Team Relays’

Drake Relays Companion App

Software Design Document

Names: Noah Dougen, Jordan Klein, Nicholas Shipley, William Bartolomucci

Date: (04/03/2020)

**TABLE OF CONTENTS**

**1. INTRODUCTION ……………………………………………………………………………………..2**

1.1 Purpose ……………………………………………………………………………………………..2

1.2 Scope ………………………………………………………………………………………………..2

1.3 Overview …………………………………………………………………………………………….3

1.4 Reference Material …………………………………………………………………………………3

1.5 Definitions and Acronyms ………………………………………………………………………....4

**2. SYSTEM OVERVIEW ………………………………………………………………………………..5**

**3. SYSTEM ARCHITECTURE …………………………………………………………………………5**

3.1 Architectural Design ……………………………………………………………………………….5

**4. DATA DESIGN ………………………………………………………………………………………12**

4.1 Data Description ………………………………………………………………………………….12

4.2 Data Dictionary …………………………………………………………………………………...14

**5. HUMAN INTERFACE DESIGN …………………………………………………………………...15**

5.1 Overview of User Interface ……………………………………………………………………...15

5.2 Screen Images …………………………………………………………………………………....17

5.3 Screen Objects and Actions ……………………………………………………………………..22

**6. REQUIREMENTS MATRIX ………………………………………………………………………24**

# 1. INTRODUCTION

# 1.1 Purpose

The purpose of this design software document is to provide future developers with knowledge of the intent of the program and how it works. Furthermore, the hope is that this design document will allow developers to recreate or further extend the scope of this software system.

Throughout this document will be details that regard the scope of the code as well as the inner workings of the code. With the code presented in this document, it may be easier to understand how the software system works as a whole, working unit--and it may, moreover, be easier to understand what its intended use is for users.

## 1.2 Scope

The software system that is being implemented in this project is a Drake Relays companion application. The intent of the application is to provide spectators of the Drake Relays a more user friendly experience as opposed to what is currently offered. Spectators of the Drake Relays, as it now stands, would need to carry around physical copies of the scheduled events, or would need to sift through multiple web-portals to find what they are looking for. The Drake Relays Companion application attempts to streamline this process for the spectator; allowing spectators a one-stop application to find all information that they need when attending the Drake Relays.

## 1.3 Overview

This software design document will outline how the developers of this application attend to achieve their goal. The document in its entirety aims to achieve clarity in how the application works--namely, how the code works together to provide the intended, end goal. This document will contain “chunks” of code to provide future developers with a “frame of mind” to show how the application is programmed with the user-experience in mind.

## 1.4 Reference Material

Throughout the development life-cycle of this software system, multiple sources were used in order to obtain a coherent application--coherent meaning, namely, that it works, and is user friendly.

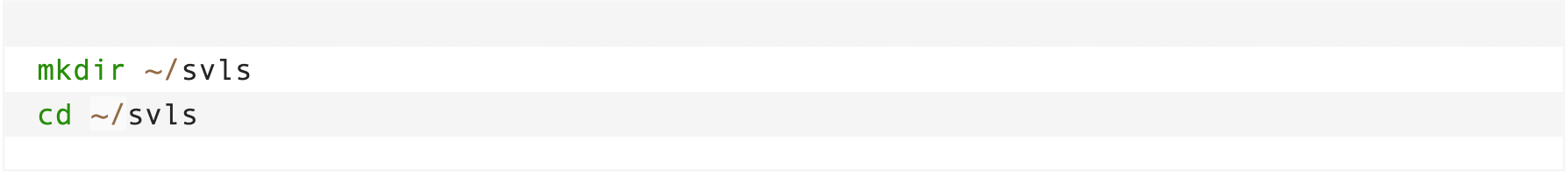
Because the software system, as a whole, requires a database in order to access, modify, and remove elements from the application, the AWS website was utilized to achieve this. The documentation can be found in the following link. <https://aws.amazon.com/blogs/database/query-your-aws-database-from-your-serverless-application/>

This article from AWS has allowed the creators of this software system to better understand how to access things externally through Amazon Web Services: specifically, how to connect an AWS database to the application which is being developed. The AWS article in question specifically outlines how to query an AWS database from a serverless application. What the creators of this software system have utilized is the portion of the article which outlines how to create the project directory for the database.

Though this software system is utilizing AWS’s services, there still may be a need for a local, small-scale server--perhaps something akin to a Raspberry Pi. The above mentioned article outlines how to create the database from a local, small-scale server. A few pictures will help outline how this is achieved.

