Full usage example of App Architecture, Retrofit, Room, Hilt and Coroutines.

Download the Starter Files for this project here
https://drive.google.com/file/d/1yM2Y3lc4S20mAoiCWNKrAisoT-Jg07q4/view?usp=sharing

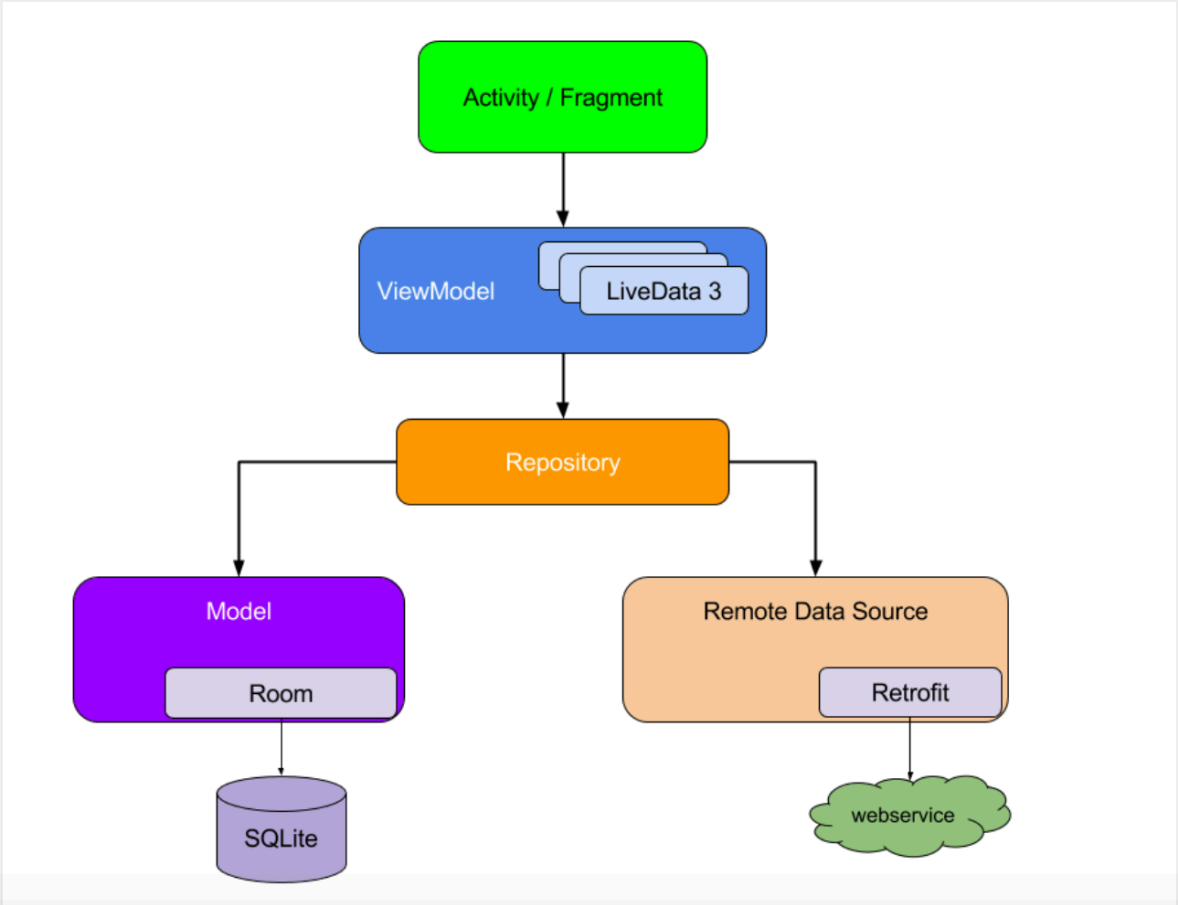
Download the starter Project from here (Optional - not necessary for this tutorial):
https://drive.google.com/file/d/1i0wSsU6s9mqnaePDqYl8Ls0MfB6GUBCS/view?usp=sharing

Download the Full App Created in this Guide:

https://drive.google.com/file/d/1sl_T-_zr8w0z7riMyvpNlReB8WcYN5N6/view?usp=sharing

Download the Generic useful classes to use in other projects:
https://drive.google.com/file/d/1fpH2qcyI0020pY0ZbB1yN69wmU0xT_NG/view?usp=sharing

We are going to put it all together now for building a clean architecture app that will communicate with both local and remote databases and will use Hilt dependency injection library for reducing boilerplate code.



