Developer Documentation

Contents

[Development environment 3](#_Toc388270757)

[Component Libraries 3](#_Toc388270758)

[Understanding the Architecture 4](#_Toc388270759)

[Model 4](#_Toc388270760)

[The controller 6](#_Toc388270761)

[The View 7](#_Toc388270762)

[Extending the Application 7](#_Toc388270763)

[Contributing to the application 7](#_Toc388270764)

[Creating new fields 7](#_Toc388270765)

[Creating new Entities 10](#_Toc388270766)

[Creating reports 10](#_Toc388270767)

[Generating via SQL 10](#_Toc388270768)

[Generating via Controllers 10](#_Toc388270769)

[Changing the Database Engine 10](#_Toc388270770)

[Configuring for Multi-user environment 12](#_Toc388270771)

# Development environment

The application was developed in strictly Java and XML using the eclipse integrated development environment (IDE). The database engine for the application was SQLite due to the requirement to run across multiple platforms and the possible future requirement to run within mobile environments. However the system is database agnostic which allows future developers to change the database engine without requiring extensive modification of the code-base of the application.

The source code versioning is managed with Git and is stored on Bitbucket as a free and open source application. This provides future developers to customize the application to their specific needs and contributing code that will improve the relevance and robustness of the application over time.

|  |  |  |
| --- | --- | --- |
| http://cdn.osxdaily.com/wp-content/uploads/2013/11/java-os-x-mavericks.png | Java | http://www.oracle.com/technetwork/java/javase/downloads/index.html |
| http://help.eclipse.org/juno/topic/org.eclipse.platform.doc.user/whatsNew/images/icon.png | Eclipse | https://www.eclipse.org/downloads/ |
| http://www.unixstickers.com/image/cache/data/stickers/git/git_badge.fw-600x600.png | Git | https://www.eclipse.org/egit/ |
| http://upload.wikimedia.org/wikipedia/commons/thumb/3/38/SQLite370.svg/382px-SQLite370.svg.png | SQLite | https://bitbucket.org/xerial/sqlite-jdbc |
| https://drupal.org/files/project-images/8251351500_4e9811a400_z.jpg | Bitbucket | https://bitbucket.org/kyledef/dammeri |
| http://www.jgit.org/egit2.png | EGit | http://www.eclipse.org/egit/download/ |
|  |  |  |

# Component Libraries

The application was developed with a number of libraries. This section will highlight the components and libraries used. This will help developers identify with library is useful for:

1. Extending a particular functionality
2. Diagnosing a problem
3. Improving the performance and/or capability of the application
4. Upgrade components for security fixes.

|  |  |  |  |
| --- | --- | --- | --- |
| Library | Version | URL | Description |
| Better beans binding | 1.3.0 | https://kenai.com/projects/betterbeansbinding/ | Increase the performance of the model UI generation |
| JCalendar | 1.4 | http://toedter.com/jcalendar/ | Used to develop the calendar/date picking functionality of the interface. |
| Metawidget | 3.8 | http://metawidget.sourceforge.net/ | Metawidget facilitates the model gui binding. Used to generate the data entry forms by the fields in the model of the application. |
| ORMlite | 4.48 | http://ormlite.com/ | The object relational mapping library that allows binding of the table entities with the data models of the application. |
| SQLite JDBC | 3.7.15-M1 | https://bitbucket.org/xerial/sqlite-jdbc | Provides the database connectivity functionality allowing the java code to communicate with the database |
| Super-csv | 2.1.0 | http://supercsv.sourceforge.net/ | Allows the export to csv to give some excel like compatibility |
| Swing worker | 1.2 | https://code.google.com/p/jfxtras/downloads/detail?name=swing-worker-1.2.jar | Facilitate greater control of concurrency to help increase performance. |
| Apache POM |  |  | Conversion to Excel |

# Understanding the Architecture

The application is divided into 3 main components based on the MVC architectural design. These components are grouped into packages to help ease the maintenance of code.

These components are:

1. Model: The logical data entities that exist within the system. A group of these logical entities may be needed to form the physical data entity that the end user will be familiar with.
2. Controller: Responsible for handling the major logic relating to each of the logical operations within the system.
3. View: The view will handle the logic and operations that allow the user to interact with the underlying functionality.

## Model

The model was developed to wrap the data elements within the application and allow its management using an object oriented paradigm. The translation between the Object and Relational representation of the data is handled by a middleware Object-Relational Mapper (ORM) library called ORMLite. ORMLite was select as it was lightweight, feature-full and compatible with android/SQLite.

