Android programming

TX00CK66 Sensor Based Mobile Applications

Lecture Retrofit and LiveData

Jarkko.Vuori@metropolia.fi

Introduction

- Today’s mobile applications are Internet dependent
- Web services are used for supporting mobile application functionalities
  - Internet dependent monetization models
  - App usage statistics, data synchronization, social components, score tables, …
- Backend servers for push notifications

- Web Service
  - self contained and self describing application components that can be used by other applications and can be communicated by using open protocols (http://w3schools.sinsixx.com/webservices/ws_intro.asp.htm)

JSON Parsing

- JSON
    - JSON (JavaScript Object Notation) is a lightweight data-interchange format (used e.g. in WebServices)
        - It is easy for humans to read and write
        - It is easy for machines to parse and generate
    - It is gaining popularity against XML (an another data-interchange format)
    - https://www.json.org/json-en.html
- Gson library
    - convert Java Objects into their JSON representation (Serialization) and vice versa (Deserialization)
    - http://howtodoinjava.com/apache-commons/google-gson-tutorial-convert-java-object-to-from-json/

```
{
 "name":"John",
 "email":"John@mail.com",
 "address":{"city":"New York","state":"USA"}
}
```

JSON Parsing using Gson

The output is

{"address":{"city":"New York","state":"USA"},"email":"John@mail.com","name":"John"}

```
import com.google.gson.Gson

object GsonTester {
    object Model {
        data class Employee(val name: String, val email: String, val address: EmployeeAddress)
        data class EmployeeAddress( val city: String, val state: String)
    }

    fun test() {
        // Setting values of Employee POJO
        val employee = Model.Employee( "John",
                                    "John@mail.com",
                                   Model.EmployeeAddress(  "New York", "USA")
                                   )

        val gson = Gson()
        val json = gson.toJson(employee)

        println(json)
    }
}
```

Kotlin data class feature is an elegant solution (getters and setters are automatically generated)

Generating Java class from JSON text

- Simple tool to create java objects from JSON
  - http://www.jsonschema2pojo.org/
  - Plain Old Java Object (POJO) is an ordinary Java object, not bound by any special restriction and not requiring any class path
- The problem with the automated tools is that it gives Java classes, not Kotlin classes
  - but you may use automated Java -> Kotlin converter in Android Studio to remedy this
    - But it does not understand to use Kotlin new data class concept
- There is an experimental Json to Kotlin converter tool available, https://www.json2kotlin.com/
- There is also an interesting Json data type creator which is able to produce output in various languages (including Kotlin) and serialization frameworks (plain datatypes also) https://app.quicktype.io/
- There is also an interesting plug-in (e.g. Android Studio) https://plugins.jetbrains.com/plugin/9960-json-to-kotlin-class-jsontokotlinclass-/

```
// This is plain old Java class
// (for making an object)
public class Person {
    private final String firstName;
    private final String lastName;
    private final Integer Age;
}
```

