CODE STANDARDS

Android Development

Table of Contents

1. Project Structure

1.1. Structure

- When starting a new project, Android Studio automatically creates some of these files for you, as shown in below figure, and populates them based on sensible defaults.

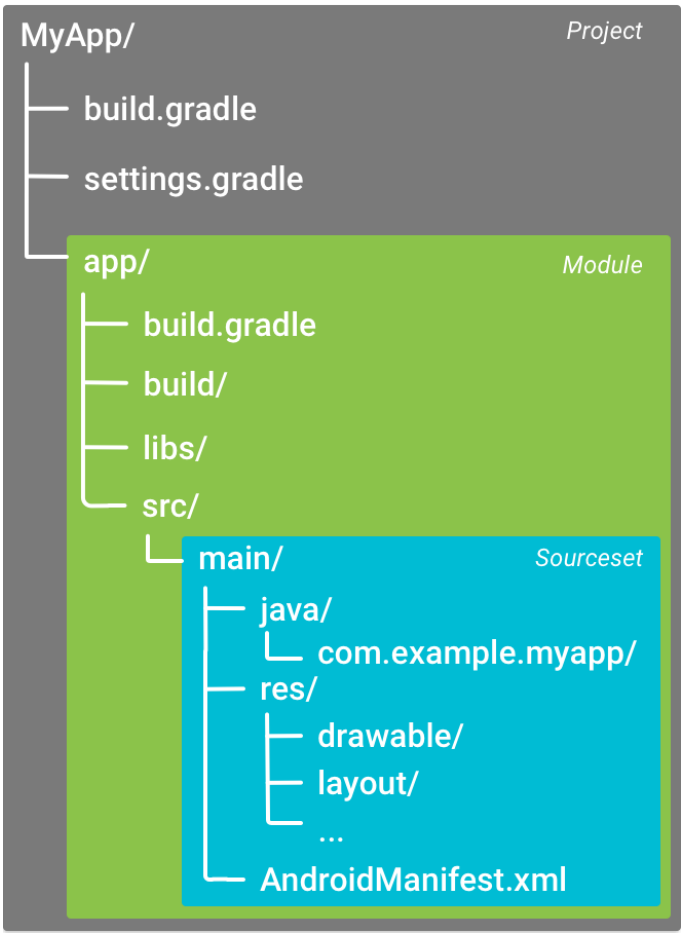

Figure 1.1.1

- There are a few Gradle build configuration files that are a part of the standard project structure for an Android app. Before you can start configuring your build, it is important to understand the scope and purpose of each of these files, and the basic DSL elements they should define.
- The Top-level Build File

- The top-level build.gradle file, located in the root project directory, defines build configurations that apply to all modules in your project.

- The Module-level Build File
  - The module-level build.gradle file, located in each ***project/module/*** directory, allows you to configure build settings for the specific module it is located in.

- Configure project-wide properties
  - For Android projects that include multiple modules, it may be useful to define certain properties at the project level and share them across all the modules. You can do this by adding extra properties to the ext block in the top-level build.gradle file.

```
buildscript {...}

allprojects {...}

// This block encapsulates custom properties and makes them available to all
// modules in the project.
ext {
    // The following are only a few examples of the types of properties you
can define.
    compileSdkVersion = 26
    // You can also create properties to specify versions for dependencies.
    // Having consistent versions between modules can avoid conflicts with
behavior.
    supportLibVersion = "27.0.2"
    ...
}
...
```

Figure 1.1.2

To access these properties from a module in the same project, use the following syntax in the module's build.gradle file (you can learn more about this file in the Figure 1.1.3).

```
android {
  // Use the following syntax to access properties you defined at the
project level:
  // rootProject.ext.property_name
  compileSdkVersion rootProject.ext.compileSdkVersion
  ...
}
...
dependencies {
    compile "com.android.support:appcompat-
v7:${rootProject.ext.supportLibVersion}"
    ...
}
```

Figure 1.1.3

1.2. Package

- The Package name should be descriptive enough to understand the project element. It should be unique identifier for project itself. The example below explains how the packaging will goes to the project with descriptive naming.

