Micheal Willard
932480626
CS 496
Final Exam

1. Overview

This documentation discusses Baseball Game Scheduling app. The documentation discusses the 6 Quality Attributes of software engineering. The 6 attributes are discussed in the context of how the Baseball Game Scheduling app and API meet and don't meet them.

2. Usability

One of the great things a bout developing in Xcode for iOS is that Apple is very keen on user interfaces. This makes achieving high levels of usability very easy. This application was design with the iOS Human Interface Guidelines [1] in mind. Xcode gives the user interface objects and a storyboard layout to verify the the views achieve acceptable levels of usability.

This application from the first Login View attempt to keep the views uncluttered and as minimalistic as possible. A mobile app is constrained by screen size, so keeping screen clutter to a minimum is essential. The Login Screen presents the user with 2 text fields which are clearly labeled, and two clearly marked buttons. A user can only login here or go create an account. The text input fields are checked for valid input, not just as a way to prevent errors within the app, but to allow the user to whether their password was wrong, the email doesn't exists or if they accidentally hit the Go button before filling in the fields.

If the user needs to create an account, they are taken to the Create Account View. This view contains 5 input fields, a Create button and a link to the Login View. Knowing screen real estate is at a premium, the form is laid out with reasonable spacing. The text fields themselves tell the user what values should be entered in them, as opposed to using clutter-some labels. A couple input controls are placed in here. One is to ensure all fields have been populated, but the second is to ensure that the user enters a password they can remember. This is done by having two password fields. The application checks the two input passwords to verify that they match and alerts the user if they don't.

After the user logs in, they are presented with the Profile View. It has a clearly marked Logout button and their name so they know they are logged in. One of the keys of usability is familiarity. Any user who is familiar with mobile applications is going to be used to having navigation cues placed on the top navigation bar of each view. This concept is tenant of the iOS Design Guidelines and a feature in the Xcode Storyboard Editor. This application makes use of the navigation bar throughout, on this page "Done" tell they user where to click to escape out to the rest of the application.

The Schedule Table View is what is accessed after closing the Profile View. The user is presented with a familiar table view with the games laid out and labeled. A bit of a leap of faith in user familiarity is taken on this page. It is assumed that a user is familiar enough with mobile applications that they know they can touch on a table cell and something may happen. That is how a user would segue to the Game Detail View in this case. Additionally, the navigation bar in this view has a familiar "+" symbol to add a game and a "Profile" button to go back to the Profile View.

Selecting a Table Cell will take the user to the Game View. Here they can view the game or navigate back using the aptly named "Back" button. The game information is displayed as well as an interactive map object. The User should be familiar enough with embedded map objects to know they can interact with it. Additionally, there is text prompting the user as to whether or not they are attending. Right below that text is a toggle switch. Toggling the switch updates the users status and immediately updates the text saying whether or not they are attending. That sort of interactive feedback should provide good clues to the user that they are using it correctly.

Back at the Table View, the user can select the "+" symbol to allow them to create a game. The Table View being minimalist to the point that it only displays games, should be the user's clue that the add button refers to adding a game. Thus we can just use "+" in lieu of the more wordy "Add Game". When the user goes to the Add Game View, the top navigation bar has a "Cancel" and "Save" button. Again, not just familiar keywords are used but those are preset in the Xcode editor for consistency. The Save button is disabled until all text input fields are filled, since they are required fields.

3. Portability

The backend is very portable, however the application is not portable at all. The front end is design and developed for iOS. Porting the code over to a new system would be very challenging for most developers as it is written in the iOS proprietary language of Swift 3.0. There interfaces and program flow would be about all that could be salvaged.

The interface for the backend is portable in a few ways. Any non-relational database could be used with the python source code. The python source code is not native to any one system, with the exception of the google ndb definition that is declared in the python files. There are a number of other non-relational databases that offer similar services and are incorporated in a similar way.

Another way it is portable, and perhaps the way it is most portable is the usage of the API interface. The API is not native to any specific system. Any application or language that supports some form of JSON parsing, which is nearly all modern languages and IDEs, can Create, Read, Update and Delete data from the database through the API. Simple URI interfaces ensure this is the case. As long as a developer has visibility to the UIR nodes, they can interact with it.

Finally, the way the applications itself interacts with the backend has portability attributes. The source code itself can easily be adapted to interface with an API that provides the same services. IF the API were to be moved to a new server access point with new URIs, only one line of code needs to change. Of course this isn't the case for if the API changes it's structure significantly. For instance: if accessing game information from an API required specific inputs or returned information that was formatted differently from the current API, then some more changes would be needed.