Retrofit

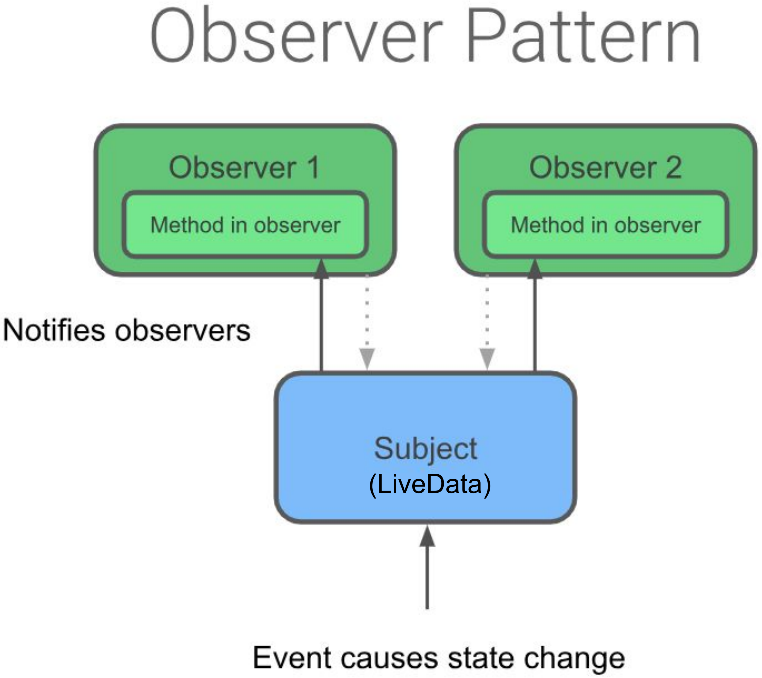
- Retrofit is a REST Client for Android and Java/Kotlin by Square
  - Representational State Transfer (REST) compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using a **uniform and predefined set of stateless** operations
    - stateless protocol means that each request message can be understood in isolation, i.e. neither sender nor receiver stores any information about the state of the communication - it is easy protocol to implement (and quite efficient if the probability of a communication error is low)
  - Square (https://squareup.com/) is a financial services, merchant services aggregator, and mobile payment company based in San Francisco, California
    - CEO (and co-founder) is Jack Dorsey, who is also CEO (and co-founder) of Twitter
- It makes relatively easy to retrieve and upload JSON (or other structured data) via a REST based web service
- In Retrofit you configure which converter is used for the data serialization
- Typically for JSON you use Gson, but you can add custom converters to process XML or other protocols
- Retrofit uses the OkHttp library for HTTP requests

Retrofit implementation (of Application)

- Type-safe REST client for Android and Kotlin/Java
- turns your REST API into a Kotlin/Java interface
- upon downloading the data is parsed into POJO which must be defined for each “resource” in the response
- flexible in message format
- uses compile-time annotation processor
- New versions are able to use coroutines (starting from the Retrofit version 2.6.0)
    - which means it is LiveData compatible

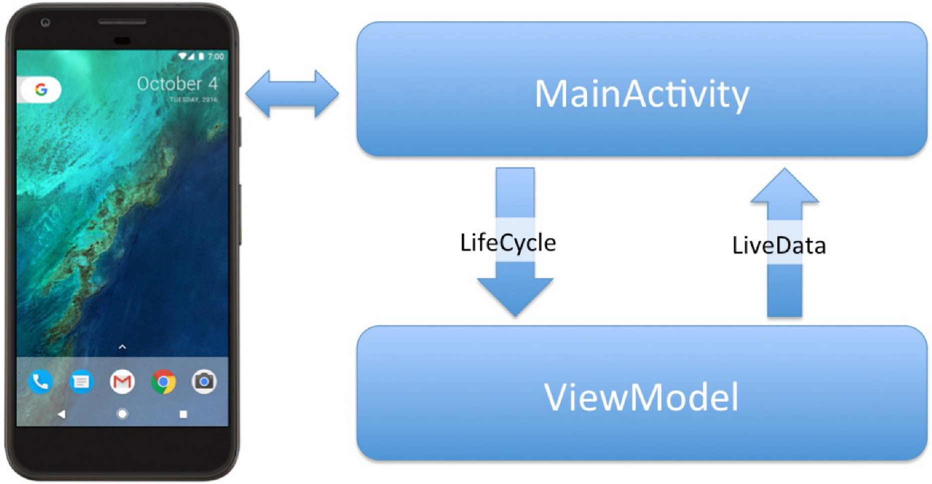
LiveData

- LiveData is Android Architectural Component which is a lifecycle-aware observable data holder class (= implements observer pattern)
  - ensures LiveData only updates app component observers that are in an active lifecycle state
- LiveData handles also the synchronization between threads (or coroutines)



ViewModel

- In order to separate program logic (ViewModel) and the actual UI (MainActivity), we have the concept of ViewModel
  - ViewModel subscribe to Activity through its LifeCycle object to get notified on LiveCycle events
  - Activity subscribe to ViewModel through it’s LiveData object to get data updates