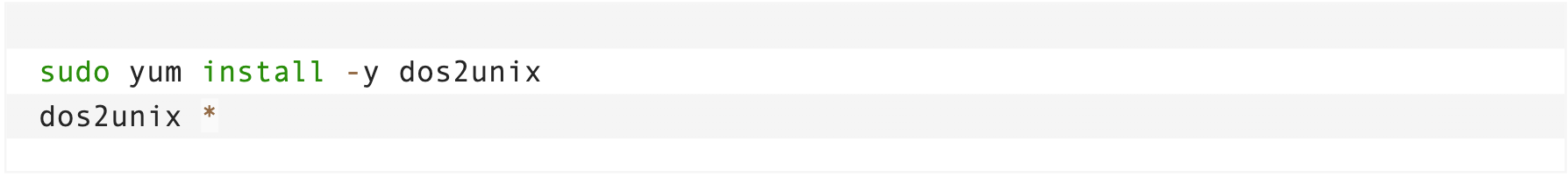
Firstly, a project directory will need to be created for the database; this can be achieved through the following command line command:

Fig 1: Project Dir. Creation



After the project directory is created, Amazon suggests that the software developer(s) download a few scripts in order to use the service, which is shown in the following figure.

Fig 2: Installing Service User



To avoid redundancy, it will be stated that, after a few more steps, outlined in the linked article, the serverless application will now be available; a database will be ready to query, and the database will be ready to have values inserted into it.

1.5 Definitions and Acronyms

|  |  |
| --- | --- |
| **Term** | **Definition** |
| AWS | AWS stands for Amazon Web Services. AWS will be the database back-end of the Drake Relays Companion app. |
| DRCA | DRCA stands for Drake Relays Companion Application. |
| Kivy | Kivy is an open-source Python library capable of creating mobile applications. |
| RDS | RDS stands for relational database service. In terms of AWS, this service will allow the DRCA to utilize SQL for database querying. |
| ARDS | ARDS is Amazon Web Service’s relational database feature. |
| High-permission user | A user who can edit results of events. |
| Low-permission user | The average user who can only influence the “interest” column |

2. SYSTEM OVERVIEW

The DRCA allows users to view Drake Relay events and express interest in them. It does so by storing the events in an RDS and letting a user access that information through an app developed using Kivy.This is accomplished mainly through an object-oriented approach. Events are stored in the RDS prior to each Relay’s week. Within the app, users are split between high and low-permission. This grants certain users the ability to edit the results of events.

