# Design

Within this chapter I will be discussion the design choice I have made whilst implementing this project and the reasons for making them choices. This chapter will also include details on the user interface design.

## Overall Architecture

I intended to design the system in a modular format thus the system contains four main packages with each package containing a tests package. Keeping code modulated enhances the reusability of the code and also makes the code easier to maintain. In this section I will be listing each package and explaining their purpose within the application.

## Activities package

Activities are Androids views; an activity provides a method of displaying information on screen for the user. The activities package contains the implemented classes for the activities.

## Models package

The models package contains the model classes for each of the database tables. A model represents an item created from the database, for example a drug or a piece of information about a drug. Keeping all models within one package allows the models to be reused in other applications.

## Data download package

The data download package contains the Robospice requests classes, which are responsible for downloading the individual API URLs and then parsing the data obtained. Each class within this package downloads a specific API XML file. Having all the data download classes in one package makes it easy to add new classes to download additional data in the future.

## Database package

The database package contains the class that is responsible for the interacting with the SQLite database. This class is responsible for fetching, creating and deleting data from the database. Having this class within its own package will allow the database and database code to be reused within other applications. If the code within the database helper class becomes to large using a package will also allow the code to be broken into smaller classes.

## Tests packages

Each package contains a package called tests. Within each test package is classes testing each class of the original package through both JUnit tests and Android instrument tests

## XML edibility

As the application will be given to a team of developers who may not have extensive experience in Android development it was essential that I make the basics of the application editable by an inexperienced developer, to achieve this all string values have been set in one XML file (strings.xml). Having done this, any text within the application can be changed by editing this XML file. As this application is produced for NHS Wales they may in the future want to translate the application to Welsh, to adding Welsh language support would be relatively easy due to this design choice.

To further improve the customisability of the application all API URL’s and the structure of the XML files is set within a single XML file (data\_download.xml). This makes the task of changing API URL’s and the format of the data simple.

## Data storage

One of main decisions when designing the application was deciding how the data for the application would be stored, within this section I will discuss the possible methods of storing the data and why I chose the method I chose.

## Store the data as XML files

Downloading the XML from the XML API URLs and storing the XML file on the device was one of the methods I considered. This method would allow for a quicker download task, as the task would not have to parse the XML. This method would also ensure the data was exactly as it was downloaded.

This is not a great method of storing the data on the device, as the device would have to parse all the XML files every time the data was needed be used. This method would also be slow at searching and ordering the data, as this method provided no indexing functionality.

## Store the data as a serialised object

For this method the application would parse the XML file into Java objects and then save the object onto the device through serialisation. This method would be quicker at runtime than storing the as XML files as the data will already be parsed when opening the data.

There are several issues with this method, the first issue being that this method does not allow indexing of the results and therefore searching the data will be slow. Another issue with this method is the entire dataset will need to be loaded into the devices memory when using the data, which could cause some devices to run out of memory.

## Store the data within an SQLite database

The Android SDK comes with a set of libraries for creating and interacting with SQLite databases. SQLite database can be indexed and are therefore quicker than the above-mentioned methods for searching the data.

I decided to use an SQLite database to store the data for the application. The data is parsed from XML into usable objects and then stored in the SQLite database. Using an SQLite database allowed me to only load needed data into memory by using the WHERE operator when searching for data. Using an SQLite database also helped keep the data in organised structure, which would allow the data to be used by other applications if needed.

The main issue with using SQLite database is that the stored data may be larger than if the data was stored as a serialised object, but if the size of the database became an issue the database could be compressed.

## Database design

As I had decided to use a database to store the data, the next step was to design the database structure. The database tables have been normalised into third normal form. SQLite foreign key constraints were only introduced into the Android SDK at API version 16, as I wanted the application to have the largest reach possible I decided not to use foreign key constraints and therefore monitor foreign key relations manually.

The first table to be designed was the Drugs table; the purpose of this table is to contain all the drugs that are downloaded. Within the original designs I had planned to place all drug information within the drugs table, but as the drug information’s changed between drugs I decided it would be better to create a separate table for drug information’s.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table name** | **Data type** | **Allow null** | **Purpose** |
| id | Integer | No | Holds the drugs ID that is set from the downloaded XML. This is the primary key. |
| name | String | No | Contains the drugs name. |

**Table showing `drugs` table structure**

The next table to be designed was drug information’s table; the purpose of this table was to hold a single piece of information about a drug, such as the method of administration or the flushing guidelines. Some drug information’s also contain a header helper. Header helpers are extra help information explaining the header. These are displayed to the user upon clicking the header or information button next to the header.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table name** | **Data type** | **Allow null** | **Purpose** |
| id | Integer | No | Holds the drug information id, this is an auto incrementing value. This is the primary key |
| drug\_id\* | Integer | No | The id of the drug this information belongs to. This is a foreign key to the drugs table id. |
| header\_text | String | No | The header for the piece of information |
| header\_helper | String | Yes | The helper information for the drug information if it exists |
| section\_text | String | No | The main body of information for the drug information. |