The general design of the model is presented in figure 1 bellow.

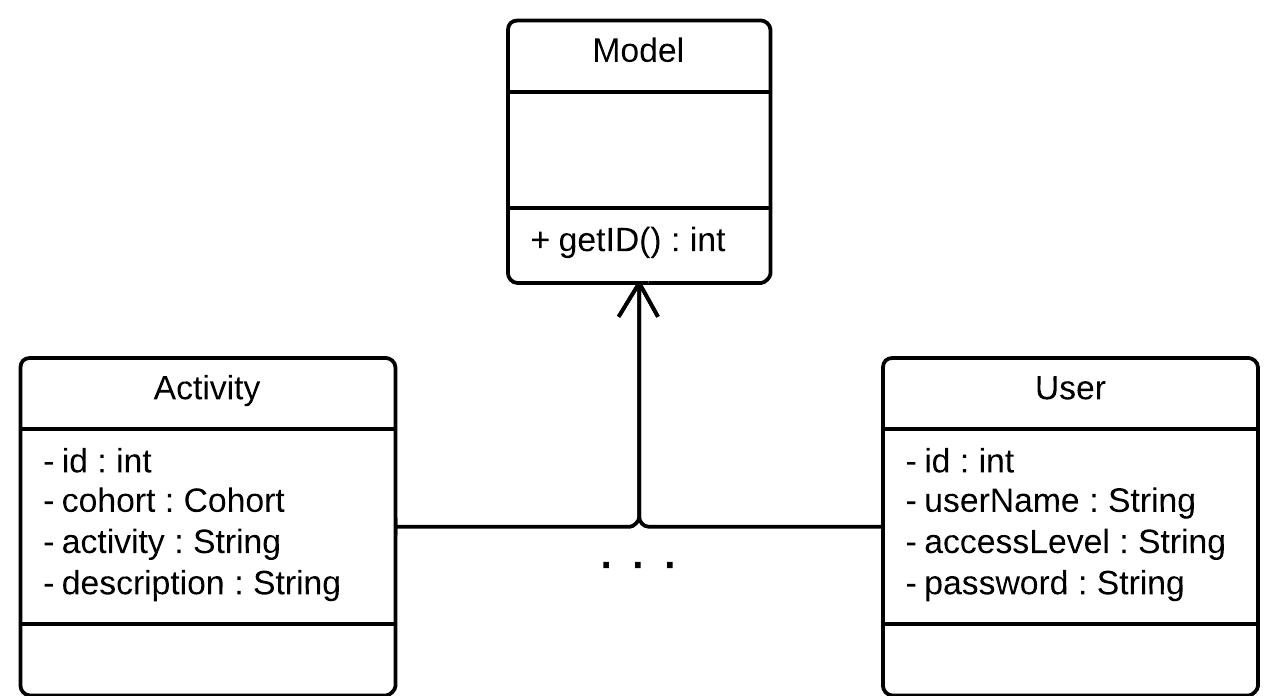


Figure - Model

Each model between the Activity model and the User Model will contain their own fields based on its requirements to store data. Each of these fields typically relates to a corresponding table within the database. A detail of the implementation of the framework is discussed in the creating new fields section of the manual.

