

UNIVERSITY COLLEGE LONDON
MSc FINANCIAL SYSTEMS ENGINEERING

CharityWare - Final Report

Authors:

Vasileios ANGELOPOULOS
Yanika FARRUGIA
Alex MARTIN
Irina MNOHOGHITNEI
Christos TOLAKIS

Supervisor:

Dr. Dean MOHAMEDALLY

April 16, 2013

1 Executive Summary

CharityWare is an Open Source Web and Android-based framework, which helps charities with data collection in remote regions of the world. As an example, charities are interested in gathering information regarding children that have been harmed or have been made homeless or regarding medical supplies that are urgently required to reach a specific region. The information is gathered through forms created through the web portal, which can then be populated by volunteers by using their Android device. When sufficient information has been gathered, statistical analysis can be performed and the generated reports can be extracted using Excel.

CharityWare has been developed in Java and contains both an Android and a Web-Application. The Web-Application consists of Prospective Charities, Registered and Verified Charities and System Administrator Views. Forms can be created by the verified charities that have registered their interest in the framework, in order to suit their specific needs in terms of data retrieval. Once forms are completed, volunteers can login to their charity using the Andoird application, where they will find all forms created by that particular charity, which they can populate with the data acquired on the field. The Charity administrators can then view the data that has been populated in the forms they created and generate relevant statistics.

CharityWare has been released in two increments. The first release (January) has been tested during [University College London \(UCL\)](#) International Testing Week, by UCL students in conjunction with NII students ([Japan's National Institute of Informatics](#)). The findings, briefly presented in the Testing section, have been addressed by the developers and have resulted in the second release (March) of Charityware, which is now stable.

The CharityWare User Manual contains information regarding the use the application (both Web and Android) by the general public, as well as technical installation instructions aimed for future developers. The CharityWare source code can be downloaded from [GitHub](#) and the Web Application can be reached at charityware.cs.ucl.ac.uk.

2 Introduction

2.1 Client

CharityWare consists of a Web-based and an Android-based application and has been developed by University College London MSc Financial Systems Engineering students, under the supervision of Dr. Dean Mohamedally. The source code of the application is scalable, therefore plans are being devised to carry on the project to future generations of students who can build on top of it by adding extra functionality, as required by the business.

The direct client of CharityWare can be any charity who registers to have access to the product. In the final release of the product, upon verification of the registration, charity administrators will have the opportunity to brand their instantiation of the framework using charity-specific logos and themes. Administrators will then be able to create forms needed by their charities and deploy them to be used by their volunteers via the Android app. Once enough data has been gathered, they will be able to generate statistics reports which can then be used to pitch their strategy to the relevant bodies in order to help their charity's cause.

2.2 Overview of the Problem

Charities have always been facing the problem of managing the wealth of data their volunteers are gathering from vast geographical regions. What usually happens is that information regarding the charity's target (for instance homeless children or scarce medication) is kept on paper records that are carried around by the volunteers as they explore their territory of choice. Even if all paper records successfully reached the charity's headquarters, it could be the case that specific records are illegible and thus cannot be used.

Unfortunately, it is often the case that parts of the paper records are lost or damaged by weather, lack of organisation or carelessness. The end result is that very sensitive information is lost and this usually negatively impacts the life of the people in need, from whom the data was initially collected.

Once the information successfully reaches the charity's headquarters, an important amount of time will be spent on organising the records and devising a meaningful analysis of the situation the volunteers encountered in their journey. Because the information is not digitally accessible and thus organised automatically using a computer, a lot of important details can be overlooked by the charity administrators as they need to process a great amount of personal information in the shortest time possible.

Should the charity be specialised on quick response actions (medication needed urgently in a specific part of the globe or helping homeless children during a very cold winter night), the administrators surely do not have much time to spend on such mundane

chores that could be easily and much quicker dealt with by computarising the entire data gathering and data analysing process.

2.3 Proposed Solution

CharityWare is an Open-Source Web and Android-based framework, which help charities with their data collection from around the world. The application is scalable and supports charity-individual branding. Furthermore, once registered with the system, each charity can create their own personalised forms via the web-portal. Volunteers can then login to their specific charity account, gain access to the charity-specific forms and populate them with data gathered in the field, such as name, date of birth, location and, if applicable, they can even upload a photo of the subject they are creating an entry for.

Using CharityWare saves charity administrators an important amount of time, as they can access the forms with the data populated by the volunteers directly through the web portal. Administrators can view the structure of the forms as well as their contents and they can also delete any unnecessary elements. Furthermore, statistics can be generated based on the information inputted by the volunteers so they can have a better view of what things look like in the region.

The framework on which CharityWare has been developed is scalable and supports adding new features easily. CharityWare has been released in two increments. The first one consisted of fair functionality of the website side and minimal functionality on the Android side. The second iteration has been prompted by International Testing Week and the code has been redesigned in order to be uniform and allow future development. The further sections will explain in more detail what the structure of the code is and what tools have been used to devise it.

3 System Requirements

ReqNo	Requirements	Description
1	Website	
1.1	Website Implemented using CMS	
1.1.1	Customise CSS	Css page styled correctly for charity ware
1.1.2	Customise Control	
1.2	Paypal integration	Vistors can pay online through paypal
1.2.1	Charity should provide unique link to their PayPal accunt	Link directly
1.2.2	Dynamic UI Events Add, Edit, View Events Forms Confirm Form Generate Form (includes Save Form) Only admin can edit Login Password Generation Confirmation Recovery Username Registration Recovery Charity Worker Login View forms Donations Populate forms Search	Website should rescale to be displayed on both phones and tablets
1.3	Administration Panel	
1.3.1	Insert New Users Allocate Users Privileges History Password Reset Approve Decline Requests	
1.3.2	Generate Reports	Reports a generated based on access logs and statistics of each charities forms
1.4	REST Web Service	

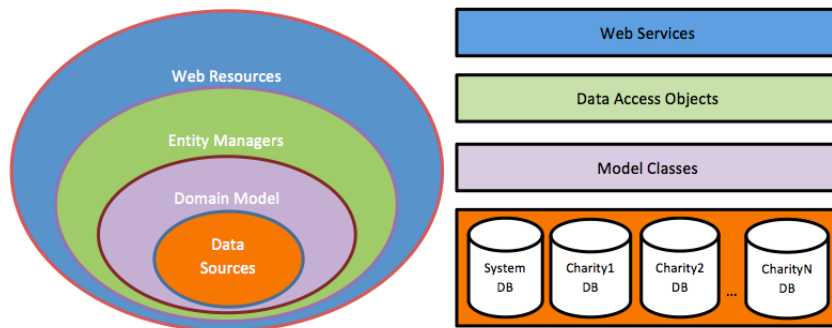
1.4.1	Insert and Select operations	Inserting and selecting data from the DB
2	Android	
2.1	Application accessible from Mobile Phones and Tablets	UI should rescale to be displayed on both Phones and Tablets
2.2	Offline SQLite Integration	Prototype to integrate the system for offline use when online DB not available. When there are not internet connection, allow as much use of the system permissible
2.3	HTML+CSS caching	Cache website for offline use Implement prototype first. Once we have the webservice working, the forms webpages will be cached for offline use
3.1	Database	
3.1.1	SQL Database	
3.1.1.2	User	create all the information relevant to users
3.1.1.3	Privileges	define and enforce user privileges in DB
3.1.1.4	Charity Details	Description of each charity
3.1.2	DB Hosting and DB Connections with Android, Website	Db's are connected to android so the client can access the DB for logging in
3.1.3	DB Schema Design in MySQL	should contain Tables and Relationships
3.1.3.1	DB Schema for the System	Tables for Users, Privileges, Charity Details , Constraints and Relationships, Primary and Foreign Key
3.1.3.2	DB Schema for generic form builder	Maintain flexibility for reporting purposes
3.2	Reporting and Statistics	
3.2.1	Exporting reports statistics to Excel R	data can be exported to either excel or r
3.2.1.1	Integrate Reporting and Statistics API for Java with DB	
3.2.2	Generate SQL according to CMS Customisations	
3.2.3	Create Query Builder Functionality	

4	Integration	
4.1	Web Service Integration	Establish Rest services between website and DB
4.2	Android	Integrate with DB
4.3	Web Site	Integrate with DB
5.1	Testing	
5.1.1	Unit testing and JUnit	creation of tests cases in Junit
5.1.2	Stress testing (JMeter)	Try to test framework under extreme circumstances using JMeter
5.1.2	Android testing	Test application on tables and phones. UI is tested on different device to see if any scaling issues have arised
5.1.3	Selenium testing	Automated testing for the web-site.
5.2	Documentation	
5.2.1	Design Patterns and System Architecture	Document all design patterns used and their impact on the overall architecture of the application
5.2.2	Implementation Decisions	Document all implementation decisions made by the team
5.2.3	Testing Procedures	Document all testing procedures used and draw conclusions based on their results
5.2.4	Testing Coverage	Document testing coverage achieved and draw conclusions regarding the relevance of the testing performed
5.2.5	User Manual	Produce an introduction to using the application. Include all important features and step by step guides
5.2.6	Technical Manual	Produce an extensive documentation on what tools have been used, what versions and how to set up the Dev Env

4 System Architecture and Design

4.1 Service Oriented Architecture

Our Web Services have been built upon 4 fundamental layers, where there is a code hierarchy between them, allowing their information retrieval only if they have been accessed by their neighbour higher layer. This layered information hiding allows to the code components to be encapsulated making them easily maintainable, extendable and as much as possible independent from the others. Additionally, this code hierarchy made the implementation of every Web Service as a single similar coding workflow process giving to the team to scale the system's functionality easily and fast. Furthermore, one other significant similarity across all the layers is that the code has been divided to the 2 main aspects: i) the system data and the ii) charity data.



1. Data Sources

The data source layer represents the permanent storage of the system's data. As our system communicates with it through the use of the Hibernate Framework following the ORM design pattern and making the database operations automatically, it would be insightful to show some core mappings between the data sources layer and the domain model. Additionally, as the system interacts with 2 types of databases (system and charity) it would be beneficial to divide them as following:

4.2 System Database

The system database is responsible to store all the system-related data that are necessary for the system to handle charity registrations, rejections, approvals and charities' monitoring capabilities. Specifically the core tables of this schema are the following:

- [*Table*: Charity, *Class*: system.model.Charity]: The list of all (handled or potentially in the future handled) charity registrations by the system administrators.

- [*Table*: Users, *Class*: system.model.User]: The list of all users that can access the system's WebSite performing the appropriate administration activities for the system or their assigned charity depending on their type and data.
- [*Table*: user_type, *Class*: system.model.UserType]: The available types of the users that can access the system's WebSite. Specifically there are 2: a) Charity Administrator and b) System Administrator.

