Android Project Guidelines

1. Project Guidelines

1.1 Android Studio

Please use this standard settings.jar that includes common standard styles like spacing, formatting lines, and Macros that does this three important actions : Formatting lines, Optimized imports, Field and Class member ordering

1. Open your AS - Go to File - Import Settings - Choose settings.jar
2. Make sure to Do Command + S to trigger Macros(Formatting lines, Optimized imports, Reordering) for any .java and .xml files changes

1.2 Credential Keys

For security reason, all credential keys and build signature are stripped out from repo. It need to be defined manually as described below :

1. Open/Create the generic gradle.properties on your machine. (the file should be located on ~/.gradle/gradle.properties)
2. Write following code

```
certificateAppLocation=[your_key_location]
certificateAppStorePassword=[your_store_password]
certificateAppReleaseKeyAlias=[your_key_alias]
certificateAppReleaseKeyPassword=[your_key_password]
```

3. Copy your .jks file into the project folder

1.3 Git

Please follow git flow branching model http://nvie.com/posts/a-successful-git-branching-model/

1.4 Project Structure

1.4.1 Java Structure

For the high level hierarchy, the project is package by layer, for instance : app, domain, data

But, For the presentation/app level, the project is package by feature/screen, for instance : login, home, etc.

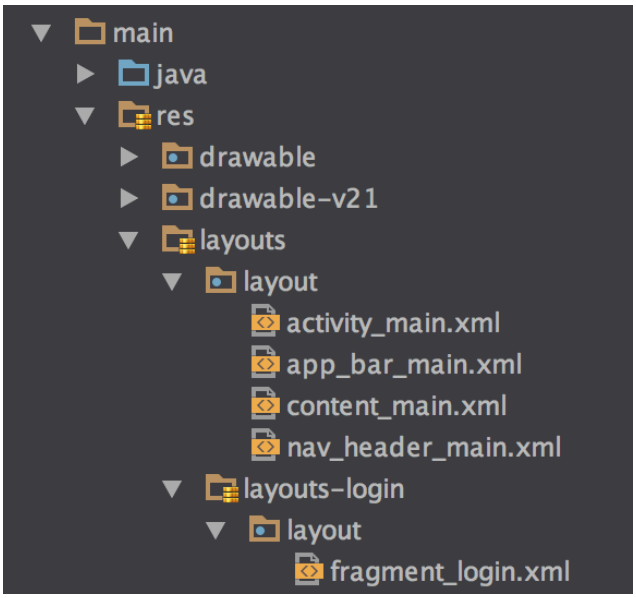
1.4.2 Layout Structure

For layout source directory, please use nesting layout by putting layout files in separate folder respectively. Steps below :

1. Open your AS - Change the Hierarchy View into Project
2. Add this lines to app/build.gradle for example you want to add login : src/main/res/layouts/layouts-login

```
sourceSets {
    main {
        res.srcDirs = [
                'src/main/res/layouts/layouts-login',
                'src/main/res/layouts',
                'src/main/res'
        ]
    }
}
```

3. Then, Create folder with the same name under layouts folder



1.5 File Naming

1.5.1 Class Files

Any classes that you define should be named using `UpperCamelCase`, for example:

```
BaseActivity, BaseFragment, NetworkManager, UserFragment
```

Any classes extending an Android framework component should always end with the component name, for example:

```
DraggableImageView, SignUpActivity, RateAppDialog
```

1.5.2 Resources Files

When naming resource files you should be sure to name them using lowercase letters and underscores, for example:

```
activity_main, fragment_user, item_product
```

1.5.3 Layout Files

When naming layout files, they should be named starting with the name of the Android Component that they have been created for, for example:

Component	Class Name	Layout Name
Activity	MainActivity	activity_main
Fragment	MainFragment	fragment_main
Dialog	RateDialog	dialog_rate
Widget	UserProfileView	view_user_profile

2. Code Guidelines

2.1 Java Language Rules

2.1.1 Never Ignore Exceptions

Avoid not handling exceptions and leave it empty, for example :

```
public void setUserId(String id) {
    try {
        mUserId = Integer.parseInt(id);
    } catch (NumberFormatException e) {
  // bad
 }
}
```

This gives no information to both the developer and the user, making it harder to debug and could also leave the user confused if something goes wrong.