Before we begin diving into our code let's look at the remote database. We will use one of the most Common web service for testing: the rick and morty API. You can explore it here https://rickandmortyapi.com just visit the docs and look the REST API, we have the characters, locations and episodes data, we will get all the characters and show them nicely in a RecyclerView and by clicking on each character we will show it in details.

So our base url will be:

https://rickandmortyapi.com/api

And we will access the /character resource

Lets look at the retuned JSON:
You can use this site for json formatting - just paste the full url
https://jsonformatter.curiousconcept.com
https://rickandmortyapi.com/api/character

```
{
    "info":{ },
    "results":[ ]
}
```

```
{
   "info":{
      "count":671,
      "pages":34,
      "next":"https://rickandmortyapi.com/api/character?page=2",
      "prev":null
   },
   "results":[
      {
         "id":1,
         "name":"Rick Sanchez",
         "status":"Alive",
         "species":"Human",
         "type":"",
         "gender":"Male",
         "origin":{
            "name":"Earth (C-137)",
            "url":"https://rickandmortyapi.com/api/location/1"
         },
         "location":{
            "name":"Earth (Replacement Dimension)",
            "url":"https://rickandmortyapi.com/api/location/20"
         },
         "image":"https://rickandmortyapi.com/api/character/avatar/1.jpeg",
```

```
"results":[
   {
      "id":1,
      "name":"Rick Sanchez",
      "status":"Alive",
      "species":"Human",
      "type":"",
      "gender":"Male",
      "origin":{ },
      "location":{ },
      "image":"https://rickandmortyapi.com/api/character/avatar/1.jpeg",
      "episode":[ ],
      "url":"https://rickandmortyapi.com/api/character/1",
      "created":"2017-11-04T18:48:46.250Z"
   },
```


Even though we need only the JSON objects in the result array in order to simplify the Gson Builder factory we will create data classes for the character, the info object and the root object which will contain a list of characters and the

total info.

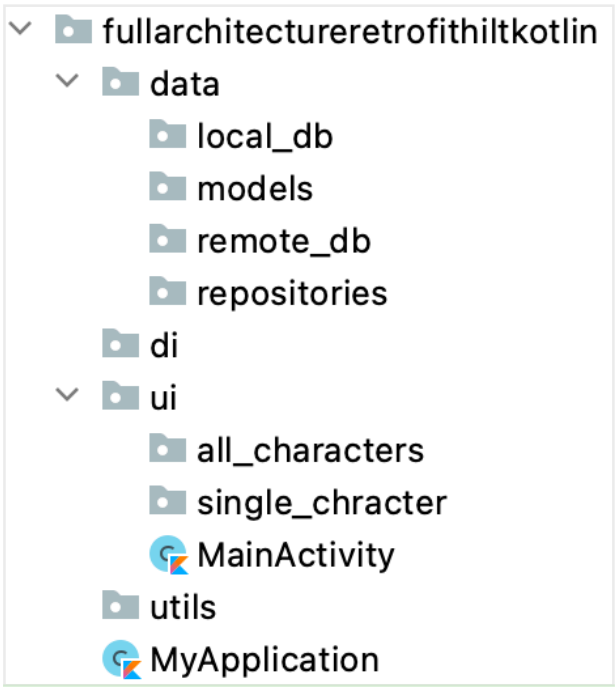
Step 1 - Project Setup

Go ahead open a new Android Studio Project and create the following project structure:

In our Project we will divide our project to:
data - it will include our models, local and remote database data operations and repositories.
di - it will include all of our injected dependencies and we will to that with the help of Hilt.
ui - all of out UI related components and their ViewModels (yes view models goes there).
utils - all of the helper classes and project related general functions.
Go ahead and create the packages mentioned above and their sub folders.

Move your MainActivity to the ui package and in the root package create your Application class for Hilt and add it's name to your manifest file.

