <u>PK</u> : The user's firebase identifier.      |
| alias           | character varying (20)      | Not Null   | The user alias.                                  |
| contact_list_id | smallint                    | Not Null   | <u>FK</u> to Contact_List.contact_list_id        |
| email           | character varying[ ] (20)   | Null       | The user's email address                         |
| first_name      | character varying (20)      | Not Null   | The user's first name                            |
| last_name       | character varying (20)      | Not Null   | The user's last name                             |
| birth_date      | bigint                      | Not Null   | The user's date of birth                         |
| phone_num       | character varying [ ] (11)  | Null       | The user's phone number                          |
| organization    | character varying (255)     | Null       | The user's organization                          |
| user_events     | character varying [ ] (300) | Null       | List of all events that user is participating in |
| user_desc       | text                        | Null       | The description of the user                      |

### Contact\_List Database Table

This table contains a list of other users and groups that are included a user's social network.

| Column Name              | Data Type                   | Null Value | Description  |
|--------------------------|-----------------------------|------------|--|
| contact_list_id          | smallint                    | Not Null   | <u>PK</u> : The numerical, system-generated identifier that contacts a user with their social network contact data             |
| contact_names            | character varying [ ] (128) | Null       | The list of all users' contacts' firebase_ids  |
| memberships              | character varying [ ] (20)  | Null       | The list of all groups that the user is a member of  |
| sent_friend_requests     | smallint [ ]                | Null       | The list of all invite_ids where a certain user has sent a contact request to another user and is still pending approval       |
| received_friend_requests | smallint [ ]                | Null       | The list of all invite_ids where a certain user has received a contact request from another user and is still pending approval |
| sent_group_requests      | smallint [ ]                | Null       | The list of all invite_ids where a certain user has sent a contact request to a group and is still pending approval            |
| received_group_invites   | smallint [ ]                | Null       | The list of all invite_ids where a certain user has received an invite to a group and is still pending approval                |
| sent_event_invites       | smallint [ ]                | Null       | The list of all event invites that a certain user has sent and is still pending approval                                       |
| received_event_invites   | smallint [ ]                | Null       | The list of all event invites that a certain user has received and is still pending approval                                   |

## Event Database Table

This table contains all the event data for the current, shared calendar and schedule data.

| Column Name               | Data Type                   | Null Value | Description   |
|---------------------------|-----------------------------|------------|---|
| event_id                  | character varying (300)     | Not Null   | <u>PK</u> : A unique event name identifier  |
| description               | text                        | Null       | A description of the event  |
| participating_users       | character varying [ ] (128) | Null       | List of firebase_ids of all users participating in an event   |
| event_admins              | character varying [ ] (128) | Not Null   | List of firebase_ids of all users that have administrative control over users (the user responsible for event creation should by default have admin capabilities) |
| whitelist                 | character varying [ ] (128) | Null       | List of firebase_ids of all users that have permission to see an event  |
| blacklist                 | character varying [ ] (128) | Null       | List of firebase_ids of all users that do not have permission to see an event   |
| start_date                | bigint                      | Not Null   | The start date for an event   |
| end_date                  | bigint                      | Not Null   | The end date for an event   |
| event_creator_firebase_id | character varying (128)     | Not Null   | <u>FK</u> to Profile.firebase_id  |



### Repeat\_Event Database Table

This table contains all the repeated event data. Note: This table inherits 'Event' table fields as well.

| Column Name  | Data Type             | Null Value | Description   |
|--------------|-----------------------|------------|---|
| rep_event_id | smallint              | Not Null   | <u>PK</u> : A unique repeat event identifier  |
| rep_type     | character varying (7) | Not Null   | This determines what type of repetition an event will have. Types can be 'daily', 'weekly', or 'monthly'  |
| week_arr     | character varying (7) | Not Null   | This number declares which days of the week that an event is to occur in. Note: this is a binary number, so only zeros and ones are allowed in it |
| start_time   | bigint                | Not Null   | The time of day that an event starts  |
| end_time     | bigint                | Not Null   | The time of day that an event ends  |

### Group Database Table

This table contains data for all groups in the system.