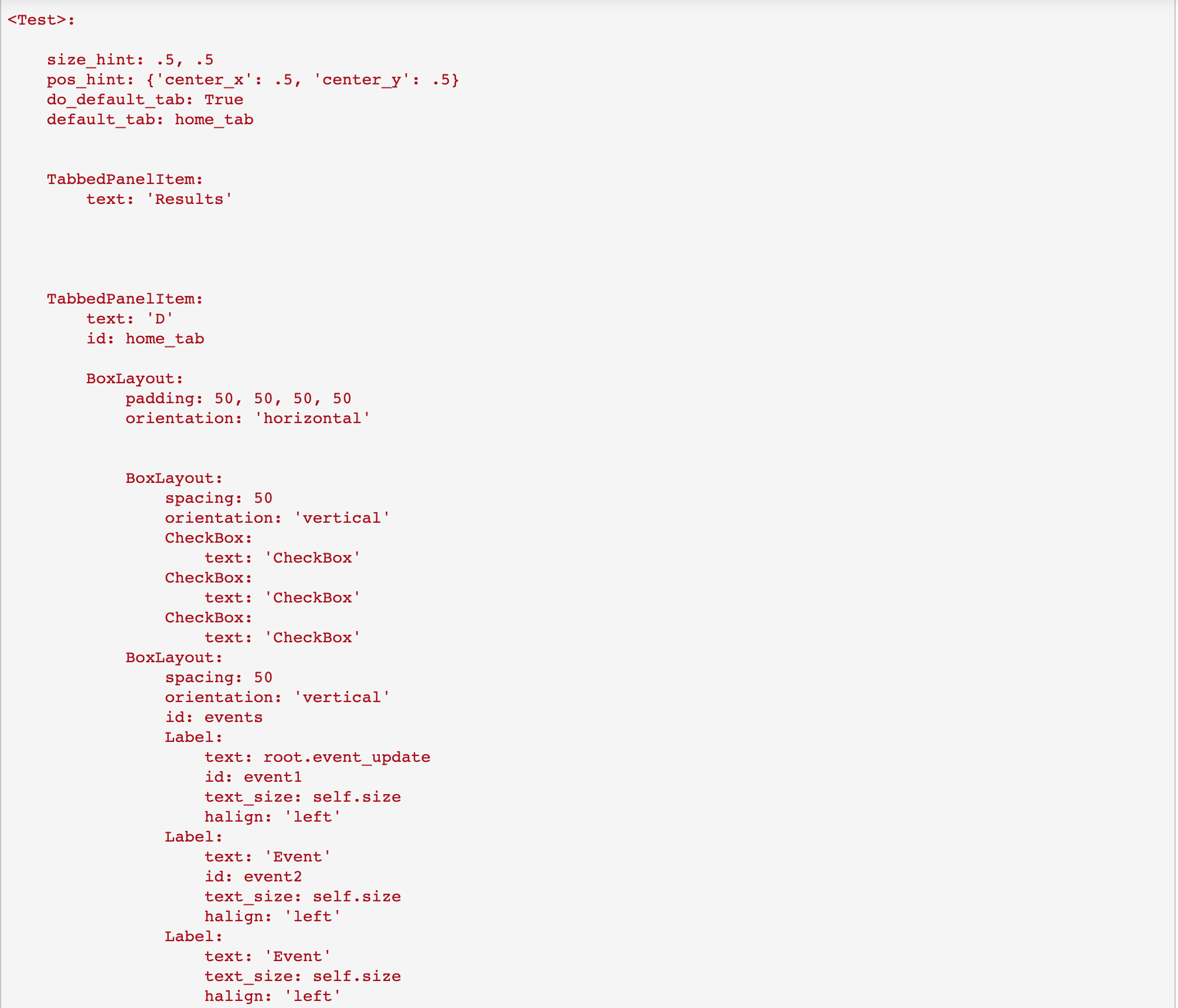
# 3. SYSTEM ARCHITECTURE

## 3.1 Architectural Design

The developers of this software system are utilizing the Python framework Kivy. Kivy allows developers to create cross-platform applications with relative ease. Because the main intent of this application is for it to be a mobile application, it is designed in a like manner--the application design follows typical mobile application style.

The application contains 2 tabs, which can be clicked by the user. The tabs of the application bring the user to different pages, each page serving its own purpose. With Kivy, this is relatively easy. With Kivy, a mobile application can be created with tabs built-in through the .Kv file. Kivy allows developers to take many routes to achieve their goal, but, to obtain a clear and easy-to-use experience for users, the developers of this software system have decided to go with Kivy’s “TabbedPanelItem.” A tabbed panel item, in Kivy’s framework, is what most people experience when they use a mobile application. The tabbed panel item creates multiple tabs across the top of a mobile device which can be clicked, and, when clicked, brings the user to a new screen. A sample of what the .Kv file looks like is shown below.

Fig 3: Kivy TabbedPanelItem



In this case, the application, as a whole, is named “Test.” Within the “Test” application there is a “TabbedPanelItem.” The tabbed panel item is what creates tabs on the top of the mobile screen. Embedded within the “TabbedPanelItem” are nested “BoxLayout” items. BoxLayout items allow the software developer(s) to add items within the tab’s screen--meaning the screen that is present when a user is on the tab.

In the code shown, one of the “BoxLayout” contains checkboxes. These checkboxes are the “user-interest” buttons. When the checkbox is pressed, and then submitted, our database will receive the information, allowing the database manager to recognize, and then output, the relative interest in an event.

The “BoxLayout” below the checkboxes contain labels, which list the events that are being held on the current day, or on a future date.

In order to update upcoming Drake Relays events when the application boots, or when the “refresh” button is pressed, the main class, as well as a secondary class/function is utilized, as shown in the following figures.

