Welcome!

This guide explains how the android smart phone app,

ComposeMVVMSample, was designed, how it is expected to be used, by android smart phone app developers, such as yourself and how it works under the covers.

This app was designed to help anyone use and understand how jetpack compose and mvvm apps are supposed to be created.

It is expected that, after renaming the sample, you will have the start of a well designed app.

ComposeMVVMSample has functionality and structure to help with common issues that all projects need, such as:

- communication between the user interface and the data being displayed, in such a way as to not do long running operations on the ui thread, which would lock up the app.
- Logging and other custom operations, via extension functions, **Extensions.kt**
- coroutine management
- Prefs.kt class to handle persisting data easily, when a database is not needed.
- Helpful, custom composeables **ui\composeables\core**
- repo repository to handle long running operations, such as fetching data from the web, other apps or just long running operations.

Follow the steps on the next few pages to get everything installed.

Quick Start

1) Open Android Studio and Clone Sample Project

a) Open Android Studio.
b) Make sure no projects are open.
c) Press the “Clone Repository” button.
d) Enter url and destination directory.

Note: **Parent Directory** does not have to be: *C:\github*
and **Name** does not have to be *MySample,*
use any name for your new project, just no spaces.

Enter the following:
URL: https://github.com/pstorli/ComposeMVVMSample.git
Directory: C:\github\MySample

e) File → Close Project.

2) Adjust Project to be your own.

Using this app, this sample project, as the foundation for any Android App that you want to create is kind of like buying a furnished house with plumbing and electricity versus what you get when you say “New Project” in Android studio, a cheaper house that has only walls and a roof, no plumbing, electricity or furniture!

a) Open a file explorer and go to the location where you downloaded the “MySample” application,

For me it was dir: **C:\github\MySample**

b) Delete the **.git** directory.

It is hidden, so you may need to
show hidden files and dirs, in your file explorer.

c) Adjust the strings XML file.

1) Open the strings XML file, in a text editor

For me it was dir: C:\github\MySample\app\src\main\res\values**Strings.xml**

2) Find the string resource typically named **app_name** and change its value to your new app name.

a) **Is:**

<string name="app_name">**ComposeMVVMSample**</string>

b) **change to:**

<string name="app_name">**MySample**</string>

d) Update the **build.gradle** file. You must update the **application ID** in your module's build.gradle file.

Locate build.gradle. My gradle file was at:

C:\github\MySample\app\build.gradle.kts

Modify applicationId. Find the android block and update the applicationId to your new, full package name.

Before:

```
android {
    namespace = "com.pstorli.composemvvmsample"
        compileSdk {version = release(36)}

      defaultConfig {
        applicationId = "com.pstorli.composemvvmsample"
        minSdk = 24
        targetSdk = 36
        versionCode = 1
        versionName = "1.0"
```

After:

```
    android {
    namespace = "com.mycompany.mysample"
        compileSdk {version = release(36)}

      defaultConfig {
        applicationId = "com.mycompany.mysample"
        minSdk = 24
        targetSdk = 36
        versionCode = 1
        versionName = "1.0"
```

e) Check the **AndroidManifest.xml**

While the steps above usually handle everything, it's good practice to ensure the package attribute in your **AndroidManifest.xml** file is pointing to the new package name.

1. **Locate AndroidManifest.xml:**

   Open up file:

   C:\github\MySample\app\src\main\AndroidManifest.xml

2. **Verify the package:** Make sure the package attribute at the top of the file reflects your new package name.

   Change, in two places, from:

   android:theme="@style/Theme.ComposeMVVMSample"

   to:

   android:theme="@style/Theme.MySample"

f) Adjust the idea workspace file for your project.

1) Open up file:

   C:\github\MySample\.idea**workspace.xml**

2) Replace

   "**ComposeMVVM**" with "**My**"

3) Then, Replace

   "**pstorli** with "**mycompany**"

f) Adjust the, hidden, idea name file for your project.

1) Open up file:

C:\github\MySample\.idea**.name**

2) Replace

"**ComposeMVVMSample**" with "**MySample**"

g) Rename directories.

Note: **mycompany, mysample** and **MySample**, are just suggested, names should be **case sensitive** and **have no spaces or other special characters**, so use whatever names you like.