| Column Name             | Data Type                   | Null Value | Description   |
|-------------------------|-----------------------------|------------|---|
| group_name              | character varying (20)      | Not Null   | <u>PK</u> : A unique group name identifier                          |
| group_admin             | character varying [ ] (128) | Not Null   | All group administrators (there can be more than one administrator) |
| group_members           | character varying [ ] (128) | Not Null   | All group members, which includes group administrators              |
| sent_group_invites      | smallint [ ]                | Null       | The list of all invite_ids that have outgoing invitations           |
| received_group_requests | smallint [ ]                | Null       | The list of all invite_ids that have incoming joining requests      |
| group_desc              | text                        | Null       | The description of the group  |

### Invite Database Table

This table is the base model for event invites, group invites, and friend requests and contains the invite\_id and the moment that the invitation was sent.

| Column Name | Data Type | Null Value | Description   |
|-------------|-----------|------------|---|
| invite_id   | smallint  | Not Null   | <u>PK</u> : A unique, sequence-generated number created by the system that identifies invites |
| time_sent   | bigint    | Not Null   | The time that the invitation is sent  |

### Group\_Invite Database Table

This table contains all invite data about group-to-user invites. Note: This table inherits 'Invite' table fields as well.

| Column Name  | Data Type                   | Null Value | Description                        |
|--------------|-----------------------------|------------|------------------------------------|
| group_name   | character varying (20)      | Not Null   | <u>FK</u> to Event.group_name      |
| invitee_list | character varying [ ] (128) | Null       | All invitees' (to a group) aliases |

### Group\_Request Database Table

This table contains all invite data about user-to-group requests. Note: This table inherits 'Invite' table fields as well.

| Column Name | Data Type               | Null Value | Description                      |
|-------------|-------------------------|------------|----------------------------------|
| sender_id   | character varying (128) | Not Null   | <u>FK</u> to Profile.firebase_id |
| group_name  | character varying (20)  | Not Null   | <u>FK</u> to Group.group_name    |

### User\_Request Database Table

This table contains all information about user-to-user requests. Note: This table inherits 'Invite' table fields as well.

| Column Name | Data Type               | Null Value | Description   |
|-------------|-------------------------|------------|---|
| sender_id   | character varying (128) | Not Null   | <u>FK</u> to Profile.firebase_id. This is the alias of the user that sent a request     |
| receiver_id | character varying (128) | Not Null   | <u>FK</u> to Profile.firebase_id. This is the alias of the user that received a request |

### Event\_Invite Database Table

This table contains all invite data about a specified event. Note: This table inherits 'Invite' table fields as well.

| Column Name   | Data Type                   | Null Value | Description                                      |
|---------------|-----------------------------|------------|--|
| event_id      | character varying (300)     | Not Null   | <u>FK</u> to Event.event_id                      |
| invited_users | character varying [ ] (128) | Null       | List of aliases of all users invited to an event |

## Refinement

The initial data models went through several changes during the subsystem implementation and merging of subsystem components to maintain functionality and remove unnecessary fields and tables. Such model changes include the insertion of "Group\_Request" and "Repeat\_Event", the deletion of "Admin" and "Calendar", and editing table fields and keys.

The most impactful change in the database subsystem that had several ramifications on "MyAlmanack" was the switch from MySQL to PostgreSQL. PostgreSQL was the more suitable database system to use with Heroku and allowed for the database to be more manageable and simpler to query due to its unique data types and functions

## Scrum Backlog

The scrum backlog can be viewed using this link:

<https://app.zenhub.com/workspaces/capstone-product-backlog-5c6ee83bbfd2a674d5c47df5/board?repos=167442195>

## Coding

This subsystem involves mostly back-end object-oriented programming due to the nature of database management. The handling of data in the application involves insertions, updates, searches, and deletions, thereby making it mandatory for all database-related coding to be object-oriented.

Query development involved the use of both the Python 3 language and PostgreSQL. Simple queries such as fetching were performed using Django models since they allowed for readability and writability. More complex queries were performed by executing raw SQL statements through Django to make working with Postgres uncomplicated for the system.

