Android Mobile Application Development
Week 7: Data Persistence and Firebase

Data Persistence

Most apps require data to be stored persistently, meaning across app launches. There are two approaches to data persistence. Data can be persisted locally on the device for data only accessed by the one user in the Android app, or saved in the cloud if the data will be accessed or aggregated across users, or accessed through different platforms.

Local Data Storage

https://developer.android.com/training/data-storage

There are multiple local data storage approaches on Android. The approach you pick should be based on

- What kind of data you need to store
- How much space your data requires
- How reliable does data access needs to be
- Whether the data should be private to your app

App-specific storage

https://developer.android.com/training/data-storage/app-specific

All Android devices have two file storage areas for files in your app that other apps don't, or shouldn't, access: "internal" and "external" storage. These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage). Many devices now divide the permanent storage space into separate "internal" and "external" partitions.

Internal file storage

Internal storage is best when you want to be sure that neither the user nor other apps can access the data. No permission is needed to read and write files to internal storage.

Pros

- Always available
- By default files saved are private to your app
- Other apps and the user can't access the files
- Starting in Android 10(API 29) internal storage directories will be encrypted

Cons

- Should check the amount of free space before saving files
- Hard to share data
- Internal storage might have limited capacity

External file storage

External storage often provides more space than internal storage for app-specific files. Other apps can access these files if they have the proper permissions.

Pros

- Often provides increased capacity

Cons

- Might not always be available as it might resides on a physical volume that the user might be able to remove
- Must verify that the volume is accessible before using

Both internal and external storage provide directories for an app's persistent files and another for an app's cached files.
In both internal and external storage files are removed when the user uninstalls your app

For files that you want to persist past the lifetime of an app you should use shared storage instead.

Shared storage
https://developer.android.com/training/data-storage/shared
Use shared storage for user data that should be accessible to other apps and saved even if the user uninstalls your app.

- Use the MediaStore API to store media content in a common location on the device
- The Storage Access Framework uses a document provider to store documents and other files in a specific directory

Key-Value Data
https://developer.android.com/training/data-storage/shared-preferences.html
Shared Preferences (different than preferences through Settings)
For small amounts of data that doesn't require structure saving the data as key-value pairs is a good choice.
The Shared Preferences API has been around since API 1and stores data as key-value pairs in an unencrypted XML file in internal storage.

- Keys are always of type String
- Values must be primitive data types: boolean, float, int, long, String, and stringset
- You can use a single file or multiple files
- You can use the default or a named preference
- Preference data can be deleted from the device by the user
- Data is deleted when the app is uninstalled

Some downsides of SharedPreferences include:

- Lack of safety from runtime exceptions
- Lack of a fully asynchronous API
- Lack of main thread safety
- No type safety

DataStore
https://developer.android.com/topic/libraries/architecture/datastore
The new DataStore library has been added to Android Jetpack and provides an improved local data solution using key-value pairs. It is now suggested over Shared Preferences.
DataStore provides two different implementations.

1. Preferences DataStore to store data using key-value pairs
2. Proto DataStore to store data as instances of a custom data type using protocol buffers for structured data

Both DataStore implementations store data asynchronously, consistently, and transactionally, overcoming most of the drawbacks of SharedPreferences. DataStore uses coroutines and Flow to store data asynchronously.

Databases

https://developer.android.com/training/data-storage/room
Relational databases (RDBMS) are a good choice to store structured data. They handle complex data with relationships and enforce data integrity. They are also a good fit for larger amounts of data that will be accessed and manipulated often.
In a RDBMS database data is organized in a collection of tables with defined relationships. This structure and organization is referred to as the database schema.
A table is organized into columns that represent what is stored in the table. Each row in a table contains one record of data.
Android's SQLite database is an RDBMS well suited for persisting large amounts of structured data locally. Similar to internal storage, Android stores your database in your app's private folder and therefore is not accessible to other apps or the user.
The Room library provides an abstraction layer over SQLite that makes it much easier to work with SQLite. It is highly recommended instead of using SQLite APIs directly. It is one of the architecture components that is included in Jetpack.