ViewModel and LiveData

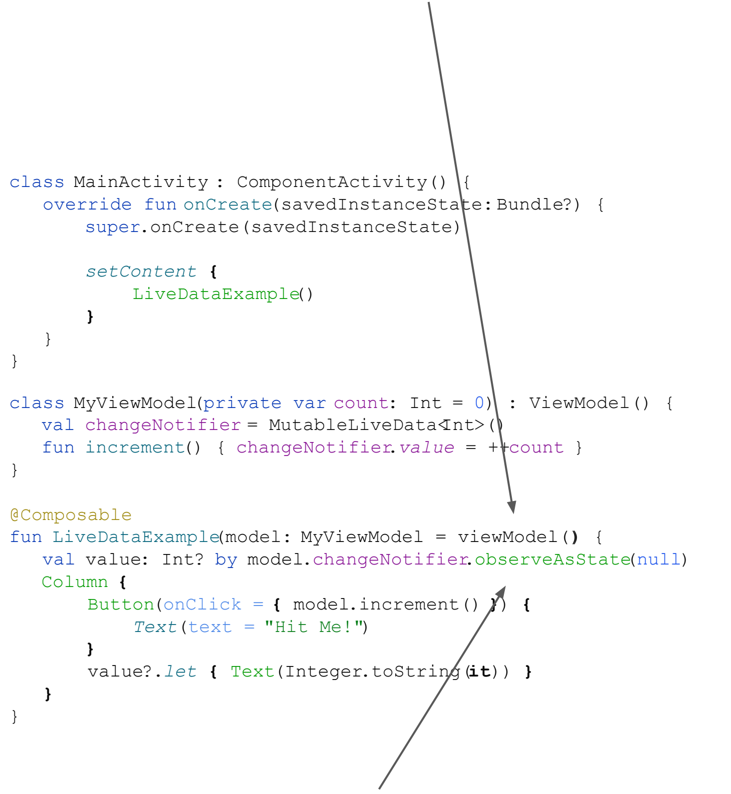
- This ViewModel (MyViewModel) holds only one integer value and methods to manipulate it, i.e. the program logic
  - it also contains LiveData which is observable
- MainActivity requests notifications when the LiveData changes its value
  - these notifications are propagated by calling the changeObserver lambda function
- Clicking the main layout makes the increment() method to be called

```
class MyViewModel(private var count: Int = 0) : ViewModel() {
   val changeNotifier = MutableLiveData<Int>()
   fun increment() { changeNotifier.value = ++count }
}

class MainActivity : AppCompatActivity() {
   private val viewModel: MyViewModel by lazy {
       ViewModelProvider(this).get(MyViewModel::class.java)
   }

   private val changeObserver =
       Observer<Int> {
              value -> value?.let { incrementCount(value) }
       }

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_main)

       viewModel.changeNotifier.observe(this, changeObserver)
       layoutMain.setOnClickListener { viewModel.increment() }
   }

   private fun incrementCount(value: Int) {
       txtNumber.text = (value).toString()
   }
}
```

remember: implementation "androidx.lifecycle:lifecycle-viewmodel-compose:$lifecycle_version"
in build.gradle (module)

LiveData in Jetpack Compose

- In layout system (imperative UI world), LiveData observer function is called when the value has been changed
  - When observing a LiveData you would explicitly instruct the view to change the necessary value (e.g. txtNumber.text = (value).toString())
- In Jetpack Compose (declarative UI world), when you assign a value to a UI element, the UI gets redrawn automatically when that value changes
- Jetpack Compose's State<T> is responsible for this automatic "redraw" (in Compose it's called "recomposition")
  - So all you need to do is convert your LiveData to a State
  - LiveData's observeAsState() does this conversion



```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            LiveDataExample()
        }
    }
}

class MyViewModel(private var count: Int = 0) : ViewModel() {
    val changeNotifier = MutableLiveData<Int>()
    fun increment() { changeNotifier.value = ++count }
}

@Composable
fun LiveDataExample(model: MyViewModel = viewModel()) {
    val value: Int? by model.changeNotifier.observeAsState(null)
    Column {
        Button(onClick = { model.increment() }) {
            Text(text = "Hit Me!")
        }
        value?.let { Text(Integer.toString(it)) }
    }
}
```

remember: implementation "androidx.compose.runtime:runtime-livedata:<current version>"
in build.gradle (module)