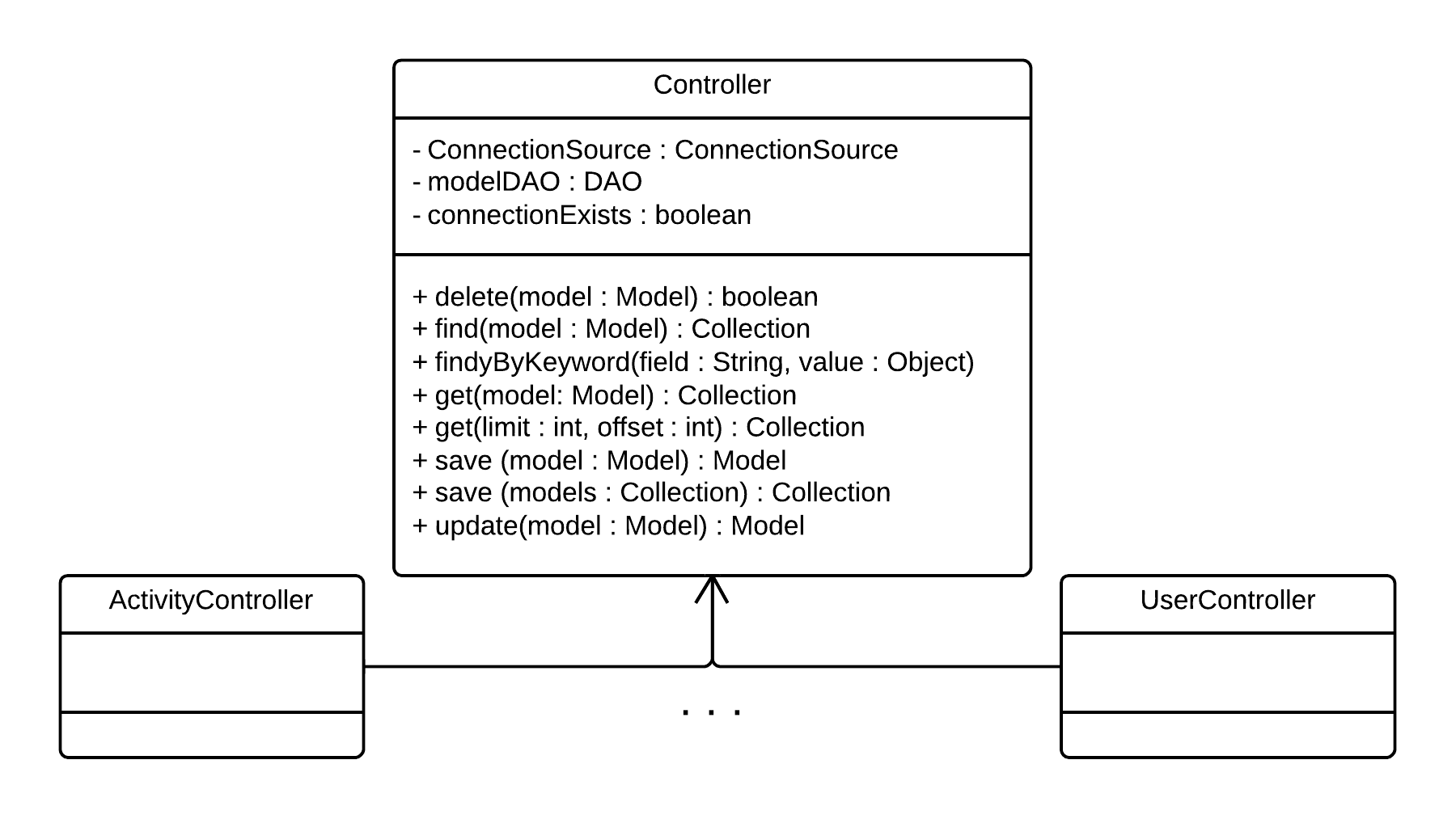


Figure - Controller

## The controller

The main controller parent class contains the majority of functionality for performing CRUD (create, read, update and delete) functionality of the database. Each specific controller will therefore implement functionality that is specific to that entity within the application. A common area of specialization in the specific controller is in the find functionality.

Because the fields are different between the models, each controller will implement the find based on the values that can be passed in to be queried.

We will explain the general composition of the specific controller using the User Controller as an example.



Figure – User Controller

One of the important things to note is that the Controller constructors are private. We are using a simple version of the singleton patter and therefore the static getInstanceMethod on line 41 will handle the creation process for this class. Due to the expensive process of creating connections and setting up the class, the application attempts to use only one instance of these controllers to make the expensive operations impact on the application less obvious.

Each controller will implement the init() method illustrated from line 49. This method will create the connection to the specific table that the respective model is related to. The data manipulation operations are handled by the Data Access Object (DAO) which is initialized on line 52.

Another area that each specific controller must define is the validate method illustrated from line 57. This method will check model to ensure that required information is supplied and that is presented in an acceptable format. The method will return “ok” if the model passes its validation.