Local Data Storage Advantages:
- No internet access needed
- Local control
- No fees
- Speed not network dependent

Local Data Storage Disadvantages:
- Subject to device failure/loss
- No backup/recovery ability

Cloud Data Storage
Setting up your own database in the cloud, and handling data syncing, takes a lot of work. Services that handle this for you such as Heroku and Firebase have become very popular as they provide platforms in the cloud for app data management.

Cloud Data Storage Advantages:
- Cross-platform access
- Ability to aggregate data
- Can handle large amounts of data
- Often supports local offline access as well
- Recovery/backup ability

Cloud Data Storage Disadvantages:
- Internet access needed
- Fees involved
- Downtime
- Performance network dependent

Firebase
https://firebase.google.com/
Google's Firebase is a backend-as-a-service (BaaS) that allows you to get an app with a server-side real-time database up and running very quickly. Providing this as a service means you don't have to set up a database, write all the code to synchronize the data, and figure out security and authentication. Firebase stores data as JSON in the cloud in a NoSQL database.

With SDKs available for the web, Android, iOS, and a REST API it lets you easily sync data across devices/clients on iOS, Android, and the Web.

Features: https://firebase.google.com/products/

- Database
- Cloud Storage
- Authentication
- Hosting
- Cross platform support – web, iOS, Android

Firebase was founded by Andrew Lee and James Tamplin in 2011, officially launched in April 2012, and purchased by Google two years later.

Cloud Firestore

https://firebase.google.com/docs/firestore

Cloud Firestore is a flexible, scalable NoSQL database for mobile, web, and server development. It keeps your data in sync across client apps through real-time listeners and offers offline support for mobile and web.

Get Started

https://firebase.google.com/docs/firestore/quickstart

Get started by logging in with your Google login to create a Firebase account

Create a new Firebase project called Recipes.

A project in Firebase stores data that can be accessed across multiple platforms so you can have an iOS, Android, and web app all sharing the same project data. For completely different apps use different projects.

Firebase has two databases, the Realtime Database (Firebase) and Cloud Firestore. Both are noSQL databases but Firestore is their newer version so we'll use that. (Be careful of the difference between Firebase and Firestore online as they're different, have different structures, methods, and are not compatible with each other.)

Then you'll be taken to the dashboard where you can manage the Firebase project.

https://console.firebase.google.com

From the console's navigation pane, select **Database**, then click **Create database** for Cloud Firestore.

- Start in test mode.
  - Good for getting started with the mobile and web client libraries
  - Allows anyone to read and overwrite your data.
- Select the multi-regional location nam5(us-central)

Data Model

https://firebase.google.com/docs/firestore/data-model

Cloud Firestore is a NoSQL, document-oriented database. Unlike a SQL database, there are no tables or rows. Instead, you store data in documents, which are organized into collections.

Documents

- Document is the unit of storage
- A document is a lightweight record that contains fields, which map to values.
- Each document is identified by a unique id or name
- Each document contains a set of key-value pairs.
- All documents must be stored in collections.

- Documents within the same collection can all contain different fields or store different types of data in those fields.
- However, it's a good idea to use the same fields and data types across multiple documents, so that you can query the documents more easily.
- The names of documents within a collection are unique. You can provide your own keys, such as user IDs, or you can let Cloud Firestore create random IDs for you automatically.

Collection
- A collection is a container for documents
- A collection contains documents and nothing else.
- A collection can't directly contain raw fields with values, and it can't contain other collections.
- You can use multiple collections for different related data (orders vs users)

You do not need to "create" or "delete" collections. After you create the first document in a collection, the collection exists.

Every document in Cloud Firestore is uniquely identified by its location within the database. To refer to a location in your code, you can create a *reference* to it.
A reference is a lightweight object that just points to a location in your database. You can create a CollectionReference to a collection or a document.

```
val recipeRef = db.collection("recipes")
```

Let's create a collection and the first document using the console so we know our collection exists.

In the console go into Firestore Database | Cloud Firestore | Data
Create a collection called recipes.
Now create a document using auto id with the fields name, and url.

