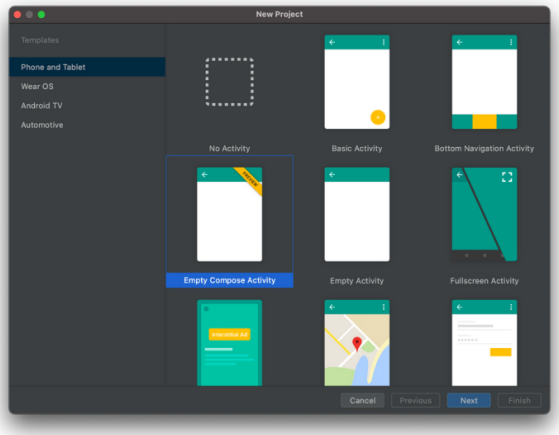
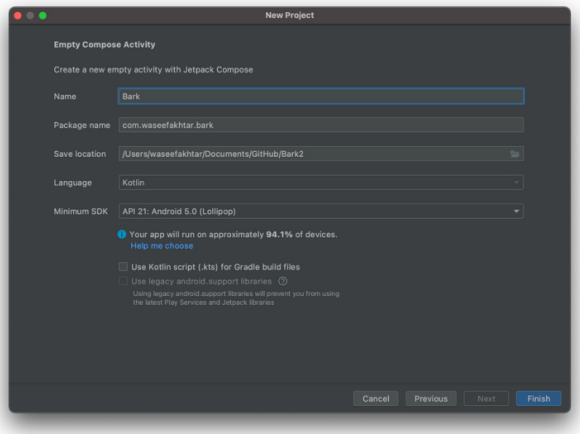
Jetpack compose : An easy way to recyclerView

Project Setup

To get things started, here's what you do:

- Open a new project.
- Select an *Empty Compose Activity* Project Template and give your app a name. This would create an empty Android project.





Running the project

Before we start writing our first line of Jetpack Compose code, let's run our current project set up by AS for us. Since we're using unstable/preview versions of Jetpack Compose and AS, chances are, there are some unknown issues that you might encounter along the way. So it's always a good idea to run your project after each change.

In my case here, after running the project for the first time, I ran into this:

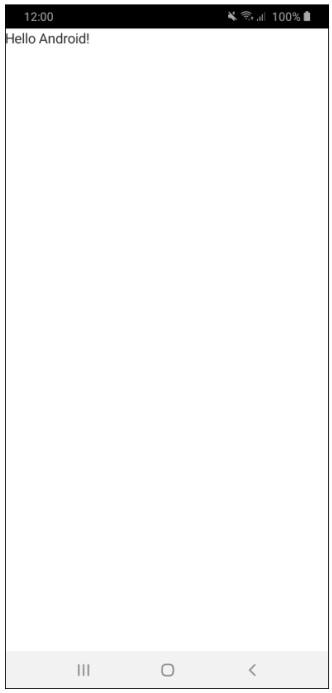
```
An exception occurred applying plugin request [id: 'com.android.application']
> Failed to apply plugin 'com.android.internal.application'.
   > com.android.builder.errors.EvalIssueException: The option 'android.enableBuildCache' is deprecated.
     The current default is 'false'.
     It was removed in version 7.0 of the Android Gradle plugin.
     The Android-specific build caches were superseded by the Gradle build cache (https://docs.gradle.org/current/userguide/build_cache.html).

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output. Run with --scan to get full insights.
```

In order to solve it:

1. Open gradle.properties.
2. Remove the line android.enableBuildCache=true.

Upon running the project again, you should see a sample Compose app that AS has built for us.



Writing our first line of Compose

In order to start writing our app, we need to first structure our app to what I call Jetpack Compose conventions since I've often seen it as a common structure among Google Codelabs.

First things first:

1. Open `MainActivity.kt`.
2. Create a new composable function under your `MainActivity` class.

```
@Composable
fun MyApp() {
    Scaffold(
        content = {
            BarkHomeContent()
        }
    )
}
```

3. Import `Scaffold` into your file if it's not imported automatically and is shown as an unresolved reference.

What is Scaffold?

If you read Scaffold's definition, it is mentioned that Scaffold implements the basic material design visual layout structure in Compose. So it's generally a good idea to start your screen structure with Android's own visual layout structure.