In the end it should look like this:




```
<application
    android:name=".MyApplication"
```

Also in the utils package a create Constants class to hold out Base url on which we will add the specific resource for each GET call (remember to add the / at the end of the path:

```
class Constants {

    companion object {
        const val BASE_URL = "https://rickandmortyapi.com/api/"
    }
}
```

If you are already in your Manifest don't forget to the add the Internet install time permission used for our retrofit calls:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Now let's go to our project and app Gradle files and get all the 3rd party libraries dependencies:

In your project Gradle file we just need to add Hilt:

```
dependencies {
      ...
      classpath 'com.google.dagger:hilt-android-gradle-plugin:2.38.1'
   }
```

In your app Gradle file we need add the plugins:

```
id 'kotlin-kapt'
id 'dagger.hilt.android.plugin'
```

And a whole bunch of stuff under our dependency :

```
//Retrofit
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'

//Lifecycle
def lifecycle_version = "2.3.1"
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"
implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"
implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"
implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'

//Kotlin Coroutines
def coroutines_android_version = '1.5.2'
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:
$coroutines_android_version"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:
$coroutines_android_version"
```

```
//Hilt
implementation 'com.google.dagger:hilt-android:2.38.1'
implementation "androidx.hilt:hilt-lifecycle-viewmodel:1.0.0-alpha03"
kapt 'com.google.dagger:hilt-android-compiler:2.38.1'
kapt "androidx.hilt:hilt-compiler:1.0.0"

//Room
def room_version = "2.3.0"
implementation "androidx.room:room-runtime:$room_version"
implementation "androidx.room:room-ktx:$room_version"
kapt "androidx.room:room-compiler:$room_version"

//Navigation
def nav_version = "2.3.5"
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"

//Glide
implementation 'com.github.bumptech.glide:glide:4.12.0'
kapt 'com.github.bumptech.glide:compiler:4.12.0'
```

Go ahead now sync your project

Step 2 - Create your models

Create all the data Classes we have talked about in the beginning.
When you create your models do it carefully and look at the Json response. You don't have to include properties for each JSON field in the response but why not, maybe we will need it, if you are lazy we only need the **id, name, gender, status and species** but make sure you use the exact same names for the properties as in the response or retrofit won't be able to create your objects.
If for some reason you want a different name you can use @SerializedName([JSON field name])- same as @ColumnInfo we have seen in Room.

```
@SerializedName( value: "image")
val picture : String,
```

One last thing pre-plan your room annotations in your Data classes

In the end it should look like this (for this project the classes don't have to be data class but it is wiser):

```
@Entity(tableName = "characters")
data class Character(
    @PrimaryKey
    val id : Int,
    val name : String,
    val status : String,
    val species : String,
    val type : String,
    val gender : String,
    val image : String,
    val url: String,
    val created: String
    ) {
}
```

```
data class Info(
    val count : Int,
    val pages : Int,
    val next : String,
    val prev : Any
)  {
}
```

```
data class AllCharacters(
    val info : Info,
    val results : List<Character>
    ) {
}
```

Step 3 - Adding Hilt

Let's start with Hilt. Why do we need Hilt you can ask but think of all the dependencies we have here. The view Model will need the repository, the repository will require both the local and the remote database. Each database requires his service and so on. This is just a simple app, the more complex it gets the more dependencies it has.

This is where Hilt comes to rescue, we can auto inject each dependency and not have to worry about writing unnecessary code. We just need to tell Hilt where we need a dependency and where to get it from, it will connect the dots and take care of the object creation and their lifecycles.

First add the **@HiltAndroidApp** to your application file

```
@HiltAndroidApp
class MyApplication() : Application() {

}
```

Create both of your fragments and add to them @AndroidEntryPoint annotation meaning they can get the Hilt dependencies.

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
```

```
@AndroidEntryPoint
class SingleCharacterFragment : Fragment() {
}
```

```
@AndroidEntryPoint
class AllCharactersFragment : Fragment() {
}
```

Also create their View Models and add the @HiltViewModel and the @Inject constructor

```
@HiltViewModel
class AllCharactersViewModel @Inject constructor() : ViewModel(){
}
```