## Testing

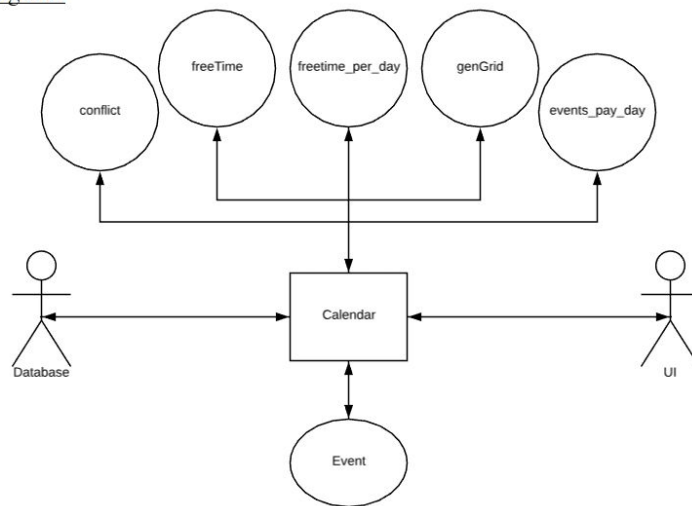
For the testing of the functionality and reliability of data table fields and the overall database schema, data queries were formulated and executed using the pgAdmin 4 Postgres GUI. This proved to be very effective in database management for the application because it showed what data fields were necessary for functionality and reduced the complexity of the tables themselves.

When testing query methods using the Django framework, the database was in constant need of wiping and re-insertion in order to make sure multiple, differing queries that were being executed in a single method were working properly together. To make sure that each method would not crash the final overall system, method-calls were placed specifically in UI methods during execution time to determine if said methods caused errors and to visually confirm, within the command prompt, that the methods were returning the data in the appropriate format.

## 7.3 Subsystem: Calendar Data – Pooya

### Design and model

Calendar Data Flow Diagram:



This subsystem consists of a combination of complex string manipulation algorithms and Unix time addition and subtraction to calculate and sorted lists of events or free time by day in a given range of dates. It also generates the numbers of the days on the calendar for the UI subsystem to be able to display the right day inside the right box on the calendar grid.

#### **Redefined**

Initially this subsystem was supposed to be coded in Django, but it was switched to JavaScript since it's more compatible for what we need it to do and is closer to HTML.

A pro is that it's easier to implement in the overall system.

A con is that I had a minor setback since the subsystem was initially coded in a different language.

#### **Scrum Backlog Link**

#### **Code**

This subsystem uses Object Oriented Programing and uses models and structures to retrieve and manipulate user data to be stored back into the database or be displayed by the user interface.

#### **User Training**

For user training and testing, a datasheet of dummy data was created in Excel to mimic user event information and test the functions with the dummy data for error checking.

## 7.4 Subsystem: Authentication – Hao

### Initial Design

The subsystem initially did not intend to handle any front-end interaction (namely the login interface). The user interface subsystem was supposed to design the interfaces responsible for handling user login and registration. The subsystem was to reside in the back-end, take in user information submitted through login interfaces provided by the user interface subsystem, and use the Firebase Admin SDK to authenticate them with Django's model authentication back-end.

There was no clear outline of the model at the time the initial model was drafted. See the Refinement subsection below for an updated, comprehensive sequence diagram, which clearly defines the control flow of the subsystem.

### Data Dictionary

The subsystem does not have a data dictionary applicable. The only data that would need to persist on the back-end would be the user's Firebase ID, which is handled by the database subsystem.

### Scrum Backlog

The scrum backlog can be viewed using this link:

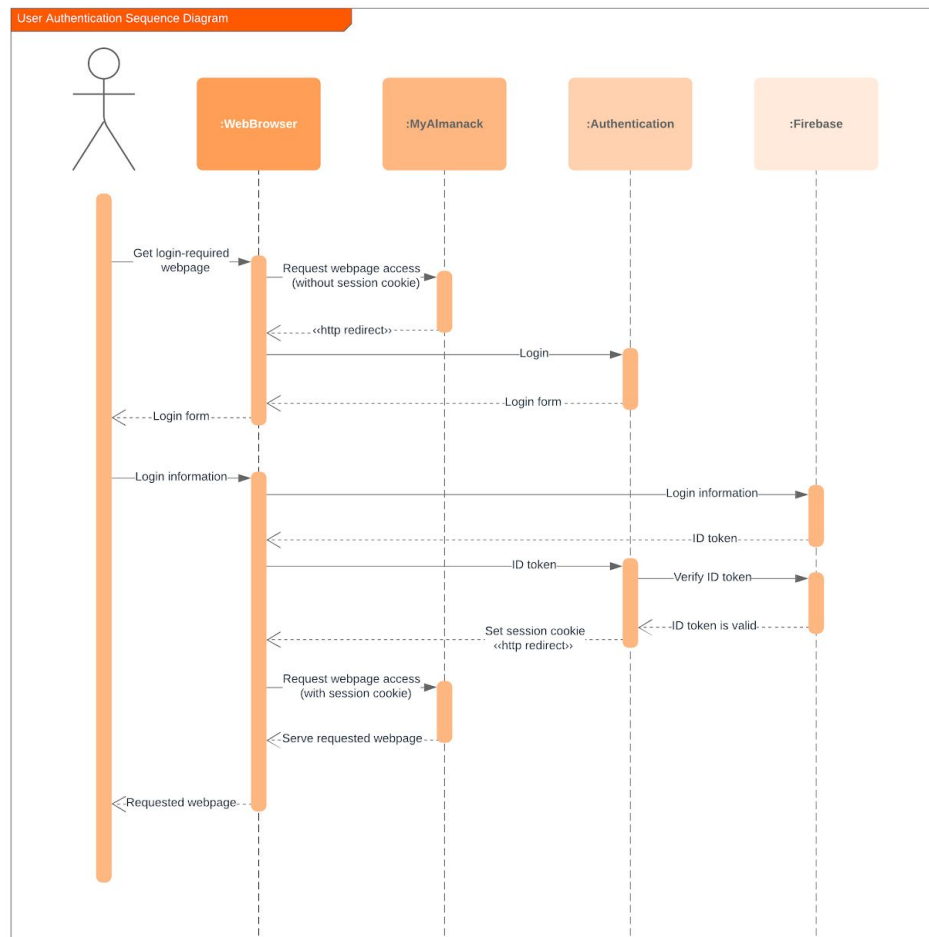
<https://app.zenhub.com/workspaces/capstone-product-backlog-5c6ee83bbfd2a674d5c47df5/board?repos=167442195>

## Refinement

Due to the nature of Firebase's authentication system, support for multiple federated authentication services (i.e. Google, Facebook, GitHub, Twitter, etc.) would require creating individual user interfaces to handle each login service. Creating these interfaces from scratch would not only require a significant amount of time, it would also expose the application to security risks due to poorly implemented security measures (or a lack thereof). Luckily, Google also provides a pre-built open-source web interface that can easily be added to any web application: FirebaseUI Web. Not only does it provide a consistent user interface across the entire range of login services, it also ensures best, industry-standard security practices.

### MyAlmanack Authentication

Hao Zhang | April 11, 2019



## Coding

The subsystem has both a front-end and a back-end component. The front-end component displays a user interface which the user will interact with to log into the application using one of the many available federated authentication services available. The back-end component will help propagate the login process by verifying the temporary ID token generated by FirebaseUI Web, and create a session cookie to authenticate the user.

The front-end component is mostly written in HTML, JavaScript, and CSS; and is mostly handled by the code provided by FirebaseUI Web. A small section is written in Python to allow Django to serve the login view when needed.

The back-end component is written in Python. There are no classes in the subsystem with the exception of a static middleware class used to make sure that user sessions are valid, and redirect them to the login page if or when they become invalid. An 80-character page width limit is used to keep the code readable on narrow displays (and sites such as GitHub). Comments have been provided throughout the code to guide the reader through the subsystem's actions.

## User Training

There is little to no user training required for the subsystem. When a user uses the front-end user interface to log into the application, they would simply choose one of the federated authentication services with whom they have an account with (i.e. Google or Facebook). They would then be shown a new user interface provided by the chosen authentication service, which they should be familiar with if they have an account with them. Once they return to the application upon successful login, there is no further interaction required with the subsystem.