## The View

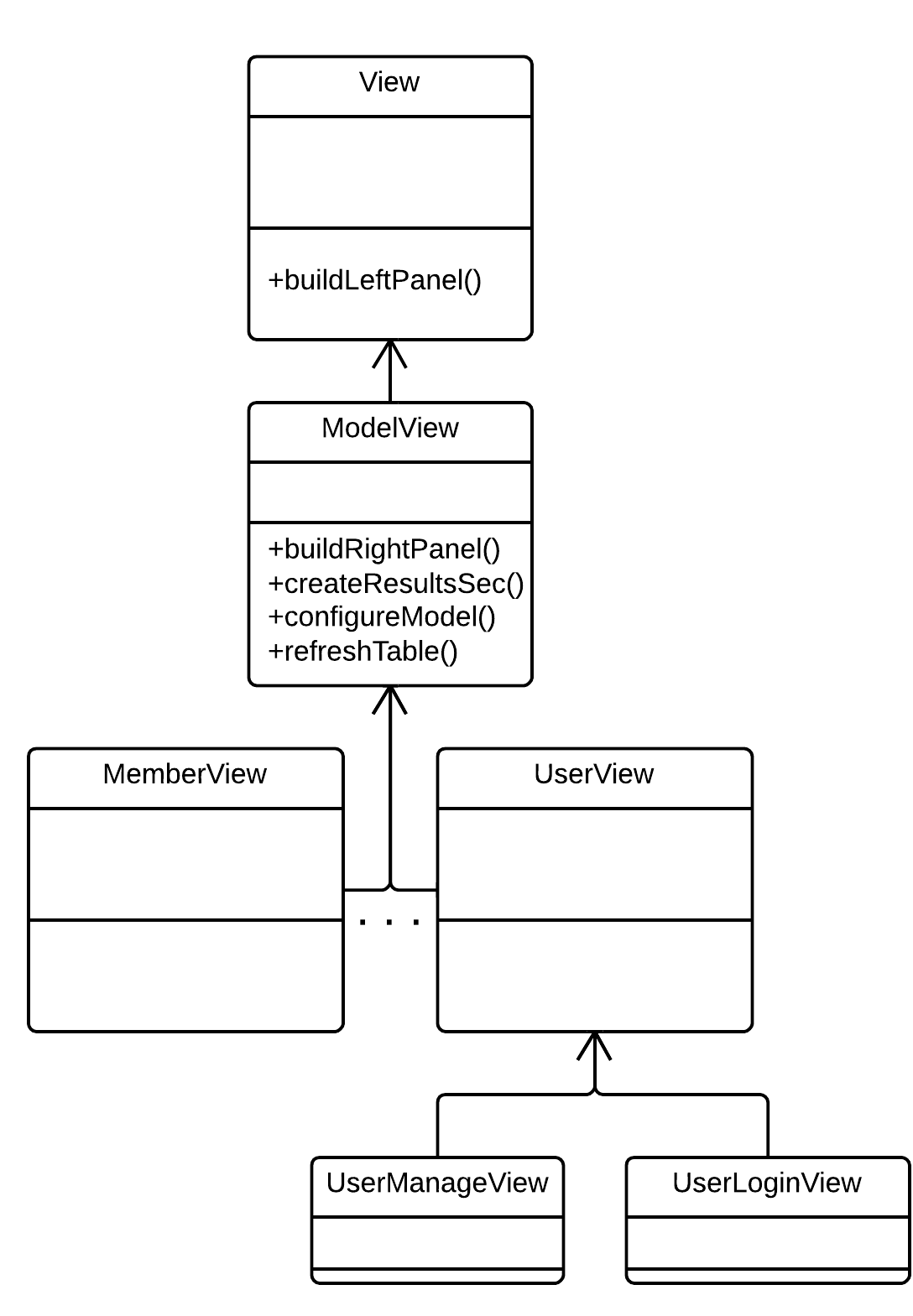


Figure - View

The view configuration is more complicated that both. However the organization is based on the concept of specialization on the basis of functionality.

The view is a general class that configures the look and feel of the application and defines operations that are performed when the screens of the applications are opened and closed.

To deal with managing the operations of the models we developed a ModelView class that configures the view with the required parameters to display the model and its related information to the user.

The ModelView uses the MetaWidgets library to automatically develop an interface based on the fields that are specified in the class. In most cases the specific user interface can be developed from the ModelView. However there may be additional user interface elements that may relate to a single model. In this case we can use a class to define the common operations and then provide sub-classes that will use the common base for the desired class. An example of this configuration is provided in the UserView views. Both the UerManageView and the UserLoginView use the UserView as a base to develop the UI elements for displaying the screens to the user.

# Extending the Application

## Contributing to the application

The application is developed in an open manner with continuous feedback from various stakeholders within the development process. The expectation is that improvements, bug fixes and new features will be contributed back to the application to improve the quality of the application.

The following gives an indication of some of the steps to contribute to the source code of the application.