Note that because we're in test mode our security rules are public and you'll see the following in the rules tab:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write;
    }
  }
}
```

You'll want to change this when not in test mode or when using authentication.

Read Data
There are two ways to retrieve data stored in Cloud Firestore. Either of these methods can be used with documents, collections of documents, or the results of queries:
- Call a method to get the data.
- Set a listener to receive data-change events.

Methods:
https://firebase.google.com/docs/firestore/query-data/get-data
Get a single document:
```
val docRef = db.collection("cities").document("SF").get()
```

Get a single document and convert it to an object of your model class (field names must match) use `.toObject<Class>()`

Get multiple documents from a collection. You then need to loop through the result or access the one you want. You can use the various `where()` methods to define your query.

Listeners:
https://firebase.google.com/docs/firestore/query-data/listen
When you set a listener, Cloud Firestore sends your listener an initial snapshot of the data, and then another snapshot each time the document changes.

You can listen to a document with the `addSnapshotListener()` method (note this is often referred to as the `onSnapshot()` method since that's what it's called on other platforms). When listening for changes to a document, collection, or query, you can pass options to control the granularity of events that your listener will receive.

An initial call using the callback you provide creates a document snapshot immediately with the current contents of the single document. Then, each time the contents change, another call updates the document snapshot.
Local writes in your app will invoke snapshot listeners immediately. This is because of a feature called "latency compensation." When you perform a write, your listeners will be notified with the new data *before* the data is sent to the backend.

Queries:
https://firebase.google.com/docs/firestore/query-data/queries
Cloud Firestore provides powerful query functionality for specifying which documents you want to retrieve from a collection or collection group. These queries can also be used with both the methods and listeners above.
By default, a query retrieves all documents that satisfy the query in ascending order by document ID. You can specify the sort order for your data using `orderBy()`, and you can limit the number of documents retrieved using `limit()`.

Write Data
https://firebase.google.com/docs/firestore/manage-data/add-data
When you write a document to a collection in Cloud Firestore each document needs an identifier. You can explicitly specify a document identifier or have Cloud Firestore automatically generate it. You can also create an empty document with an automatically generated identifier, and assign data to it later.

`.set()` creates or overwrites a single document and requires a document identifier.
`.add()` adds a document and automatically generates a document identifier for you.
`.update()` lets you update some fields of a document without overwriting the entire document.

Delete Data

`.delete()` deletes a specific document
Note that deleting a document does NOT delete any subcollections it might have.

To delete an entire collection or subcollection in Cloud Firestore, retrieve all the documents within the collection or subcollection and delete them. This is not recommended from a mobile client.

Access Data Offline
https://firebase.google.com/docs/firestore/manage-data/enable-offline
Cloud Firestore supports offline data persistence. A copy of the Cloud Firestore data that your app is actively using will be cached so your app can access the data when the device is offline. You can write, read, listen to, and query the cached data. When the device comes back online, Cloud Firestore synchronizes any local changes made by your app to the Cloud Firestore backend.

- For Android and iOS, offline persistence is enabled by default. To disable persistence, set the PersistenceEnabled option to false
- For the web, offline persistence is disabled by default

Android Studio Setup
https://firebase.google.com/docs/android/setup
New project called Recipes
Empty Compose Activity template.

1. Create a Firebase project
Chose existing Recipes database or create a new one
In the Firestore Database go to the Rules tab and make sure the documents permissions for both read and write are set to true so our app can access the database.

```
match /{document=**} {
  allow read, write;
}
```

2. Register your app with Firebase
In the Firebase console click the Android icon or Add app
Enter your app's application id which is the same as the package name which you can get from your Gradle files, Android manifest, or top of your MainActivity file.

3. Add the Firebase configuration file
In project settings click Download google-services.json to obtain your Firebase Android config file (google-services.json).
Go into the Project view.
Move your config file into the app-level directory of your app.

4.Add Cloud Firestore to your app
In your project-level `build.gradle` file add the Google services plugin as a dependency.

```
buildscript {
    ...
  dependencies {
    classpath 'com.google.gms:google-services:4.3.14'
  }
}
```

In your `app/build.gradle` file apply the Google services plugin. Order might matter so add this one last.