```
@HiltViewModel
class SingleCharacterViewModel @Inject constructor() : ViewModel() {
}
```

Now since we don't own the retrofit classes and we can't created an Injected constructor we must build the Retrofit module inside our di package. There we

must annotate by @Provides each of the dependencies we need to provide, this will be our bag of dependencies. The class will be installed in the SingletonComponent, meaning it will be available for all of the app. We also use the @Singleton annotation in cases where we want only one instance of the provided dependency. For now we just add a function that returns a single instance of the retrofit and also his constructor's Gson dependency.

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {

    @Provides
    @Singleton
    fun provideRetrofit(gson: Gson) : Retrofit {
        return Retrofit.Builder().baseUrl(Constants.BASE_URL)
            .addConverterFactory(GsonConverterFactory.create(gson)).build()
    }

    @Provides
    fun provideGson() : Gson = GsonBuilder().create()


}
```

One more thing. I want to explain more about the Gson. Here we are not doing any specific deserialization meaning our response JSON object will be directly mapped into the corresponding object - the json object has two keys one for the info and one for the result and so does the Kotlin data class. But if we wanted to map the result to a character class without the extra classes on the way then we would have needed to go into the root object get the array under the key results and fit each json object there to the character class - this is custom deserialization. For this we would have need to pass a custom Gson converter factory and not the standard one like we did here.

A nice and very simple tutorial on how to do this can be found here: https://www.woolha.com/tutorials/retrofit-2-define-custom-gson-converter-factory

We will get back to this module and add provider functions to the rest it's of the data parts. But now let's go ahead and create them, only then we can supply dependency for them.

Step 4 - Room Database

We will start From the easy part - Room Database

Besides the retrofit work, we are also interested in storing and getting data from a local database. We need it in order to show at least something to our users when they are out of connection or before the data is fetched - with slow connection. We are going to use it as a *cache* for our system.

Add your Dao and create functions to retrieve a character by id and all characters, and functions to insert a character or character list. Please make the insert functions suspended and the only thing we need is to execute it from a coroutine scope. Room will take care of their implementation including their background capabilities. Please note that the fetching functions don't have to be suspended because LiveData is already asynchronous - it is working on the IO Dispatchers.

Our Dao should look like this:

```
@Dao
interface CharacterDao {

    @Query( value: "SELECT * FROM characters")
    fun getAllCharacters() : LiveData<List<Character>>

    @Query( value: "SELECT * FROM characters WHERE id = :id")
    fun getCharacter(id : Int) : LiveData<Character>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCharacter(character: Character)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCharacters(characters : List<Character>)

}
```

And our app database will look like this

```
@Database(entities = [Character::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {

    abstract fun characterDao() : CharacterDao

    companion object {

        @Volatile
        private var instance : AppDatabase?  = null

        fun getDatabase(context: Context) : AppDatabase =
            instance ?: synchronized( lock: this) {
                Room.databaseBuilder(context.applicationContext,AppDatabase::class.java, name: "characters")
                    .fallbackToDestructiveMigration().build().also { it: AppDatabase
                        instance = it
                    }
            }
    }
}
```

Please note:
fallbackToDestructiveMigration() - tells Room that if the database version had changed and No Migration guide is found not to throw an exception but instead delete the old table and recreate it.

Step 5 - Retrofit calls

We are all done with the caching of the data and move to fetching. Let's create our data fetching service (same as the Dao in Room). This interface will be called CharacterService and it will contain two functions: both annotated with @Get("[path]"), both suspended meaning they can be executed on a background thread and both return the retrofit Response object with the Kotlin class that that json object should be parsed to. Amazing what could be achieved in a single line of code!
The code should look like this:

```
interface CharacterService {

    @GET( value: "character")
    suspend fun getAllCharacters() : Response<AllCharacters>

    @GET( value: "character/{id}")
    suspend fun getCharacter(@Path( value: "id") id: Int) : Response<Character>
}
```

Note:
You can use The @Query annotation to add the functions parameters and append them as the Query parameters.
For example:
If we want to add to the base URL this path "/maps/api/gecode/json?" With that parameters: "address=90210&sesnsor=false"
We will create the following Get call

```
@GET("/maps/api/gecode/json?")
```

suspend fun getLocationInfo(@Query("address") String zipCode,@Query("sensor") boolean sensor);

Retrofit also offers a returned Call that can be executed async but here the function itself is suspended so we don't need to use retrofit's background execution queue and execute Call on it.

The functions returns Response. When we invoke it we can check if it is successful and it has a data that can be null for both cases and a message that won't be null if there was an error and in that case there is also his code. More then that some exceptions can occur while executing the retrofit call (meaning not getting the response at all).
It seems like allot of different branching when all we care about is Success, Failure or Loading to be checked in a nicely organize when() {} clause. And we are going to do allot of work to achieve that, but this work will done once for all Requests. It will be Generic enough to serve us in every retrofit invocation.