(a) Charity Database

In every charity registration that is getting approval from the application administrator, it is auto-generated a new Database schema related only with this specific charity. The core tables of this schema are the following:

- [*Table*: field_selection, *Class*: charity.model.FieldSelection]: The available field selections in the existing fields of the charity's forms
- [*Table*: field_type, *Class*: charity.model.FieldType]: The type of the existing fields of the charity's forms
- [*Table*: filled_form, *Class*: charity.model.FilledForm]: The filled data that correspond to the existing charity's forms
- [*Table*: form, *Class*: charity.model.Form]: The existed charity's forms
- [*Table*: form_fields, *Class*: charity.model.FormFields]: The fields of the existing fields of the charity's forms
- [*Table*: form_permissions, *Class*: charity.model.FormPermissions]: The available user permissions in the existed charity's forms
- [*Table*: form_type, *Class*: charity.model.FormType]: The types of the existed charity's forms
- [*Table*: user_type, *Class*: charity.model.UserType]: The types of the existed charity's users
- [*Table*: Users, *Class*: charity.model.User]: The existed (active and non-active) users of the particular charity

It would be also important to be mentioned the fact that all of these tables have the corresponding automated-generated tables (with the extension _ver) for logging purposes of the particular database operations that have been performed on them.

4.3 Domain Model

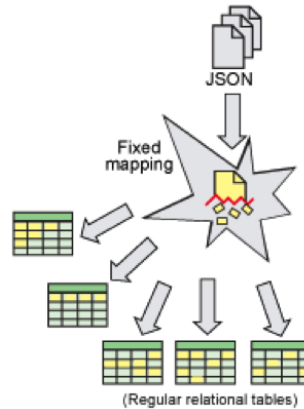
The domain model is consisted of all the POJO Beans that represent the application's data in our application domain. Their usage is double:

(a) Application Data Modelling

The model classes are the responsible medium and representation of the data that are stored in the system's databases. In essence, they provide an object-oriented modelling of the underlying Entity-Relationship diagrams that exist in the charity and system schemas. The main reason for this data modelling transformation was the necessity to develop Web Resources that they will be smoothly bound with the system's back-end that in our case is 2 separate database schema types.

(b) Message Data Exchange

In the Web Services invocation process, the model classes have been used in a significant degree as the main generic context that could be sent from the server-side. The main reason that the project team decided to use the model classes as generic context inside of the transmission messages (JSON) was their easy and direct handling by the client-side code teams (Android, WebSite) just by providing to them the appropriate jar file with the corresponding compiled model classes.



4.4 Entity Managers

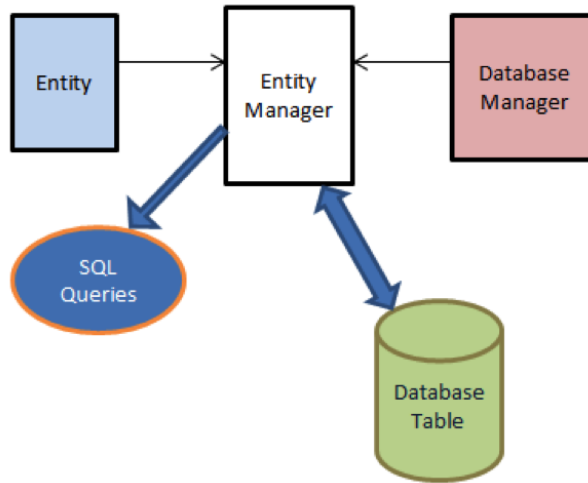
The Entity Managers form the particular data access objects that take access to the corresponding data resources. It consists of this particular part of code that is

responsible for the fundamental database operations (select, insert, delete, update) with the back-end system. The team implemented this part of software taking seriously into consideration that the project in future will need either database or domain modelling changes. For that particular reason, the Entity Manager for each permanent stored domain Entity formed with the same methodology. Specifically, each Entity Manager is consisted of 2 fundamental parts:

- Entity (*Packages* : system.model, charity.model)
The entity gives to the EntityManager all the particular type of data that is intended to handle in its interactions with the corresponding database table in the back-end.
- Database Manager (*Package* : shared.dao.util)
The Database Manager supports the EntityManager with all the fundamental software stuff that is responsible for the Database Session Management and the Transaction Management. It also supports the EntityManager with the generation of simple and abstract DB queries such as simple select, update and insert statements.

and provides 2 main operations:

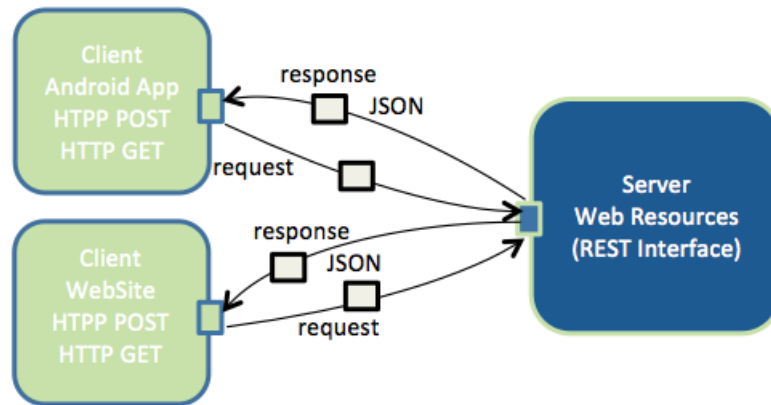
- SQL Queries: Customized to the Entity SQL queries that give to the Entity-Manager the appropriate SQL stuff in order to communicate effectively with the Back-end.
- Database Interaction: Invocation of the particular database operation (data retrieval, insertion, update) providing the required communication channel between the application's middleware and the back-end system.



4.5 Web Resources

The Web Resources form the Server's external communication interface to the existing (and the potential future) Web Clients providing, the whole management process of the permanently stored data. Specifically, the Web Resources that are implemented with the REST design pattern are available to get accessed via the appropriate HTTP Request methods such as GET and POST. Additionally, the main type of the exchanged messages that are sent as response from the implemented Web Services is the JSON. JSON technology was selected due to the following 2 significant factors:

- (a) The client-side code could efficiently and easily parse this type of messages using existing Frameworks such as jQuery to retrieve the JSON objects from the website and GSON for android application.
- (b) The JSON technology can bind very easily in its generated messages any type of Object (either primitive or user defined) with the existing implemented libraries. This feature makes the Server's interaction with the clients much more flexible and scalable for future amendments and functional change requests.



4.6 Design Patterns

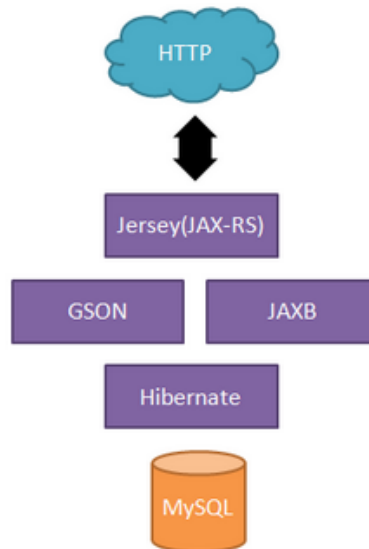
The main design patterns that the project team followed in order to implement the Web Services are the following:

4.6.1 REST

To provide access to the database information it was decided that the data should be made available via RESTful services (Representational State Transfer). To do this we settled upon a number of API's that would provide us with some key features, these were:

- Object Relational Mapping(ORM) - where we selected to use the Hibernate API, this allowed us to directly map our relational database schema into Java Objects making data access from Java more intuitive to the developer.
- Jersey - Provided us with an API for the JAX-RS reference implementation for hosting RESTful services, allowing us to easily capture and parse URL's and HTTP headers into Java.
- JAXB - Provided us with the capability to automatically parse JSON objects that were received via the HTTP protocol and captured by Jersey straight into the corresponding Java objects. This was useful as we could apply the JAXB notation to the same objects that were being used by Hibernate allowing us to parse them straight into the final objects we required. (The creation of a Form is probably the best example of this)
- GSON - GSON was also required to parse Google's JSON format inside of Java.

The below image shows how the technology stack of API's is roughly used to go from a object being passed or sent via HTTP through to the database. A Huge benefit of using A RESTful approach is that it is platform agnostic when it comes to the data it is receiving via HTTP as long as it is in the correct format.



4.6.2 DAO

Data Access Object (DAO) is an object that provides an abstract interface to the system's persistence mechanism – that is a MySQL database. By mapping application calls to the persistence layer, DAOs provide some specific data operations without exposing details of the database. This isolation separates the concerns of what data accesses the application needs, in terms of domain-specific objects and data types (the public interface of every Entity Manager), and how these needs can be satisfied with a specific DBMS, database schema, etc. (the implementation of every Entity Manager with the assistance of the Hibernate Framework).

The main advantage of using data access objects is the relatively simple and rigorous separation between two important parts of an application that can and should know almost nothing of each other, and which can be expected to evolve frequently and independently. Changing business logic can rely on the same DAO interface, while changes to persistence logic do not affect DAO clients as long as the interface remains correctly implemented.

Briefly, the rest advanced features that Data Access Objects (Entity Managers) provides in the system are the following:

- They can be used in a large percentage of the Web Services implementation - anywhere data management is required.
- They hide all details of data storage from the Web Services implementation.
- They act as an intermediary between the Web Resources and the back-end system. They move data back and forth between objects and database records.
- They allow ripple effects from possible changes to the persistence mechanism (MySQL database) to be confined to a specific area.

4.6.3 Entity Abstraction

Entity Abstraction is a design pattern, applied within the service-orientation software design which provides guidelines for designing reusable services whose functional contexts are based on business entities (Entity classes in system's domain model). The Entity Abstraction pattern advocates that logic that relates to the processing of business entities

- is separated from the process-specific single purpose logic
- is designed as independent logic that has no knowledge of the overall business process in which such logic is being utilized

Specifically, the Entity classes (domain model packages) contain functionality, including the CRUD functions (Create – Read – Update - Delete), that is only relevant to the physical or logical business entities (system's or charity's data), represented by their corresponding functional context. Apart from the identification of different actions, the Entity classes contain all the particular relationships between the business entities whether part of the current business process or not. By looking across all the different business processes that our system provides, the Entity classes can be packed with extra data/functionality that may be required by some of them. An important source of information for identifying such relationships are the Entity Relationship Diagrams that have constructed by the Database team, as they physically display the relationships between different tables and also identify the different attributes of entities that form the basis of the relationships among the tables. This kind of relationships between the business entities has been implemented with the assistance of the Hibernate Framework using appropriately the property mappings in the construction of the hbm.xml files.

4.7 WebSite

4.7.1 System Design

1. Java Server Page Code

The JSP code was used to provide us with some basic features, the first and most obvious is dynamic web content based on data, parameters etc. In addition to this

it was used to provide each user with a persistent session after login. The session variables are used to hold various state data about the active session including login and other meta data about the logged in user.

Although covered in more detail in the Web service section of this document it is worth noting that the JSP accesses the REST services in much the same way as the client side code. This is the JSP performs a HTTP request to the given URL of the service and parses the response. The difference between the JSP code and the javascript is that the JSP parses the returned JSON into Java object for ease of use in the code.