Example: *com.(YourAppName).ui* - which contains all the activities

Example: *com.(YourAppName). fragment* - which contains all the Fragment

Example: *com.(YourAppName). notification* - which contains all notification related java files

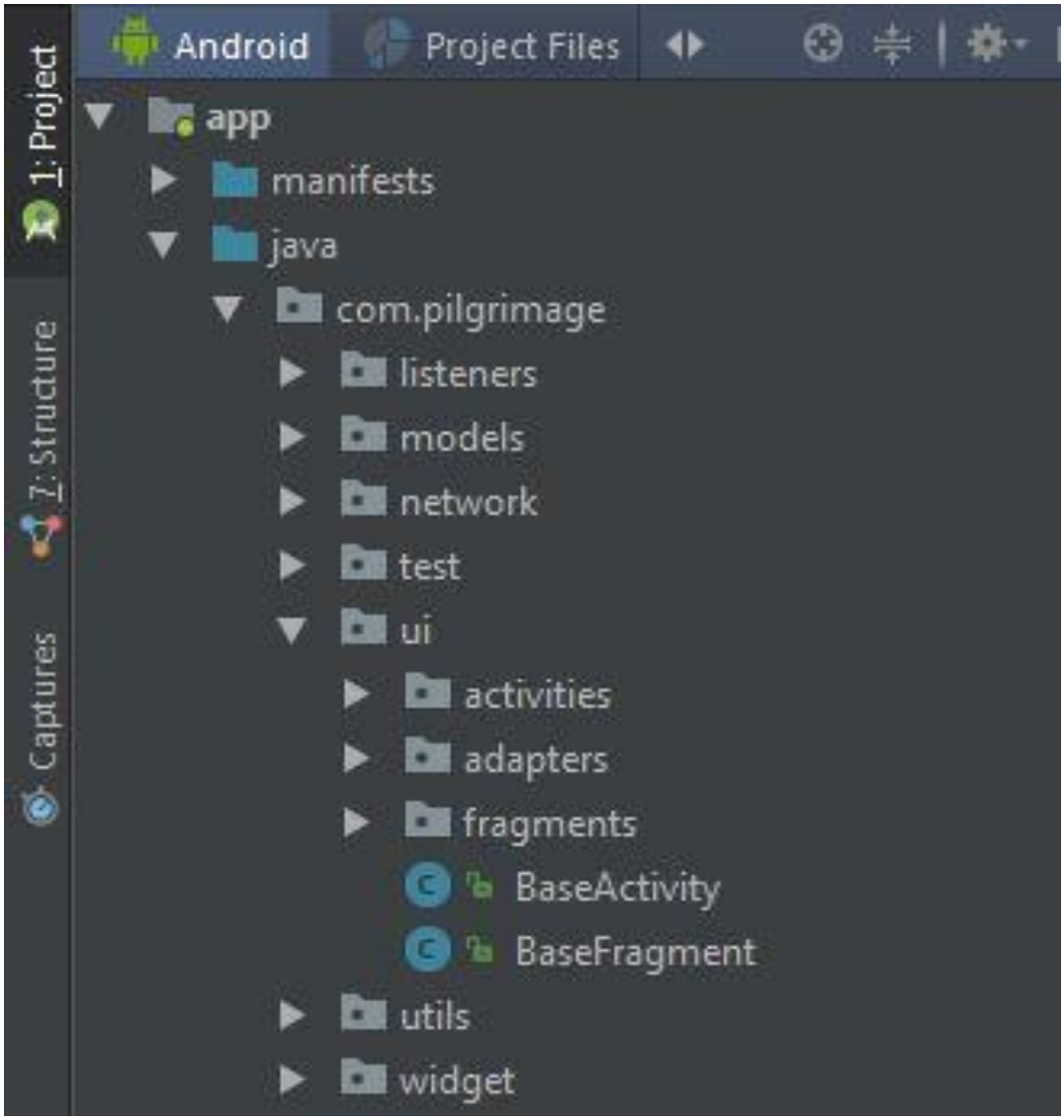

Figure 1.2.1

2. Naming Rules

2.1. Class files

- Class names are written in UpperCamelCase.
- For classes that extend an Android component, the name of the class should end with the name of the component.
- For Example: SignInActivity, SignInFragment, ImageUploaderService, ChangePasswordDialog.