1) Rename project name directory:

from:

C:\github**MySample**\app\src\main\java\
com\pstorli**composemvvmsample**

to:

C:\github\MySample\app\src\main\java\
com\pstorli**mysample**

2) Rename company name directory:

from:

C:\github**MySample**\app\src\main\java\
com\pstorli

to:

C:\github**MySample**\app\src\main\java\
com**mycompany**

3) New project directory should look like:

C:\github**MySample**\app\src\main\java\
com**mycompany****mysample**

h) Open MySample Project in Android Studio.

1) Edit → Find → Replace in files.
"**pstorli**" with "**mycompany**"

2) Edit → Find → Replace in files.
"**composemvvmsample**" with "**mysample**"

3) Edit → Find → Replace in files, case sensitive.
"ComposeMVVMSample" with "MyS**ample**"

4) Select: File → **Invalidate Caches and Restart**

i) Integrate version control with github

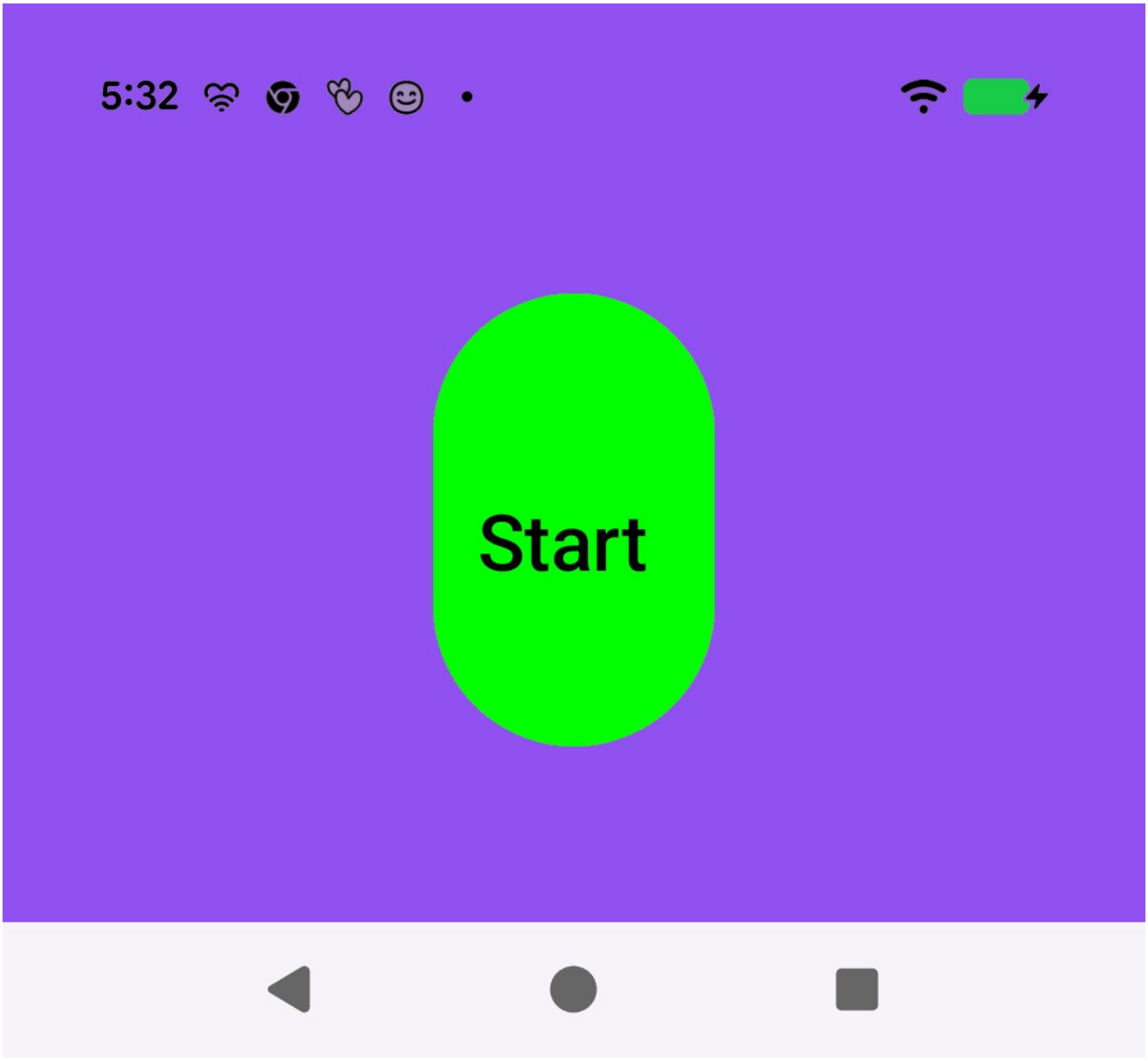
1) Open ***MySample*** Project in Android Studio.