**Table showing `drugInformations` table structure**

The next table designed was the drug indexes table; this table contains the drug indexes. A drug index is an extra name for a drug. A drug may have more than one index. The drug indexes are what the user will be shown in the list of searchable drugs.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table name** | **Data type** | **Allow null** | **Purpose** |
| id | Integer | No | Holds the drug information id, this is an auto incrementing value. This is the primary key |
| drug\_id\* | Integer | No | The id of the drug this information belongs to. This is a foreign key to the drugs table id. |
| name | String | No | The name for the index |

**Table showing `drugIndexes` table structure**

The final table to be designed was the table containing the drug calculation information’s. This table contains all the information needed to perform the infusion rate or dose calculations. As SQLite does not support boolean types I had to use integer values (0 or 1) for if the patient weight required and time is required columns.

|  |  |  |  |
| --- | --- | --- | --- |
| **Table name** | **Data type** | **Allow null** | **Purpose** |
| id | Integer | No | Holds the calculators id, this is an auto incrementing value. This is the primary key |
| drug\_id\* | Integer | No | The id of the drug this information belongs to. This is a foreign key to the drugs table id. |
| infusion\_rate\_label | String | No | The label used for the infusion rate calculator |
| Infusion\_rate\_units | String | No | The units of the calculated infusion rate. |
| dose\_units | String | No | The units of the calculated dose |
| patient\_weight\_req | Integer | No | If the patients weight is required for the calculation (1 represents true, 0 false) |
| time\_req | Integer | No | If the time is required for the calculation (1 represents true, 0 false) |
| factor | Integer | No | The factor used within the calculation |

**Table showing `drugCalcs` table structure**

## Downloading of data

The next part of the design process was to decide how the data for the database would be downloaded. The download process had to be able to run in the background on the device as the download can take several minutes, therefore a user is likely to minimise the application. In this section I will discuss the methods of downloading the data that I considered.

## Download the data using an AsyncTask

It would have been possible to download the data using an AsyncTask as they AsyncTasks create a new thread meaning they can be used for HTTP requests. Within the Android documentation it states that AsyncTasks should not be used for long running tasks (more than a few seconds), due to this reason I decided not to use AsyncTasks for downloading the data. Another reason against using AsyncTasks is that they are attached to the activity that created them, so if the user minimises the application the task will also be killed, thus not allowing background downloading.

## Download the data using a service

The Android documentation suggests using a service for any long running task. A service runs within its own thread therefore allowing the download to continue working whilst in the background. A service will also place an item in the devices notification panel letting the user know that the service is still running. Using a service to download the data would have worked and been a good solution if implemented correctly.

A service is only a basic class for running a process in the background, it provides no implementation for alerting the view of its progress, nor does it contain any error handling should the tasks fail. The reason I chose not to download the data using a basic service is that I would need to implement a large amount additional code to handle all areas of failure and to notify the view of the downloads progress.

## Download the data using a Robospice service

Robospice is an open source library released under the Apache licence that simplifies the implementation of long running tasks. Robospice runs on top of the Android service providing implementation for frequently used features needed when implementing services. Robospice makes notifying the view of progress simple and also allows for multi-threading meaning that simultaneous downloads can occur. Another useful feature that Robospice contains is its error handling features; Robospice retries a task a set amount of times before finally alerting the main thread of its failure.

I decided that using a Robospice service to handle the downloading of data would be the best solution. Using a Robospice service meant I would be able to focus on the body of the task instead of worrying about implementing the various features needed for the service. Robospice does allow for caching, but as the data will only be downloaded once every few months I decided using an un-cached service would be acceptable.

## Design pattern used

When planning the application it was planned to create the application using a Model-View-Controller (MVC) design pattern. Once I began developing within Android it became apparent that a true MVC approach was not possible within Android. Although a true MVC approach was not possible I attempt to follow the MVC pattern throughout my design, thus creating a package for models, a package for activities (views) and separate packages for everything else (controllers). I believe that by modulating the code, the maintainability and reusability of the code has been increased.

## Database interactions class

A single database interaction class was designed. The purpose of this class was to handle all interactions with the database, thus keeping all SQL code into a single class. This class contains the method for creating and retrieving data, whilst also containing the methods for creating and updating the database.

I wanted to be able to test the all methods within the database class without altering the original database, to achieve this a separate constructor was creating which prepends a suffix to the database files name, thus making the test database separate from the original one.

Currently the amount of methods with this isn’t very large, but if the application grows it would be ideal to split this class into separate classes that handle different section of the database, for example a separate class for each table within the database.