Fig 4: Main Class

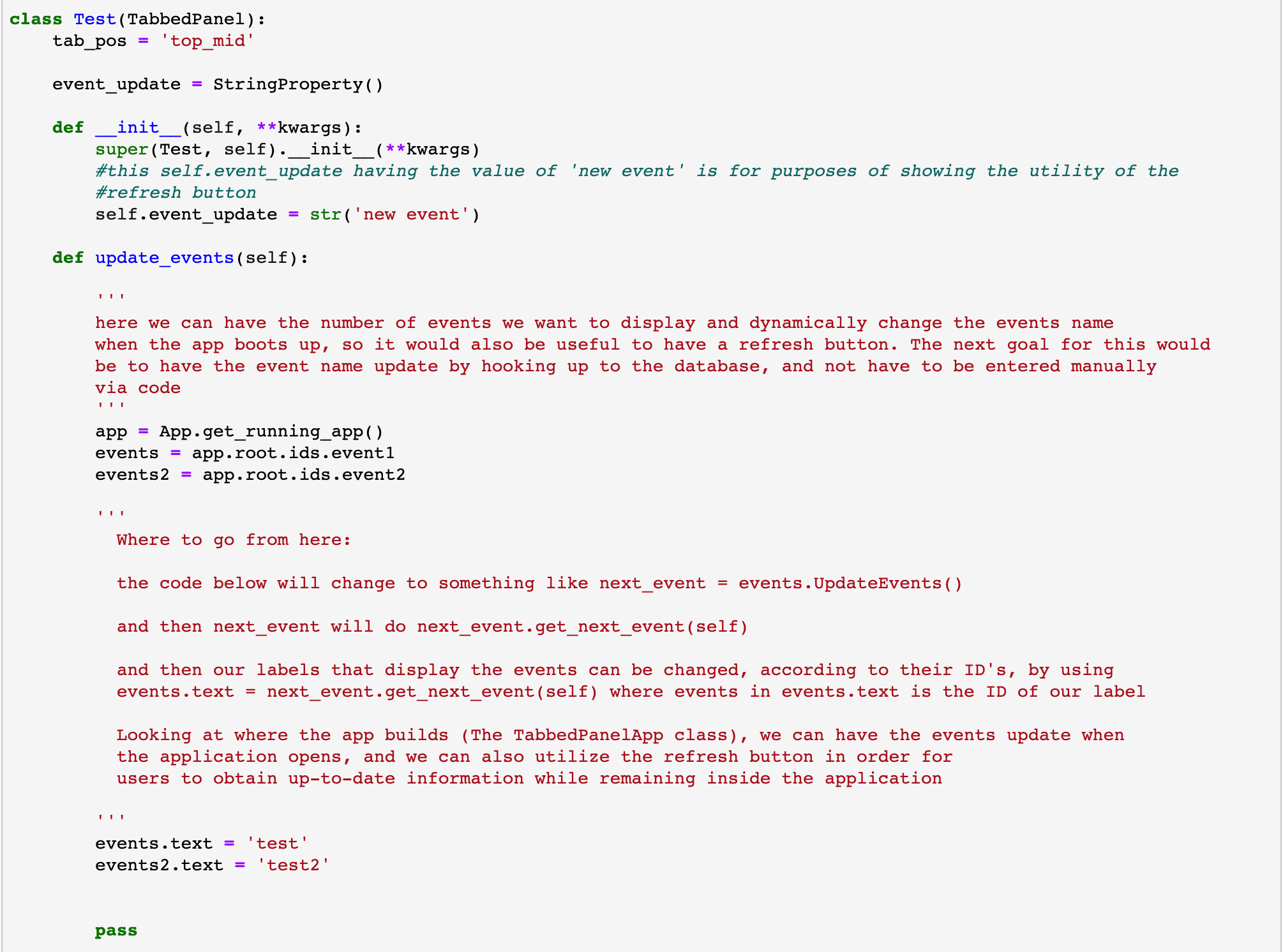


Figure 4 shows the main class’s code. The class “Test” is what ultimately builds every time the application is booted up. Comments are used in this figure to show the ultimate goal of the application, and the design of the code to be implemented.

Within the Test class are an \_\_init\_\_ function and an update\_events function. The \_\_init\_\_ function is what “creates” the application when it is started via the user opening it. The \_\_init\_\_ function creates labels on the main tab; the main tab being where the upcoming events are listed.

The update\_events function, on the other hand, is what allows for the application to update automatically, or when the refresh button is pressed. The update\_events function gets the current state of the application through the line of code:

app = App.get\_running\_app()

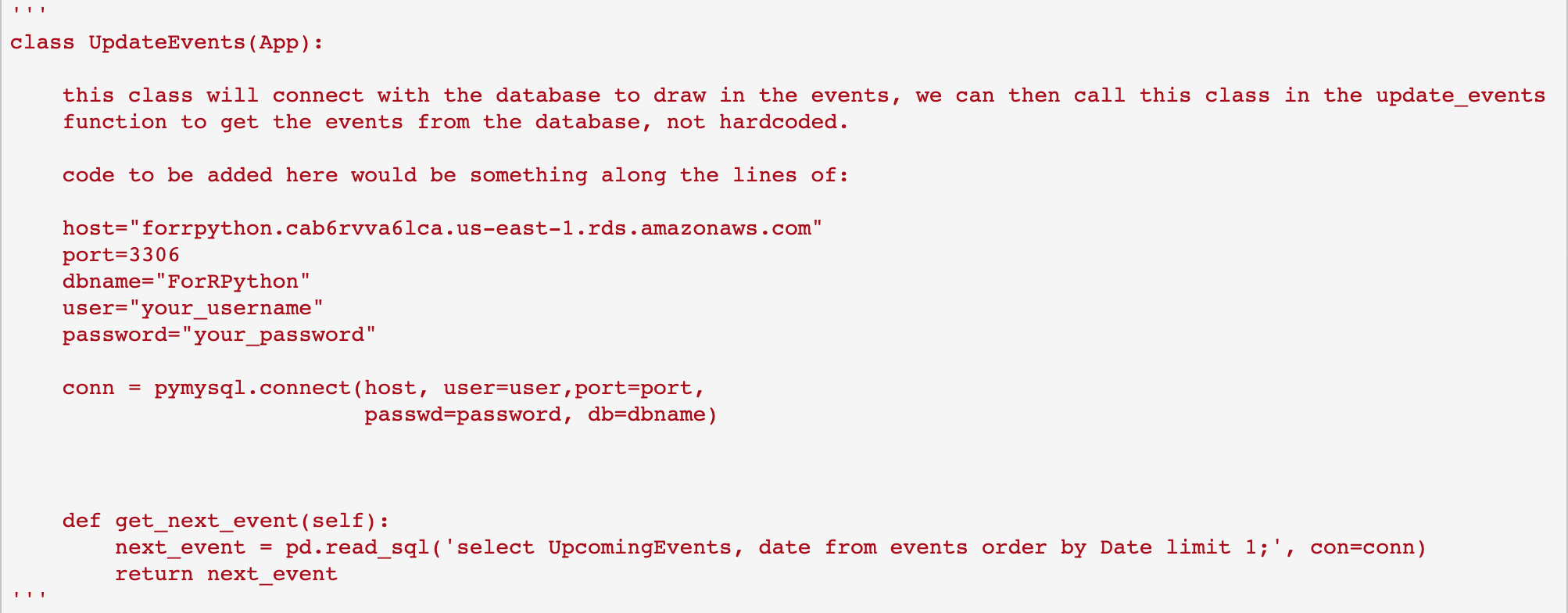
After the variable named “app” is created, the root of the application (specifically the “base” of all the data contained in the current application) can be utilized in order to receive events, and then display the events via the labels in the .Kv file.

