**Design Issues Elaboration**

*(Refer to High-Level Design Issues)*

*Maintainability*

* *i.1 API Assurance*

API creates an issue for us when it comes to the maintainability of the application. Because we have no control for the well-being of the external sources we are using to populate our application with information, we may be at the risk of losing major functionality if one of those API providers shutdown. Because of this, over the course of development, alternative API sources must be ready within our library to maintain any longevity without any downtime of function. API providers also limit the amount of API calls we can query. This can be remedied in the near future. If we start to receive more clients and begin making a revenue, that revenue can be used to purchase more access to the troubled APIs. In the meantime, the code has to be constructed so that we limit the amount of API calls and optimize information gathering.

* *i.2* *Using a 3rd Party Database (Firebase)*

Because Firebase is being provided as a service to us, we are tethered to their success. If they shut down their services, than effectively are shutdown as well. In order to combat this, in the future we need to develop a method to transfer the server to another server or by housing our own Firebase Server prior to any big event. Regardless of any backend logic that can be written, the structure at which Firebase provides its service is also unable to receive personal treatment. As technologies become more advanced, we are again at the mercy of Firebase to update their services.

*Testability*

* *i.3 Difficulty Testing Accuracy of Algorithms That Tailor to Health Needs*

Phoodbuddy has a reliance on retrieving external data through provided APIs. Our algorithms that will be sifting through the data will have to accurately take into consideration Health Profile issues that users may have. Because we are not actually providing the data internally, we have less control of how accurate the data being retrieved is. Testing that our algorithm is useful when we cannot 100% rely on the data provided makes handling errors a troublesome task. The solution is to focus time on creating a robust algorithm that not only checks for required components, but also checks the components themselves to ensure that they are accurate.

* *i.4 Multi-Platform Testing*

Phoodbuddy is currently supporting three modes of operation: Web, Androids, and Windows. This requires that we develop a solution that tests the functionality across all three interfaces. We must develop the logic layer so that it handles the queries coming from different platforms to and from the database. The traffic associated with this system must communicate without fault. Testing that the independencies of the sub-systems work properly is a concern that will be addressed during in the testing documentation.

*Performance*

* *i.5 Dependencies on 3rd party database systems*

A lot of database components are not “in-house”. Phoodbuddys database will be managed by Firebase on one of their servers. Our data being retrieved by API calls from FatSecret, Walmart, Facebook, Google, Twitter, and FitBit is also being run on servers external from our system. We are dependent on the performance of external systems that we have little control over. As a result, when it comes to our applications performance, we are unable to predict exactly how effective our queries to the databases will be. We can, however, run tests to estimate the average query times from these external databases. This information can assist in optimizing our internal testing to compensate for the delay that will inevitably transpire from the external database.

*Portability*

* *i.6 Changing Databases and Resulting Effects*

Even though we hope that Firebase continues to serve us, there may come a time where their services will no longer be active. If this happens, the option of migrating to another database architecture may become the most appropriate solution. With this in mind, when developing the Logic Layer and PHP backend layer, we must be careful to design in modularly so that classes from them can be modified to suit another database architecture.

*Safety*

* *i.7 Misrepresented Data Retrieved from External Sources*

One of Phoodbuddys main focuses is to cater to the users’ preferences and health concerns to optimize their experience. These variables can affect which recipes users receive. As the implementers of this data, we must concern ourselves with any inaccurate data that may lead a user to consume a recipe or product that may harmful to their current health status. For example, a user that is allergic to peanuts cannot consume a variety of Nutella because it contains Peanut extract. Therefore our solution to misguiding users into harmful decisions will be to optimize the searching algorithm for retrieving the recipes as well as warning the users of our inability to provide 100% reliable information. However, we will try to benchmark ourselves so that returned requests are at least above a permitted percentile (To be decided)