## Download data singleton

It had been decided that the downloading of data would be executed within a Robospice service. Robospice services like Android services run within there own thread and are therefore are not attached to a specific activity. Although Robospice services allow for the passing of progress from the service to the activity, they do not natively support the passing of data.

To pass data between the download services and the download activity I designed a singleton class. A singleton class is a class that can only have one object of itself instantiated; a method is created within the class for the purpose of retrieving the single instantiated object. Using a singleton for the download progress meant that both the service and the activity would be using the same object, thus allowing data to be passed between them.

## Calculation class

The calculation class is the class responsible for handling the dosage and infusion rate calculations. I intended to create a powerful class that could handle both the validation and calculation of the calculations.

A method called validate was designed, this method checks that all data used for the calculation is present and correct. The method then returns an integer for the result of the validation; this integer can either represent an error (which the user must change), a warning (which the user has the option to change or continue) or success.

Once the validate method has been called, if the result is a success or a warning and the user opted to continue the calculation will be performed.

Keeping the validation and calculation within the same class means that the calculation code be reused within other projects. Using this approach also allowed for more rigorous testing as the validation and calculations could be tested together whilst Unit testing.

## Preferences class

Android provides developers with a class for storing values on the device. This class is called SharedPreferences; a shared preference is a key-value pair accessed by providing the key as a string. SharedPreferences are useful for storing single pieces of information on the device without creating a separate table in the database for them. Android also provides security on top of SharedPreferences, preventing other applications for accessing your applications SharedPreferences.

The preferences class was used as a wrapper class for inserting, updating and deleting SharedPreferences. The preferences class is used within the application to store the users credentials, the date of the last database update and if the database was successfully downloaded.

Although Android provides some security for SharedPreferences, they cannot stop a malicious user from accessing the data should they have physical control of the device. Therefore it is important that secure information is not stored within the preferences without being encrypted.

## Authentication class

This class was designed to provide authentication with the NHS server. This class contains methods for validating a users username and password and retrieving the user credentials so that they can be used when updating the database. The authentication class also provides a method for checking whether a user is logged in, this is used when the application is first opened and decides if the login or main view should be opened.

## Splash activity

When the application first opens the application has to decide whether to open the login screen, the main screen or the update screen. Android does not allow any logic to be processed before the first activity is displayed; this caused an issue, as there was no way of opening the correct activity without first opening an activity.

One solution to this problem would be put the logic within the main activity and if the user needed login or download the database they could be redirected. This would not be an ideal solution as for a brief amount of time the main activity would be should, causing a flicker.

I decided that designing a splash activity would be the best way to solve this problem. The splash activity briefly displays the NHS logo on a neatly presented background whilst the application decides where to redirect the user, the user is then redirected to the correct screen, without any flickering.

## Common activities

Android provides the user with menu options when the user presses the menu button on the device. This menu contains items such as update, logout and exit. The menu buttons in most applications vary between activities, but within this project the menu items are the same throughout the application, the only time they change is when the user is logged out.

It is possible to create the menu item’s manually within each activity but I want to design the application so that the code is DRY. To achieve this I created a class that inherited androids Activity and implemented the Exit and About buttons for that class, this class could then be inherited within activities that needed to display the Home and About buttons. I then created an another class which inherited from that class adding buttons that can be sign once the user is logged in (Logout, Browse, Calculator).

Using this approach makes it easy to add items to the menu of all activities. If an activity needs a custom option adding to the menu this can be done easily within the activity.

## User interface design

I planned to design an application that was both aesthetically pleasing and easy for the novice user to user. Within this section I will discuss how I achieved both of these objectives.

## Android developer design guidelines

During the background and analysis stage of the project I read the majority of the design guidelines laid out by Android. These guidelines taught me how to create applications that would scale for a multitude of devices whilst only implementing the design once.

The guidelines taught me to use density-independent pixels (dp), for expressing layout and positioning of view object that need to scale as the density increases. If you were to layout an object using normal pixels, the object would use the same amount of pixel no matter how large the screen was, thus if viewing on a large screen with a large amount of pixels the object would appear small. Using density-independent pixels means the object will automatically scale as the density of the screen does.

I also learnt to use scale-independent pixels for fonts and text. Scale-independent pixels act just like density-independent pixels but they also scale for the users font setting, thus allowing the font to be larger if the user sets a large font within their device settings. Using scale-independent pixels increasing the accessibility of the application.

The guidelines also taught me about layout parameters such as match parent and wrap content. Match parent makes the layout parameter an object match its parents width or height. Using match parent is useful when you want a button to fill then the screen. The wrap content layout parameter allows the size of the views width or height to expand depending on the size of its contents This is useful when displaying information to the user which may be more than one line.

Using these learnt Android design techniques allowed me to design a user interface that is fluid and expands well over a multitude of device size, of varying densities. They also allow the views to be rotated depending on the devices orientation automatically.

## Designing the interface to be easy to use