Referencing Fig 3, we can see that the event labels have the ID’s “event1” and “event2.” These labels are dereferenced from obtaining the root of the application, and the label’s specific ID.

To iterate how the events will dynamically update, Fig 3 can be referenced again along with Fig 4. When the application is booted up in its current state, the events will show as shown in section 5.2 Event Tab. That is, the event labels will display as what they are (currently) hardcoded to be. However, upon reopening the application, or pressing the refresh button, the labels change depending on what events are upcoming according to the database’s “EVENT” table, and the current date and time.

To achieve this through code, the following figure can be referenced:

Fig 5: Update Events Class



This class, “UpdateEvents,” queries the database which holds the events--specifically, it will query the EVENT table to determine what events should appropriately be shown. Within the UpdateEvents class is the get\_next\_event function.

The get\_next\_event function is where the core of the update functionality comes into play. The get\_next\_event function queries the database, and references the events in the database to determine which events are closest to the current date and time. The function then returns the next\_event variable.

As seen in the comments of Fig 4, Fig 5’s function/class will be utilized by creating a new instance of the Class through the code:

next\_event = events.UpdateEvents()

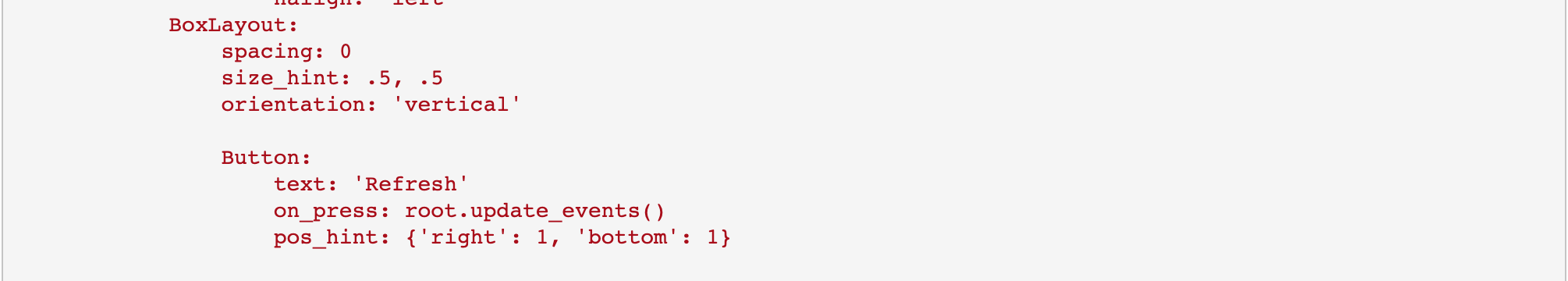
Where the next\_event variable is arbitrarily named and “events” is the label which is shown in the main tab of the application. After the initialization of next\_event, the next\_event variable can then be utilized to update the labels which contain the information about upcoming events by using the code:

next\_event.get\_next\_event(self)

Because the next\_event variable is contained within the Test class, in the update\_events function, the label that is updated via the variable will happen upon the booting of the application. However, if a user has not rebooted the application, they can utilize a refresh button to keep up-to-date with events.