2. **Client Code** (HTML/CSS for the User Interface, JavaScript handling users' requests)

The client side is formed of HTML/CSS and JavaScript, depending on the individual that created a given section dictates the use of the JQuery Library or straight Javascript. Some dynamic content of the site is achieved through clever use of AJAX requests to provide dynamically updating content to the page. This involves performing the appropriate POST/UPDATE/DELETE request to a REST service before reloading a given element/s to the page. The invocation of the web services are best done using the jquery library, this is due to it providing some easy methods for constructing the HTML headers and handling the returned JSON objects.

<http://api.jquery.com/jquery.post/>
<http://api.jquery.com/jquery.get/>
<http://api.jquery.com/jquery.ajax/>

The structure of these requests should match the Java objects that they are going to be automatically parsed into. For example one of the best ways to construct the correct JSON object to be passed to a web service is to first get the service to output the java object in JSON format allowing the inspection using a browser Dev tool such as Firefox Firebug. The attributes returned by the service can be accessed using the usual object dot notation.

4.7.2 Code Structure

The main design pattern in use with the JSP code is MVC (Model View Controller) which is explained in more detail in the final section. This is a reasonably obvious choice when creating dynamic web pages in Java being able to use the object orientation of Java to form the Model then the client server model of JSP with HTML and CSS to form the controller and view.

4.7.3 Design Patterns

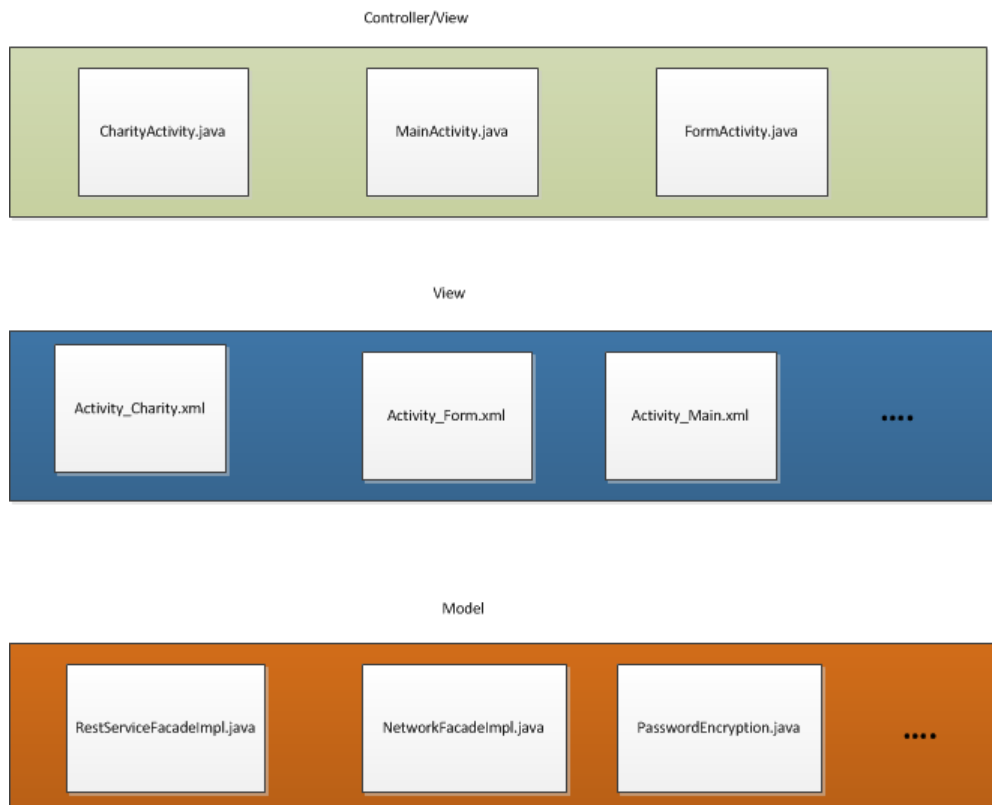
1. MVC

The **Model** is based of a number of domain model class that abstract away from all the functional code to produce a simple java object with a hint of domain logic, that is structured using constructors, getters and setters for the respective data attributes.

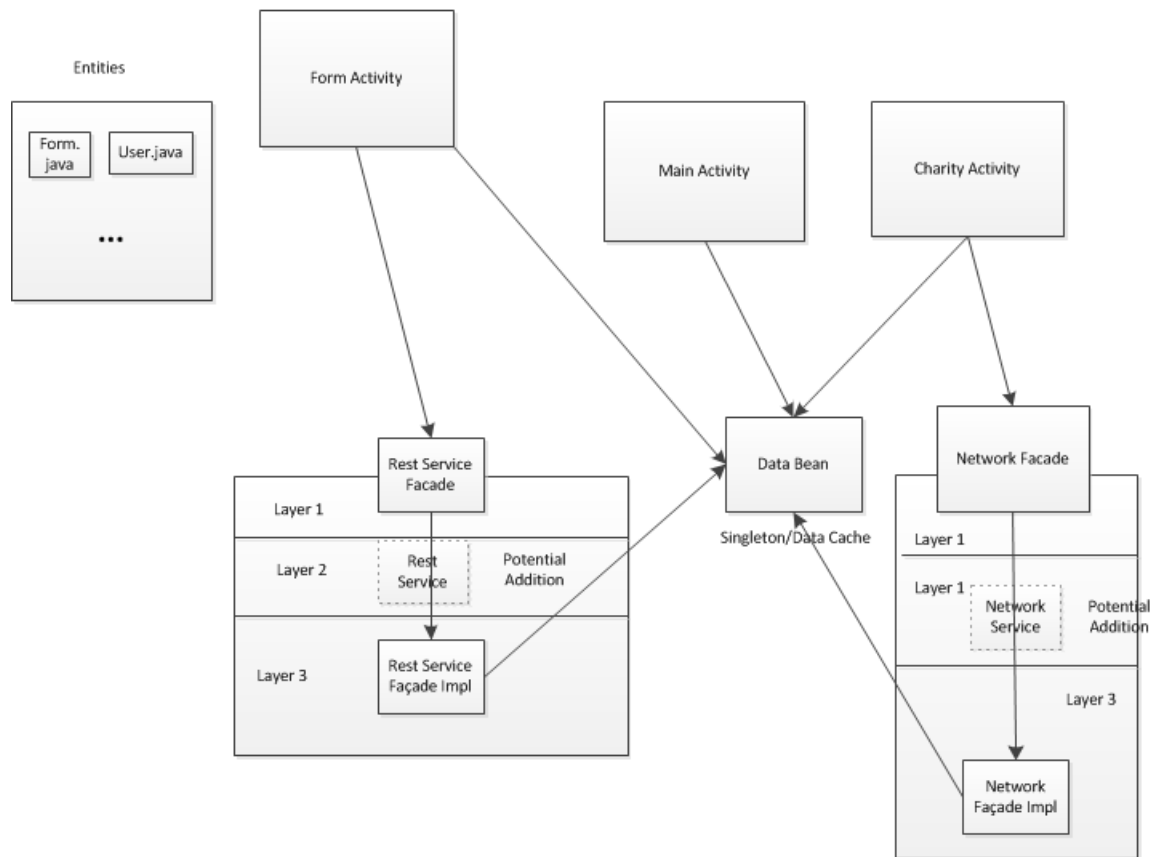
The **Controller** is based of a number of web servlets with minimal functionality, that's to request procedure from to the business logic layer and send a response back to the JSP pages. Therefore the controller is simply utilised to wire the communication between the functional business logic and the almost static JSP pages.

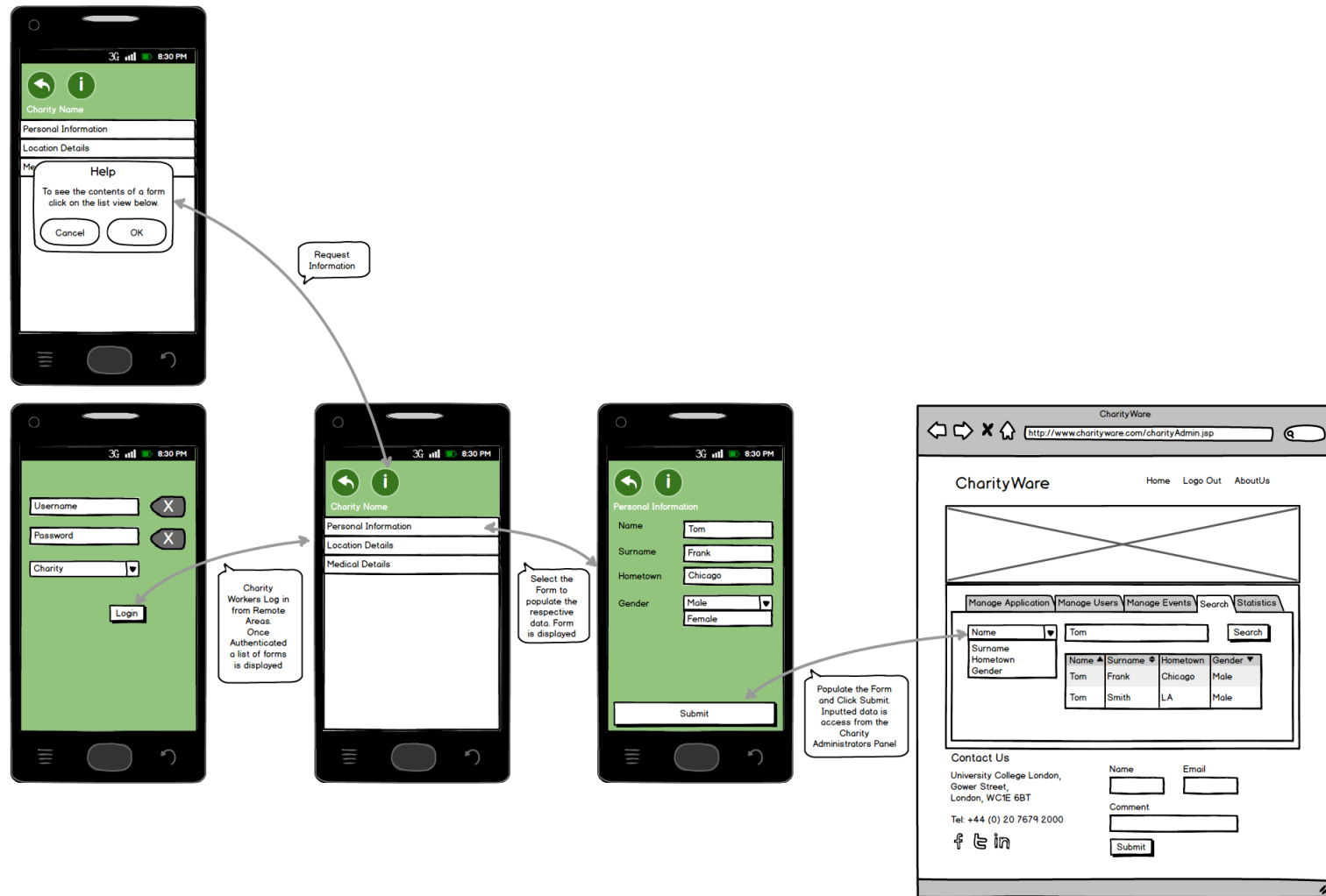
The **View** is composed of JSP pages that encapsulate the response from the business layer through the controller using HTML5, CSS3 and Javascript with the use of jQuery. With the use of JSP and Javascript we are able to iterate through the JSON object returned by the controller and display data within the presentation layer.