Here how to handle the error appropriately by:

- Showing a message to the user notifying them that there has been an error
- Setting a default value for the variable if possible
- Throw an appropriate exception

2.1.2 Never Catch Generic Exceptions

Catching exceptions generally should not be done:

```
try {
 context.startActivity(intent);
} catch (Exception e) {
 handleError();
}
```

It is very dangerous because it means that Exceptions you never expected (including RuntimeExceptions like ClassCastException) get caught in application-level error handling. It obscures the failure handling properties of your code, meaning if someone adds a new type of Exception in the code you're calling, the compiler won't help you realize you need to handle the error differently. In most cases you shouldn't be handling different types of exception the same way. - taken from the Android Code Style Guidelines

2.1.3 Don't keep unused imports

Sometimes removing code from a class can mean that some imports are no longer needed. If this is the case then the corresponding imports should be removed alongside the code.

2.1.4 Use TODO Comments

Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect. TODOs should include the string TODO in all caps, followed by a colon:

```
// Bad.
// TODO: Implement request backoff.

// Good.
// TODO(George Washington): Implement request backoff.
```

A TODO isn't a bad thing - it's signaling a future developer (possibly yourself) that a consideration was made, but omitted for various reasons. It can also serve as a useful signal when debugging.

And TODOs should have owners, otherwise they are unlikely to ever be resolved.

2.2 Java Style Rules

2.2.1 Never use Hungarian notation

Every day new Java code is written for Android apps and libraries which is plagued with an infectious disease: Hungarian notation. The proliferation of Hungarian notation on Android is an accident and its continued justification erroneous. - Jake Wharton

```
private int mMonthOfYear; // bad

private int monthOfYear; // better
```

2.2.2 View Field Naming

Our naming convention for widget in layout is `{initial_widget}_{context1}_{contextN}` and for widget in java code is the same order but with camelCase, for example:

Component	Java	Layout XML
TextView	tvDisclaimer	tv_disclaimer
EditText	etUsername	et_username
Button	btnLogin	btn_login
RelativeLayout	rlBackground	rl_background

2.2.3 Avoid naming with container types

Leading on from the above, we should also avoid the use of container type names when creating variables for collections. For example, say we have an arraylist containing a list of userIds:

Do:

```
List<String> userIds = new ArrayList<>();
```

Don't:

```
List<String> userIdList = new ArrayList<>();
```

To avoid confusion If and when container names change in the future.

2.2.4 Treat Acronyms as words

Any acronyms for class names, variable names etc should be treated as words - this applies for any capitalisation used for any of the letters, for example:

Do	Don't
setUserId	setUserID
String uri	String URI
int id	int ID
parseHtml	parseHTML
generateXmlFile	generateXMLFile

2.2.5 Use standard Brace Style

Please use 1TBS(One true brace style), for example :

```
if(condition) {
 action(); // This is 1TBS, do this!
}


if(condition)
 action(); // This is Bad!


if(condition) action(); // Better, only if the line is shorter than the
max line length
```

2.2.6 Ternary Operators

When appropriate, ternary operators(usually called Elvis Operator) can be used to simplify operations.

For example, this is easy to read:

```
userStatusImage = signedIn ? R.drawable.ic_tick : R.drawable.ic_cross;
```

and takes up far fewer lines of code than this:

```
if (signedIn) {
    userStatusImage = R.drawable.ic_tick;
} else {
    userStatusImage = R.drawable.ic_cross;
}
```

2.2.7 Field Ordering

Any fields declared at the top of a class file should be ordered in the following order:

1. Enums
2. Constants
3. Dagger Injected fields
4. Butterknife View Bindings
5. public global variables
6. default variables
7. protected variables
8. private variables

For example:

```
public static enum {
    ENUM_ONE, ENUM_TWO
}

public static final String TAG = "MainActivity";

@Inject SomeAdapter someAdapter;

@BindView(R.id.text_name) TextView nameText;

public String someString;

protected String someString;

String someString;

private String someString;
```

Using this ordering convention helps to keep field declarations grouped, which increases both the locating of and readability of said fields.