**Design Prototype Results**

* *Interface Class Diagram(Figure A.1)*

The design was formatted so that it can be a universal template for how all 3 platforms will function on its user interface component. Essentially, the user will always land on a splash page which will invoke the login process. One design issue was how the user was to be authenticated into the system. We concluded that the user can gain entry to the dashboard of the application through verification from Facebook, Google+, or an internal account on our system. (*Refer to ?.?)* Once they pass this system and are verified, they gain entry to the dashboard. This sub-system comprises of the pages the user can surf through in order to obtain some sort of desired functionality. They can browse user recipes, establish a weekly recipe plan, analyze their health/Fitbit data, and set up grocery lists. Users can also customize their profile, including their aesthetics of their dashboard. This can all be navigated using the collapsible NavBar menu

* *Frontend Logic Class Diagrams (Figure A.2, A.3)*

This design was established to show all the necessary functional calls that each component will have to utilize, either to read and write from the database or to make queries to the external components which will be providing a lot of the data that the user will be interacting with (recipes, deals etc.) A lot of the design issues here was whether to focus on an internal driven system where users populate the Phoodbuddy database with data or retrieve from outside sources. *(Refer to ?.?)* Android and Windows will be using the interfaces provided to make connections to the databases from here, since there is very little functionality in place for developing this logic in the backend for these platforms. The web platform will be making the connections to the APIs from the PHP layer (*Refer to “High Level Design – Component Diagram”)* We have the API calls for our use collected from *FatSecert*, *Walmart*, *FitBit*, *Facebook*, *Google*, and *Firebase*. We also have the other functions that will encapsulate the processes of reading and writing particular sets of data to Firebase; In the web system, those calls are handled by the AJAX interface to be done in the PHP layer.

* *PHP Class Diagram (Figure A.7)*

The PHP backend layer was written so that it can clearly be distinguished what general operations were to be performed in respect to the actual functionality that’s needed to be performed for the user. Since the relationship between the front-end logic-layer needs to stay consistent with the PHP layer, a lot of the design philosophy that the two will be using will be similar. This diagram also establishes the relationship with the API that will be providing the information external from our system. This includes the connection to the *FatSecret*, *Walmart, FitBit, Facebook, Google,* and *Firebase* APIs. The connection to Firebase is shown by the PHP class called “PHP-Firebase “centered in respect to all other classes. “This class will have the logic for sending and retrieving data from the database. The other classes surrounding this class will display the logic for interpreting data for each of the main Sub-Components that the frontend logic layer has to implement.

* *Database: Firebase Design Diagram (Figure A.8)*

Firebase is established as a NoSQL hierarchical database which organizes data into a system of directories. It stores its data as a constant JSON string. It can read and write data using two methods: By file path or by priority. This system can be compared to by the hierarchy of a file browser on someone’s computer. When a node is retrieved in Firebase, that node and all of the children associated it with are retrieved at once and sent back to the caller as a JSON String. This means that the design is optimized by “flattening” (Restricting the depth size of a node to prevent unnecessary data collection). The keys that follow each initial root directory is used to create the relationships across multiple root directories to organize data for different groups, whether that be users or recipes. This allows us to develop a relationship between nodes but still keep data retrieval small by separating components that don’t have to be called up all at once.

* *Interface Languages Diagrams (Figure A.4, A.5, A.6)*

These diagrams were drawn up to show the relationship of languages that make up the interface layer for each individual platform leading to the logic layer. The overall arching interface diagram (*Refer to Figure A.1)* represents the functionality the user expects using the interfaces, and all platforms must incorporate those functions. The environments in which this will be accomplished (coding languages) are represented by these diagrams.