4.8 Android



A snapshot of the Android app's architecture





4.9 System Design

The diagram below depicts the high level design of CharityWare application, using wireframes/mockups.

The web application handles three aspects:

Prospective Charities View

Prospective Charities have the ability to view information about the framework, find out how they can register their interest and also register their interest to use the CharityWare Framework. By registering their interest the system's administrator is notified. The system administrator checks and verifies that the charity is legitimate and is granted or denied access to the framework according to the verification results. Once granted access the charity administrator will gain permission to create an android application.

Accessible Pages: default.jsp, info.jsp, login.jsp, register.jsp and registerHelp.jsp

Registered Charities View

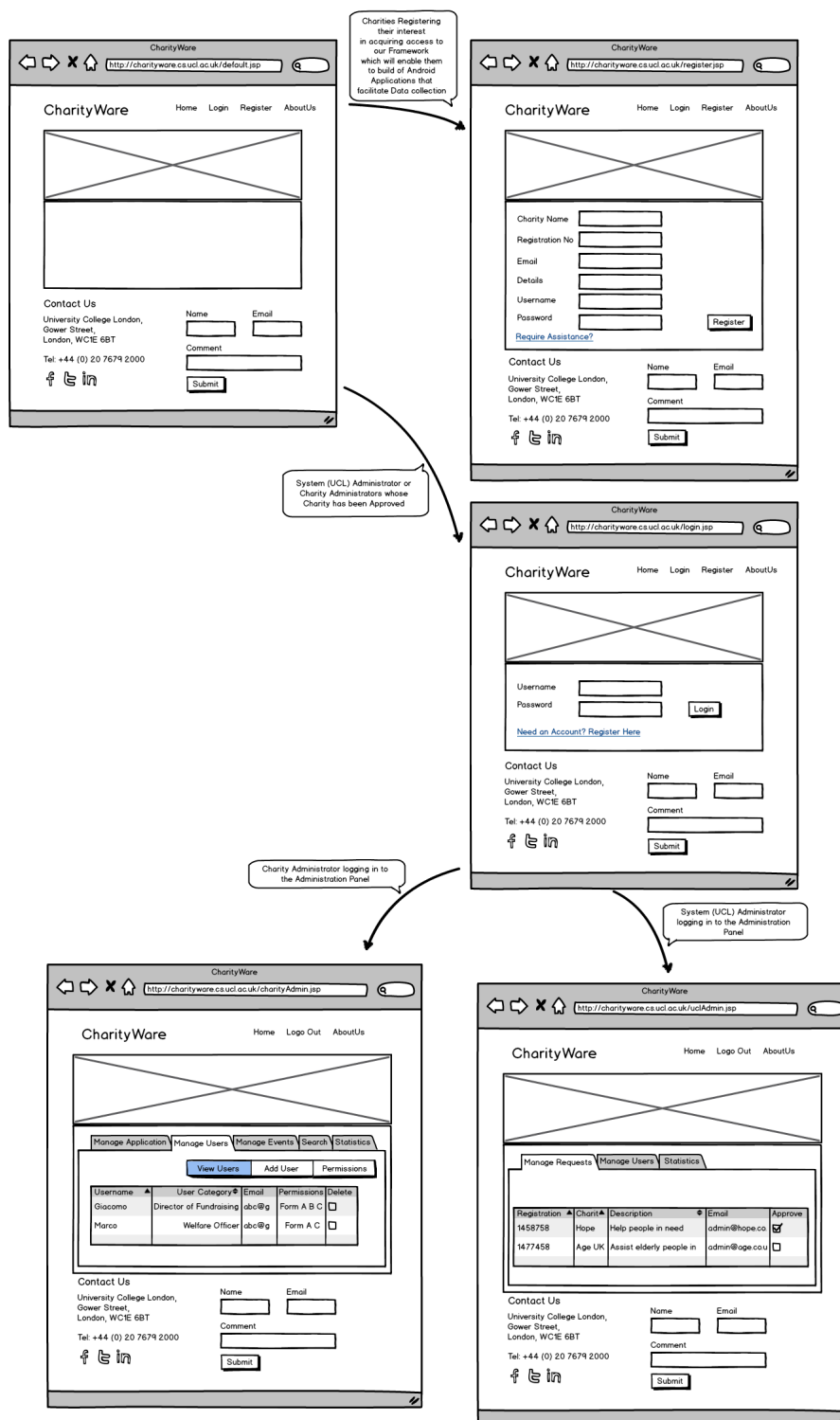
Once a charity has been verified by the system administrator, the charity administrator will be granted access to the Framework. The charity administrator has the functionality to customise the android application by generating of a number of different forms of varying types such as input, search, events or donation forms. The forms will be published on an android application provided to charity's workers in order to facilitate data collection from remote areas. The charity administration panel will also provide additional functionality such as data analytics, search and user access control functionality.

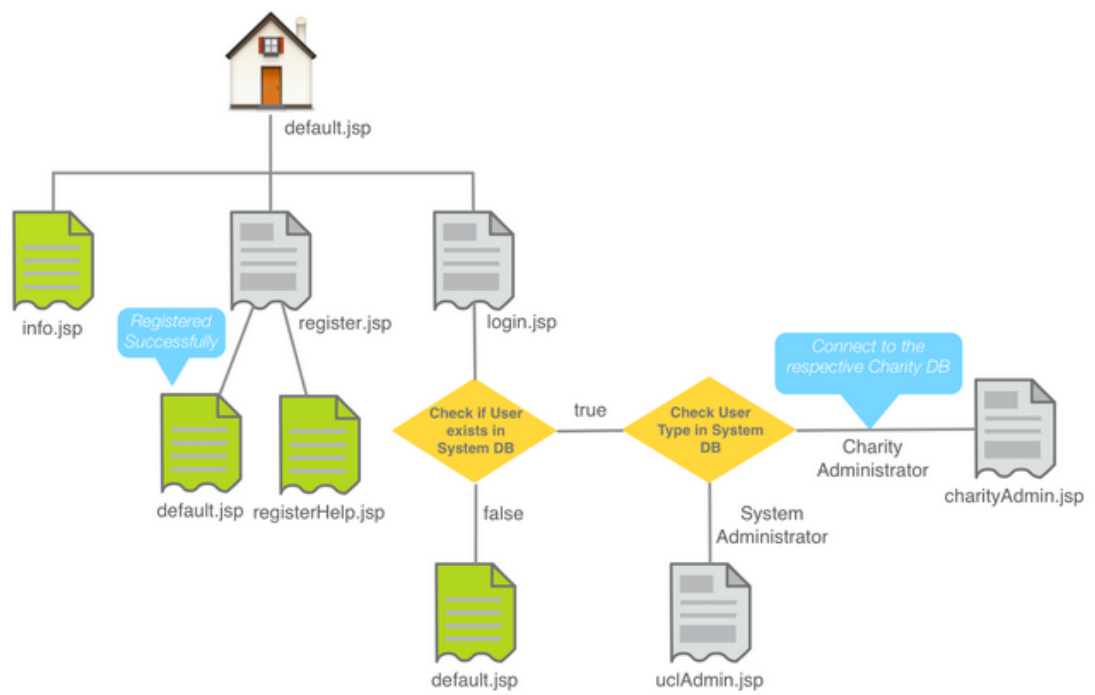
Accessible Pages: default.jsp, info.jsp, login.jsp, charityAdmin.jsp

System Administrator View

The system administrator is responsible for monitoring the charity registrations and verifying charities based on the details provided. The system administration panel will also provide additional functionality such as data analytics and user management.

Accessible Pages: default.jsp, info.jsp, login.jsp, uclAdmin.jsp





5 System Development

5.1 Web

During the system implementation phase, the team took into consideration some additional factors apart from the system design, in order to construct a system that could be assumed as a state-of-the-art software project. These factors are the following:

1. Use of Design Patterns

Design patterns play a significant role in the success of a software project as it can drive thoroughly the development team to write code with the most appropriate and state-of-the-art way. It is not only crucial in the easy and rapid development of a new software project but play a key role for its general maintenance, implementation of future potential enhancements and general adaptability to future requirements change. For these reasons and as the server-side code is one the most important parts of this project being in the first position of potential changes/enhancements in the future, the development team tried to research exhaustively all the possible candidate design patterns that could be followed in this particular software solution. The use of the selected design patterns (that they will be described in the next section) has influenced significantly the way that the code has been structured as the project's software has followed specific software development conventions and methodologies.

2. Use of existing Frameworks and creation of Black Boxes

In order to construct as easier, fast and scalable as possible the domain classes of our project linking them with the existed relational database schemas, the development team decided to use the Hibernate Framework. One great advantage using this Framework is the fact that we can provide a DB-independent mapping between our domain model and the back-end. With this way our system could operate in the future across any future Database platform change modifying only the Hibernate Dialect. Hibernate is an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. Hibernate solves object-relational impedance mismatch problems by replacing direct persistence-related database accesses with high-level object handling functions. Additionally, it helps significantly developers to utilize POJO-style domain models in their applications in ways extending well beyond Object/Relational Mapping. More information about this Framework can be found in its official [website](#).

As the model classes and the Web Clients were implemented in the Server side development teams (cooperation between the Web Services and Database teams),

it was incredibly valuable to provide a state-of-the-art interface to the client teams (Android, WebSite) for the appropriate Web Services calling and handling. For this particular reason, beyond the fundamental implementation of the RESTful Web Services API publishing the corresponding HTTP paths, we distributed a .jar file with the compiled classes of the domain model and some particular Web Clients objects. Creating with this way a black box for the client teams, the invocation process of the implemented Web Services was become simple, scalable, maintainable and clearly independent from the client application domain.

3. Code Factors

Significant code factors that have been taken into consideration as our software solution is based in state-of-the-art object oriented software development concepts are the following:

(a) Creation of a generic software solution

One of the most significant factors that the project's team took into consideration was the mission to create a software solution for charities that can be easily adapted to new aspects in terms of system requirements engineering and software development. For that particular reason the developed software not only provides the required functionality but can be easily used as a software Library (.war or .jar file) from new software development teams being a black box as itself.

(b) Scalability

The software implemented with such a way in order to scalable to many charities in terms of data management and client application development. Specifically, as any new charity that is getting approval of its registration, is mapped to its separate database schema, its data and session management is not entangled with the database schemas of the other registered charities. This feature that provides data and session isolation makes our software product to be scalable to any amount of stored charities' data.

(c) Maintenance

The code has been implemented with the view of the future need of modification on it due to the correction of existed faults, performance improvements or functionality enhancements. For this particular reason, the project team gave significant attention to the creation of logical boundaries between the components and building the appropriate interfaces for in the interconnection of the separate code fragments (Java packages).