First create your Wrapper class for the Response.
The base logic is to create a sealed Status object that works on a generic covariance that will be its data. From this sealed class there will be three derived classes: Success, Error (which will hold an additional error message) and Loading. We will create the Resource class with Status as her property and three Factory methods which we will create the suitable Response object with each status option :

```
class Resource<out T> private constructor(val status: Status<T>) {

    companion object {
        fun<T> success(data : T) = Resource(Success(data))
        fun<T> error(message: String, data : T? = null) = Resource(Error(message,data))
        fun<T> loading(data : T? = null) = Resource(Loading(data))
    }
}

sealed class Status<out T>(val data: T? = null)

class Success<T>(data: T) : Status<T>(data)
class Error<T>(val message:String, data: T? = null) : Status<T>(data)
class Loading<T>(data: T? = null) : Status<T>(data)
```

When we get an instance of this Resource wrapper class it will be very easy to check:

```
when (it.status) {
     is Success -> {//do something with the data that is not null

    }

    is Error -> {//read the error message (data may be null)

    }

    is Loading -> {//wait (data may be null)

    }
}
```

More than that, because Status is sealed the compiler will warn us if we forget to check any of it's subclasses.

So the wrapper class is ready now we need to actually call the function and wrap the Response with that Resource.
To do this in A Generic way we will create a base class with getResult function that will get the Response and return the appropriate Resource. Then we will inherit from that generic class to a specific data source for our service and call the generic getResult with functions from our service. Our data source will have an @injected constructor that will get the specific service and create wrapper functions for each of the service function. Don't worry about the injection we will provide the injected service in our AppModule

While we created our Resource classes in the util package generate this two classes in the remote db folder.

our new added code will look like this:

```
abstract class BaseDataSource {

    protected suspend fun <T> getResult(call : suspend () -> Response<T>) : Resource<T> {

        try {
            val result  = call() //we invoke the given suspended function(we are suspended also)
            if(result.isSuccessful) {
                val body  = result.body()
                if(body!=null) return Resource.success(body)
            }
            return Resource.error( message: "Network call has failed for a following reason: " +
                    "${result.message()} ${result.code()}")

        }catch (e : Exception) {
            return Resource.error( message: "Network call has failed for a following reason: " +
                    (e.localizedMessage ?: e.toString())
            )

        }
    }
}
```

```
@Singleton
class CharacterRemoteDataSource @Inject constructor(
    private val characterService: CharacterService) : BaseDataSource() {

    suspend fun getCharacters() = getResult { characterService.getAllCharacters() }
    suspend fun getCharacter(id : Int) = getResult { characterService.getCharacter(id) }
}
```

Step 6 - Repository

The very last thing we have to do in terms of our data is to create the Repository.
First let's explain our policy for local and remote data fetching:

- First we need to let our LiveData know that we are looking for the Character, so that should be the LOADING state.
- Then, we would like to get that character from the local data source, because it is faster than getting it from the internet. If it finds it, we are changing the state to a SUCCESS
- Regardless of the result of the local database operation, we would want to keep our app synched, so we are fetching the characters from the internet as well (but remember that the ui thread won't be blocked and the user can already see the correct characters information).
- Finally, we need to save our result from the remote call in the local database, in order to keep it updated.

To achieve that we can use a Generic get function that receives three functions as parameters: One for local fetching, one for remote fetching and one for saving data. Then we will use the LiveData coroutine builder to create scope for

running our suspended functions synchronously and get back the result as a LiveData object (please note that we need to tell the LiveData builder we want to run our job on the Dispatchers.IO because this default LiveData scope is the Main UI thread where emit is called). We are creating a scope where the suspended functions can wait for each other although they run in a background thread, and updating the data they fetched using emit which is called on the Main UI thread for anyone who is observing these returned LiveData. You can also emit multiple values from the block. Each emit() call suspends the execution of the block until the LiveData value is set on the main thread.
You can read more here:
https://developer.android.com/topic/libraries/architecture/coroutines#livedata

We will use both emit() and emitSource() function from within the LiveData scope. These functions defined by the LiveDataScope interface and are used to update the LiveData value or it's source. Meaning emit will be used to update the LiveData stored value and emitSource the LiveData itself and then each change will be auto updates by the LiveData. These are both suspended functions meaning it will pause the scope until the LiveData is updated.
Remember: Inside a suspended function, calls to other suspended functions behave like normal function calls. We stop and wait. We work on the same coroutine that can be stoped and continued.