4. Reliability

The backend of this application is built on Google App Engine. Google App Engine is relatively reliable for cloud infrastructure, they offer certain uptime guarantees. Its a backend built more for affordability. The equipment is not extraordinarily fast or powerful by modern computing standards.

This is a relatively simple system back to front, so we can mitigate some reliability issues there. If the cloud does fall short on it's guarantees, the Google Cloud Platform does have excellent dashboards and accessibility for viewing the status of your servers. Their guarantees on uptime put you in the range of 95-99.95% monthly uptime. The higher end of that is 99.00 to 99.95% which is acceptable for this type of application and puts you in the three nines level of reliability.

Google App Engine additionally has built-in code replication that occurs automatically. This is needed for the load balancing features. The GAE servers can copy the code over to new server instances and continue running the backend application there. In addition to code replication, it also has data replication. This API is only configure for simple backups. The scope of this application was small enough that code replication and server traffic aren't factors at the moment. But the ability to expand with ease is there. For example, an application like this would be of most use if it could be scaled to accommodate multiple teams across the country.

The non-relational database (ndb) used in the Google App Engine back end implements smart caching to help with speed reliability. However, it can also be prone to latency. During testing of the application in the iOS simulator, some latency may have been noticeable. However, given the relatively small dataset that may have been more related to the limitations of the the iOS simulator program.

The latency is also an issue when writes and reads are occurring within a second of each other. To overcome some of this issue, the client application loads the database data each time a new page view is presented. The data is stored in local objects, so that when you switch from the Game Table View to the Game Detail View, the data is consistent. If a change occurred while you were in the Game Detail View, you will see that change when you navigate back to the Game Table View.

The Game Add data is the only that can be affected by multiple people at one time. Part of controlling that is to have a player's attending or not attending status to only be controlled by that logged in user.

5. Performance

To address performance within the iOS application, some design considerations were made. Since all the data in the app is retrieved from URI based API, it was important to make the distinction between when we need to retrieve new data and when we should work with local data. It was decided that each time the Game Table View loads, new data should be retrieved from the API backend. The GET call is made to the API, then that data is stored into local Game class objects. The table is then built using the objects. A poor design would have been to make a call to the API when each table cell is built. With this way, we make one call to the API, retrieve all the data then build using the local objects.

The second design decision regarding when to access the API addressed what information is displayed then the Game View is pulled up by selecting a Game Table cell. There were two options here. A new call could be made to the API for that specific Game ID, or the existing game object could be displayed. From a performance standpoint, it would be much quicker to use the local data stored. As discussed in the Reliability section, the trade-off here is that the game data may not be up to date. That trade-off was deemed acceptable.

Those two design decisions also help minimize the load on the server as well. Minimizing the load on the server greatly improves the backend performance. We are discussing the operation of one person running one instance of the app, however in service this app could be millions of users running instances of the app. If we are making dozens of API calls that we don't need, we are stressing the server. The data is also returned using a standard format in JSON. This keeps the size of the messages the server is delivering down to a minimum.

The Google App Engine backend generates its own pointer based indexes. These indexes allow for fast data retrieval from the non-relational database. This was all handled automatically by Google App Engine, no additional changes were made from the default settings of the system.

6. Security

Having all the data stored in the backend API, the onus of data integrity and availability fell to the design of the API. The information contained in the non-relational database can only be accessed through API endpoints. Most of these endpoints are restricted to GET access. The few points of API entry that allow data manipulation are controlled strictly thought both the interface of the iOS client app and the handling of the commands in the python backend code. Additionally, the client app requires a user to have an account with a password to access the app beyond the Login or Create Account Views.

For the GET access, this is straightforward. The data is simply accessed within the app through controlled calls. No user input affects those calls. This ensures the integrity of the data, as the access points always provide exactly what is needed. However, the availability of the data in the APIs could be vulnerable to availability issues. If a entity had access to the API URIs, they could easily affect operations through a DDOS attack. This vulnerability wasn't addressed due to the scope of the project.

For the PUT and POST calls within the app the integrity varies a little bit. The PUT calls are restricted to a user being able to select whether or not they are attending a specific game. This shows a level of data integrity, as only the user who is logged into the app can change their status. Of course requiring the user to be logged in with an email and password is it's own form of security. Ensuring that only the user is able to affect their status for a game keeps other users or evil entities from changing the user's data. Again, an entity with access to the API URIs could exploit the vulnerability on the backend. If the scope of this project was expanded, the backend could be updated to perform some form of authorization when a PUT or POST request is made. But for now it is assumed the iOS clients interface is protection enough.

The POST request doesn't restrict any user type. Any user with an account can create game objects. One potential change would be to modify the accounts system to have privileges. It would be designed so only administrators could create games. That's a relatively simple design change, but for purposes of this project and demonstrations it wasn't needed at this time.