(d) Modularity

The functionalities of the Web Services have been divided into a number of components separating logically the code based on what it is intended to perform. Great example of this motion is the separation of the classes across all the layers of the system (aforementioned layers in Section A) by the code's relativity to the system's or charity's data manipulation. This can be shown emphatically by the naming of project's Java packages where their name starts with the word either `charity.*` or `system.*`.

(e) Cohesion

The functionalities that can be retrieved from every class in the system's packages are strongly related with the data that are encapsulated in this particular class. Great examples of this motion are the classes of the domain model and the Entity Managers. Specifically, the methods that exist in every domain model class are related in a very high degree only with constructors, getters and setters of its included data attributes and the methods of every Entity Manager class are related only with the particular database operations of the corresponding Entity.

(f) Coupling

The software components have been built upon the strong perspective of low code coupling. For this reason, the code packages have been developed with the view that potential changes in one module usually will not force ripple effects of changes in other modules. Specifically, they developed with a way where the code is either inter-dependent between packages along with their data inter-dependence or strongly coupled only between classes of the same package without disrupting other external classes. This has been achieved either by making a flow of code in terms of package inter-dependence or by defining the appropriate code interfaces in terms of class inter-dependence.

(g) Extensibility

The software product implemented with such a way in order to be easily extendable either on its core functionalities or by scaling on new client-side applications. This type of scaling could be done either by creating new apps for other mobile OS (such iOS, Symbian, Blackberry, etc) or by creating new UI applications tailored for specific charities that they have been already registered to the CharityWare.

4. Code Appendix

Apart from the general description and outline of the project's code structure, in order to provide a more-in-depth insight of the code implementation to the fu-

ture developers of this application, the team created a particular mapping of the code with the aforementioned system's technical aspects that were analysed.

(a) Model Classes

The model classes have been implemented in the following code packages:

- `charity.model`
- `shared.model.util`
- `system.model`

(b) Entity Managers

The entity managers have been implemented in the following code packages:

- `charity.dao.hbm`
- `charity.dao.managers`
- `shared.dao.util`
- `system.dao.hbm`
- `system.dao.managers`

(c) Web Resources

The Web Services have been implemented in the following code packages:

- `charity.services`
- `system.services`

(d) Web Clients

The Web Clients have been implemented in the following code packages:

- `charity.clients`
- `shared.clients.util`
- `system.clients`

5.2 Android

5.2.1 Code Structure, Paradigms & Methodology

The Android application's main structure was necessitated by the established standards and limitations of the Android SDK. As such the code structure follows firmly the MVC(Model-View-Controller). In particular, all view-related elements corresponding to each of the application's activities are found under the `res/layout`, the object

model lies under the `env/entities` directory and the controller files/activities reside under `com/webviewprototype`. It must be noted that although the MVC pattern is mandated by the mechanics of the SDK, there can be considerable overlap of view and controller functionality is concerned, particularly as far as dynamic creation and adjustment of visual elements is concerned. As such, a portion of front-end functionality can be found under relevant methods in the controller files/activities.

Apart from the standard Android principle of MVC, several conscious decisions by means of design pattern implementation in order to facilitate information/data flow and to minimize the complexity of the working code. Furthermore, several design decisions were driven not only by the need to conform to the baseline requirements of the application but also to pave the way for future amendments and enhancements by creating an extensible and modifiable interface framework.

The Android app for Charityware is required to request, format, display and post-process data on demand to the user. In order to minimize unnecessary database access overheads, a static data cache (or bean as it is called in our app) was created to store form/charity data retrieved via the SOA framework. This design decision, apart from conforming to sound programming practices, was also precipitated by the app's intended use. Charity workers are expected to swiftly retrieve forms outdoors and re-use as much data as possible without having to re-load or forcing busy pedestrians to wait due to lagging issues. Moreover, the conscious design choice to not incorporate local storage functionality, was also a motivating factor for the consolidation and upgrade in significance of the bean, since in case of internet outages the Charity worker will be able to navigate across forms and their fields without having to go through layers of loading phases.

The `bean(DataBean.java under env.Entities)` was implemented using the Singleton design pattern. Form-specific information, such as form names, form and filled field content are stored there among other relevant data requiring restricted global access.

Early in the development phase of the app, we decided to create a set of layers to encapsulate interface specification and functionality in discrete separate levels, provided by the **Façade** design pattern. The adherence to this prominent design paradigm can be attributed to the following factors:

1. The **Façade** pattern provides a single unified point of access to the interface methods. Since the scope of the Android app is to become increasingly complex over time, the **Façade** pattern simplifies the complexity and handling of separate functionality interfaces and associated sub-systems. Thus the number of dependencies is reduced by forcing all callers to instantiate the Façade interface. For example, in our case there are multiple asynchronous tasks, triggered during execution that may be called by multiple sources. To eliminate the chance of conflicts and

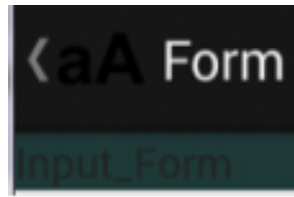
to enhance transparency and tractability, the `AsyncTasks` were divided into functionality families(`Network Polling/REST service call`) and inserted into a corresponding `Facade` implemented interface(`NetworkFacadeImpl.java`, `RestServiceFacadeImpl.java`) . Within the subsystem, further subsystems can be defined (e.g. the existing `RestServiceFacadeImpl.java` could be further split into implementation classes).

2. Due to the nature of incremental development, there was the need to make consistent changes on the classes interacting with the REST services without affecting the client classes. Since the chosen pattern reduces the coupling between calling and called classes, any intended changes on post-processing and data allocation could be carried out without causing unintended problems.
3. Although not explicitly part of the pattern's guidelines, the `Facade` pattern lends itself to multiple layering and associated levels of abstraction, mirroring in part the approach adopted by the REST/SOA framework architecture for the Charityware website. Although, this was not implemented in the current build of the Android app, additional layers(such as a dedicated `Service` layer between the `Processing` layer and the `Abstract` layer) would enable the app to be effortlessly integrated into an envisioned broader family of Charity Android apps. As such, we have ensured that most of the existing functionality are portable and deployable across related projects with minimal alterations.
4. The `Facade` pattern adheres to the imperative of OO-programming for extensibility and flexibility. Since related functionality can be mapped to a specific interface in the `Facade`, one can add additional `Facades` for local ORM storage, an extension of the Android app for future consideration.

Finally, another design pattern employed was the `Adapter` pattern. Its use was enforced by the way Android populates list-like elements (e.g. `SimpleAdapter`) and was heavily modified to compensate for the programmatic generation of visual elements for each row according to the input type of each field of the selected form. `FormListAdapter.java` adapts the formatted data received by the `façade` interface as input to determine which custom row type to instantiate (`Calendar`, `TextBox`, `Spinner` etc.). When a match is identified a custom data/view element container(e.g. `AndroidDropDownField.java`) of the list row is used to accommodate for the unique characteristics of the input type and then a suitable view `FXML` file(e.g. `.list_view_dropdown_layout.fxml`) is used to wrap the container.

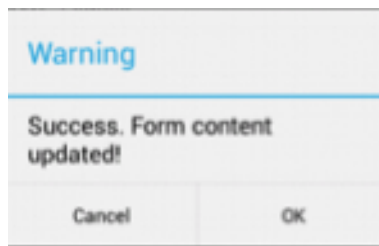
As far as UI code is concerned, Nielsen's design principles exerted significant influence. UI design and underlying code conformed to industry standards and related applications (Standards and Consistency) and employed a minimalist aesthetic to prevent irrelevant visual elements from diminishing the visibility of critical information by diverting the user's locus of attention (Aesthetic and Minimalist Design). Presented text is expressed

in natural language, eschewing technical terminologies, with error messages being presented in an understandable, user-friendly manner (Match between the System and the Real World). The Recognition rather than Recall heuristic is fulfilled by displaying all the necessary functionality at each screen. For example, the charity's name whose forms are listed in the Form screen is displayed underneath the Form title and similarly the form title is visible in the Form Field screen so that the user won't have to memorize which charity/field he/she has previously selected. As far as Error Prevention is concerned, all input fields are supported by validation mechanisms, preventing users from entering illegal input.

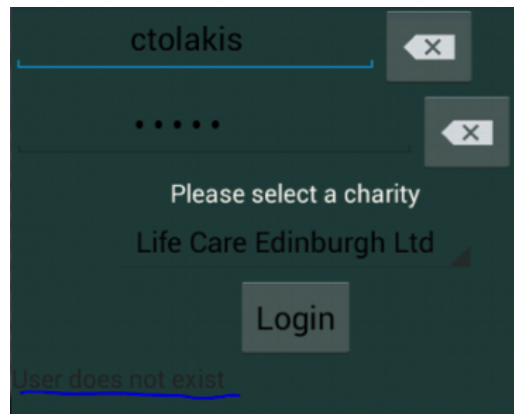


Recognition rather than Recall

The application caters to mostly casual users of technology and in conjunction with the minimalist design approach, no immediate need for the provision of accelerators was diagnosed (*Flexibility and Efficiency of Use*). *Visibility of System Status* was partially followed, in the sense that success and error messages are displayed to users. Progress bars associated to loading could not be properly implemented due to the fact that the data was transmitted as a batch from the REST framework and that an AsyncTask cannot interact with the main display thread of Android.

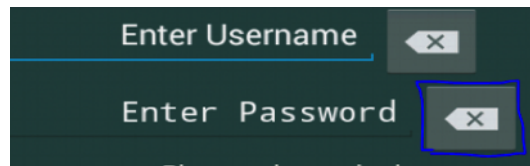


Visibility of System Status

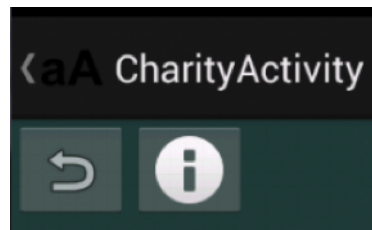


Visibility of System Status/Match Between the System and the Real World

User Control and Flexibility relates to the ability of the user to revert his/her actions by providing revert/back and canceling buttons. The aim of the heuristics is to prevent the user from navigating to an unwanted state by mistake. In our app, this is represented by the presence of back/refresh and canceling buttons as evidenced by the figures below.



User Control and Freedom



User Control and Freedom

5.2.2 Functionality

The functionality offered by the Android app presently is:

1. Charity worker log-in for multiple charities he/she may be enrolled with.
2. Real-time connection polling.
3. Local password hashing for verification purposes.
4. Retrieval and listing of forms associated with the selected charity.

5. Form Refreshing/Re-loading functionality.
6. Form Field generation.
7. Form Field filling.
8. Form Field submission.

As mentioned before, the originally planned local storage functionality in the device was not incorporated in the current build. Storing retrieved forms and form fields was envisioned as a way to enable the worker to continue filling forms whilst off-line. However, due to the performance overheads inherent in keeping a large volume of data in the mobile device and the express requirement that the application should be light-weight and easy to use, the feature never left its planning stages. Nevertheless, sufficient preparatory work was done, by implementing the network polling routine and deciding that ORMLite is the ideal candidate for an ORM mobile-friendly framework. Given the extensible nature of the **Façade** framework, only a couple of additional classes need be added without introducing additional dependencies.