**Design Decisions /w Associated Risks**

*Database Provider: Firebase over Cloudboost (Figure A.8)*

The design decision over which database to use was narrowed down to two database services. Ease of use was an important decisions when it came to narrowing down these services. Cloudboost and Firebase are both a Database as a service, or a DBaaS. When it came down to it, we required a platform which provided some sort of web-hosting. Cloudboost currently does not provide this service, however Firebase did. Both Database architectures were NoSQL variants, but Cloudboost was more rooted to trying to emulate a SQL style relationship system, defining one-way and two-way relationships with built in applications. Firebase on the other hand is a NoSQL database focused on hierarchal data, similar to a file structure. The risk is working with a database architecture unfamiliar with the team. This can lead to issues with retrieving data and testability. However, we will adapt to that solution.

*Switching from tracking orders to tracking deals*

After doing more research, the team decided that it would not be feasible to link our application to being able to make or track orders provided from different retailer API, such as Walmart’s API. It is impossible to do so in this sense, so switching to tracking “deals” within Walmart’s system was more feasible. This will help solve a lot of design issues that were presented when trying to find a method of incorporating the orders into our system. This helps improve the testability and reusability of this system.

*Using API external sources instead of creating an internal recipe manager*

The reliance on 3rd party sources was a tough call. Unfortunately, given the time frame of this project, we were forced to populating a lot of the user experience with information ready to go. This has a few pros and cons. The cons are that we are now relying on external parties that provides some of the main functionality of our application. The longevity of the product may be subject to the success of the companies being relied upon. The pros include less coding on our side, resulting in less development time to integrate the system. It also means that our application is more scalable where we don’t have to put so much stress on actual data storage. This allows our use of Firebase to remain free for longer, before having to commit when we need an upgrade in data hosting.

*Windows: Coding for UWP*

For the last year, Microsoft has really been pushing its new operating system Windows 10, to become their all-encompassing platform. UWP, or universal windows platform, is what is supported by the windows 10 architecture, but with the niche that it is not backwards compatible with Windows 8 operating systems. The user market for Windows phones is already a low percentile, but restricting to UWP makes that even shorter. However, in the long run Microsoft will have to push all of its Windows 8 users to the new architecture, making this marginal difference obsolete, and rendering any Windows 8 component useless. This increases longevity and usability of the UWP component.