Within the .Kv file, and within the nested BoxLayout features, there is a refresh button, which, when pressed, calls the Test class’s update\_events function. And, because the update\_events function contains an initialized instance of the UpdateEvents() class, which calls the get\_next\_event function, when the refresh button is pressed, the tab’s labels will be updated accordingly. Meaning that the tab’s labels will be updated according to what events are closest to the current date/time when the button was pressed. This can be seen in the following figure of the .Kv file.

Fig 6: Update Button



# 

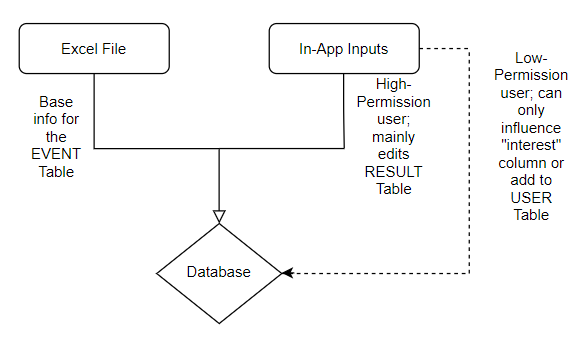
# 

# 4. DATA DESIGN

## 4.1 Data Description

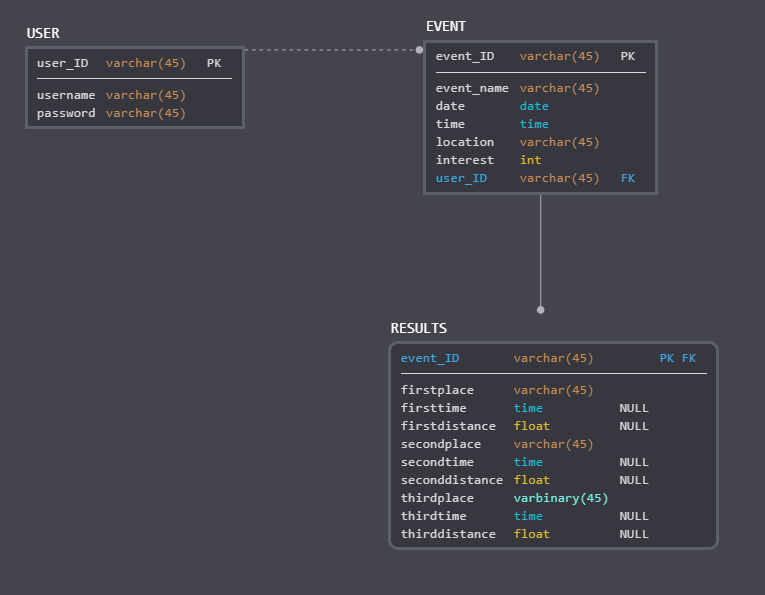
Below is a template of data input for the Relay’s App:

Fig 7: Data Input



Below is a template for the database design that will be implemented for this project:

Fig 8: Database Design



There are two methods of entry for data in the app. The first is data read directly into the database before the app is launched. This data contains information for every event of Drake Relays and is stored inside an excel document with six columns: the event’s ID, the event’s name, the date of the event, the time of the event, the location of the event, and an event’s interest level (default to zero when entering in events). MySQL code will read in the data prepared in an excel sheet to fill in the database’s EVENT table.

The second method of input is In-App input. Users will be allowed to make varying In-App inputs based on the permissions of their account. A low-permission user will only be able to make an account, which will add to the USER Table, or add to the interest column of an event. They are meant to only view the app with limited ability to make actual changes to the data.

A high-permission user, such as the database administrator, will be able to manipulate the data in any Table. However, they will mainly update the RESULTS table since the other Tables are filled by different means. In the RESULTS Table, a high-permission user can enter the name of the first, second, and third-place takers along with their corresponding time or distance score (these are optional so an event doesn’t have to fill in a column that doesn’t apply to a sport).

## 4.2 Data Dictionary

* Login Class(App)- Creates page seen when starting the app; allows users to login to the app or create a new account.

login(self)- Sends login information to the database and enters the app if the information is found; checks privilege level of a user.

new\_account(self)- allows a user to add a new account to the USER Table.

* MySQL

CREATE TABLE- Used to create the tables in the database

LOAD DATA LOCAL INFILE- Used to load data on the events from an excel file into the EVENTS Table.

* SendInterest Class()- Determines if a user has pressed the interest button for an event.

send\_interest(self)- Updates the interest column of an event in the EVENT Table with +1 if a user has not already done this action.

* TabbedPanelApp Class(App)- Builds the app using the information in <Test>.

build(self)- Function used to build the app.