Finally, the primary security feature is the account system. A user creates an account with a password. That password is stored on the databases and is not accessible through any API URI. The only way the password is accessed form the database is through backend code not visible to an external user. This is the form of the ad-hoc authorization system built for the app. The Login View sends a POST request to the server. The request includes the email address and password of the user. The back-end code queries the database against the email address, and checks the password stored in the database for that entry. If they match, a "token" is sent back to the client app in the form of an HTTP response. The response contains the user's account info like their name, email, phone, etc. An incorrect email, password combination returns simple failure response, again not providing any details on the password.

7. Interoperability

The Google App Engine backend provides all data responses in JSON format. JSON is a very common standard that nearly every modern programming language has libraries that can interact with the JSON objects. In the case of this application, Swift 3.0 has JSON parsing functions in it's libraries. If the program had been written in C++ or JavaScript, any of those languages would interact with the back-end in the exact same way. This ensures the back-end portion is fully interoperable for any future application platforms that would be developed for. Google App Engine doesn't currently conform to the Cloud Data Management Interface (CDMI) Standard. That standard would be a higher level of interoperability as it defines interaction standards.

8. Sources

1. iOS Human Interface Guidelines, https://developer.apple.com/ios/human-interface-guidelines/overview/design-principles/

9. Appendix

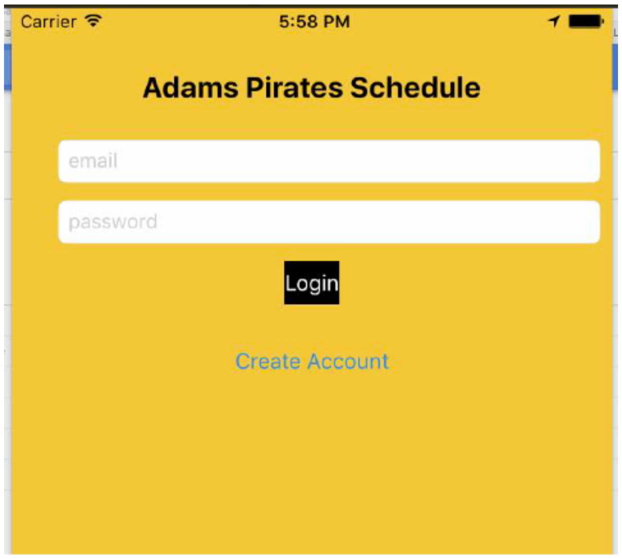


FIGURE 1: LOGIN VIEW

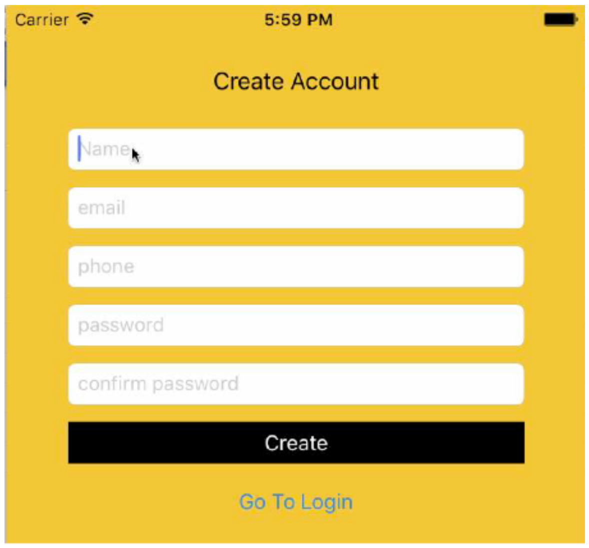
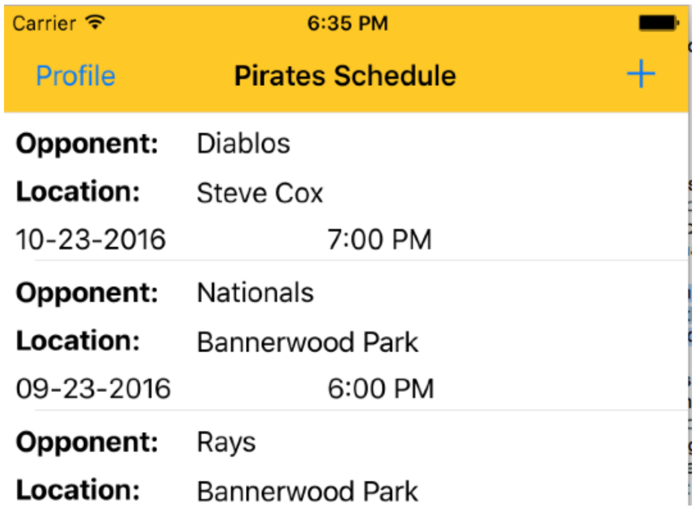