So our helper function will look like that (put it a general DataFunction Kotlin file located in the util package):

If you are wondering why you see two generics it is because we need to distinguish the value stored in the LiveData from the value retuned by the call, for example in a single function call we are getting all the characters from the remote db we get a Response<AllCharacters> while from the local db we get LiveData<List<Character>>. The A represent the retrofit's generic while the T is the Room generic.

```
fun <T, A> performFetchingAndSaving(localDbFetch: () -> LiveData<T>,
                                    remoteDbFetch: suspend () -> Resource<A>,
                                    localDbSave: suspend (A) -> Unit) : LiveData<Resource<T>> =

    liveData(Dispatchers.IO) {  this: LiveDataScope<Resource<T>>
        emit(Resource.loading()) //this will tell the live data we are loading

        val source  = localDbFetch().map { Resource.success(it) } //mapping the returned
        //live data his wrapper class
        emitSource(source) //setting the new source for live data

        val fetchResource = remoteDbFetch()

        if(fetchResource.status is Success)
            localDbSave(fetchResource.status.data!!) //saving the data will also update ui any change of data
        //because we used emitSource with the local db livedata
        //since it is LiveData
        else if(fetchResource.status is Error)
        {
            emit(Resource.error(fetchResource.status.message))
            emitSource(source) //because each call to emit remove the previous value
            //we want to last valid value to be stored even if it is empt
        }
    }
```

And last but not least we will write the Character Repository. It's @Injected constructor will get the local and the remote services by Hilt and will use this function above to do all the work. Please note that we use the @Singleton scope so one instance of the repository is for all of the app. This repository will be auto created and injected later on to our view model.

```
@Singleton
class CharacterRepository @Inject constructor(
    private val remoteDataSource : CharacterRemoteDataSource,
    private val localDataSource : CharacterDao
){

    fun getCharacters() = performFetchingAndSaving(
        {localDataSource.getAllCharacters()},
        {remoteDataSource.getCharacters()},
        {localDataSource.insertCharacters(it.results)}
    )

    fun getCharacter(id : Int) = performFetchingAndSaving(
        {localDataSource.getCharacter(id)},
        {remoteDataSource.getCharacter(id)},
        {localDataSource.insertCharacter(it)}
    )
}
```

Add put them all together in you AppModule dependency bag
Remember if you @provide a class then all of its constructor parameters also have to be provided!
Our final AppModule should look like that:

```
@Module
@InstallIn(SingletonComponent::class)
object AppModule {

    @Provides
    fun provideGson() : Gson = GsonBuilder().create()

    @Provides
    @Singleton
    fun provideRetrofit(gson : Gson) : Retrofit {
        return Retrofit.Builder().baseUrl(Constants.BASE_URL)
            .addConverterFactory(GsonConverterFactory.create(gson)).build()
    }

    @Provides
    fun provideCharacterService(retrofit: Retrofit) : CharacterService =
        retrofit.create(CharacterService::class.java)

    @Provides
    @Singleton
    fun provideLocalDataBase(@ApplicationContext appContext : Context) : AppDatabase =
        AppDatabase.getDatabase(appContext)

    @Provides
    @Singleton
    fun provideCharacterDao(database: AppDatabase) = database.characterDao()

}
```

When we want to create our Retrofit service to execute our queries we can call create on our retrofit instance. It will create an implementation of the API endpoints defined by the service interface.
Please also note that we are using **@ApplicationContext** that allows hilt to provide application context without having to explicitly specify how to obtain it.
And also note that we don't need to provide the CharacterRepository and the CharacterRemoteDataSource since we they have the @Inject constructor meaning Hilt can generate these classes without the need to explicitly tell him how.

Thats It for our data and the dependencies! Now all we have to do is the easy part UI!