## Testing

The subsystem has been tested to be within specification. At one point, a bug caused invalid sessions to lead to infinite redirects to the login page, causing the web browser to eventually give up loading. A fix was applied shortly after, and the bug has not been noticed again in environments using the updated code.



## 7.5 Subsystem: Authorization – Hao

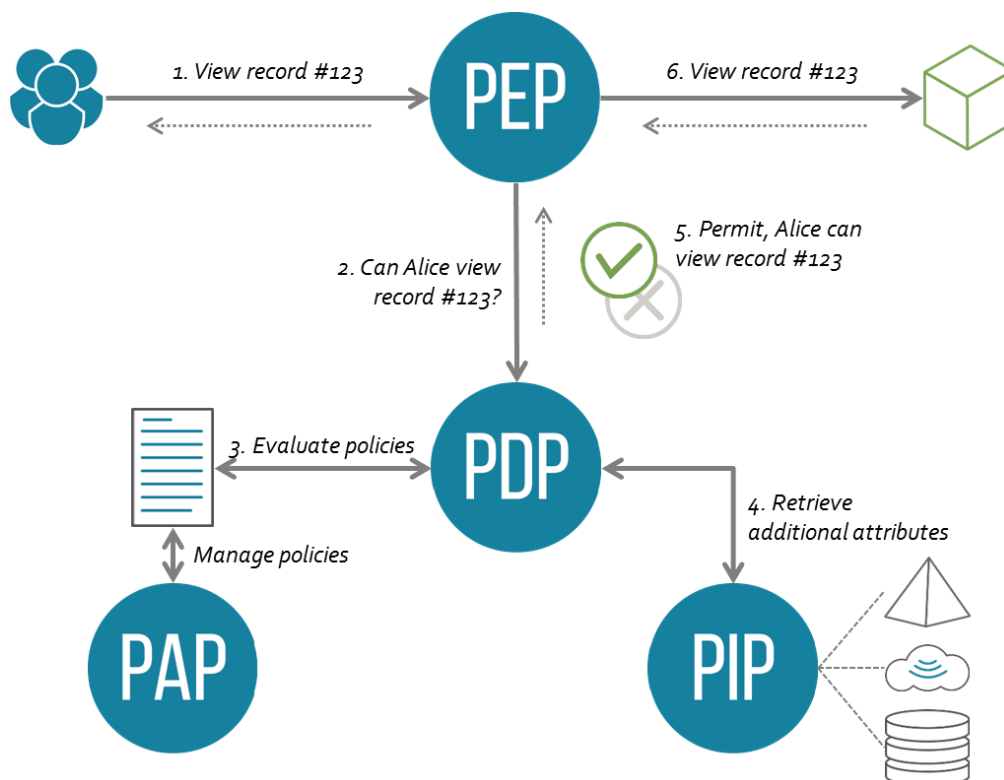
### Initial Design

The subsystem was originally designed to conform to a basic and generic attribute-/policy-based access control model. According to [Wikipedia](#), an ABAC model should contain the following:

- Policy enforcement point - provides an interface for other subsystems to enforce permission authorizations
- Policy decision point - evaluates appropriate policies for given authorization requests
- Policy information point - used during policy evaluation to retrieve data relevant to the evaluation of authorization requests

The attributes that make up an authorization request can be broken down into the following:

- Subject - the user or object asking for permission
- Action - the action that the subject is attempting to act upon the resource
- Resource - the user or object that the authorization request is acting on
- Context - additional details that may affect authorization request evaluation



The system's design was also initially heavily-influenced by this Stack Exchange thread: <https://softwareengineering.stackexchange.com/questions/351620/what-is-a-suggested-roadmap-towards-implementation-of-a-simple-attribute-based-a>

## Data Dictionary

The subsystem does not have a data dictionary applicable. The subsystem only performs permission checking, not modification. Therefore, it only relies on what the database subsystem provides, and does not save persistent data.