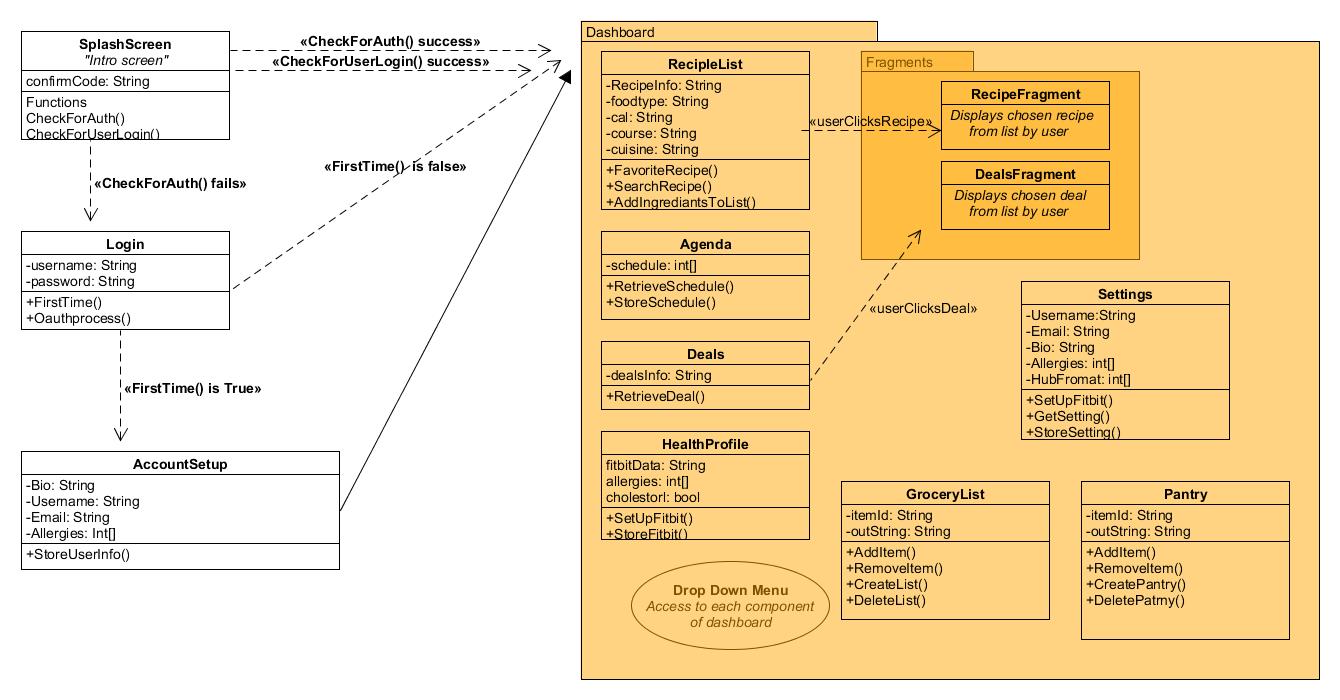
*Web-Based iOS development*

Lack of team experience and knowledge forces us to exempt iOS to be a part of the development process in terms of mobile application. However, creating a dynamic web portal for the application allows us not to alienate the apple market. This however comes with the risk that if your web client ever goes down, we lose our Apple market.

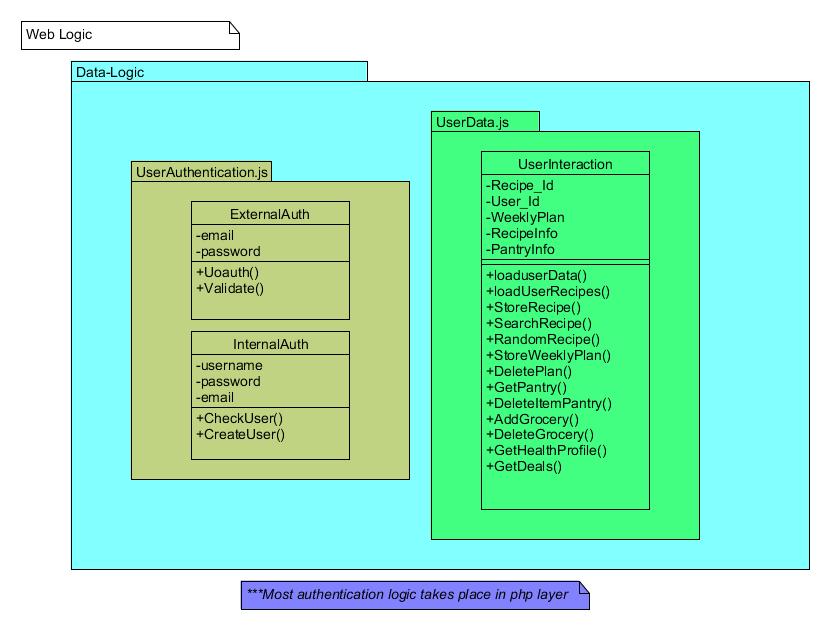
Web: Additional PHP backend for security

Because the Web component is in use for desktop and iOS markets, security on this front must be at its best. Therefore, alongside Firebases built in security measures, we have incorporated an additional layer of PHP to optimize the security. The risks with this is putting an increase in development time for the web component of Phoodbuddy, but it ensures that our iOS and desktop market has more security since we have to manage information coming from two different platforms.