```
plugins{
      id 'com.google.gms.google-services'
}
```

Also add these libraries to your `app/build.gradle` file:

```
implementation platform('com.google.firebase:firebase-bom:31.1.0')
implementation 'com.google.firebase:firebase-firestore-ktx'
implementation 'androidx.lifecycle:lifecycle-viewmodel-compose:2.5.1'
```

The Firebase Android BoM (Bill of Materials) enables you to manage all your Firebase library versions by specifying only one version — the BoM's version. The BoM automatically pulls in the individual library versions mapped to BoM's version. All the individual library versions will be compatible. When you update the BoM's version in your app, all the Firebase libraries that you use in your app will update to the versions mapped to that BoM version.
The second dependency is for the cloud firestore kotlin library.
The last dependency is for the compose viewmodel library.

You will need to add other dependencies for other Firebase functionality (authentication, etc).

Model class

Go into your project's Java folder select the recipes folder and create a new package called model and add a new Kotlin class called Recipe for our model. Our model will match our database document structure with name and url properties. We will also have an id property to store the unique document id.

The @DocumentID annotation marks a property so Firestore will automatically map the document ID to the id attribute. We need this unique id so later we know which document the user is trying to access, such as in a delete or update, in Firestore.

Firestore also requires an empty constructor method so I added a secondary empty constructor as well.

```
data class Recipe(@DocumentId val id: String, val name: String, val
url:String) {
    constructor(): this( "", "", ""){}
}
```

Database

Create a package called util and a class called RecipeDatabase that will handle all interaction with Firebase.

Here we create constants for the Firebase database and the recipes collection. The app knows which database to reference from the data in your GoogleService-Info.plist file.
We also create a List to store all our recipes.

We create the getRecipes() method to add a snapshot listener on our recipes collection. This listener will get notified whenever there is a change to the data in the collection and run the lambda expression passing it a snapshot of the data in the collection. Here we clear the recipes list and then iterate through

the snapshot taking each document and adding it to our recipes list. The `toObject()` method converts the document to an object of type Recipe which it can infer from the type of our recipes list. It also uses the document id for the id property because of the `@DocumentId` annotation. Note that our class property names match the names we used in our documents. If that's not the case you will either need to map them or assign each property individually.

We also create methods that add and delete a document using Firebase's add() and delete() methods.

```
class RecipeDatabase {
    //Firestore instance
    private val db = Firebase.firestore

    //recipe collection
    private val recipeRef = db.collection("recipes")

    var recipes = mutableStateListOf<Recipe>()

fun getRecipes() {
        recipeRef.addSnapshotListener { docSnapShot, error ->
            if (docSnapShot != null) {
                //clear list to avoid duplicates
                recipes.clear()
                for (doc in docSnapShot) {
                    //add to list
                    recipes.add(doc.toObject())
                    Log.d("get", doc.getId())
                }
                Log.d("get", recipes.size.toString())
            } else
                if (error != null) {
                    Log.e("get", "listener error", error)
                }
        }
}

fun addRecipe(recipe: Recipe){
    recipeRef.add(recipe)
}

fun deleteRecipe(id: String){
    recipeRef.document(id).delete()
}

}
```

If your project doesn't recognize Firebase.firestore go back to the setup steps and make sure you downloaded the google-services.json file and put it in the app level directory of your app.

ViewModel

Now we'll create our view model class.
Select the model package and add a new class called RecipeViewModel.
The view model includes methods to add and delete a recipe from the database. The method to delete takes an id because that's what the Firestore delete() method will need.

```
class RecipeViewModel: ViewModel() {
    private val recipedb = RecipeDatabase()

    var recipeList = recipedb.recipes

    init {
         recipedb.getRecipes()
    }

    fun addRecipe(recipe: Recipe){
        recipedb.addRecipe(recipe)
    }

    fun deleteRecipe(id: String){
        recipedb.deleteRecipe(id)
    }

}
```

Since we want to set up the database listener immediately, we call getRecipes() in the initializer method. When the listener is attached it runs the first time so our recipes list gets populated.

Screens
Add these strings to your strings.xml resource file.