Potential enhancements include:

1. Local storage for forms and fields, courtesy of SQLite and ORMLite. The transition to using local storage instead of REST content will be determined by the output of the polling routine.
2. Form statistics and evaluation to help charity workers make informed decisions as to the entropy of information received by individuals belonging to oft queried population groups.

5.3 Database Schemas

Assumption: A charity has only one charity administrator by default, whose details are stored in the User table in the System Database.

To provide a solid backbone to the Charity Framework, a scalable design that enables the required flexibility yet does not compromise the integrity and consistency has been adopted for the database schemas.

Two types of schemas have been designed for different yet complementary purposes:

- **System Schema**

This schema was designed to handle all the transactions directly related to the Charity framework, including Charity registration details, Framework Users and feedback related to the framework.

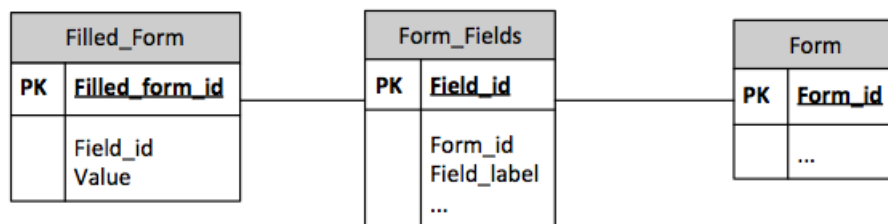
- **Charity Schema**

To handle the requirements of the Charities mainly, to have an individual schema for every charity and to be able to generate dynamic forms this schema was designed. The creation of this schema is based on whether a charity has been verified.

5.3.1 Design Approach

A relational model has been adopted for the System Database, where attributes have been normalised to minimise any data redundancy and dependency. This decision has been taken to enforce consistency.

As for the Charity Database a combination of Entity Attribute Value Model in conjunction with a Relational Model was adopted. The entity attribute value approach is a simple design that facilitates the storage of dynamic fields specified by charity administrators to represent the data they wish to capture. The table that is modelled on this approach is called Filled_Form, as can be depicted from the Figure below the table captures a map of all the field's values being submitted from the android application using the Field_Id and Value. The rest of the tables within this schema make use of a normalised relational model approach.



5.3.2 Retaining Data

In order to handle data from being deleted from either of the schemas, an isActive field has been added where appropriate and is set to 0 if the entry is no longer required. As for fields within forms that are no longer required to capture data the isActive for that respective field is set to 0 whilst the data that has already been captured for the respective fields will still be available for retrieval.

5.3.3 Object-Oriented Domain Model

The Hibernate library was used to develop the data object model layer for both database schemas, this facilitates the development of transparent persistent classes using object-oriented programming model. Hibernate Envers also facilitated the implementation of auditing of persistence classes which replicate the schema's relational model. Similar to

subversion each database table is replicated using the following syntax `TableName_VER`, and uses the concept of record versions by logging data for each revision using a revision entity.

5.3.4 Schema and Hibernate Configuration File Generation

Since every charity requires its own schema upon verification of a charity by the System Administrator, a schema for the respective charity is generated (Naming convention: the word “Charity” concatenated with the `Charity_ID`). This is carried out by creating a RESTful web service that triggers the execution of the `spSchemaGeneration(DBName)` using HQL. Along with the database schema, the hibernate configuration file for the specified database schema is also generated.

5.3.5 Data Dictionary

Event				
Field	Type	Allow Null	Default Value	Description
<u>Event_Id</u>	int(11)	No		PK: Autogenerated
Event_Name	varchar(255)	No		Stores the Name of the Event
Event_Description	text	No		Stores the Description of the Event
Event_Location	varchar(255)	No		Stores the Location of the Event
Event_Date	date	No		Stores the Date of the Event
Event_Time	time	No		Stores the Time of the Event
User_Id	int(11)	No		User who Created the Event

Field_Selection				
Field	Type	Allow Null	Default Value	Description
<u>Field_Selection_Id</u>	int(11)	No		PK: Autogenerated
Field_Id	int(11)	No		The Field that offers a selection
Field_Selection_Value	varchar(255)	No		Stores the selection values

Field_Type				
Field	Type	Allow Null	Default Value	Description
<u>Field_Type_Id</u>	int(11)	No		PK: Autogenerated
Field_Type	varchar(100)	Yes		Stores the values of the Field_Types
Field_DataType	varchar(100)	Yes		Stores the data type of the Field_Types
Field_Description	varchar(200)	Yes		Stores the Description of the Field Type
isActive	tinyint(1)	No	1	Stores whether the Field Type is used

Form				
Field	Type	Allow Null	Default Value	Description
<u>Form_Id</u>	int(11)	No		PK: Autogenerated
Form_Type_Id	int(11)	Yes		FK: Form Type being assigned
Form_Name	varchar(100)	Yes		Stores the Form Name
Date_Created	datetime	No	01/01/1970 00:00	Stores the Date when the form was created
URL	varchar(250)	Yes		This field is only required if there are corresponding files. Currently not used.
isActive	tinyint(1)	No	1	Stores whether the Form is used

Form_Permissions				
Field	Type	Allow Null	Default Value	Description
<u>Form_Id</u>	int(11)	No		Composite Key
<u>User_Type_Id</u>	int(11)	No		Composite Key
Permission	varchar(100)	Yes		Stores a description of the Permission
isActive	bit(1)	Yes		Stores whether the Form Permission is used

Form_Type				
Stores the type of forms that can be added to the Android application				
Field	Type	Allow Null	Default Value	Description
<u>Form_Type_Id</u>	int(11)	No		PK: Autogenerated
Form_Type	varchar(100)	Yes		Stores the Form_Type Name
isActive	tinyint(4)	No	1	Stores whether the Form Type is used

Mailing_Group				
Stores the Mailing Groups Names				
Field	Type	Allow Null	Default Value	Description
<u>Mailing_Group_Id</u>	int(11)	No		PK: Autogenerated
Mailing_Group	varchar(255)	No		Stores a description of the Mailing Group

Mailing_List				
A linking table that stores the Mailing Groups the User is assigned to				
Field	Type	Allow Null	Default Value	Description
<u>Mailing_Group_Id</u>	int(11)	No		Composite Key
<u>User_Id</u>	int(11)	No		Composite Key

User_Type		Stores the different User Types a User can be part of		
Field	Type	Allow Null	Default Value	Description
<u>User_Type_Id</u>	int(11)	No		PK: Autogenerated
User_Type	varchar(100)	Yes		Stores the User_Type Name
User_Type_Description	varchar(200)	Yes		Stores the User_Type Description
isActive	tinyint(1)	No	1	Stores whether the User Type is used

Users		Stores data related to the user including credentials. This Table is used for Login from the Android Application		
Field	Type	Allow Null	Default Value	Description
<u>User_Id</u>	int(11)	No		PK: Autogenerated
Username	varchar(20)	No		Stores the User's Username
User_Type_Id	int(11)	No		FK: User Type being assigned
User_Password	varchar(250)	No		Stores the User's Encrypted Password
Salt	varchar(10)	No		Stores the User Password's Salt for Encryption
User_Email	varchar(250)	Yes		Stores the User's Email
Date_Created	timestamp	No	CURRENT_TIMESTAMP	Stores the Date the User was created
isActive	tinyint(1)	No	1	Stores whether the User is used

access_log		Stores Data in relation to the User Access to the System. Android functionality not yet created		
Field	Type	Allow Null	Default Value	Description
<u>Access_Log_Id</u>	int(11)	No		PK: Autogenerated
Form_Id	int(11)	No		FK: Form Accessed
Field_Id	int(11)	Yes		FK: Field Accessed
User_Id	int(11)	No		FK: User Accessing the Field & Form
Access_Start_Time	datetime	Yes	01/01/1970 00:00	Time Started Accessing
Access_End_Time	datetime	Yes	01/01/1970 00:00	Time Stopped Accessing
Device	varchar(100)	Yes		Stores the IP Address of the Device it is being used from
Location	varchar(100)	Yes		Location from where the Application is being accessed
isOnline	tinyint(4)	No	0	Is the User currently using the App
access_start_date	date	Yes		Last time User started accessing the App
access_end_date	date	Yes		Last time User stopped accessing the App

filled_form		Stores the Data that has been populated through the Android Application in the respective form_fields. Using a Key Value Approach		
Field	Type	Allow Null	Default Value	Description
<u>Filled_Form_Id</u>	int(11)	No		PK: Autogenerated
Field_Id	int(11)	Yes		FK: Field being populated
Value	varchar(250)	Yes		The Value populated in the field
User_Id	int(11)	Yes		FK: User populating the field
Record_Id	int(11)	No		A number assigned on submitting the form
isActive	tinyint(1)	No	1	Stores whether the entry is used