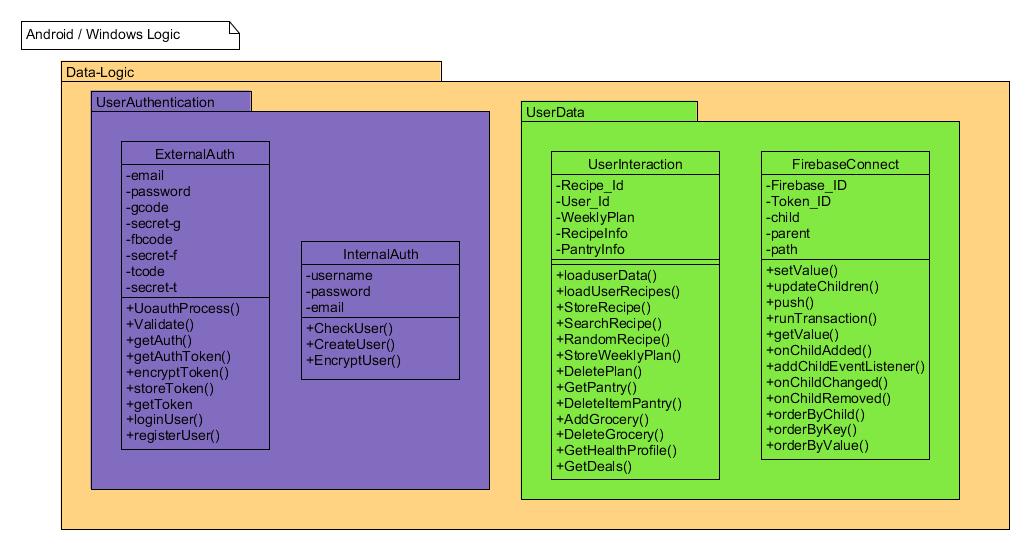
**Detailed Design Information**

***Figure A.1: Frontend Interface Class Diagram***

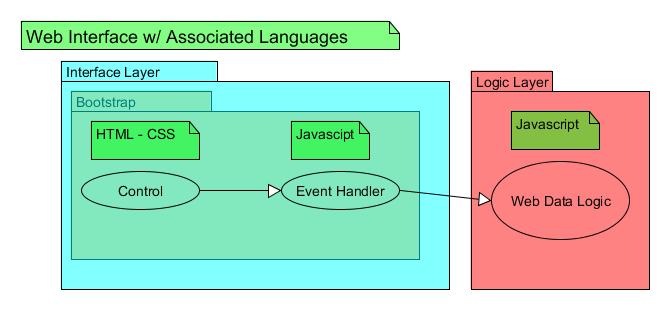
***Figure A.2: Web Logic Layer***

****

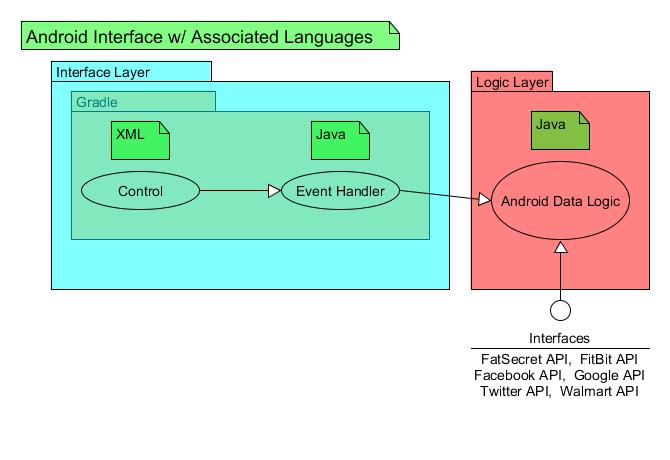
***Figure A.3 Android / Windows Logic Layer***