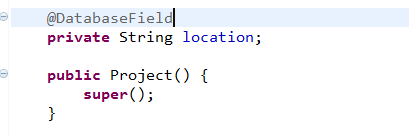
1. Install the Java 7 Java Development Kit (JDK). <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
2. Install eclipse by downloading from the URL provided above. Extract eclipse into a folder that it will be easily found. Open the folder and run the eclipse executable found within the folder.
3. Install Git or a Git application to manage the development process. For windows we recommend using “SourceTree” to manage the development process. (<http://blog.bitbucket.org/2013/03/19/introducing-sourcetree-git-client-microsoft-windows/>) An instructional video that highlights how to setup and configure Git using SourceTree can be found at: <https://www.youtube.com/watch?v=1lBdlh3AGSc> or <http://dotzlaw.com/tutorials/servoy-tutorials/servoy-tutorial-git-sourcetree/>
4. Install the EGit plugin into the eclipse installation. This will allow the development workflow to be managed from eclipse.
5. Clone the repository found at the bitbucket URL specified above.
6. There are a number of online resources that discuss how to manage repositories of Git. A number of quality resources can be found at <http://sixrevisions.com/resources/git-tutorials-beginners/>

## Creating new fields

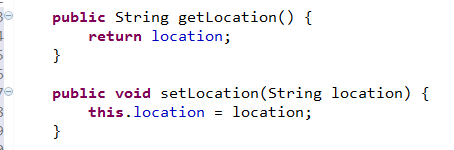
For the addition of additional fields that will be a simple or primitive data type (String, numbers, currency etc.) will require modification to the SQL generation script “db.sql” with the fields in the table to be modified. This is followed by a simple addition to the model class and modifying the “metawidget-metadata.xml” to specify the order that the newly created field should be placed on the interface.

Example:

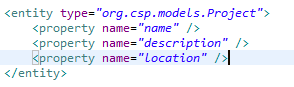
If a location was to be added to the project we first begin by adding the location field as a String to the project class.



Note that we added the “@DatabaseField” annotation just above the location string specified. This allows the system to know that this field will be saved to the a corresponding field in the projects table with the same name. There are additional functionality with this annotation field that can allow the database field and the application field to differ. Examples of this can be found in the Cohorts class.



After create the field, we are required to create the accessors and mutators for the field created. This allows the UI library (Metawidget) to know which fields to add the data entry interface.



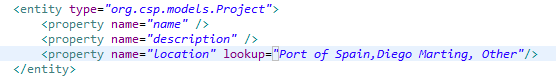
We then add the field to the “metawidget-metadata.xml” as a property of the project entity.



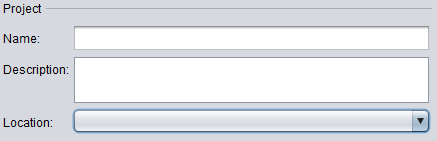
After completing this and lunching the data entry interface for the project, we now see the location field that we recently created.

If there is a standard set of locations, we may want to provide the users with the ability to choose from a list compared with entering the information manually which may be prone to errors.

To do this we can add additional information as attributes to the property we would like to extend.

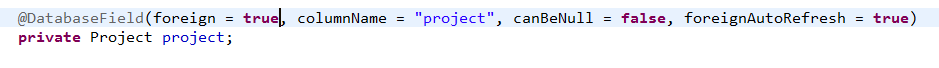


We add the lookup attribute to the property and add the options separated by commas. This now changes the interface and the text field is converted to a drop down box as follows:



If there is a more complex addition of a field a good example to follow will be the Project model within the Cohort model.

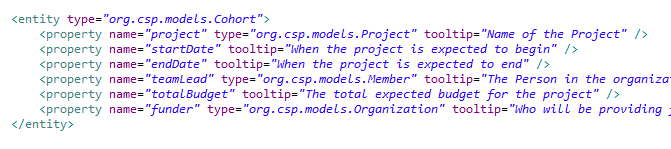
We add the Project as an attribute of the Cohort Class and specify using the annotation field that this is field in the cohorts table in the database.



Note that there are additional bits of information specified in the annotation. The meaning of these additional information can be found in the ORMlite documentation.

In order to establish the relationship between these two models, we use the speficiation “foreign = true” and “foreignAutoRefresh=true” that will ensure that the ORM makes some of the linkages for us automatically. These linkages can include loading the project and its related data when the cohort is loaded.

Similar to the location we add the Project as a property of the Cohort entity.



Notice that unlike the location we are now required to specify the type attribute of the Project. This helps the system recognize that this is a sub-model and will figure out how to properly represent this object.

## Creating new Entities

The current application is developed based on what has been determined as common

## Creating reports

Developers have a number of methods to create their own custom reports. Here we will consider two strategies; using SQL code and using functionality exposed by the objects.