2) Share Project on github: **(Version Control System)**

Go to the menu bar and **select**

Git → GitHub → Share Project on GitHub

3) The Sample application:



not to scale

The sample application, MySample, shown here in the ready state.

If you press the **start** **button**, the **background color**, **purple** here,

will start changing, every few seconds.

Nothing earth shattering, but under the covers, some amazing things are going on.

It is important to understand that the UI thread must never be stopped or blocked, or your app will display the deadly "***app not responding***" dialog.

The app starts out in MainActivity.create, on the ui thread.

The first thing we do is instantiate the **ViewModel** class,

val viewModel: ViewModel **by** *viewModels*()

Which will be passed to our composeables on the constructor.

StartScreen (**viewModel)**

The ViewModel class is used as a central location to store project data.

The composeables use the viewModel to get data they need, and the repository uses the viewModel class as a place to store data perhaps retrieved in the background.

Which makes data sharing easy, because everyone knows where it is.

And, you get added protection for storing your data in the viewMoidel,

Your data can survive context switches, such as the phone being rotated from portrait to landscape.

Only quick things should be done on the ui thread,

such as asking the viewModel for data, no logic performed here.

What does happen is that if the viewModel data is modified, the ui layer, composeables, are notified and update themselves accoredingly.

Logic can go in the viewModel class, but can also be delegated by creating viewModel children, such as these viewModel variables:

```
// Used to request long running data
// Repository stuff, do long running things in the background.
var repo = Repo (this)

// Used to run coroutines in the background.
var vh    = VMHelper (this)

// Used to persist data , even after app shut down.
@Suppress("unused")
var prefs = Prefs(app) // Preferences, initialize first
```

1) We launch a composeable, **StartScreen**, in
com.mycompany.mysample.ui.composeables.screens.StartScreen.kt

- When the composeable starts, it sets up to listen to any changes in the view models, backColor variable.

With this line of code:

```
modifier = Modifier.fillMaxSize().background (viewModel.backColor)
```

- The buttons text color is also tied to the viewModels buttonColor

```
containerColor  = Color (viewModel.buttonColor.value)
```

- It then also ties pressing the button on the main screen with executing the

Method **toggleRunning** in class ***VMHelper.kt***

viewModel.vh.toggleRunning()

The method **toggleRunning** toggles the running state of a background task. Let's examine this is greater detail. First the code.