```
<string name="recipeName">Recipe name</string>
<string name="recipeURL">URL</string>
<string name="addRecipe">Add Recipe</string>
<string name="add">Add</string>
<string name="recipeAdded">Recipe Added</string>
<string name="action">Action</string>
<string name="cancel">Cancel</string>
<string name="delete">Delete?</string>
<string name="deleteRecipe">Recipe Deleted</string>
<string name="yes">Yes</string>
<string name="no">No</string>
```

Our MainActivity class will use the same structure as in our Book list app.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            RecipesTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                ) {
```

```
                    ScreenSetup()
                }
            }
        }
    }
}

@Composable
fun ScreenSetup(){
    RecipeScreen()
}
```

Create a package called screens and a file called RecipeScreen that will have the composables for our UI. This too will have a very similar structure as our Book list app. This is because we've used a well structured MVVM architecture (model view viewmodel) so changing the data repository layer doesn't affect the model or the view. The only changes are:

- When adding a recipe I let Firestore auto generate the ID instead of using the UUID class to randomly generate one
- The delete function uses the id to delete a document so it's passed an id instead of the whole Recipe object

I removed the Toast from the onClick event handler and left it empty for now since we'll be using it for something else later.

```
@Composable
fun RecipeScreen() {
    val viewModel: RecipeViewModel = viewModel()
    val context = LocalContext.current
    var showAddDialog by remember { mutableStateOf(false) }

    Scaffold(
        backgroundColor = MaterialTheme.colors.surface,
        floatingActionButton = {
            FloatingActionButton(
                onClick = {showAddDialog = true}
            )
            {
                Icon(Icons.Filled.Add, contentDescription = "")
            }
        },
        content = {
            if (showAddDialog){
                addRecipeDialog(context, dismissDialog =
{showAddDialog = false}, {viewModel.addRecipe(it)})
            }
            LazyColumn(
                contentPadding = PaddingValues(
                    vertical = 8.dp,
                    horizontal = 8.dp
                )
```

```
            )
            {
                items(viewModel.recipeList, key={recipe ->
recipe.id}) { recipe ->
                    // , key={recipe -> recipe.id}
                    RecipeItem(item = recipe, context,
{viewModel.deleteRecipe(it)})
                    //, {viewModel.deleteRecipe(it)}
                }
            }
        }
    )
}

@Composable
fun addRecipeDialog(context: Context, dismissDialog:() -> Unit,
addBook: (Recipe) -> Unit){
    var nameTextField by remember {
        mutableStateOf("")
    }
    var urlTextField by remember {
        mutableStateOf("")
    }

    AlertDialog(
        onDismissRequest = { dismissDialog},
        title={ Text(text = stringResource(id =
com.example.recipes.R.string.addRecipe), style =
MaterialTheme.typography.h6) },
        text = {
            Column(modifier = Modifier.padding(top=20.dp)) {
                TextField(label = { Text(text= stringResource(id =
com.example.recipes.R.string.recipeName)) }, value = nameTextField,
onValueChange = {nameTextField=it})
                Spacer(modifier = Modifier.height(10.dp))
                TextField(label = { Text(text= stringResource(id =
com.example.recipes.R.string.recipeURL)) },value = urlTextField,
onValueChange = {urlTextField=it})
            }
        },
        confirmButton = {
            Button(onClick = {
                if(nameTextField.isNotEmpty()) {
                    addBook(Recipe("",nameTextField, urlTextField))
                    Toast.makeText(
                        context,

context.resources.getString(com.example.recipes.R.string.recipeAdded)
,
                        Toast.LENGTH_SHORT
```

```
                    ).show()
                }
                dismissDialog()
            })
            {
                Text(text = stringResource(id =
com.example.recipes.R.string.add))
            }
        },dismissButton = {
            Button(onClick = {
                dismissDialog()
            }) {
                Text(text = stringResource(id =
com.example.recipes.R.string.cancel))
            }
        }
    )
}

@Composable
fun deleteRecipeDialog(context: Context, dismissDialog:() -> Unit,
item: Recipe, deleteBook: (String) -> Unit){
    AlertDialog(
        onDismissRequest = { dismissDialog},
        title={ Text(text = stringResource(id = R.string.delete),
style = MaterialTheme.typography.h6) },
        confirmButton = {
            Button(onClick = {
                deleteBook(item.id)
                Toast.makeText(
                    context,

context.resources.getString(R.string.deleteRecipe),
                    Toast.LENGTH_SHORT
                ).show()
                dismissDialog()
            })
            {
                Text(text = stringResource(id = R.string.yes))
            }
        },dismissButton = {
            Button(onClick = {
                dismissDialog()
            }) {
                Text(text = stringResource(id = R.string.no))
            }
        }
    )
}
```

```
@OptIn(ExperimentalFoundationApi::class)
@Composable
fun RecipeItem(item: Recipe, context: Context, deleteBook: (String) -
> Unit) {
    var showDeleteDialog by remember { mutableStateOf(false) }

    Card(
        elevation = 4.dp,
        shape = RoundedCornerShape(10.dp),
        backgroundColor = MaterialTheme.colors.primary,
        contentColor = MaterialTheme.colors.onPrimary,
        border = BorderStroke(2.dp, color =
MaterialTheme.colors.primaryVariant),
        modifier = Modifier
            .padding(8.dp)
            .fillMaxWidth()
            .combinedClickable(
                onClick = {},
                onLongClick = { showDeleteDialog = true }
            )
    ) {
        Column(
            modifier = Modifier.padding(16.dp)
        ) {
            Text(text = item.name, style =
MaterialTheme.typography.h6)
        }

    }
    if (showDeleteDialog){
        deleteRecipeDialog(context, dismissDialog = {showDeleteDialog
= false}, item, deleteBook)
    }
}

@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    RecipesTheme {
        RecipeScreen()
    }
}
```