FIGURE 2: CREATE ACCOUNT VIEW


Carrier 6:35 PM

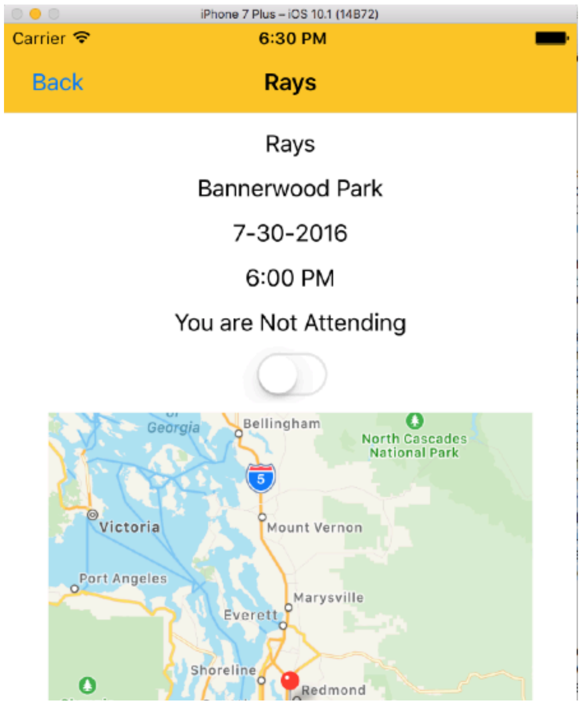
Profile **Pirates Schedule** +

Opponent: Diablos
Location: Steve Cox
10-23-2016 7:00 PM

Opponent: Nationals
Location: Bannerwood Park
09-23-2016 6:00 PM

Opponent: Rays
Location: Bannerwood Park

FIGURE 3: GAME TABLE VIEW

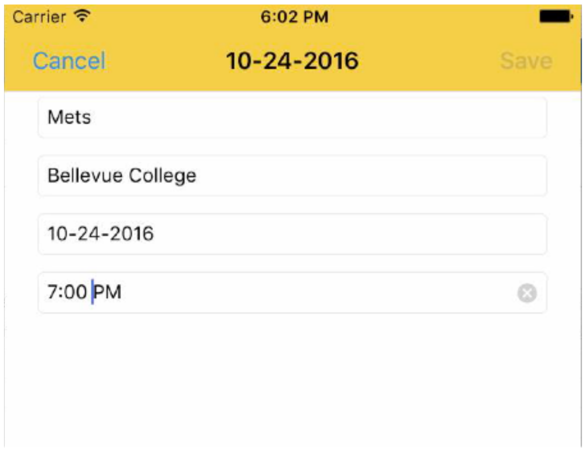

iPhone 7 Plus – iOS 10.1 (14B72)

Carrier 6:30 PM

Back **Rays**

Rays
Bannerwood Park
7-30-2016
6:00 PM
You are Not Attending

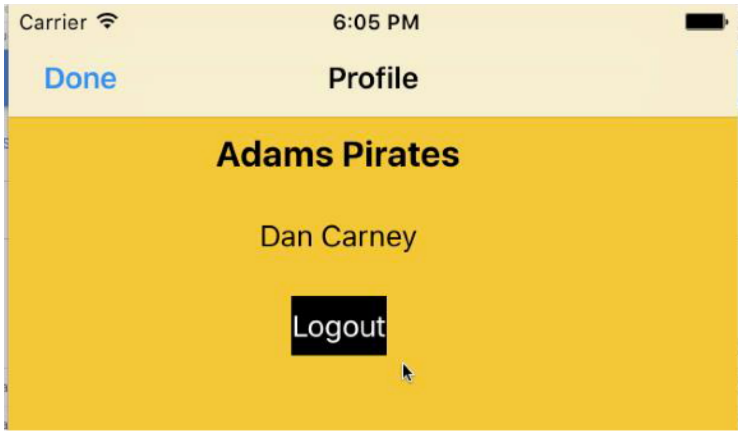
FIGURE 4: GAME DETAIL VIEW


Carrier 6:02 PM

Cancel **10-24-2016** Save

Mets
Bellevue College
10-24-2016
7:00 PM

FIGURE 5: GAME ADD VIEW


Carrier 6:05 PM

Done **Profile**

Adams Pirates
Dan Carney
Logout

FIGURE 6: PROFILE VIEW