## Refinement

The subsystem has mostly conformed to the original design with only minor changes. Other access control models were considered (i.e. role-based access control), but they did not fully serve the purpose of the application as they did not allow for the fine-grained control possible with an attribute-based access control model.

Throughout the lifetime of the subsystem's development, layers of abstraction were created to provide better isolation of components. For example, a database model wrapper layer was created to isolate unexpected database model changes to a single location.

## Scrum Backlog

The scrum backlog can be viewed using this link: <https://app.zenhub.com/workspaces/capstone-product-backlog-5c6ee83bbfd2a674d5c47df5/board?repos=167442195>

## Coding

The subsystem is written in Python using heavy object-oriented programming principles. Child classes may extend from as deep as three parent classes, inheriting multiple repeating attributes and methods. Although there is heavy abstraction of classes, it is not done without a purpose. In many cases, it is done to ensure that the API provided to the other subsystems cannot accidentally take in incorrect object classes. This helps identify errors and inconsistencies quickly in the subsystem if and when they occur.

## User Training

The subsystem is invisible to the end-users. Only members of the MyAlmanack team will be able to directly use and interface with the subsystem's provided APIs. For the other team members, a file (located at */docs/authorization/actions.txt*) documenting the available authorization actions and the type of attributes each of them will accept. Example code has been provided in the form of Django view files (located in */authorization/views/*) that can be used as a reference as to how the API should be interfaced with.

## Testing

The testing of the subsystem involved creating testing Django view files (located in */authorization/views/*) to test the authorization actions against their expected authorization results with respect to the data in the database. After the authorization policies have been adjusted to fit the expected application behavior, the user interface team worked to integrate with the authorization subsystem by using the provided API and sample code within their own subsystem, and performing their own tests afterward. When unexpected or inconsistent behavior was found, the user interface team would reach out with the expected behavior and the actual result to aid in debugging the subsystem. Over time, the subsystem maintained good stability while refactors were performed; and more features were added to accommodate for changes in the overall application.

## 8. Complete System

### Final Product

The MyAlmanack web application can be found using this link:

<https://my-almanack.herokuapp.com/>

*Note: This website may be taken down shortly after the end of the course when the Heroku subscription is ended.*

### Source Code

The source code for MyAlmanack can be found on GitHub using this link:

<https://github.com/capstone-dt/MyAlmanack/>

### User Manual

The user manual for MyAlmanack can be viewed using this link:

<https://docs.google.com/document/d/13ZuJnqWwPUyGauaBBw8FEleipY5ZMiBJNie-2SJwktA/edit?usp=sharing>

## **Team Members**

### **Justin Oakley**

Justin came up with the idea for the application after running into difficulty while attempting to schedule the first team meeting during the ideation phase of the project. Since he had real-world experience with database systems in the IT industry, Justin took on the role to handle creating the database to the project's needs and providing the user interface team with the queries they need. Justin also took on the role of finding and maintaining the web and database servers on Heroku to host the project's back-end.

### **Michael Resnik**

Michael worked on the user interface subsystem which helps visualize all other subsystems. He developed the calendar interface from scratch so that it would support all functions that MyAlmanack set out to achieve for calendar interaction and display. He made popup dialogs for each view and worked on connecting all subsystems together in different languages, as well as data management for communication between subsystems. He made the profile picture loading / saving mechanism for different users and all dropdowns used in MyAlmanack.

### **Mark Bures**

Mark also worked on the user interface subsystem. He was responsible for the initial design and ongoing development of the subsystem. He styled each web page and made sure that consistency was maintained across MyAlmanack with different color management techniques.

### **Pooya Motee**

Pooya worked on the calendar data subsystem to provide the user interface team the necessary functions behind calendar interactions. Such functions included methods for: freetime calculations, finding the numbers to populate the month calendar view, repeat event splitting, and event conflicts.

### **Hao Zhang**

Hao worked on the authentication and authorization subsystems to ensure that the users were able to smoothly log into the application; that session management is handled securely; and that users can only perform actions that they are allowed to. He made sure that the application had some coherent security practices in place by setting up some of the available security measures in Django and helping the team avoid pushing secret keys to GitHub. Hao also helped with maintaining the back-end server and ensuring that it was ready for presentation during different phases of the project.