Step 7 - UI

First copy the four xml files found on the starter into your res/layout folder
Please note a few things:

1. In the activity_main we have a custom tool bar this for setup with the navigation component
2. Still in activity_main We Have the FragmentContainerView please notice his id and the nav graph id - when you create the nav graph use this id or change it here to what you will use

Go ahead and add your navigation graph to your resources with name corresponding the the Container mentioned above. Add both of your fragments and create an action between them like here:

Add your view binding to the app Gradle file

```
viewBinding {
    enabled = true
}
```

Define your view binding and pass the root view.
Get your Navigation controller And connect you toolbar to your navigation component for the purpose of showing the current fragment label in the app bar and navigating back. Your MainActivity should look like that:

```
@AndroidEntryPoint
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        val navHostFragment  = supportFragmentManager.findFragmentById(R.id.nav_host_fragment)
                as NavHostFragment
        val navController = navHostFragment.navController
        val appBarConfiguration = AppBarConfiguration(navController.graph)
        binding.toolbar.setupWithNavController(navController,appBarConfiguration)

    }
}
```

Let's start with all the characters and their RecyclerView adapter - use Glide for the pictures and also create an interface to pass the item click to the fragment who will implement the interface. When clicked pass the character id. In this example I prefer to get the listener in the adapters constructor and only it. The list of characters will be updated through setCharacters function you will add to the adapter.

Here is the full Adapter code:

```
class CharactersAdapter(private val listener : CharacterItemListener) :
    RecyclerView.Adapter<CharactersAdapter.CharacterViewHolder>() {

    private val characters = ArrayList<Character>()

    class CharacterViewHolder(private val itemBinding: ItemCharacterBinding,
                              private val listener: CharacterItemListener)
        : RecyclerView.ViewHolder(itemBinding.root),
        View.OnClickListener {

        private lateinit var character: Character

        init {
            itemBinding.root.setOnClickListener(this)
        }

        fun bind(item: Character) {
            this.character = item
            itemBinding.name.text = item.name
            itemBinding.speciesAndStatus.text = "${item.species} - ${item.status}"
            Glide.with(itemBinding.root)
                .load(item.picture)
                .circleCrop()
                .into(itemBinding.image)
        }

        override fun onClick(v: View?) {
            listener.onCharacterClick(character.id)
        }
    }
```

```
    fun setCharacters(characters : Collection<Character>) {
        this.characters.clear()
        this.characters.addAll(characters)
        notifyDataSetChanged()
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CharacterViewHolder {
        val binding = ItemCharacterBinding.inflate(LayoutInflater.from(parent.context),parent, attachToParent: false)
        return CharacterViewHolder(binding,listener)
    }

    override fun onBindViewHolder(holder: CharacterViewHolder, position: Int) =
        holder.bind(characters[position])


    override fun getItemCount() = characters.size

    interface CharacterItemListener {
        fun onCharacterClick(characterId : Int)
    }
}
```

Now for the final stage: Your Fragments and their View Models!

Let's start with AllCharacters - The ViewModel should supply the list of characters. All it needs in its Injected constructor is the repository. We will create a single characters property and get it from the repository. This will be

the observable LiveData.

```
@HiltViewModel
class AllCharactersViewModel @Inject constructor(
    characterRepository: CharacterRepository) : ViewModel(){

    val characters = characterRepository.getCharacters()
}
```

In the Fragment get your view bindings. Don't forget that the fragment outlives it's views so release your binding in the onDestroyView. Don't worry about the hassle, in the end you will get a property delegate that does this automatically while observing the corresponding fragment's lifecycle events.

Create the adapter and implement his Listener, the item click should perform the navigation's one and only pre-made action and pass a bundle containing the supplied character id.
Get the RecyclerView set it's layout manager and the created Adapter above. Now observe the characters from your view model and upon invocation check your status. in case of loading show the progress bar, In case of Success hide it set the adapter's characters, and in case of an error hide it and prompt the error message in a Toast.

```
@AndroidEntryPoint
class AllCharactersFragment : Fragment(),CharactersAdapter.CharacterItemListener {

    private val viewModel : AllCharactersViewModel by viewModels()

    private var _binding : CharactersFragmentBinding? = null
    //private val binding : CharactersFragmentBinding by autoCleared()
    private  val binding get() = _binding!!

    private lateinit var adapter: CharactersAdapter

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = CharactersFragmentBinding.inflate(inflater,container, attachToParent: false)

        return binding.root
    }
```