4. Replace your sample Hello World greeting by calling `MyApp()` inside onCreate.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
        BarkTheme {
            MyApp()
        }
    }
}
```

Next, we need to write our content that we added to our Scaffold's content parameter, `BarkHomeContent()`.

But first, we do know that we need to display a list of puppies with some sort of detail for each puppy and perhaps a picture to go along with it. In order to do so, we need to create a Data class that holds information for each puppy and a Data Provider that provides us with a list of puppies in their correct structure to be displayed in our list.

Setting puppies for adoption

In a real scenario, our data would generally be provided by a backend through some sort of RESTful API that we need to work with asynchronously and write a different flow for. But for learning purposes, we're going to fake our data and write all our puppies information and add their pictures in our app itself.

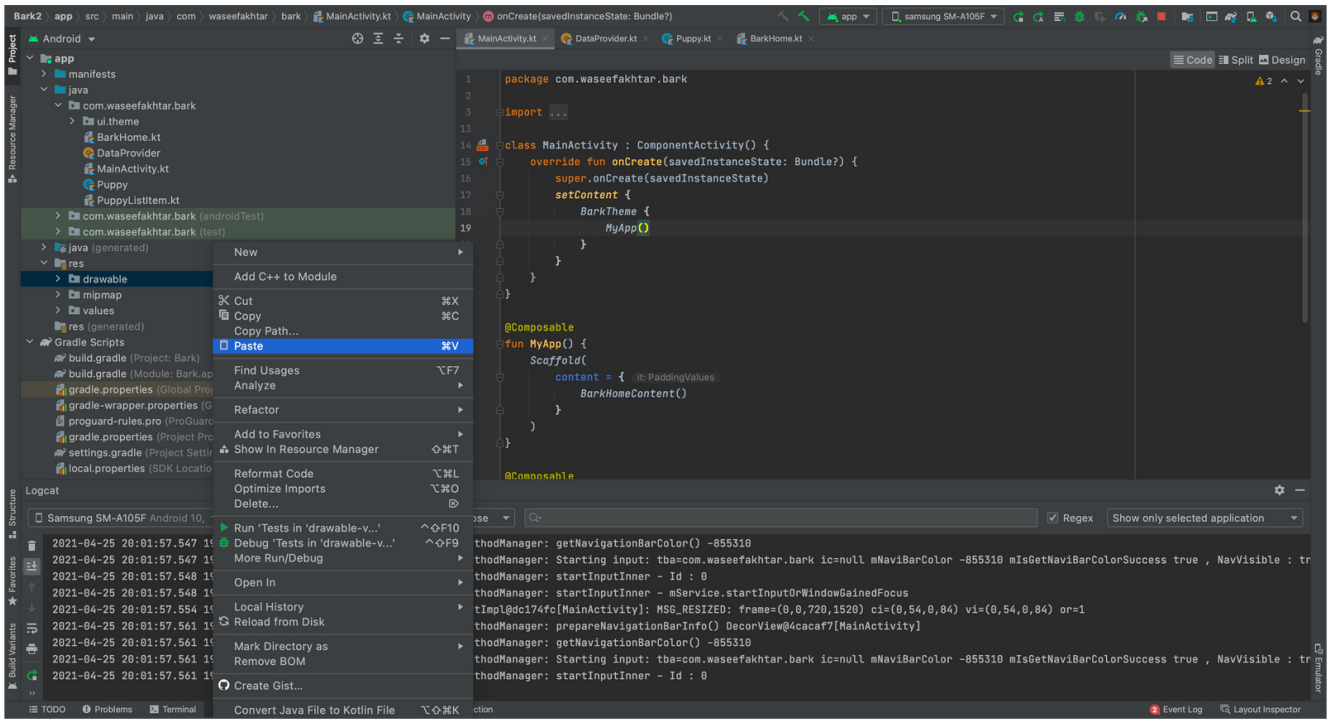
In order to do so:

1. Create a new class called `Puppy.kt`.
2. Write a data class with fields of all the properties that we're going to have in order to populate our list items with:

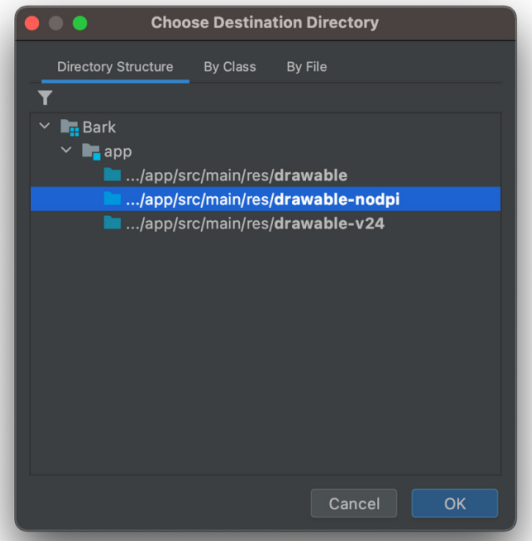
```
data class Puppy(
    val id: Int,
    val title: String,
    val sex: String,
    val age: Int,
    val description: String,
    val puppyImageId: Int = 0
)
```

After downloading,

1. Select all the files.
2. Copy the files.
3. In Android Studio, under */res, select /drawable* and paste all your files.


Bark2 app src main java com waseefakhtar bark MainActivity.kt MainActivity onCreate(savedInstanceState: Bundle?)
app
samsung SM-A105F
Android
MainActivity.kt
DataProvider.kt
Puppy.kt
BarkHome.kt
Code Split Design
app
manifests
java
com.waseefakhtar.bark
ui.theme
BarkHome.kt
DataProvider
MainActivity.kt
Puppy
PuppyListItem.kt
com.waseefakhtar.bark (androidTest)
com.waseefakhtar.bark (test)
java (generated)
res
drawable
mipmap
values
res (generated)
Gradle Scripts
build.gradle (Project: Bark)
build.gradle (Module: Bark.ap
gradle.properties (Global Proj
gradle-wrapper.properties (G
proguard-rules.pro (ProGuard
gradle.properties (Project Pro
settings.gradle (Project Settir