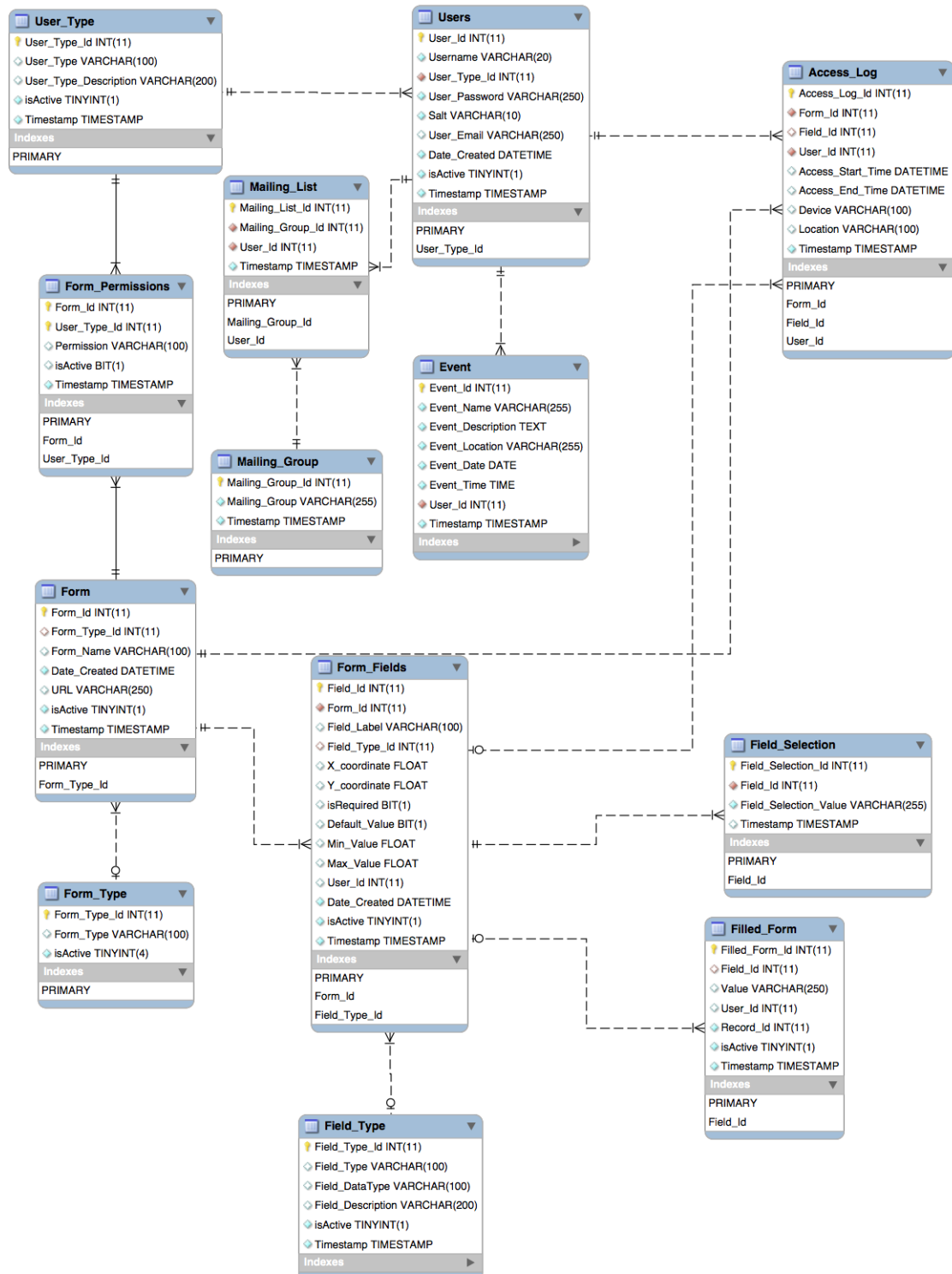
form_fields		Stores the fields that make up the Forms in the Android Application for the specific charity.			
Field	Type	Allow Null	Default Value	Description	
<u>Field_Id</u>	int(11)	No		PK: Autogenerated	
Form_Id	int(11)	No		FK: Form the field has been assigned to	
Field_Label	varchar(100)	Yes		Stores the Field Name	
Field_Type_Id	int(11)	Yes		FK: Field Type been assigned to the Field	
X_coordinate	float	Yes		This attribute was created in case WYSIWYG interface is adopted. Currently not yet implemented	
Y_coordinate	float	Yes		This attribute was created in case WYSIWYG interface is adopted. Currently not yet implemented	
isRequired	bit(1)	Yes		This determines whether this field is required to be populated	
Default_Value	bit(1)	Yes		This determines the default value with which the field is populated	
Min_Value	float	Yes		This determines the minimum value with which the field is populated	
Max_Value	float	Yes		This determines the maximum value with which the field is populated	
Creator_Name	varchar(11)	Yes		Stores the Username of the Person creating the field	
Date_Created	datetime	No	01/01/1970 00:00	Stores the Field was added to the Form	
isActive	tinyint(1)	No	1	Stores whether the Form Field is used	

5.3.6 Relationships

The tables below describe the relationships in both schemas.

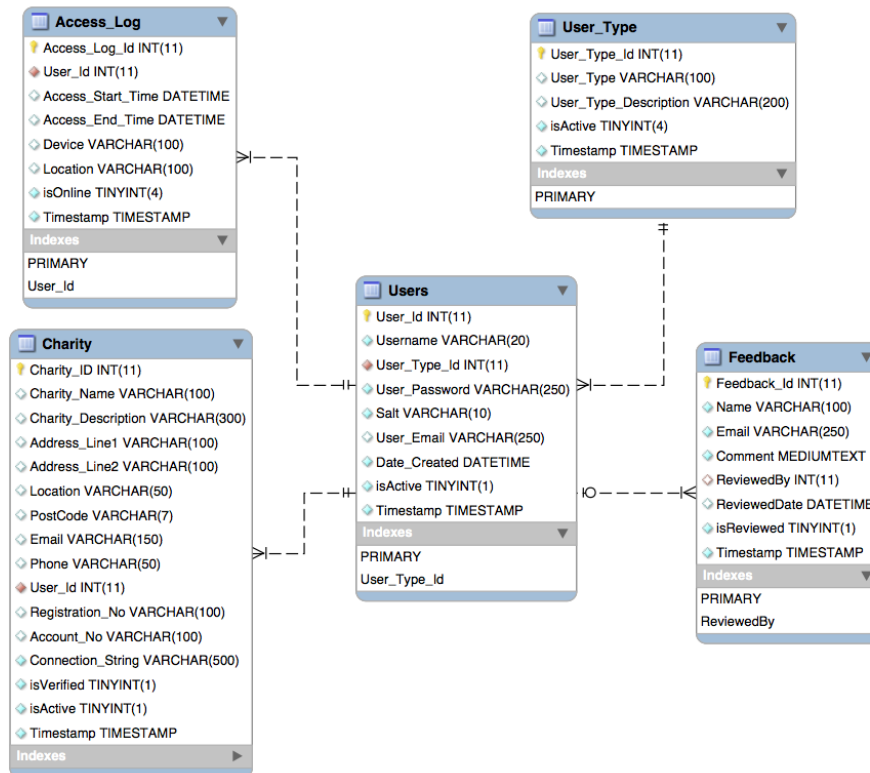
Charity Schema Relationships

Constraint Name	Table	Field	Reference Table	Foreign Field	Update Cascade
FK_Access_Log_Form_Fields_ID	Access_Log	Field_Id	Form_Fields	Field_Id	✓
FK_Access_Log_Form_ID	Access_Log	Form_Id	Form	Form_Id	✓
FK_Access_Log_Users_ID	Access_Log	User_Id	Users	User_Id	✓
FK_Event_User_Id	Event	User_Id	Users	User_Id	✓
FK_Field_Selection_Form_Field	Field_Selection	Field_Id	Form_Fields	Field_Id	✓
FK_Filled_Form_Users_Users_ID	Filled_Form	User_Id	Users	User_Id	
FK_Filled_Form_Form_Fields_ID	Filled_Form	Field_Id	Form_Fields	Field_Id	✓
FK_Form_Form_Type_ID	Form	Form_Type_Id	Form_Type	Form_Type_Id	
FK_Form_Fields_Field_Type_ID	Form	Form_Type_Id	Form_Type	Form_Type_Id	✓
FK_Form_Fields_Form_ID	Form	Form_Id	Form	Form_Id	
FK_Form_Permissions_Form_ID	Form_Permissions	Form_Id	Form	Form_Id	✓
FK_Form_Permissions_User_Type_ID	Form_Permissions	User_Type_Id	User_Type	User_Type_Id	✓
FK_Mailing_List_Mailing_Group_Id	Mailing_List	Mailing_Group_Id	Mailing_Group	Mailing_Group_Id	✓
FK_Mailing_List_User_Id	Mailing_List	User_Id	User	User_Id	✓
FK_User_User_Type_ID	User	User_Type_Id	User_Type	User_Type_Id	



System Schema Relationships

Constraint Name	Table	Field	Reference Table	Foreign Field	Update Cascade
FK_Access_Log_Users_ID	Access_Log	User_Id	Users	User_Id	✓
FK_Charity_User_ID	Charity	User_Id	Users	User_Id	✓



6 System Testing

6.1 First Release

The main testing for this release was performed during international testing week where both the Charityware and Schoolware projects. These were tested by the entire of the FSE and SSE MSc Students along with some guests from Japan's National Institute of Informatics. A number of key problems were highlighted during this process:

- The website was missing functionality leading to unused or buggy code where something had not been implemented yet.
- Security problems the most prominent being SQL injection.
- Display issues across UCL Admin and the charity administrators calendar.
- Mismatching code styles and architectures in places leading to a sometimes confusing mix of technologies.

These issues were resolved during the refactoring effort that has taken place between the first and second builds.

6.2 Second Release

The main problems that this release faces have been outlined below, allot of the problems stem from simple man hour constraints that were present within the project.

6.2.1 Web HTML compliance and CSS

The current Web HTML and CSS are not compliant with any of the major standards, this is not due to the HTML being totally invalid e.g. open close tag pairings, but the problem mostly lying in the use of incorrect meta tags and attributes within elements. This is not a huge issue as it works correctly on Safari, Chrome, Firefox and IE however needs to be addressed in future releases of the code. It would also be a good idea to ensure that the CSS is brought up to scratch against standards to help reduce the current amount of redundant or styling.

6.2.2 Web Service Security (Web and Android)

Currently the web services do not implement any form of security at all for both Web and Android. This could be resolved with the implementation of something like Amazons Oauth which would cause the client communicating with the service to be authenticated. This in addition to HTTPS where the data needs to be fully secured would resolve the bulk of the security concerns in this area.

6.2.3 Website session security

Currently the session data is stored within the default HTTP session cookies, this for obvious reasons is not a secure solution and needs to be replaced with one of the popular alternatives.

6.2.4 Form's Security

Most of the forms provided on this site do not have any protection against people attempting to spam the system with data. This is quite a broad security point as it incorporates multiple attacks from Injection to denial of service however all user input entry points need to be assessed with security in mind.

6.3 Testing Summary

Overall a more stringent testing methodology should have been used where by the code is tested as it is written, hopefully this will save time in future when bug fixing the code.

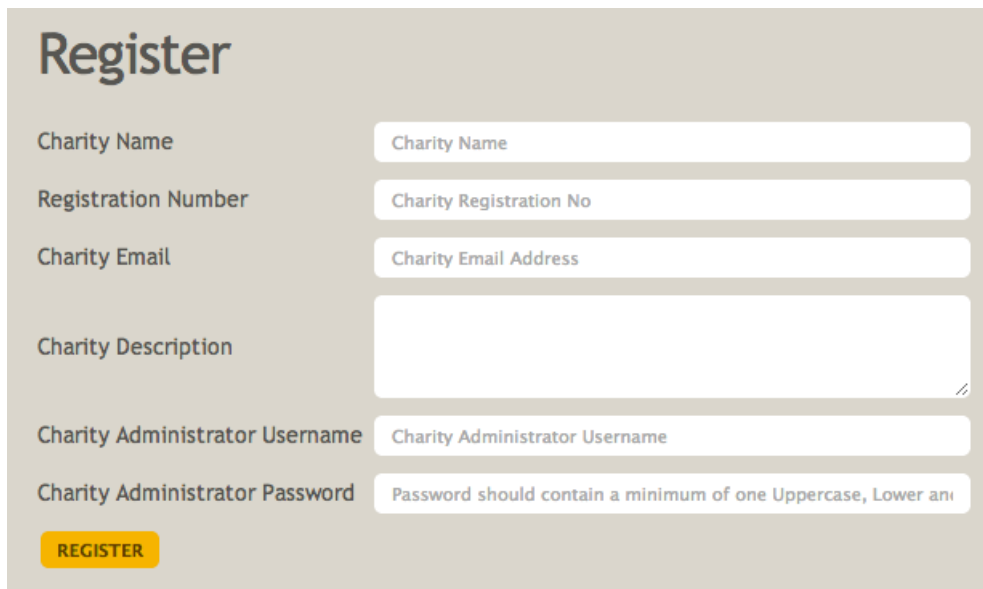
Appendix

A USER MANUAL

A.1 Public

A.1.1 Register as Charity

To register to use the charityware system you first need to register your charity with us. This can be done by filling in a short application form that can be found at charityware.cs.ucl.ac.uk and looks like the image bellow. Once registration has been completed you will need to wait for a charity administrator to review your application and contact you.