You should now be able to run your app and see your Firebase data as well as add and delete recipes. When you add or delete in the app you'll see the database data in the console change. If you make any changes through the console your app should automatically update.

Error Handling

In RecipeDatabase we'll add in error handling.

In getRecipes() we wrap `addSnapshotListener` in a try/catch block so if an error is thrown it will pass to catch().

```
fun getRecipes(){
     try {
         recipeRef.addSnapshotListener { docSnapShot, error ->
             if (docSnapShot != null) {
                 //clear list to avoid duplicates
                 recipes.clear()
                 for (doc in docSnapShot) {
                     //add to list
                     recipes.add(doc.toObject())
                     Log.d("get", doc.getId())
                 }
                 Log.d("get", recipes.size.toString())
             }
         }
     } catch (error: FirebaseFirestoreException){
         Log.e("get", "listener error", error)
     }
}
```

In addRecipe() and deleteRecipe() we add success and failure listeners for error handling per the documentation. This could also be achieved using try/catch.

```
    fun addRecipe(recipe: Recipe){
        recipeRef.add(recipe)
            .addOnSuccessListener { documentReference ->
                Log.d("add", "DocumentSnapshot written with ID:
${documentReference.id}")
            }
            .addOnFailureListener { e ->
                Log.e("add", "Error adding document", e)
            }
    }


    fun deleteRecipe(id: String){
        recipeRef.document(id).delete()
            .addOnSuccessListener {
                Log.d("delete", "DocumentSnapshot successfully
deleted")
            }
            .addOnFailureListener { e ->
                Log.e("delete", "Error deleting document", e)
            }
    }
```

Network Connections

Each Android application runs in its own process in a single thread which is called the **Main thread** or **UI thread.** Prior to Ice Cream Sandwich (API 14/15), Android apps could perform any operations from the main UI thread. This led to such situations where the application would be completely unresponsive while waiting for a response. Now there are stricter rules regarding performing operations -- tasks have 100-200ms to complete a task in an event handler. Operations that take longer could be killed by the WindowManager or ActivityManager processes and the user will receive an application not responding (ANR) message.
In Android the main thread is responsible for drawing and updating the UI on the screen. In scenarios where we're not in control of how long the operation will take we shouldn't perform them on the main thread. Instead we should perform them on a thread that runs in the background so our main UI thread never freezes or errors out.
An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread.

Kotlin Coroutines
https://developer.android.com/kotlin/coroutines
Kotlin coroutines handle concurrency in Kotlin and can be used in Android to simplify asynchronous code and solve two main issues:

1. Long running tasks are tasks that take too long and would block the main thread.

    a. File IO
    b. Database interaction
    c. Network interaction
    d. API integration

2. Main-safety ensures that functions don't block UI responses on the main thread

A coroutine is conceptually similar to a thread in that it takes a block of code to run that works concurrently with the rest of the program. Coroutines are described as lightweight threads because creating a coroutine doesn't allocate a new thread and is not bound to any particular thread. It may suspend its execution in one thread and resume in another one. Coroutines provide similar functionality to async/await used in other languages.
Coroutines offer asynchronous programming support at the language level in Kotlin. Coroutines can *suspend* execution without blocking threads. A responsive UI is inherently asynchronous, and Jetpack Compose solves this by embracing coroutines at the API level instead of using callbacks.
Jetpack Compose offers APIs that make using coroutines safe within the UI layer.
(slides)

- Coroutines can be suspended and resumed without blocking a thread.
    - Suspending vs blocking
        - If you make a blocking call on the main thread's execution, you effectively freeze the UI
        - If you call a suspending function in the main UI thread it can be run on a different thread so it won't block the main UI thread.
    - Coroutines execute the block of code *sequentially* by default. A running coroutine that calls a suspend function *suspends* its execution until the suspend function returns.

