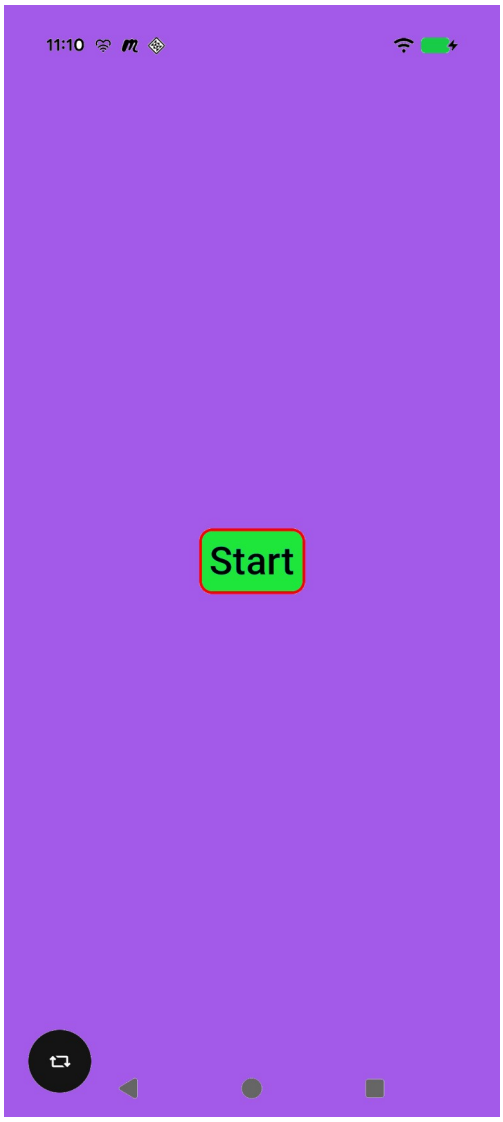
MVVM

If you press the **start** **button**, a background task, not on ui thread, changes the background color, purple here, every few seconds.



MVVM,

the sample application, shown here in the stopped state.

Table of Contents

Quick Start

This guide explains how the android smart phone app,

MVVM, 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.

MVVM 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.

Clone Sample Project

a) Open Android Studio.

If you don't have android studio, download and install it from:
https://developer.android.com/studio
This sample project was built using
Otter 2 feature Drop

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/MVVM.git
Directory: C:\github\MySample

e) File → Close Project.

f) I highly suggest you download WinMerge
to help compare projects, or versions of same project.
https://winmerge.org/downloads

Project Diagram

****

Main Packages

****

- **Domain package:** proj data
- **model package:**

  handles data src and ui / viewmodel communication

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

  VMHelper.kt class holds viewmodel logic.

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

- **repo package**

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

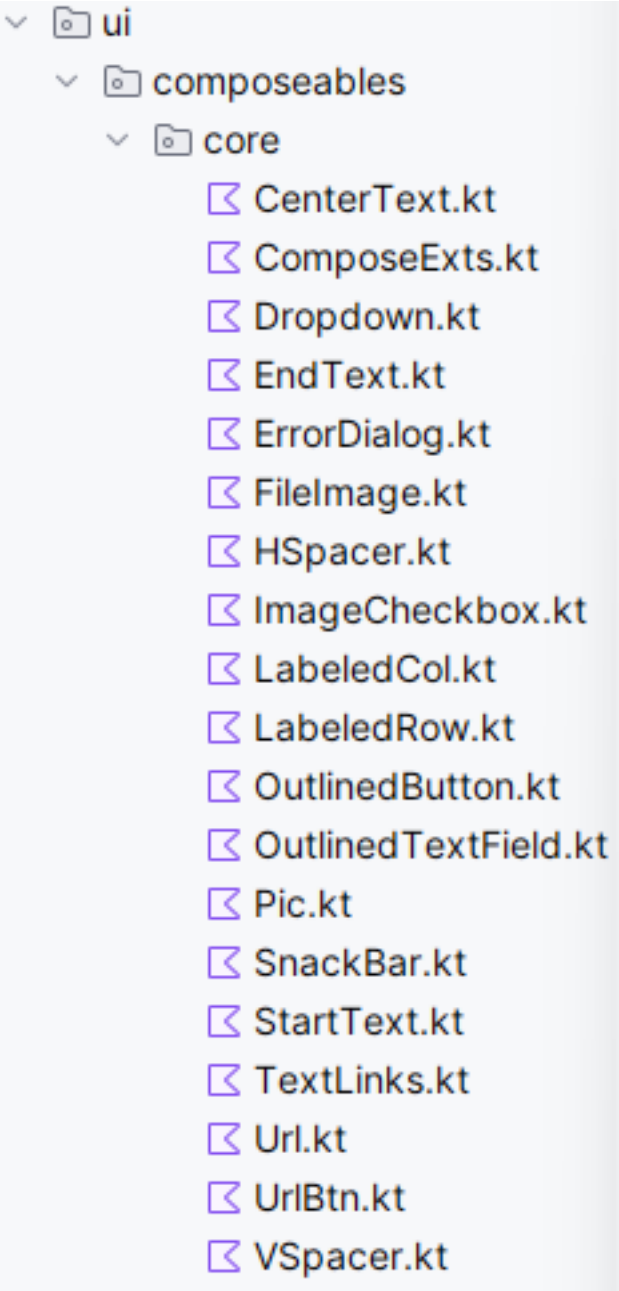
- **res package** holds resources, such as images and text strings

  drawable package Holds project image files, png and jpg

  values package Strings.xml file holds user displayable text, one file per language

ui composeables core package

holds custom, generic, composeable classes for project



Composeables are easy to use and make. Look at these examples in this directory for project independent composeables.