Retrofit implementation

- First define the data model
  - either using the automated converter (http://www.jsonschema2pojo.org/) and then using Android Studio Java -> Kotlin konversion
  - or creating the data model by hand
    - not so complicated because Kotlin data class generates getters and setters automatically
    - Retrofit does not need every field to be specified
      - only those fields you are interested of are needed

```
object Model {
   data class Employee(val name: String, val email: String, val address: EmployeeAddress)
   data class EmployeeAddress(val city: String, val state: String)
}
```

Retrofit implementation

- Remember to add the following lines to the application **module** build.gradle

```
dependencies {
        .
        .
        .
  final retrofit version = '2.9.0'
  implementation "com.squareup.retrofit2:retrofit:$retrofit version"
  implementation "com.squareup.retrofit2:converter-gson:$retrofit_version"
        .
        .
        .
}
```

- In your code, create Retrofit instance using builder of the Retrofit class
  - in this way we give parameters to the Retrofit adapter
    - URL is the URL of the Web Service endpoint
    - `GsonConverterFactory.create()` tells that we will use Gson to make conversions to/from JSON

```
private val retrofit = Retrofit.Builder()
        .baseUrl(URL)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
```

Retrofit implementation

- Define server API (HTTP methods) interface
  - @GET annotation instructs Retrofit to use HTTP GET request
    - it is possible here to give extension to the base URL endpoint
  - @Query annotation gives and additional parameter to the HTTP GET
    - if the action contains name “Enzio Benzino”, the query will be like …?name=Enzio%20Benzino…
  - Each call yields its own HTTP request and response pair
    - Make an asynchronous network request by using coroutines (function can suspend the current coroutine)
    - So when you make a call to the method `userName("Ensio Benzino")`, a HTTP GET request to the URL `https://datastorage.corporation.com/api.php?name=Enzio%20Benzino` is generated, and the returned JSON text is converted to the `Model.Employee` object returned from the method

```
interface Service {
   @GET("api.php")
   suspend fun userName(@Query("name") action: String): Model.Employee
}
```

Retrofit implementation

- Finally we need to create the service with create method

- We can pack the whole API (for the external server) to the enclosing object (here DemoApi) to keep those components together

This singleton DemoApi is used only to keep Retrofit things together in program’s namespace

```
object DemoApi {
    const val URL = "https://datastorage.corporation.com/"

   object Model {
     data class Employee(val name: String, val email: String, val address: EmployeeAddress)
     data class EmployeeAddress(val city: String, val state: String)
    }


   interface Service {
      @GET("api.php")
      suspend fun userName(@Query("name") action: String): Model.Employee
    }


    private val retrofit = Retrofit.Builder()
            .baseUrl(URL)
            .addConverterFactory(GsonConverterFactory.create())
            .build()

    val service = retrofit.create(Service::class.java)!!
}
```

https://datastorage.corporation.com/api.php?name=Enzio%20Benzino

Retrofit implementation

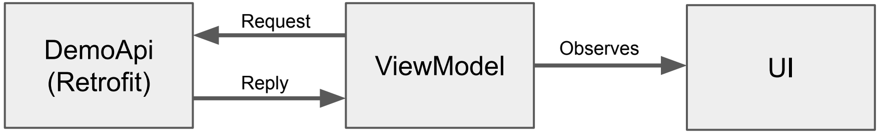
- Repository is a central location in which data is stored and managed
- MainViewModel is the ViewModel where the program logic (just a request to the server) is implemented
- when we update LiveData values from coroutines, we need to use `postValue()` method instead of direct assignment

```
class WebServiceRepository() {
   private val call = DemoApi.service

   suspend fun getUser(name: String) = call.userName(name)
}

class MainViewModel: ViewModel() {
   private val repository: WebServiceRepository = WebServiceRepository()

   val employee = MutableLiveData<String>()

   fun getUserName(name: String) {
       viewModelScope.lauch(Dispatcher.IO) {
           val retrievedEmployee = repository.getUser("Enzio Benzino")

           employee.postValue(retrievedEmployee.name)
       }
   }
}
```