local.properties (SDK Locatio
New
Add C++ to Module
Cut ⌘X
Copy ⌘C
Copy Path...
Paste ⌘V
Find Usages ⌥F7
Analyze
Refactor
Add to Favorites
Show In Resource Manager ⇧⌘T
Reformat Code ⌥⌘L
Optimize Imports ⌥⌘O
Delete...
Run 'Tests in 'drawable-v...' ^⇧F10
Debug 'Tests in 'drawable-v...' ^⇧F9
More Run/Debug
Open In
Local History
Reload from Disk
Mark Directory as
Remove BOM
Create Gist...
Convert Java File to Kotlin File ⌥⇧⌘K
package com.waseefakhtar.bark
import ...
class MainActivity : ComponentActivity() {
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContent {
BarkTheme {
MyApp()
}
}
}
}
@Composable
fun MyApp() {
Scaffold(
content = { it: PaddingValues
BarkHomeContent()
}
)
}
@Composable
Logcat
Samsung SM-A105F Android 10,
Regex
Show only selected application
2021-04-25 20:01:57.547 19
2021-04-25 20:01:57.547 19
2021-04-25 20:01:57.548 19
2021-04-25 20:01:57.548 19
2021-04-25 20:01:57.554 19
2021-04-25 20:01:57.561 19
2021-04-25 20:01:57.561 19
2021-04-25 20:01:57.561 19
2021-04-25 20:01:57.561 19
thodManager: getNavigationBarColor() -855310
thodManager: Starting input: tba=com.waseefakhtar.bark ic=null mNaviBarColor -855310 mIsGetNaviBarColorSuccess true , NavVisible : tr
thodManager: startInputInner - Id : 0
thodManager: startInputInner - mService.startInputOrWindowGainedFocus
tImpl@dc174fc[MainActivity]: MSG_RESIZED: frame=(0,0,720,1520) ci=(0,54,0,84) vi=(0,54,0,84) or=1
thodManager: prepareNavigationBarInfo() DecorView@4cacaf7[MainActivity]
thodManager: getNavigationBarColor() -855310
thodManager: Starting input: tba=com.waseefakhtar.bark ic=null mNaviBarColor -855310 mIsGetNaviBarColorSuccess true , NavVisible : tr
thodManager: startInputInner - Id : 0
TODO
Problems
Terminal
Event Log
Layout Inspector

1. When prompted with the dialog asking for which directory to add them to, select `drawable-nodpi.` (If you can't see it, you can manually create the directory under */res* or just paste your files inside */drawable)*



Now we're finally going to write out DataProvider class to structure our data for the list.

1. Create a new class called `DataProvider.kt`.
2. Write an object declaration and create a list with information about each puppy. (Feel free to copy all the text to save your time building the app)