Suspending functions

A suspending function is simply a function that can be paused and resumed at a later time. They can execute a long running operation and wait for it to complete without blocking. Regular functions are mostly executed synchronously on the main thread. Suspending functions allow us to execute jobs asynchronously in the background without blocking the main thread.

- Suspending functions can only be called by another suspending function or within a coroutine.
- A suspending function is a function marked with the `suspend` keyword

Launching coroutines

To execute a suspend function we need to create and launch a coroutine.

- CoroutineScopes define the scope when a coroutine runs and handles their lifecycle
    - Android includes predefined coroutine scopes
        - GlobalScope
            - Allows coroutines to live as long as the app
        - ViewModelScope
            - Scopes the coroutine to our ViewModel
        - LifecycleScope
            - Scopes the coroutines to the lifecycle of a LifecycleOwner such as an Activity
        - rememberCoroutineScope
            - scopes a coroutine to the composition cycle of a composable function
    - You can also create custom scopes.
- A CoroutineDispatcher is used to configure what thread pool our coroutine should be run on. These dispatchers are available through the API:
    - Dispatchers.Default
        - CPU-intensive work, such as sorting large lists or doing complex calculations
    - Dispatchers.IO
        - any input and output such as disk or networking intensive operations
            - reading and writing from files or a local database
            - network requests such as API calls or remote database
    - Dispatchers.Main
        - Dispatches work to the main thread
        - Should only be used for light work that won't block the UI
- Coroutine builders are extension functions that allow us to create and start coroutines. It creates a scoped coroutine where we can call our suspended function.
    - launch builds a coroutine and dispatches the execution of its function body to the corresponding dispatcher.
        - Coroutines started with launch won't return the result to the caller. It's our job to get the result from the suspending function
    - async starts a new coroutine and allows you to return the result as a Deferred<T> object. The deferred object is a promise that your result will be returned in the future.
        - To start the coroutine and get a result you need to call await(), which blocks the calling thread.
        - Useful for tasks where results are required from one task before the next task runs

Coroutines

Our calls to Firebase are over the network so depending on network speed the UI could be blocked while we wait for a response so we'll using the viewModelScope coroutine for these calls. We use the

Dispatchers.IO dispatcher as otherwise Main is the default for viewModelScope, which is what we're trying to avoid.

Update RecipeViewModel

```
init {
    viewModelScope.launch(Dispatchers.IO) { recipedb.getRecipes() }
}

fun addRecipe(recipe: Recipe) =
viewModelScope.launch(Dispatchers.IO){
    recipedb.addRecipe(recipe)
}

fun deleteRecipe(id: String) = viewModelScope.launch(Dispatchers.IO){
    recipedb.deleteRecipe(id)
}
```

In RecipeDatabase we could make the three functions suspend functions but since there's no need to call other functions asynchronously the suspend keyword is not needed.

Now let's see how we can load a recipe in a web page.

Activities
https://developer.android.com/guide/components/activities/intro-activities.html
An app typically has one activity for each specific task of an app. Each activity in an app is listed in the AndroidManifest.xml file.

To start an activity you need to define an intent and then use the startActivity(Intent) method to start a new activity.

Intents
https://developer.android.com/guide/components/intents-filters.html
https://developer.android.com/training/basics/firstapp/starting-activity.html
Intents request an action such as starting a new activity.