2.2. Resources files

- Resources file names are written in lowercase_underscore.

2.3. Drawable files

- Naming conventions for drawable:

Asset Type	Prefix	Example
Action bar	ab_	ab_stacked.9.png
Button	btn_	btn_send_pressed.9.png
Dialog	dialog_	dialog_top.9.png
Divider	divider_	divider_horizontal.9.png
Icon	ic_	ic_star.png
Menu	menu_	menu_submenu_bg.9.png
Notification	notification_	notification_bg.9.png
Tabs	tab_	tab_pressed.9.png

Figure 2.3.1

- Naming conventions for icons (taken from Android iconography guidelines):

Asset Type	Prefix	Example
Icons	ic_	ic_star.png
Launcher icons	ic_launcher	ic_launcher_calendar.png
Menu icons and Action Bar icons	ic_menu	ic_menu_archive.png
Status bar icons	ic_stat_notify	ic_stat_notify_msg.png
Tab icons	ic_tab	ic_tab_recent.png
Dialog icons	ic_dialog	ic_dialog_info.png

Figure 2.3.2

- Naming conventions for selector states:

State	Suffix	Example
Normal	_normal	btn_order_normal.9.png
Pressed	_pressed	btn_order_pressed.9.png
Focused	_focused	btn_order_focused.9.png
Disabled	_disabled	btn_order_disabled.9.png
Selected	_selected	btn_order_selected.9.png

Figure 2.3.3

2.4. Layout files

- Layout files should match the name of the Android components that they are intended for but moving the top level component name to the beginning. For example, if we are creating a layout for the SignInActivity, the name of the layout file should be activity_sign_in.xml.

Component	Class Name	Layout Name
Activity	UserProfileActivity	activity_user_profile.xml
Fragment	SignUpFragment	fragment_sign_up.xml
Dialog	ChangePasswordDialog	dialog_change_password.xml
AdapterView item	---	item_person.xml
Partial layout	---	partial_stats_bar.xml

Figure 2.4.1

- A slightly different case is when we are creating a layout that is going to be inflated by an Adapter, e.g to populate a ListView. In this case, the name of the layout should start with item_.
- Note that there are cases where these rules will not be possible to apply. For example, when creating layout files that are intended to be part of other layouts. In this case you should use the prefix partial_.

2.5. Menu files

- Similar to layout files, menu files should match the name of the component. For example, if we are defining a menu file that is going to be used in the UserActivity, then the name of the file should be activity_user.xml

- A good practice is to not include the word menu as part of the name because these files are already located in the menu directory.

2.6. Values files

- Resource files in the values folder should be **plural,** e.g. strings.xml, styles.xml, colors.xml, dimens.xml, attrs.xml

2.7. Resources naming

- Resource IDs and names are written in lowercase_underscore.
- IDs should be prefixed with the name of the element in lowercase underscore. For example:

Element	Prefix
TextView	text_
ImageView	iv
Button	btn
Menu	menu_

Figure 2.7.1

Image view example:

```
<ImageView
    android:id="@+id/ivProfile"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Figure 2.7.2

Menu example:

```
<menu>
  <item
        android:id="@+id/menu_done"
        android:title="Done" />
</menu>
```

Figure 2.7.3

- String names start with a prefix that identifies the section they belong to. For example registration_email_hint or registration_name_hint.If a string does not belong to any section, then you should follow the rules below:

Prefix	Description
error_	An error message
msg_	A regular information message
title_	A title, i.e. a dialog title
action_	An action such as "Save" or "Create"

Figure 2.7.4

2.8. String constants, naming, and values

- Many elements of the Android SDK such as SharedPreferences, Bundle, or Intent use a key-value pair approach so it's very likely that even for a small app you end up having to write a lot of String constants.
- When using one of these components, you must define the keys as a static final fields and they should be prefixed as indicated below.

Element	Field Name Prefix
SharedPreferences	PREF_
Bundle	BUNDLE_
Fragment Arguments	ARGUMENT_
Intent Extra	EXTRA_
Intent Action	ACTION_

Figure 2.8.1

- Note that the arguments of a Fragment - Fragment.getArguments() - are also a Bundle. However, because this is a quite common use of Bundles, we define a different prefix for them.

  - Example:

```
// Note the value of the field is the same as the name to avoid
duplication issues
static final String PREF_EMAIL = "PREF_EMAIL";
static final String BUNDLE_AGE = "BUNDLE_AGE";
static final String ARGUMENT_USER_ID = "ARGUMENT_USER_ID";

// Intent-related items use full package name as value
static final String EXTRA_SURNAME = "com.myapp.extras.EXTRA_SURNAME";
static final String ACTION_OPEN_USER =
"com.myapp.action.ACTION_OPEN_USER";
```

Figure 2.8.2

2.9. Line length limit

Code lines should not exceed 100 characters. If the line is longer than this limit there are usually two options to reduce its length:

- Extract a local variable or method (preferable).
- Apply line-wrapping to divide a single line into multiple ones.

There are two exceptions where it is possible to have lines longer than 100:

- Lines that are not possible to split, e.g. long URLs in comments.
- package and import statements.

2.10. Line-wrapping strategies

There is not an exact formula that explains how to line-wrap and quite often different solutions are valid. However there are a few rules that can be applied to common cases.

Break at operators

When the line is broken at an operator, the break comes before the operator. For example:

```
int longName = anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne
        + theFinalOne;
```
Figure 2.10.1

Assignment Operator Exception

An exception to the break at operators' rule is the assignment operator =, where the line break should happen after the operator.

```
int longName =
        anotherVeryLongVariable + anEvenLongerOne - thisRidiculousLongOne
        + theFinalOne;
```
Figure 2.10.2

Method chain case

When multiple methods are chained in the same line - for example when using Builders - every call to a method should go in its own line, breaking the line before the ".".

```
Glide.with(context).load("http://ribot.co.uk/images/sexyjoe.jpg").into(imageView);

Glide.with(context)
         .load("http://ribot.co.uk/images/sexyjoe.jpg")
         .into(imageView)
```
Figure 2.10.3

Long parameters case

When a method has many parameters or its parameters are very long, we should break the line after every comma ",".

```
loadPicture(context, "http://ribot.co.uk/images/sexyjoe.jpg",
mImageViewProfilePicture, clickListener, "Title of the picture");
```
Figure 2.10.4

```
loadPicture(context,
         "http://ribot.co.uk/images/sexyjoe.jpg",
         mImageViewProfilePicture,
         clickListener,
         "Title of the picture")
```
Figure 2.10.5

3. Code Guidelines

3.1. Do not ignore exceptions

You must never do the following:

```
void setServerPort(String value) {
    try {
        serverPort = Integer.parseInt(value);
    } catch (NumberFormatException e) { }
}
```

Figure 3.1.1

While you may think that your code will never encounter this error condition or that it is not important to handle it, ignoring exceptions like above creates mines in your code for someone else to trip over some day. You must handle every Exception in your code in some principled way. The specific handling varies depending on the case. -

3.2. Do not catch generic exception

You should not do this:

```
try {
    someComplicatedIOFunction();        // may throw IOException
    someComplicatedParsingFunction();   // may throw ParsingException
    someComplicatedSecurityFunction();  // may throw SecurityException
    // phew, made it all the way
} catch (Exception e) {                 // I'll just catch all exceptions
    handleError();                      // with one generic handler!
}
```

Figure 3.2.1

3.3. Do not use finalizers

We do not use finalizers. There are no guarantees as to when a finalizer will be called, or even that it will be called at all. In most cases, you can do what you need from a finalizer with good exception handling. If you absolutely need it, define a close () method (or the like) and document exactly when that method needs to be called. See Input Stream for an example. In this case, it is appropriate but not required to print a short log message from the finalizer, if it is not expected to flood the logs. - (Android code style guidelines)

3.4. Fully qualify imports