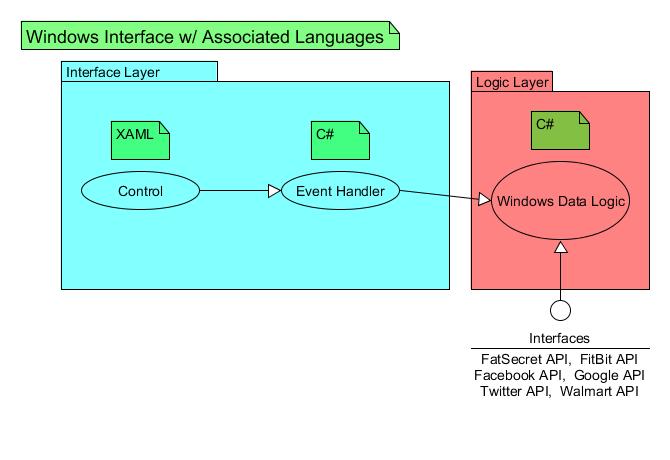
****

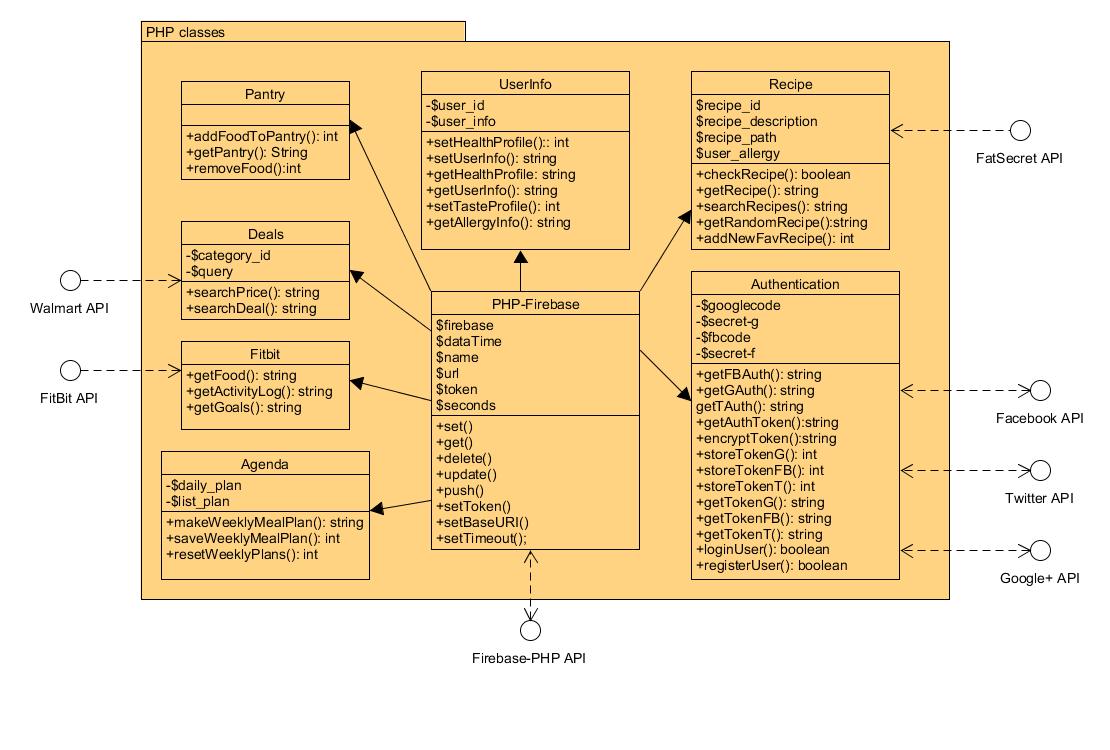
***Figure A.4: Web Interface w/ Associated Languages***