```
object DataProvider {

    val puppyList = listOf(
        Puppy(
            id = 1,
            title = "Monty",
            sex = "Male",
            age = 14,
            description = "Monty enjoys chicken treats and cuddling while watching Seinfeld.",
            puppyImageId = R.drawable.p1
        ),
        Puppy(
            id = 2,
            title = "Jubilee",
            sex = "Female",
            age = 6,
            description = "Jubilee enjoys thoughtful discussions by the campfire.",
            puppyImageId = R.drawable.p2
        ),
        Puppy(
            id = 12,
            title = "Stella",
            sex = "Female",
            age = 14,
            description = "Stella! Calm and always up for a challenge, she's the perfect companion.",
            puppyImageId = R.drawable.p12
        ),
    )
}
```

Displaying Puppies in a list

Now, going back to where we left off when calling *BarkHomeContent()* inside *MyApp()*, we're finally going to create a list item and populate our list with the data we just created.

First things first,

1. Create a new class called *BarkHome.kt.*
2. Add the composable function, *BarkHomeContent()*, inside the new class.

```
@Composable
fun BarkHomeContent() {
    val puppies = remember { DataProvider.puppyList }
    LazyColumn(
        contentPadding = PaddingValues(horizontal = 16.dp, vertical = 8.dp)
    ) {
        items(
            items = puppies,
            itemContent = {
                PuppyListItem(puppy = it)
            })
    }
}
```

3. Import all the missing references.

Note: You might notice that at this point, you might have a different version of `items` function that we need, considering that the parameter `items =` is not resolved. In that case, you need to manually import the reference for it at the top of class:
`import androidx.compose.foundation.lazy.items`.

Now, there's quite a bit going on here, let's explain it one by one.

1. On line 3, we define a *puppies* variable but with a remember *{ } keyword*. A remember function in a composable function simply stores the current state of the variable (in our case, the *puppies* variable) when the state of the list changes. This would be quite useful in a real-life scenario where the list changes from the back-end or from user events if we have any UI elements that let users change the state of the list. In our current case, we do not have such functionality but it's still a good practice to persist the state of our puppy list. To learn more about states, have a look at the docs:

2. On line 4, we call a *LazyColumn* composable. This is the equivalent of the RecyclerView that we as Android developers are quite familiar with. This honestly calls for a big celebration because of how easy it is to create a dynamic list with Jetpack Compose.

3. On line 5, inside *LazyColumn* params, we give it a nice little padding to give our items a bit of a breathing space, and

4. On lines 7-11, inside *LazyColumn*'s content, we call the *items* function that takes our *puppies* list as the first param, and a composable *itemContent* (that we're going to create next) that takes our list item composable to populate with each item in the list.

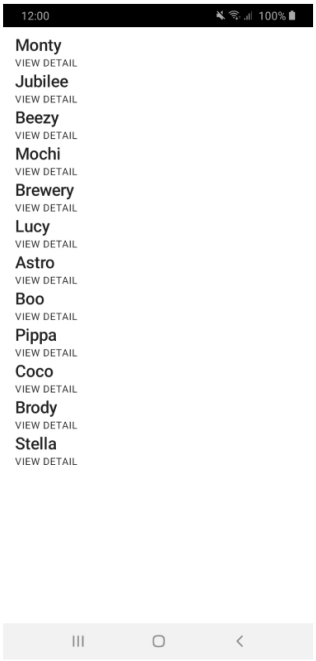
Creating a list item

Next, we're going to create our list item composable that we're going to call *PuppyListItem:*

1. Create a new Kotlin file, *PuppyListItem.kt.*
2. Write a new simple composable function in the class that takes a *Puppy* type as a param.
3. Inside the function, create a *Row* that represents a row in a list.
4. Inside the *Row*, create a column of two texts and pass in the puppy title on the first text and a view detail as the second text.