```
This is bad: import foo.*;
This is good: import foo.Bar;
```

Figure 3.4.1

3.5. Fields definition and naming

Fields should be defined at the top of the file and they should follow the naming rules listed below.

- Private, non-static field names start with m.
- Private, static field names start with s.
- Other fields start with a lower case letter.
- Static final fields (constants) are ALL_CAPS_WITH_UNDERSCORES.

Example:

```
public class MyClass {
    public static final int SOME_CONSTANT = 42;
    public int publicField;
    private static MyClass sSingleton;
    int mPackagePrivate;
    private int mPrivate;
    protected int mProtected;
}
```

Figure 3.5.1

3.6. Treat acronyms as words

Good	Bad
XmlHttpRequest	XMLHTTPRequest
getCustomerId	getCustomerID
String url	String URL
long id	long ID

Figure 3.6.1

3.7. Use spaces for indentation

Use **4 space** indents for blocks:

```
if (x == 1) {
    x++;
}
```

Figure 3.7.1

3.8. Use standard brace style

Braces go on the same line as the code before them.

```
class MyClass {
    int func() {
        if (something) {
            // ...
        } else if (somethingElse) {
            // ...
        } else {
            // ...
        }
    }
}
```

Figure 3.8.1

Braces around the statements are required unless the condition and the body fit on one line.

If the condition and the body fit on one line and that line is shorter than the max line length, then braces are not required, e.g.

```
if (condition) body();
```

Figure 3.8.2

This is **bad:**

```
if (condition)
    body();  // bad!
```

Figure 3.8.3

3.9. Limit variable scope

The scope of local variables should be kept to a minimum. By doing so, you increase the readability and maintainability of your code and reduce the likelihood of error. Each variable should be declared in the innermost block that encloses all uses of the variable.

Local variables should be declared at the point they are first used. Nearly every local variable declaration should contain an initializer. If you do not yet have enough information to initialize a variable sensibly, you should postpone the declaration until you do.

3.10. Logging guidelines

Use the logging methods provided by the Log class to print out error messages or other information that may be useful for developers to identify issues:

- Log.v(String tag, String msg) (verbose)
- Log.d(String tag, String msg) (debug)
- Log.i(String tag, String msg) (information)
- Log.w(String tag, String msg) (warning)
- Log.e(String tag, String msg) (error)

As a general rule, we use the class name as tag and we define it as a static final field at the top of the file. For example:

```
public class MyClass {
    private static final String TAG = MyClass.class.getSimpleName();

    public myMethod() {
        Log.e(TAG, "My error message");
    }
}
```

Figure 3.10.1

VERBOSE and DEBUG logs **must** be disabled on release builds. It is also recommended to disable INFORMATION, WARNING and ERROR logs but you may want to keep them enabled if you think they may be useful to identify

issues on release builds. If you decide to leave them enabled, you have to make sure that they are not leaking private information such as email addresses, user ids, etc.

To only show logs on debug builds:

```
if (BuildConfig.DEBUG) Log.d(TAG, "The value of x is " + x);
```

Figure 3.10.2

3.11. Class member ordering

There is no single correct solution for this but using a **logical** and **consistent** order will improve code learnability and readability. It is recommendable to use the following order:

1. Constants
2. Fields
3. Constructors
4. Override methods and callbacks (public or private)
5. Public methods
6. Private methods
7. Inner classes or interfaces

Example:

```
public class MainActivity extends Activity {

    private static final String TAG =
MainActivity.class.getSimpleName();

    private String mTitle;
    private TextView mTextViewTitle;

    @Override
    public void onCreate() {
        ...
    }

    public void setTitle(String title) {
    mTitle = title;
    }

    private void setUpView() {
        ...
    }

    static class AnInnerClass {

    }
}
```

Figure 3.11.1

3.12. Parameter ordering in methods

When programming for Android, it is quite common to define methods that take a Context. If you are writing a method like this, then the Context must be the first parameter.

The opposite case are callback interfaces that should always be the last parameter.

Examples:

```
// Context always goes first
public User loadUser(Context context, int userId);

// Callbacks always go last
public void loadUserAsync(Context context, int userId, UserCallback
callback);
```

Figure 3.12.1

3.13. Use self closing tags

When an XML element does not have any contents, you must use self closing tags.

This is good:

```
<TextView
    android:id="@+id/tvProfile"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

Figure 3.13.1

This is bad :

```
<!-- Don\'t do this! -->
<TextView
    android:id="@+id/tvProfile"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" >
</TextView>
```

Figure 3.13.2

3.14. Managing Images

- For loading Image from Server Glide Library is used. Glide manages cache management

```
Glide.with(context)
        .load("http://ribot.co.uk/images/sexyjoe.jpg")
        .centerCrop()
        .diskCacheStrategy(DiskCacheStrategy.ALL)
        .placeholder(R.drawable.shape_place_holder)
        .error(R.drawable.shape_place_holder)
        .into(imageView)
```

Figure 3.14.1

3.15. Managing Libraries

- A libraries that are being used in project are organized in build.gradle file of a project
- Libraries must be defined with their repository name. There should be section as well as per the group of the library.
- Libraries belonging to same group will be under the same section and there should be difference between the other group libraries.
- Group libraries version should be managed with common parameter.

```
compile "com.android.support:cardview-v7:${appCompatVersion}"
compile "com.android.support:recyclerview-v7:${appCompatVersion}"
compile "com.android.support:appcompat-v7:${appCompatVersion}"
compile "com.android.support:design:${appCompatVersion}"
compile "com.android.support:customtabs:${appCompatVersion}"
compile 'com.android.support:multidex:1.0.2'

compile('com.bignerdranch.android:expandablerecyclerview:3.0.0-RC1') {
    exclude module: 'recyclerview-v7'
}

// View Binding Libraries
compile 'com.jakewharton:butterknife:8.5.1'
annotationProcessor 'com.jakewharton:butterknife-compiler:8.5.1'

// Networking Libraries
compile 'com.google.code.gson:gson:2.8.2'
compile "com.squareup.retrofit2:retrofit:${retrofitVersion}"
compile "com.squareup.retrofit2:converter-gson:${retrofitVersion}"
compile 'com.squareup.okhttp3:logging-interceptor:3.6.0'
```

Figure 3.15.1

4. Data Base

4.1. Normalization

- Normalization is a database design technique which organizes tables in a manner that reduces redundancy and dependency of data.

SYSTEM: Hospital		**DATE:** / /	**AUTHOR:**
Source ID No.:		**Name of Source:** Prescription Record	
UNF	**1NF**	**2NF**	**3NF**
Pat No Surname Forename Ward No Ward Name Presc Date Med Code Med Name Dosage Lgth Treat	Pat No Surname Forename Ward No Ward Name Pat No Presc Date Med Code Med Name Dosage Lgth Treat	Pat No Surname Forename Ward No Ward Name Pat No Presc Date Med Code Dosage Lgth Treat Med Code Med Name	Pat No Surname Forename Ward No Ward No Ward Name Pat No Presc Date Med Code Dosage Lgth Treat Med Code Med Name

Figure 4.1.1

4.2. Schema & Table

- It divides larger tables to smaller tables and links them using relationships.