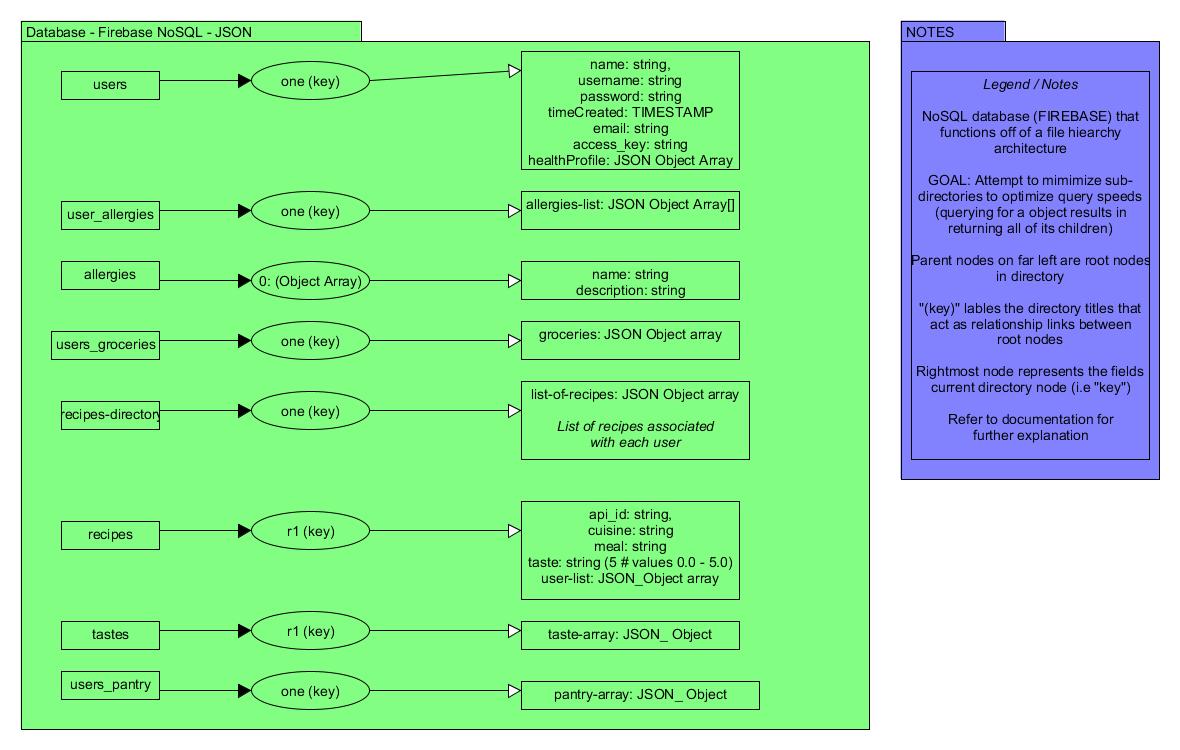
******

***Figure A.5: Android Interface w/ Associated Languages***

******

***Figure A.6: Windows Interface w/ Associated Languages***

***Figure A.7 PHP Layer***

***Figure A.8: Database: Firebase Design***

**Trace of Requirements to Design**

|  |  |
| --- | --- |
| **Requirements ID** | **Figure ID** |
| **1.10, 1.11, 1.16, 2.8, 2.12, 4.2, 4.3, 4.4, 8.2** | **A.1 User Interface Class** |
| **1.11, 2.12, 3.3, 3.4** | **A.2 Web Logic Layer** |
| **1.14, 2.7, 2.8, 2.9, 2.10, 2.11, 4.2, 6.7** | **A.3 Android/Windows Logic Layer** |
| **3.3, 3.4, 4.2** | **A.4 Web Interface Languages** |
| **1.1, 1.2, 1.3, 1.4, 1.12, 1.13, 1.15, 2.10, 2.12, 3.1, 3.5, 6.7** | **A.5 Android Interface Languages** |
| **1.9, 2.11, 2.12, 3.2, 3.6** | **A.6 Windows Interface Languages** |
| **2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 4.2** | **A.7 PHP Class Diagram** |
| **1.6, 1.7, 1.8, 2.12, 4.1, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 8.2, 8.3** | **A.8 Database Design** |

***\*Refer to SRS Documentation for Requirements ID***