```
@Composable
fun PuppyListItem(puppy: Puppy) {
    Row {
        Column {
            Text(text = puppy.title, style = typography.h6)
            Text(text = "VIEW DETAIL", style = typography.caption)
        }
    }
}
```

This is the result when running the app after creating our *PuppyListItem.*

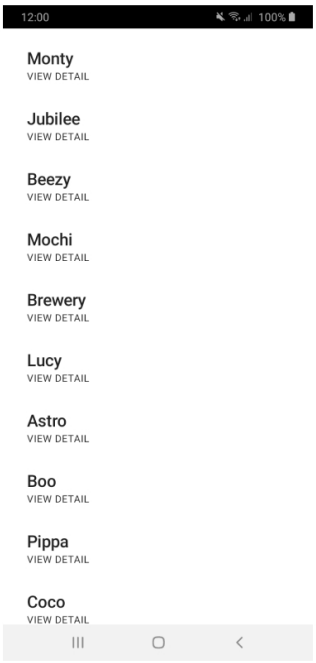


Not very nice looking. But there are easy steps to style our item.

Styling List item

1. Add a bit of a padding and make the texts full width for some breathing space.

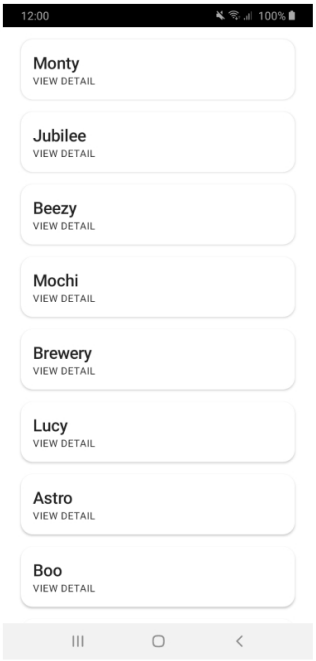
```
Row {
    Column(
        modifier = Modifier
            .padding(16.dp)
            .fillMaxWidth()
            .align(Alignment.CenterVertically)
    ) {
        Text(text = puppy.title, style = typography.h6)
        Text(text = "VIEW DETAIL", style = typography.caption)
    }
}
```


12:00
100%
Monty
VIEW DETAIL
Jubilee
VIEW DETAIL
Beezy
VIEW DETAIL
Mochi
VIEW DETAIL
Brewery
VIEW DETAIL
Lucy
VIEW DETAIL
Astro
VIEW DETAIL
Boo
VIEW DETAIL
Pippa
VIEW DETAIL
Coco
VIEW DETAIL

Styling List item

2. Surround your *Row* with a *Card* composable and style it as you please.

```
Card(
    modifier = Modifier.padding(horizontal = 8.dp, vertical = 8.dp).fillMaxWidth(),
    elevation = 2.dp,
    backgroundColor = Color.White,
    shape = RoundedCornerShape(corner = CornerSize(16.dp))
) {
    Row {
        Column(
            modifier = Modifier
                .padding(16.dp)
                .fillMaxWidth()
                .align(Alignment.CenterVertically)
        ) {
            Text(text = puppy.title, style = typography.h6)
            Text(text = "VIEW DETAIL", style = typography.caption)
        }
    }
}
```



Finally, we need to add an image for each puppy. In order to do so:

1. Create a new composable function, *PuppyImage()* under *PuppyListItem()*, passing the `puppy` param.
2. Call the *Image* composable function and style it as you please:

```
@Composable
private fun PuppyImage(puppy: Puppy) {
    Image(
        painter = painterResource(id = puppy.puppyImageId),
        contentDescription = null,
        contentScale = ContentScale.Crop,
        modifier = Modifier
            .padding(8.dp)
            .size(84.dp)
            .clip(RoundedCornerShape(corner = CornerSize(16.dp)))
    )
}
```

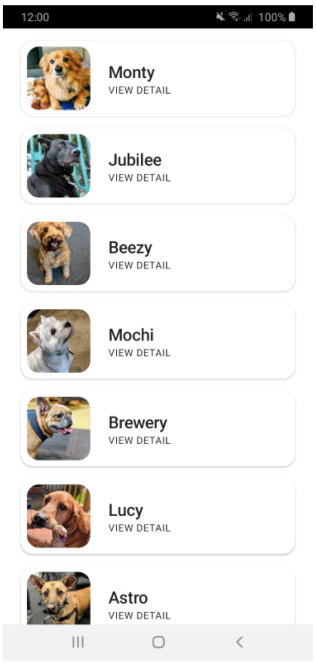
3. Finally, call PuppyImage() the first thing inside your Row in PuppyListItem()

```
@Composable
fun PuppyListItem(puppy: Puppy) {
    Card(
        modifier = Modifier.padding(horizontal = 8.dp, vertical = 8.dp).fillMaxWidth(),
        elevation = 2.dp,
        backgroundColor = Color.White,
        shape = RoundedCornerShape(corner = CornerSize(16.dp))

    ) {
        Row {
            PuppyImage(puppy)
            Column(
                modifier = Modifier
                    .padding(16.dp)
                    .fillMaxWidth()
                    .align(Alignment.CenterVertically)) {
                Text(text = puppy.title, style = typography.h6)
                Text(text = "VIEW DETAIL", style = typography.caption)

            }
        }
    }
}
```

Output


12:00
100%
Monty
VIEW DETAIL
Jubilee
VIEW DETAIL
Beezy
VIEW DETAIL
Mochi
VIEW DETAIL
Brewery
VIEW DETAIL
Lucy
VIEW DETAIL
Astro
VIEW DETAIL