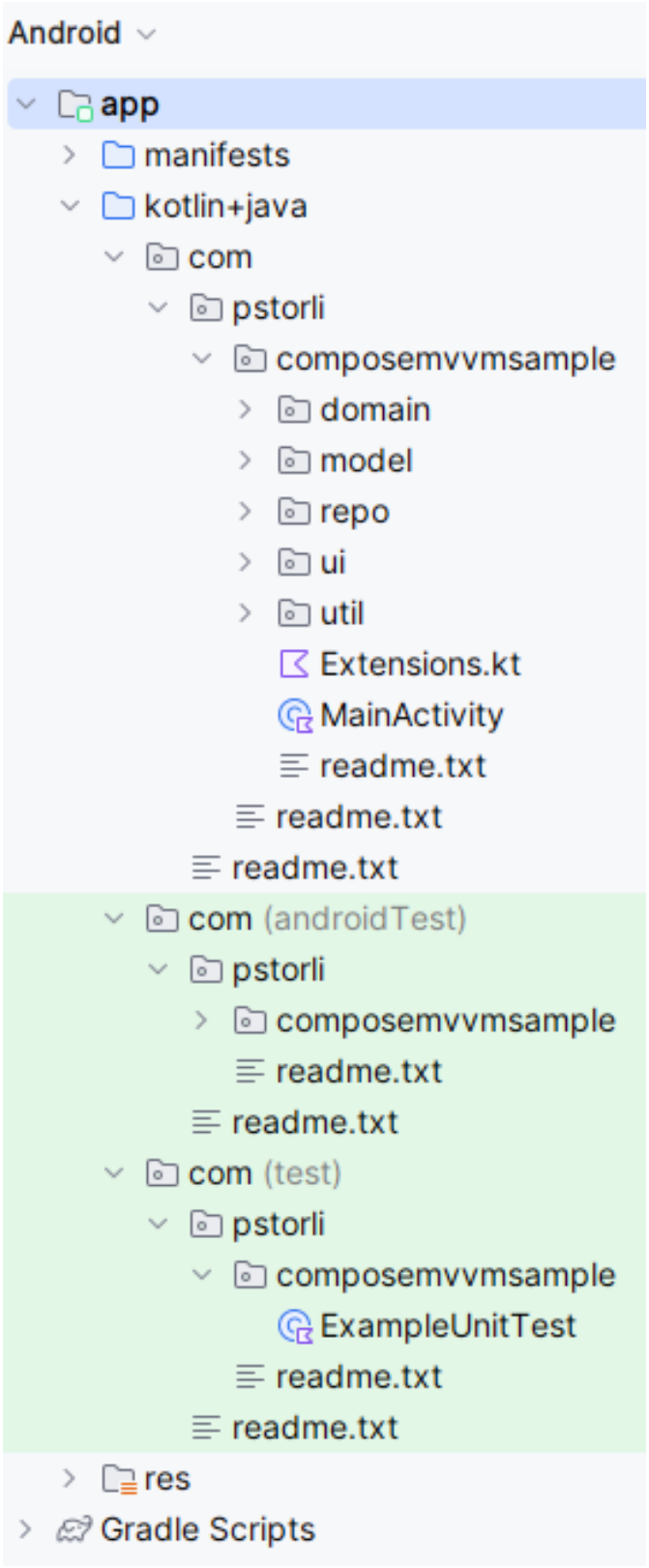
```
fun toggleRunning() {
    // 1 toggle running state
    viewModel.running = !viewModel.running

    // Are we running?
    var word: String

    // 2 Are we running?
    if (!viewModel.running) {
        // 3 Make sure that background thread running.
        // This task throws out random colors at random times.
         // In this app it is simulating outside data trickling in.
        ch.startBackgroundTask()

        // 4 Pressing button now starts background color task.
        word = viewModel.app.getText(R.string.stop).toString()
}
else {
        // 5 Pressing button now stops background color task.
        word = viewModel.app.getText(R.string.start).toString()
}
   // 6 set button background color in a background thread.
  // set the button text.
viewModel.buttonText  = word

// 7 get word color in background.
ch.getWordColorInBackground (word)}
```

In step **1**, when reverse the running state with a not !
In step **2**, we take slightly different paths if we are running or not.
In step **3**, we start background task in a co-routine using class CoHelper,
which stands for coroutineHelper
protected var ch = CoHelper (viewModel)

Note, in steps 4 and 5, we retrieve the color by calliung our repository
on a backgroound coroutine,
viewModel.buttonColor = viewModel.repo.getColorOfWord(start)

We would not normally do this, but wanted to show how to call a background thread and get data from it back. When the background coroutine is done, it notifies the viewModel

In step **4**, if we are running, set button text to stop and button color
to red.
In step **5**, If we are not running, set button text to Start and button color to
green.
In step **6**, set the button text.
In step *7, get word color in background.*



- **domain package**

  Used to hold custom project data

- **model package:**

  handles data src and ui / viewmodel communication

  ViewModel.kt has **viewmodel** which is data source, holds **repo** and **prefs**

  VMHelper.kt holds viewmodel logic.

  CoHelper.kt handles viewmodel /repo communication, via coroutines, used to perform long running operations, that would otherwise lock up the ui thread.

- **repo package**

  Repo.kt for long running functions, done **not on ui thread**!

- **ui package** holds custom, generic, composeable classes core and project screens**StartScreen.kt**

  screens holds project's compose screens. sample start screen, **StartScreen.kt** and

  theme**Theme.kt**

- **util package holds utility classes**

  Prefs.kt class used for persisting data, when a database is overkill.

  Consts.kt class used for project wide constants.

NOTES:

- **App execution starts at**

  MainActivity.onCreate, MainActivity.kt, also holds the viewModel, which holds project data.

  val viewModel: ViewModel by viewModels()

- ***Project root:***
  ComposeMVVMSample\app\src\main\java
  app\kotlin+java\com\pstorli\composemvvmsample

- **To debug,**
  - Left click mouse far left in file, breakpoint will appear as a red dot.
  - Start program with bug icon, top left, of app or Run → Debug menu
  - F7 step into, F8 step over, F9 run to next breakpoint
  - when stopped at a breakpoint, you can view current state of variables.