* <Test>- Structure for the app; includes a results tab and a home tab.
* Test Class(TabbedPanel)- Used to update the event tab with Relay’s events.

update\_events(self)- Updates the events listed in the event tab using the UpdateEvents Class.

update\_results(self)- Allows a user with high-permission to update the results of an event.

* UpdateEvents Class(App)- Connects to the EVENT Table of the database and retrieves relevant events for the event tab.

get\_next\_event(self)- Retrieves events by date, starting with the next event to occur and ending with the last event that will occur.

5. HUMAN INTERFACE DESIGN

5.1 Overview of User Interface

Describe the functionality of the system from the user’s perspective. Explain how the user will be able to use your system to complete all the expected features and the feedback information that will be displayed for the user.

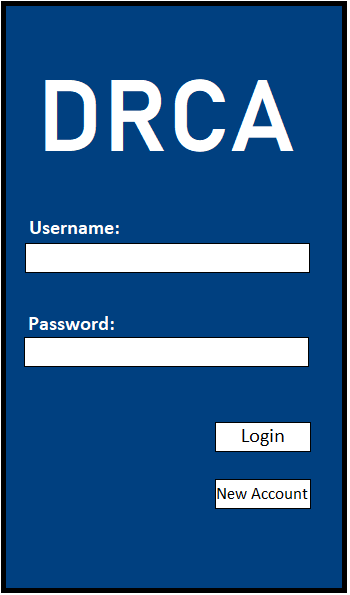
When a user opens the app, they will be brought to the Login screen. They will be presented with two inputs: one for a username and one for a password. If they have not made an account yet, they will click the New Account button. This will take them to the New Account Screen (an altered version of the Login screen). They will fill out a username and password, then they will press the Submit New Account button. This will add their username and password to the USERS Table and send the user to the Login screen.

When a user has an account, they will input their username and password, and press Login. This will take them to the main hub, the Events tab. Here, they can Look through a list of events that are displayed. To see events that will occur soon, they can press the Refresh button. Next to each event is an Interest checkbox that, if checked, adds +1 to the interest column for an event in the EVENTS Table. If they are a high-permission user, they will also have access to the Admin button.

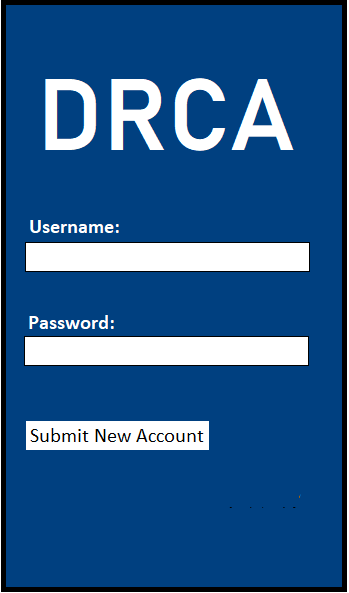
The Admin button takes a user to the Result Edit screen. Here, a user can use the inputs to submit the results of an event. They would input the event’s ID, the names of the first, second, and third place winners; and the time or distance for those winners. This data is entered into the RESULTS Table using the Submit button.

Finally, from the Main Hub, a user can click on the Results tab. Here, they will view the results of events that have been completed.