Retrofit implementation

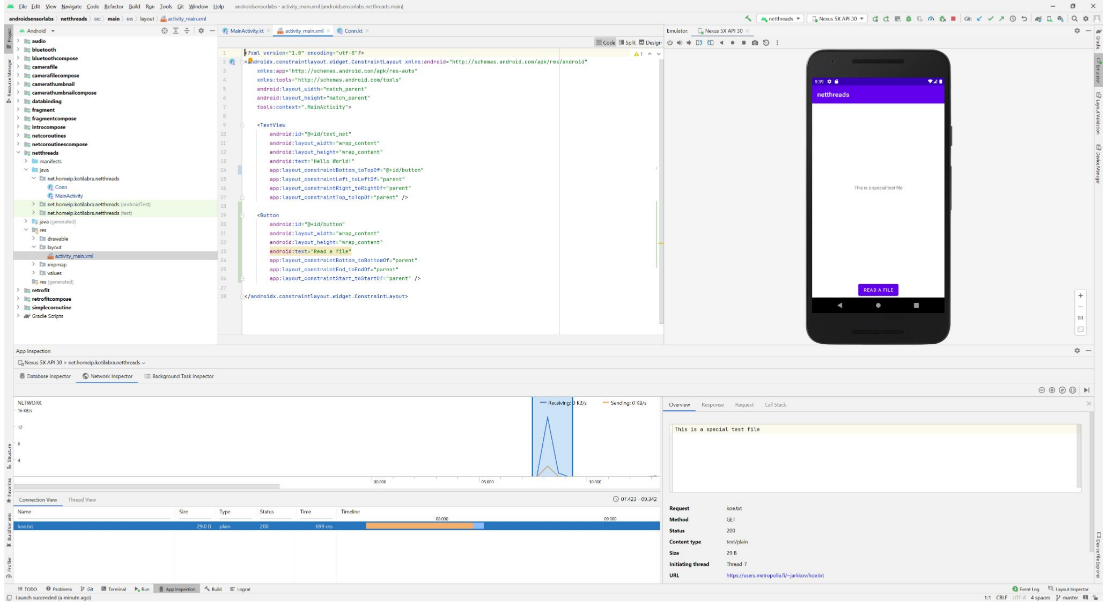
- On the Activity side, life is easy. Just register the observer to be a State in Composable

```
class MainActivity : ComponentActivity() {

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContent {
           ShowEmployee()
       }
       model.getUserName( "Enzio Benzino")
   }
}
        .
        .
        .
@Composable
fun ShowEmployee(model: MainViewModel = viewModel()) {
    val name: String? by model.employee.observeAsState(null)

    value?.let{ Text(name) }
}
```



Debugging

- Android Studio has a very elaborate profiler (called as Application Inspector) which can be used to analyze the application’s runtime behaviour
  - ○ Profiler is activated by selecting the App inspector menu item at the bottom of the screen
- Network operation can be analyzed using Network Inspector (timing, server responses, amount of data, message content, etc.)



```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/text_net"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toTopOf="@+id/button"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Read a file"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Name	Size	Type	Status	Time	Timeline
koe.txt	29.0 B	plain	200	699 ms	

```
This is a special test file
```

Request	koe.txt
Method	GET
Status	200
Content type	text/plain
Size	29 B
Initiating thread	Thread-7
URL	https://users.metropolia.fi/~jarkkov/koe.txt

Reading list

- http://www.vogella.com/tutorials/Retrofit/article.html
- http://square.github.io/retrofit/
- https://code.tutsplus.com/tutorials/getting-started-with-retrofit-2--cms-27792
- https://developer.android.com/training/basics/network-ops/connecting

- https://developer.android.com/topic/libraries/architecture/livedata
- https://www.raywenderlich.com/9217202-coroutines-with-lifecycle-and-livedata