2.2.8 Method paramater Ordering

When defining methods, parameters should be ordered to the following convention:

```
public void loadPost(Context context, int postId, Callback callback);
```

Context parameters always go first and **Callback** parameters always go last.

2.2.9 Method Chaining

When it comes to method chaining, each method call should be on a new line.

Don’t do this:

```
Glide.with(context).load("someUrl").into(imageView);
```

Instead, do this:

```
Glide.with(context)
     .load("someUrl")
     .into(imageView);
```

2.2.10 Method Spacing

There only needs to be a single line space between methods in a class, for example:

Do this:

```
public void setUserName(String name) {
    // Code
}

public String getUserName() {
    // Code
}
```

Not this:

```
public void setUserName(String name) {
    // Code
}


public String getUserName() {
    // Code
}
```

2.2.11 RxJava Chaining

When chaining Rx operations, every operator should be on a new line, breaking the line before the period, for example:

```
return dataManager.getPost()
            .concatMap(new Func1<Post, Observable<? extends Post>>() {
                @Override
                 public Observable<? extends Post> call(Post post) {
                     return mRetrofitService.getPost(post.id);
                 }
            })
            .retry(new Func2<Integer, Throwable, Boolean>() {
                 @Override
                 public Boolean call(Integer numRetries, Throwable
throwable) {
                     return throwable instanceof RetrofitError;
                 }
            });
```

This makes it easier to understand the flow of operation within an Rx chain of calls.

2.2.12 Butterknife Event Listeners

Where possible, make use of Butterknife listener bindings. For example, when listening for a click event instead of doing this:

```
mSubmitButton.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        // Some code here...
    }
  };
```

Do this :

```
@OnClick(R.id.button_submit)
public void onSubmitButtonClick() { }
```

2.3 XML Style Rules

2.3.1 Use self closing tags

When a View in an XML layout does not have any child views, self-closing tags should be used.

Do:

```
<ImageView
    android:id="@+id/iv_user_avatar"
    android:layout_width="90dp"
    android:layout_height="90dp" />
```

Don't:

```
<ImageView
    android:id="@+id/iv_user_avatar"
    android:layout_width="90dp"
    android:layout_height="90dp">
</ImageView>
```

2.3.2 Attributes Ordering

Ordering attributes not only looks tidy but it helps to make it quicker when looking for attributes within layout files. As a general rule,

1. View Id
2. Style
3. Layout width and layout height
4. Other `layout_` attributes, sorted alphabetically
5. Remaining attributes, sorted alphabetically

For example:

```
<Button
    android:id="@id/button_accept"
    style="@style/ButtonStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_alignParentStart="true"
    android:padding="16dp"
    android:text="@string/button_skip_sign_in"
    android:textColor="@color/bluish_gray" />
```

3. Gradle Styles

3.1 Versioning

Where applicable, versioning that is shared across multiple dependencies should be defined as a variable within the build.gradle project scope. For example:

```
ext {

    versions = [
            // App dependencies
            supportLib         : "25.0.0",
            butterknife        : "8.4.0",

            //Logger
            timber             : "4.3.1",

            // Rx
            rxjava             : "1.2.1",
            rxbinding          : "0.4.0",
            rxlint             : "1.0",
            rxjavaProguardRules: "1.2.0.0",

            // Network
            retrofit2          : "2.1.0",
            okHttp3            : "3.4.1",
            gson               : "2.7",
    ]
}
```

and use it in build.gradle app scope :

```
dependencies {
    compile "com.android.support:appcompat-v7:$versions.supportLib"
    compile "com.android.support:cardview-v7:$versions.supportLib"
    compile "com.android.support:design:$versions.supportLib"

    compile "com.jakewharton:butterknife:$versions.butterknife"
    apt "com.jakewharton:butterknife-compiler:$versions.butterknife"

    //Logger
    compile "com.jakewharton.timber:timber:$versions.timber"

    // Rx
    compile "io.reactivex:rxjava:$versions.rxjava"
    compile "nl.littlerobots.rxlint:rxlint:$versions.rxlint"
    compile "com.jakewharton.rxbinding:rxbinding:$versions.rxbinding"
    compile
"com.artemzin.rxjava:proguard-rules:$versions.rxjavaProguardRules"
}
```

This makes it easy to update dependencies in the future as we only need to change the version number once for multiple dependencies.