### Generating via SQL

The ORMLite library supports the querying of the database using raw SQL commands.

This operation and capability is discussed in the library documentation at: <http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite_2.html#Raw-Statements>. The DAO object of the model is related in the corresponding controller. Therefore the controller can be modified (Preferably extended) to be customized with these specific reporting functionality desired.

### Generating via Controllers

The preferred method of building controllers is through the use of the Query Builder mechanism within the ORM. The details of this capability and its methods can be found at: <http://ormlite.com/javadoc/ormlite-core/doc-files/ormlite_3.html>

Examples of the utilization of the query builder can be found in the find method of the respective controllers.

## Changing the Database Engine

The database CRUD functionality is implemented on the ORMLite library. The ORMLite library supports the following databases:

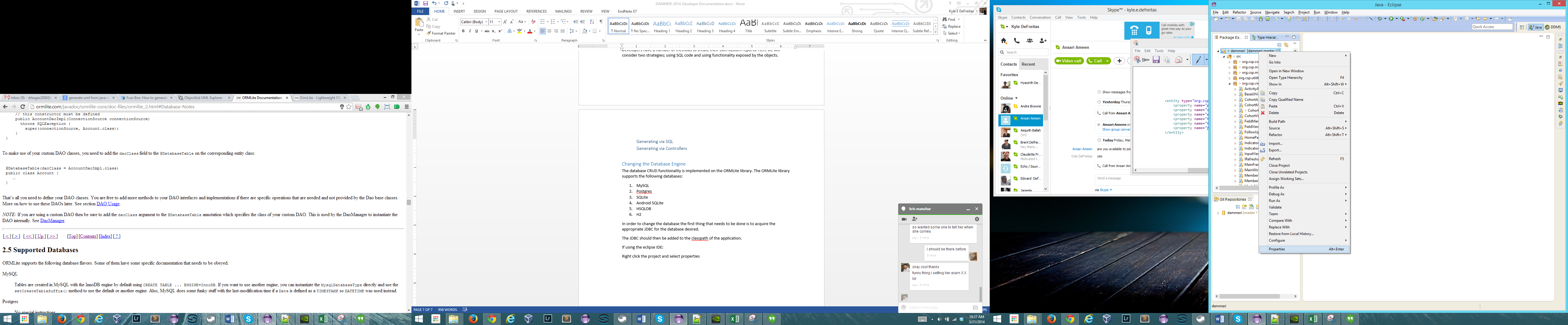
1. MySQL
2. Postgres
3. SQLite
4. Android SQLite
5. HSQLDB
6. H2

In order to change the database the first thing that needs to be done is to acquire the appropriate JDBC for the database desired.

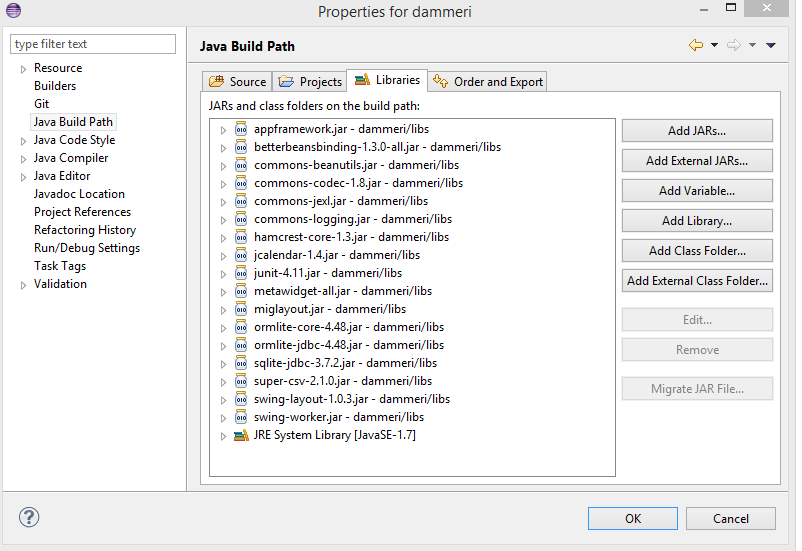
The JDBC should then be added to the classpath of the application.

If using the eclipse IDE:

Right click the project and select properties



This opens the properties dialog after which we will select the Java Build Path option in the left tree menu option. We select the Libraries tab which will present the list of libraries that are used by the application.



## Configuring for Multi-user environment

Using the MYSQL DBMS we can configure that application to work as a front-end client for a backend database stored on a centralized server or on a cloud provided such as AWS.