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    adapter = CharactersAdapter( listener: this)
    binding.charactersRv.layoutManager = LinearLayoutManager(requireContext())
    binding.charactersRv.adapter = adapter

    viewModel.characters.observe(viewLifecycleOwner) {  it: Resource<List<Character>>!
        when(it.status) {
            is Loading -> binding.progressBar.visibility = View.VISIBLE

            is Success ->  {
                binding.progressBar.visibility  = View.GONE
                adapter.setCharacters(ArrayList(it.status.data))
            }

            is Error -> {
                binding.progressBar.visibility = View.GONE
                Toast.makeText(requireContext(),it.status.message,Toast.LENGTH_SHORT).show()
            }
        }
    }
}
```

```
override fun onDestroyView() {
    super.onDestroyView()
    _binding = null
}

override fun onCharacterClick(characterId: Int) {

    findNavController().navigate(R.id.action_allCharactersFragment_to_singleCharacterFragment,
        bundleOf( ...pairs: "id" to characterId))

}
```

Our very last part of this very long journey is the detailed character fragment and it's View Model.
First, our View model should get the repository and get a character by it's id.
Now this is a tricky part. Think a little about how to solve this.

First we need a character property so we can observe it. But what will trigger the event?

We will create a character look up as a transformation of the id.
The id will have a public set function that we will invoke from the fragment.
The character field will be defined as a transformation of the id. As long as your app has an active observer associated with the character field, the field's value is recalculated and retrieved whenever id changes.

```
@HiltViewModel
class SingleCharacterViewModel @Inject constructor(
    private val characterRepository: CharacterRepository) : ViewModel() {

    private val _id = MutableLiveData<Int>()

    private val _character = _id.switchMap { it: Int!
        characterRepository.getCharacter(it)
    }

    fun setId(id : Int) {
        _id.value = id
    }
```

There is a reason why we use the internal _character - it is a mutable live data and therefore can be dangerous to expose that is why we will return only LiveData with a public character

```
val character : LiveData<Resource<Character>> = _character
```

For more reading on transformations:
https://developer.android.com/topic/libraries/architecture/livedata#transform_livedata

And that's it for the View Model

As for the fragment do the same binding as before and when your view is created get the id from the arguments, remember to deal nicely with null, and set the value in the view model value, this will trigger your character observer and upon invocation will update the ui!

```
@AndroidEntryPoint
class SingleCharacterFragment : Fragment() {

    private val viewModel : SingleCharacterViewModel by viewModels()

    private var _binding : CharacterDetailFragmentBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        _binding = CharacterDetailFragmentBinding.inflate(inflater,container, attachToParent: false)
        return binding.root
    }
```

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    viewModel.character.observe(viewLifecycleOwner) {  it: Resource<Character>!
        when(it.status) {
            is Success -> {
                binding.progressBar.visibility = View.GONE
                updateCharacter(it.status.data!!)
                binding.characterCl.visibility = View.VISIBLE
            }
            is Loading -> {
                binding.progressBar.visibility = View.VISIBLE
                binding.characterCl.visibility = View.GONE
            }
            is Error -> {
                binding.progressBar.visibility = View.GONE
                Toast.makeText(requireContext(),it.status.message,Toast.LENGTH_SHORT).show()
            }
        }
    }

    arguments?.getInt( key: "id")?.let {  it: Int
        viewModel.setId(it)
    }
}
```

```
    fun updateCharacter(character:Character) {
        binding.name.text = character.name
        binding.gender.text = character.gender
        binding.species.text = character.species
        binding.status.text = character.status
        Glide.with(requireContext()).load(character.image).transform(CircleCrop()).into(binding.image)
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null
    }
}
```

One last thing I have promised for a simpler solution to the fragment view binding.

Take AutoClearedValue.kt and copy it into your utility package. Until google will add this property delegate we will use this. This property delegate keeps track of the fragment lifecycle and upon destruction of the fragment update null value in the property

Use the by autoCleared() for your binding properties and remove the _binding and its related code.

```
//private var _binding : CharactersFragmentBinding? = null
private var binding : CharactersFragmentBinding by autoCleared()
// private  val binding get() = _binding!!

private lateinit var adapter: CharactersAdapter

override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
  //  _binding = CharactersFragmentBinding.inflate(inflater,container,false)

    binding = CharactersFragmentBinding.inflate(inflater,container, attachToParent: false)
    return binding.root
}
```