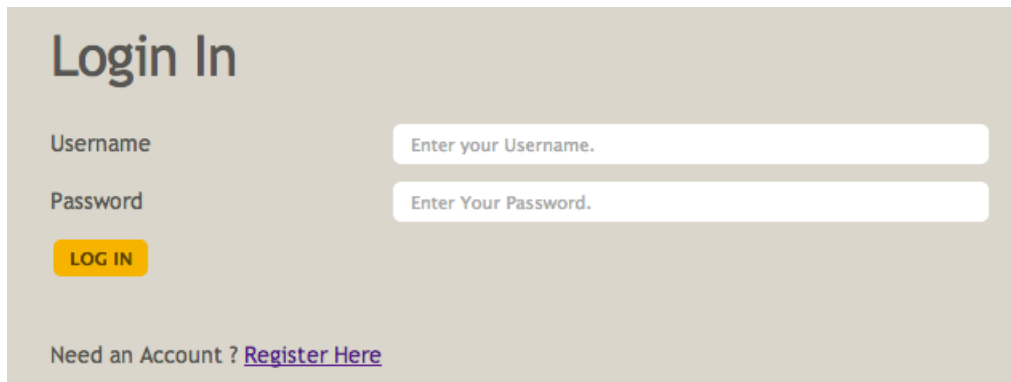
The image shows a web form titled "Register" on a light beige background. The form contains several input fields and a submit button. The fields are arranged in a two-column layout. The first column contains labels for each field, and the second column contains the input fields themselves. The fields are: "Charity Name" (text input), "Registration Number" (text input), "Charity Email" (text input), "Charity Description" (text area), "Charity Administrator Username" (text input), and "Charity Administrator Password" (text input). The password field has a placeholder text: "Password should contain a minimum of one Uppercase, Lower an". At the bottom left of the form is a yellow button with the text "REGISTER".

Label	Input Field
Charity Name	Charity Name
Registration Number	Charity Registration No
Charity Email	Charity Email Address
Charity Description	
Charity Administrator Username	Charity Administrator Username
Charity Administrator Password	Password should contain a minimum of one Uppercase, Lower an

REGISTER

A.1.2 Login

Once your application is successful you will be able to login via the login portal that can be found at charityware.cs.ucl.ac.uk. Please note this is for the charity administrators use only.



A login form with a light gray background. At the top left, the text "Login In" is displayed in a large, bold, dark gray font. Below this, there are two input fields. The first is labeled "Username" and contains the placeholder text "Enter your Username.". The second is labeled "Password" and contains the placeholder text "Enter Your Password.". Below the password field is a yellow button with the text "LOG IN" in black. At the bottom left, there is a link that says "Need an Account ? [Register Here](#)".

The following login credentials should be used for testing purposes:

User: ucladmin

Password: open

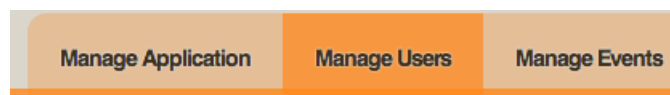
User: lcadmin

Password: open

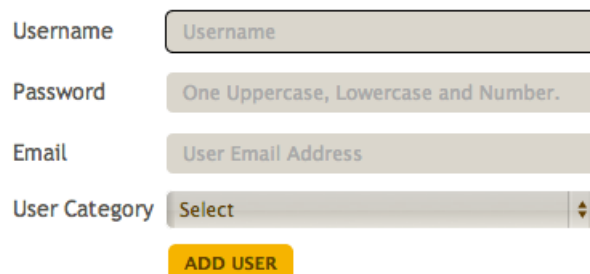
A.2 Charity Admin

A.2.1 Users

To add a charity worker to the system, first login as instructed in the previous section of this document, from there select the orange Manage Users tab and fill in the credentials of the user and click Add User.



A horizontal navigation bar with three tabs. The first tab is "Manage Application", the second is "Manage Users" (highlighted in orange), and the third is "Manage Events".



A form for adding a new user. It has four input fields: "Username" with placeholder "Username", "Password" with placeholder "One Uppercase, Lowercase and Number.", "Email" with placeholder "User Email Address", and "User Category" with a dropdown menu showing "Select". Below the fields is a yellow button with the text "ADD USER".

A.2.2 Forms

To create a Form you need to access the Manage Application tab of the charity admin portal. From here you can view the existing information about a Form or create a new one.

Manage Application Manage Users Manage Events Search Statistics

My forms
hibernate.cfg.xml Form name:

Form Wizard
Form name:

Field wizard
Field name Input type ☐ Mandatory?

Current rows:

When adding rows it is important to fill in the field name before adding the row to the form.

A.3 System Admin

A.3.1 Approve Charities

To approve or reject a charity simply click the accept or decline button found on the right hand side.

Requests Manage Accounts Statistics

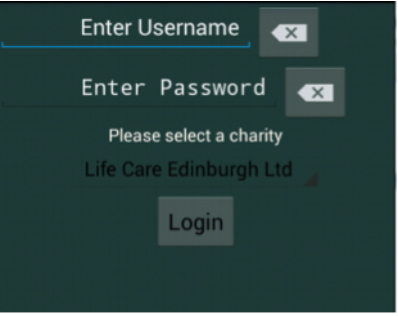
Manage Charity Requests

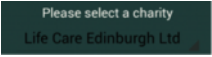
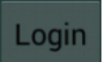
Charity Name	Charity Registration No.	Charity Email	Description	Action
Red Cross	C482963	info@redcross.co.uk	undefined	<input type="button" value="Accept"/> <input type="button" value="Decline"/>
Hope	1041258	info@hope.co.uk	undefined	<input type="button" value="Accept"/> <input type="button" value="Decline"/>

A.4 Android

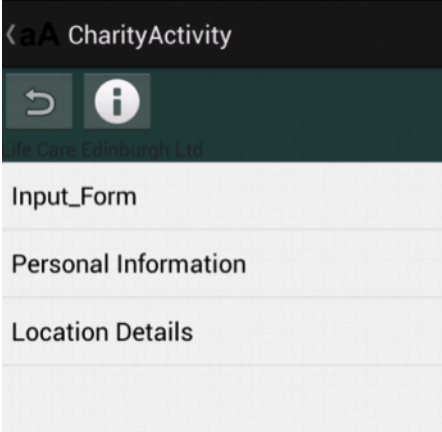
Step-by-Step guide for a complete user session will be outlined below.





A.4.1 Login Screen

The login screen has a dark teal background. At the top, there is a text input field labeled "Enter Username" with a small gray button containing an 'x' to its right. Below this is another text input field labeled "Enter Password" with a similar gray 'x' button. Under the password field, the text "Please select a charity" is displayed above a dropdown menu. The dropdown menu is currently open, showing the text "Life Care Edinburgh Ltd" with a small red triangle to its right. At the bottom center of the screen is a gray rectangular button labeled "Login".

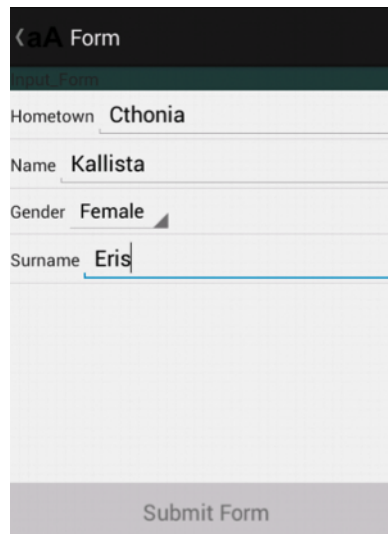
1. Enter your details in the username and password fields. If you wish to delete previous entries, click on the canceling button next to the field you want to clear.
2. Select the charity you want from the drop-down  beneath the username/password fields.
3. Click on the  button.




A.4.2 Form Screen

The form screen has a dark header bar at the top with the text "CharityActivity" and a back arrow icon to its left. Below the header is a dark teal bar containing a refresh button (a square with a curved arrow) and an information button (a circle with an 'i'). Below this bar, the text "Life Care Edinburgh Ltd" is displayed. The main content area is a light gray grid with four rows of text: "Input_Form", "Personal Information", "Location Details", and a fourth row that is currently empty.

1. From the list displayed, click on a Form title to see the fields of that form.
2. If you want to refresh the list, click on .
3. If you request help, click on .
4. The selected charity is displayed under the  button.
5. If you want to log-out, click on  next to the Charity title.

A.4.3 Form Field Screen



1. The name of the selected form is displayed beneath the Form title .
2. Suitably fill the text fields, select values from the drop-boxes, check checkboxes or pick a date from a calendar view.
3. If you want to go back to the previous screen click on  next to the Form title.
4. When ready submit the form by clicking on the  button.
5. If successful login to the CharityWare website as a Charity Administrator to view your latest entry.

B DEVELOPER MANUAL

B.1 Web Service Server Setup

B.1.1 Apache Tomcat

[Apache Tomcat 7](#) has been chosen as the preferred local server on which development took place. In order to install Tomcat 7 on your Mac, please follow the instructions in [this](#) tutorial. If you have a Linux machine, please follow [this](#) tutorial. Lastly, if you have a Windows machine, the recommended tutorial can be found [here](#) (also appropriate for Macs).

B.1.2 MySQL

[MySQL](#) is the chosen database for this project. The schemas used to generate the database can be found in the [GitHub](#) repository of CharityWare under resources. In

order to successfully run CharityWare on your local machine, you need to make sure MySQL is installed. [MySQL Workbench](#) is a useful tool if you are looking for a good MySQL GUI interface. If you're on a Mac, you should follow [this](#) tutorial or [this](#) one if you're on Linux. On Windows, you should have a look at [this](#) tutorial.

B.1.3 Java 7 (both JRE and JDK are required)

[Java](#) usually comes pre installed on the Mac. However, if your Mac does not have Java installed, you can follow [this](#) tutorial. On Linux, [this](#) tutorial should be followed and on Windows [this](#) one.

B.1.4 Eclipse EE

[Eclipse EE](#) allows easy creation of Web Projects, which is why this IDE is recommended for implementing CharityWare. Once the appropriate version has been downloaded, Eclipse can be started from the executable created by unarchiving the downloaded file.

B.1.5 Server Details

Please contact [Dr. Dean Mohamedally](#) for details regarding credentials of the server.

B.2 Android Setup

The only tool needed to run the Android app is the [Android Development Tools IDE](#). Once it is installed on a local machine, the user needs to import the android project following its download from [Github](#). From there, using a suitable Android device emulator the app can be launched.