- Provides the binding between two activities

You build an Intent with two key pieces of information

- Action – what action should be performed
- Data – what data is involved

There are two types of intents

- An explicit intent tells the app to start a specific activity
    - Usually used to start an activity in your own app
    - Provide the class name when creating the Intent
    - Call startActivity(Intent)
- An implicit intent does not name a specific activity. Instead you declare what type of action you want to perform which allows a component from another app to handle it

- Usually used to start an activity in a different app
- Provide the type of action when creating the Intent
- Call startActivity(Intent)
- Android uses intent resolution to see what apps can handle the intent
  - Compares the information in the intent to the intent filters of other apps
    - Each activity has intent filters defined in its app's AndroidManifest.xml file
    - An intent filter specifies what types and categories of intents each component can receive
    - An intent filter must include a category of android.intent.category.DEFAULT in order to receive implicit intents
    - If an activity has no intent filter, or it doesn't include a category name of android.intent.category.DEFAULT, it means that the activity can't be started with an implicit intent. It can only be started with an explicit intent using the fully qualified component name.
  - Android first considers intent filters that include a category of android.intent.category.DEFAULT
  - Android then matches the action and mime type with the intent filters
- If there is only one match between the intent and intent filters then Android starts that activity by calling its onCreate() method and passes it the Intent object
- If there is more than one match between the intent and intent filters, Android presents the user a list of apps to pick from

Creating Intents
https://developer.android.com/reference/android/content/Intent.html
An Intent object includes all the information needed to determine what activity to start

- Class name for an explicit intent
  - Context for the intent which is usually "this"
- Action for an implicit intent
- Data
  - A URI (uniform resource identifier) object references the data, usually dictated by the intent's action
  - set the data on the intent using the URI object
- Category description of the intent
  - Most intents don't need a category
- Extras enable intents to carry additional information
  - You can add extra information to your intent to pass data to the new activity using the putExtra(String, value) method
    - The putExtra(String, value) method is overloaded so you can pass many possible types
    - Call putExtra(String, value) as many times as needed for the data you're passing
    - Each call to putExtra(String, value) is setting up a key/value pair and you will use that key to access that value in the intent you're starting.

Receiving Intents
When a new activity starts it has access to the Intent passed to it

- Can access any data passed in the intent using the getExtra() methods

Using intents Android knows the sequence in which activities are started. This means that when you click on the Back button on your device, Android knows exactly where to take you back to.

Load web page

When the user taps on a recipe we want to load the recipe in a web page. We could create our own Activity class and load the web page or we can use the browser on the phone to load the web page. The first approach would be an explicit intent since we'd specify the class name in our app. The second approach will use an implicit intent so the device can use whatever app can display a url.
We'll use the second approach so the user will be prompted to choose an app to load the web page if the device has more than one capable of doing so.
It's best to start a new activity from an Activity class so we'll create a function in MainActivity that takes in a url and creates an intent to open the url in a browser.

```
fun startDetailActivity(url: String = "https://www.google.com"){
    var intent = Intent()
    intent.action = Intent.ACTION_VIEW
    intent.data = Uri.parse(url)
    startActivity(intent)
}
```

Now we need to pass this function as a callback through our composables down to RecipeItem where we'll assign it to the onClick event.
Update the following composable function headers to add a lambda as a parameter.

```
fun ScreenSetup(loadWebPage: (String) -> Unit)
fun RecipeScreen(loadWebPage: (String) -> Unit)
fun RecipeItem(item: Recipe, context: Context, deleteBook: (String) -
> Unit, loadWebPage: (String) -> Unit)
```

Now pass the function to these composable functions.
In onCreate()

```
ScreenSetup(loadWebPage = ::startDetailActivity)
```

Because `startDetailActivity` takes a parameter if we pass it as `{startDetailActivity()}` the value eventually passed as the parameter is ignored. Instead, we need to convert it to a function reference so it's passed as a lambda expression and the parameter value will be used.

In ScreenSetup()

```
RecipeScreen(loadWebPage = loadWebPage)
```

In RecipeScreen in the LazyColumn items loop

```
RecipeItem(item = recipe, context, {viewModel.deleteRecipe(it)},
loadWebPage=loadWebPage)
```

And in RecipeItem we finally call the lambda, passing the url, in the onClick event handler.

```
onClick = {loadWebPage(item.url)},
```

In the Preview

```
RecipeScreen(loadWebPage = {})
```

Now when you tap on a recipe its web page should open in a browser.
The bowser app is added to the backstack so when you tap the back button you'll go back to your app.