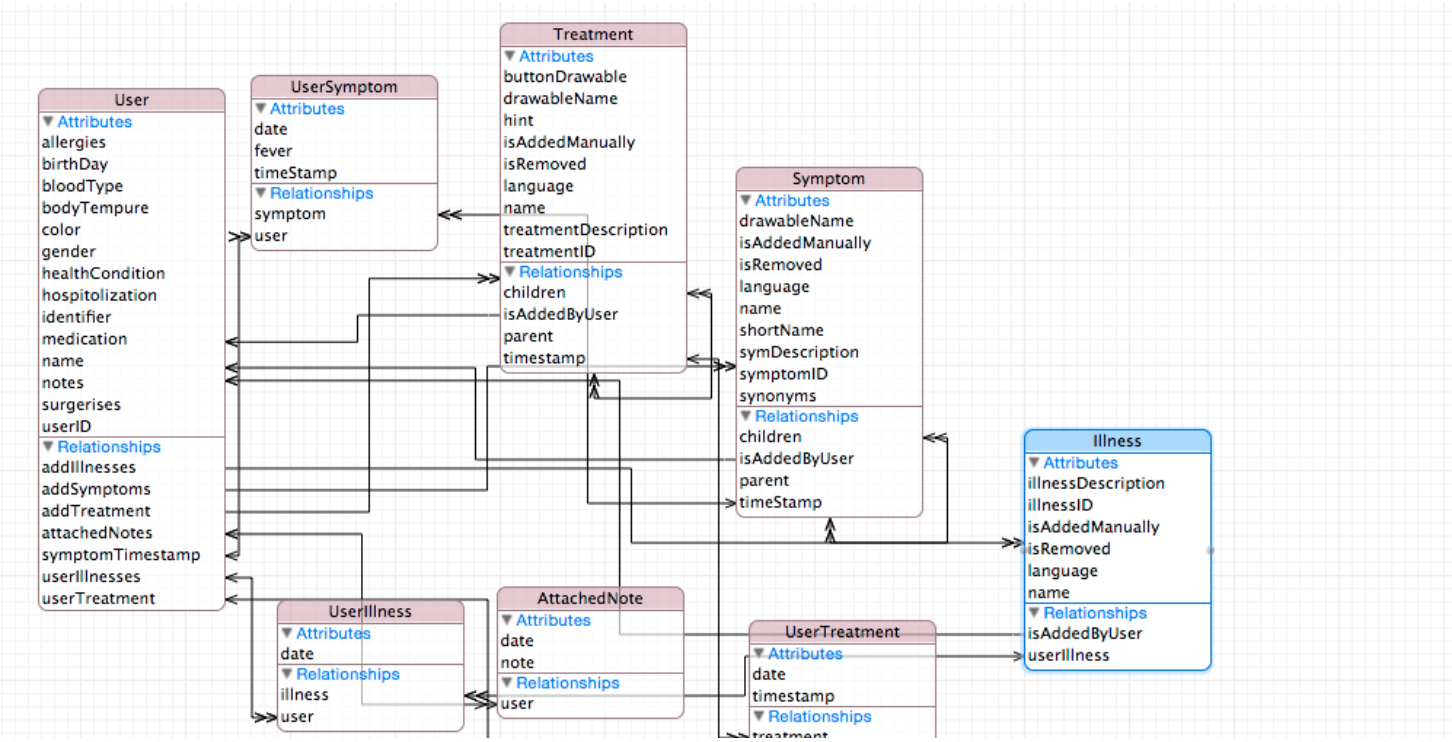

Figure 4.2.1

5. Commenting

- Inline Comment:
    - An inline comment will be used when any methods contain the main base logic which are important or for a complex logic which has been used to get some specific result.
    - In below example, an inline comment has been shown for code which has been used to apply a font style.

Inline Comment

```
public void applyFont(TextView textView,AttributeSet attrs) {
    if (attrs != null) {
        Context context = textView.getContext(); // getting contect from view
        TypedArray styledAttributes = context.getTheme().obtainStyledAttributes(attrs, R.styleable.customFont,  defStyleAttr: 0,  defStyleRes: 0);
        String fontPath = styledAttributes.getString(R.styleable.customFont_fontFamily); // getting attribute
        if (!TextUtils.isEmpty(fontPath)) {
            Typeface typeface = getTypeface(context, fontPath);
            if (typeface != null) {
                textView.setTypeface(typeface); //Applying font on textview
            }
        }
        styledAttributes.recycle();
    }
}
```

Figure 5.1

- Outer Comment:
    - An outline comment will be used outer side of any methods which will indicate the purpose of the class method, input parameters to be passed, output result return by the method and exception which can be thrown by a method.