Pay careful attention to how the composeable is created, parameters passed, and how it indicates action, such as what a button press or click does.

The composeable, **OutlinedButton.kt**, is used as the sample button in the composeable **StartScreen.kt.**

Copy a composeable in core, play around with it, make new ones, they are re-usable gui, kind of like a reusable class in kotlin.

- **ui package**

  composeables package

  core package Custom, Generic, composeables

  screens package holds project's compose screens

  StartScreen.kt A composeable sample start screen

  theme package

  Theme.kt A composeable that handles light and **dark** color schemes.

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.

****

Main Files

****

.gitignore

```
/**
 * This file has Files to not save into version control
 */
```

****

build.gradle.kts

```
// The namespace property defines the package name for your app's
// internal generated code like the R class
// (for resources) and BuildConfig class.
namespace = "com.pstorli.mvvm"
// unique identifier for your application on an Android device
applicationId = "com.pstorli.mvvm"

// What min and max android version do you support?
minSdk = 24
targetSdk = 36

// The version this app is on.
versionCode = 1
versionName = "1.0"

// This section is used to import external projects we need.
dependencies {}
```

****

ColHelper.kt

```
/**
 * This class is used to spin off co-routines which call
 * long running functions in the Repo.kt class
 * NOTE: The repo class is private, we don't want it being
 * used by anyone but this class, which knows how to launch
 * co-routines.
 */
```

****

Color.kt

```
/**
 * This class holds the colors used in the app.
 */
```

****

colors.xml

```
/**
 * Hex color codes.
 */
```

****

Consts.kt

```
/**
 * A place to put project constants.
 */
```

****

Extensions.kt

/**
 * *A place to put class extensions.*
 */
A class extension allows you to add functionality to a class that it does not have. For example to add easy exception logging,

The class, a period, then the function namespace

Like this:

```
fun Exception.logError ()
{
   Consts.logError (this.toString())
}

// This is how we use it.
   catch (ex: Exception) {
      ex.logError()
   }
```

****

MainActivity.kt

/**
This class is the applications main activity.

This is the only place we instantiate the ViewModel.kt class.
It is passed to composeables on their constructor.

The function, **onCreate**, in **MainActivity.kt**

```
override fun onCreate(savedInstanceState: Bundle?) {}
```

is called when program starts up and is on the ui thread, so no long running operations here, just call setContent to set the composeable to view,
In this case, **StartScreen.kt**
*/

****

Prefs.kt

/**
* This class can be used to store and retrieve data that persists, even when app and phone are turned off. The only way to "clean" or "clear" the preference data is by un-installing app.
*/

****

readme.txt

/** *A place to store info about developer, project and version changes.*
0000 Changes for Version X:
Changed …
*/

****

Repo.kt

/** This class handles talking to remote data sources to get data in the background, such as images from urls, websites, vpn, or even local database , which is considered slow.

This class is currently designed to be called from class CoHelper.kt
so that it is never called directly from the ui thread.
If request is needed, add methods to CoHelper.kt to get to the Repo.kt method you want to talk to.

The function getColorOfWord, normally does not need to be here, it is not long running. But it have it here as an example of how a longer running routine should be handled.

fun getColorOfWord (word: String): Color

****

StartScreen.kt

/**
This is an example of a typical composeable.
Note that the viewModel is passed in on the constructor.

// This line both sets the background color, and listens for
// any future changes in the background color in the viewModel

```
.background (viewModel.backColor),

OutlinedButton (
   name        = viewModel.btnText,
   backColor   = Color (viewModel.btnBackColor.value),
   borderColor = Color.Red,

   // Toggle running state when clicked
   onClick     = {
      // Tell the view model to toggle the running state.
      viewModel.vh.toggleRunning()
   })
```

The **toggleRunning**() call, tells the viewModel that we want to toggle the running state. The ui should only handle two things, displaying the data and notifying the viewModel of ui actions.

We want to keep logic out of the ui, as much as possible.
*/

****

strings.xml

```
/**
 Where to put strings being displayed to the user.
  Useful for internationalization.
  Strings used for logging need not go here.
 */
```

****

Theme.kt

```
/**
 * Handles switching between dark and light modes for colors.
 */
```

****

themes.xml

```
/**
 * Where to declare theme to use.
 */
```

****

Type.kt

```
/**
 * Typography, font family, font size, font weight, line height and spacing
 */
```

****

ViewModel.kt

```
/**
   Holds project data, in a way that survives context switches,
    such as changing the phone's orientation
 */
```

****

VMHelper.kt

```
/**
 * This class is used for logic that is short lived, as we are on the ui thread here. The reason is that I like to keep the viewModel as solely a data repository, and as little logic, functions, as possible.
 */
```

Make Project 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">**MVVM**</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.mvvm"
        compileSdk {version = release(36)}

      defaultConfig {
        applicationId = "com.pstorli.mvvm"
        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.

NOTE: If you are happy with '**Current**' being the **theme name**, skip this step.

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.Current"

   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

   "**MVVM**" with "**MySample**"

3) Then, Replace

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

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

Note: If file does not exist, skip this step,

but your file explorer may be hiding it from you.

1) Open up file:

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

2) Replace

"**MVVM**" 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**mvvm**

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.
"**mvvm**" with "**mysample**"

3) Edit → Find → Replace in files, case sensitive.
"MVVM" with "My**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

More Project “Code” details

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 calling our repository

on a background 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.*

Notes:

- **App execution starts at**

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

  val viewModel: ViewModel by viewModels()

- ***Project root:***
  MVVM\app\src\main\java
  app\kotlin+java\com\pstorli\mvvm

- **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.