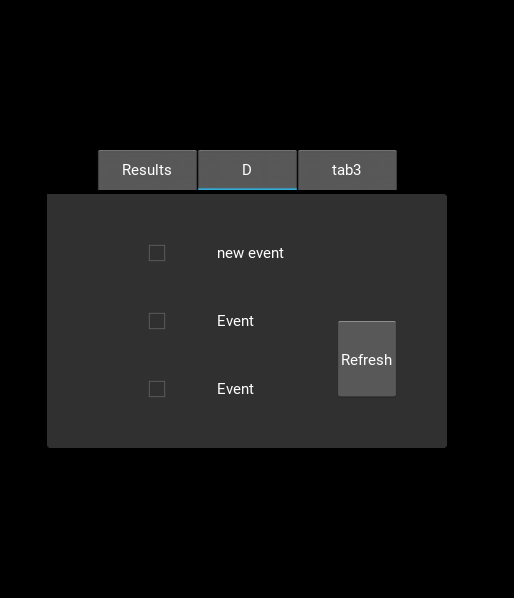
5.2 Screen Images



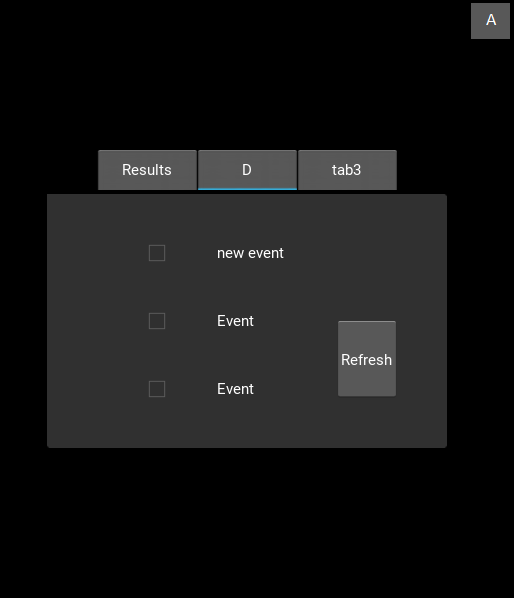
Login screen



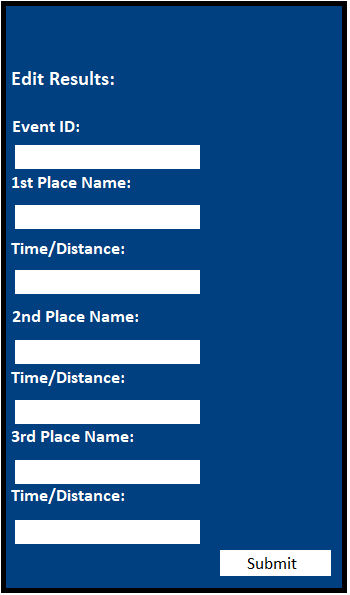
Account Creation screen



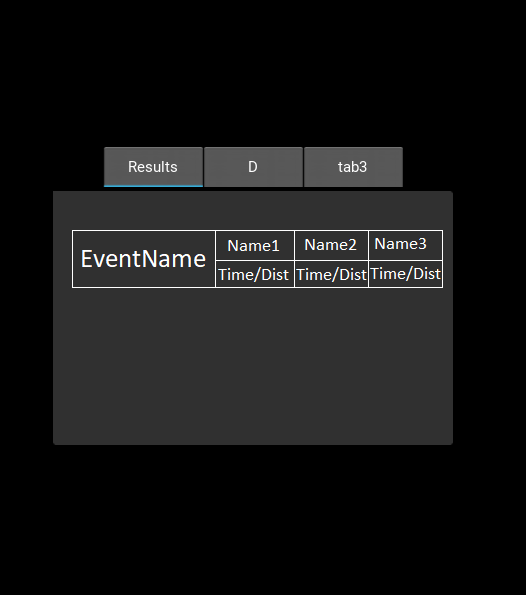
Low-permission Events tab; tab3 is temporary.



High-permission Events tab



Result Edit screen



Results tab

## 5.3 Screen Objects and Actions

Login screen:

* Text- the app’s name and labels for the inputs.
* Username input- allows a user to enter in their username; submitted information is checked against the database.
* Password input- allows a user to enter their password; submitted information is checked against the database.
* Login button- checks information submitted into the username and password inputs against the USERS Table.
* New Account button- allows a new user to create an account; takes them to the account creation page.

Account Creation Screen:

* Text- the app’s name and labels for the inputs.
* Username input- allows a user to enter a new username; submitted information is placed in the database.
* Password input- allows a user to enter a new password; submitted information is placed in the database.
* Submit New Account button- enters information submitted into the username and password inputs into the USERS Table

Events Tab:

* Text- displays information about events including title, date, and time
* Refresh button- updates the events that are displayed in the Events tab.
* Interest checkbox- allows a user to show interest in an event; an event can be checked once to add +1 to the Interest column in the EVENTS Table.
* Results tab- takes the user to the Results tab when clicked.
* Admin “A” button- takes a user to the Result Edit Screen; only viewable by high-privilege users.

Results Tab:

* Text- information on the results of events including the event’s name, the names of the first through third placers, and the times or distances of those people.
* Events tab- takes the user to the Events tab when clicked.

Result Edit Screen:

* Text- displays labels for the inputs.
* Submit button- sends updates located in the inputs to the RESULTS Table of the database.

6. REQUIREMENTS MATRIX

|  |  |
| --- | --- |
| Functional Requirement | Component/Data Structure\* |
| 1. Main Event Hub | <Test>  Test Class(TabbedPanel) |
| 4. Interest Meter | SendInterest Class()  send\_interest(self) |
| 6. Results Hub | Test Class(TabbedPanel) |
| 7. Upload Events by Date/Location | MySQL  LOAD DATA LOCAL INFILE |
| 9. Ability to Edit or Cancel Events | UpdateEvents Class(App)  get\_next\_event(self) |
| 10. Ability to Edit Event Results | Test Class(TabbedPanel)  update\_results(self) |
| 11. A Login System | Login Class(App)  login(self) |

\*See Section 4.2 Data Dictionary

Requirements Cut Due to Time Constraints/Feasibility Issues:

2. Calendar of events

3. Map of campus

5. Google API Traffic Data

8. Highlight Upcoming Events