Outer Comment example

```
/**
 * Gets a Typeface from the cache. If the Typeface does not exist, creates it, cache it and returns it.
 *
 * @param context a Context
 * @param path    Path to the font file in the assets folder. ie "fonts/MyCustomFont.ttf"
 * @return the corresponding Typeface (font)
 * @throws RuntimeException if the font asset is not found
 */
public Typeface getTypeface(Context context,String path) throws RuntimeException {
    Typeface typeface = mFontCache.get(path);
    if (typeface == null) {
        try {
            typeface = Typeface.createFromAsset(context.getAssets(),  path: "fonts/" + path);
        } catch (RuntimeException exception) {
            String message = "Font assets/" + path + " cannot be loaded";
            throw new RuntimeException(message);
        }
        mFontCache.put(path, typeface);
    }
    return typeface;
}
```

Figure 5.2

6. Image Loading

- We are using Glide image loading library which is recommended by Google. It has been used in many Google open source project as well. It provides an animated GIF support and handles image loading/caching
- In a build.gradle file of project a required library needs to be added

```
dependencies {
  compile 'com.github.bumptech.glide:glide:3.8.0'
}
```

- Basic usage of it is much simpler

```
Glide.with(context)
    .load("http://via.placeholder.com/300.png") // A image url which needs to load.
    .into(ivImage);  // Ab image object in which image needs to load.
```

- A glide library can be also used with an advance use like crop an image, display a placeholder image until an image load, resize an image to fix size.
- An example to load an image with a placeholder an error image as given below. A placeholder image will be displayed until the image loads and if there is in interruption occur e.g network and other then it will load an error image.

```
Glide.with(context)
    .load("http://via.placeholder.com/300.png")
    .placeholder(R.drawable.ic_placeholder)
    .error(R.drawable.ic_imagenotfound)
    .into(ivImg);
```

Figure 6.1

- An example to crop an image

```
Glide.with(context)
    .load("http://via.placeholder.com/300.png")
    .centerCrop() //A cropping property to crop an image from a center.
    .into(ivImg);
```

Figure 6.2

- An example to resize an image

```
Glide.with(context)
    .load("http://via.placeholder.com/300.png")
    .override(300, 200) // A property to resize an image to specific dimension.
    .into(ivImg);
```

Figure 6.3

7. API Calling Library

- We are using Retrofit as it is one of the most reliable, fast and easy to use Networking library. We have made our own Classes to reduce the long API calling code and handling the response in better way.
- It is a type-safe REST client for Android built by Square.
- Using this tool android developer can make all network stuff much easier.
- In a build.gradle file of project a required library needs to be added

```
// Newtorking LIbraris
compile 'com.google.code.gson:gson:2.8.2'
compile "com.squareup.retrofit2:retrofit:${retrofitVersion}"
compile "com.squareup.retrofit2:converter-gson:${retrofitVersion}"
compile 'com.squareup.okhttp3:logging-interceptor:3.6.0'
```

Figure 7.1

- In APIResponseListener interface it having 3 methods onPreExecute where we can write a code which needs to execute before calling an api e.g. Show a loading indicator before main logic of code execution start.
- onSuccess method will be called only once all the required conditions to get a data from server has been satisfied and api request goes successful.
- onError method will be called when an api breaks in between due to a server not reachable or internet issue or if a wrong input has been passed e.g. In a forgot password api if we pass an email which hasn't been registered with an app then it will return an error message: "No such user found registered with this email."

```
NetworkHelper changeStatusHelper = new NetworkHelper(mContext, ApiInterface.ENDPOINT_CHANGE_AVAILIBILITY_API);
changeStatusHelper.setResponseListener(new ResponseListener() {

    @Override
    public void onPreExecute() { showProgress(getString(R.string.str_please_wait)); }

    @Override
    public void onSuccess(int statusCode, JSONObject jsonObject) {
        stopProgress();
        Logger.Error("Success");
        if (HelperFunctions.isErrorCritical(jsonObject, mContext) == null) {
            session.setFlagFromKey(SessionManager.KEY_USER_AVAILABLE, swAvailability.isChecked());
        } else {
            new ErrorDialog(getActivity(), HelperFunctions.isErrorCritical(jsonObject, mContext)).show();
        }
        updateDriverAvailibility();
    }

    @Override
    public void onError(int statusCode, String message) {
        stopProgress();
        Logger.Error("Error");
        session.setFlagFromKey(SessionManager.KEY_USER_AVAILABLE, !swAvailability.isChecked());
        swAvailability.setChecked(!swAvailability.isChecked());
        mDailogs.showSnackBarForLong(mRecyclerView, message);
    }
});

changeStatusHelper.executeRequest(requestModel);
```

Figure 7.2