Hilt: dependency injection

https://www.youtube.com/watch?v=bbMsuI2p1DQ&ab_channel=PhilippLackner

What is Dagger-Hilt

Dependency injection is a design pattern. It is about giving an object its instance variables.

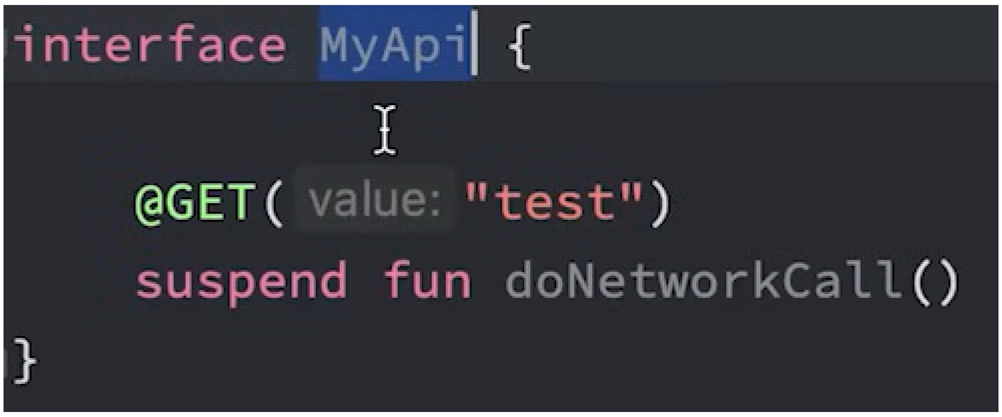
```
// Constructor injection
class Test(val name: String)
val test = Test("Philip") //passing "Philip" is injecting the
```

Dagger-Hilt is a library from Google that gives us flexibility and tools to dependency injection.

Setting up the structure

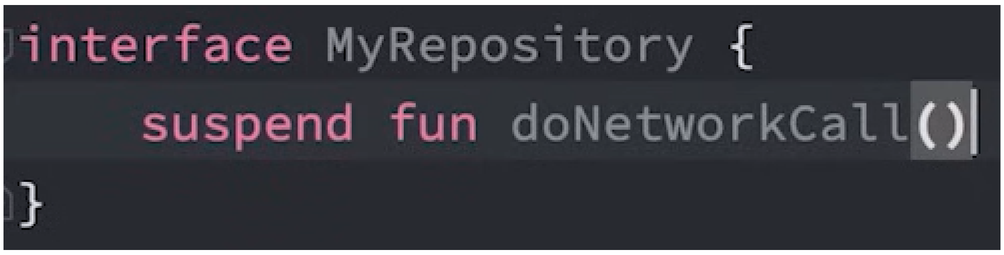
Import the dependencies of ViewModel, Dagger-Hilt and Retrofit.

Create a dummy API for learning purposes in data.remote.MyApi:



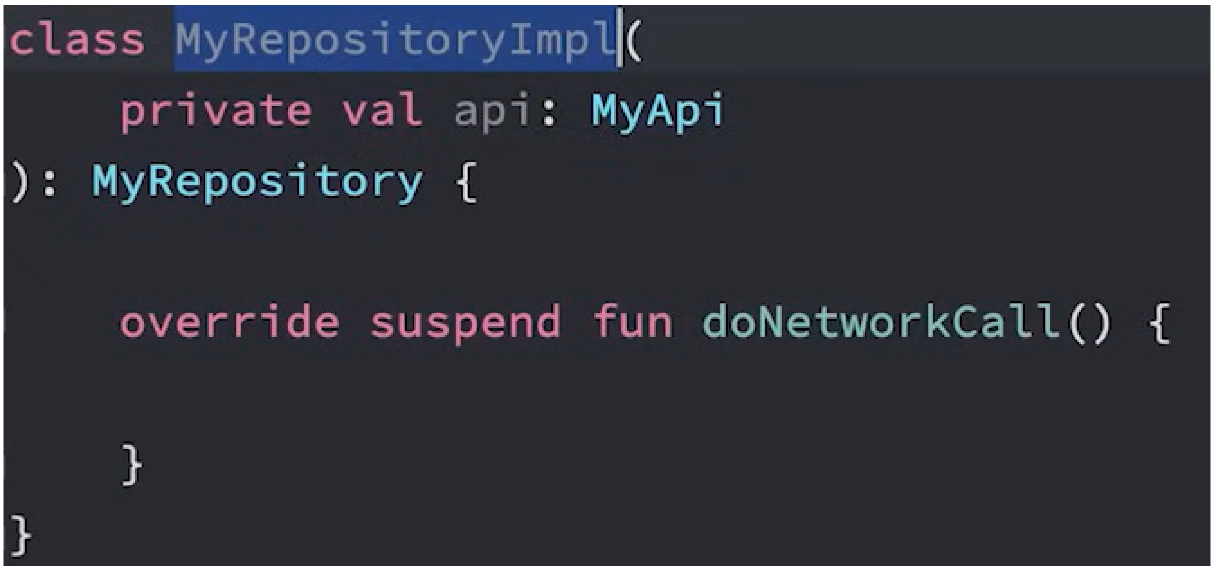
```
interface MyApi {

    @GET( value: "test")
    suspend fun doNetworkCall()
}
```

Create the packages domain.repository and inside the interface MyRepository.



```
interface MyRepository {
    suspend fun doNetworkCall()
}
```

Create the implementation of that interface in data.repository.



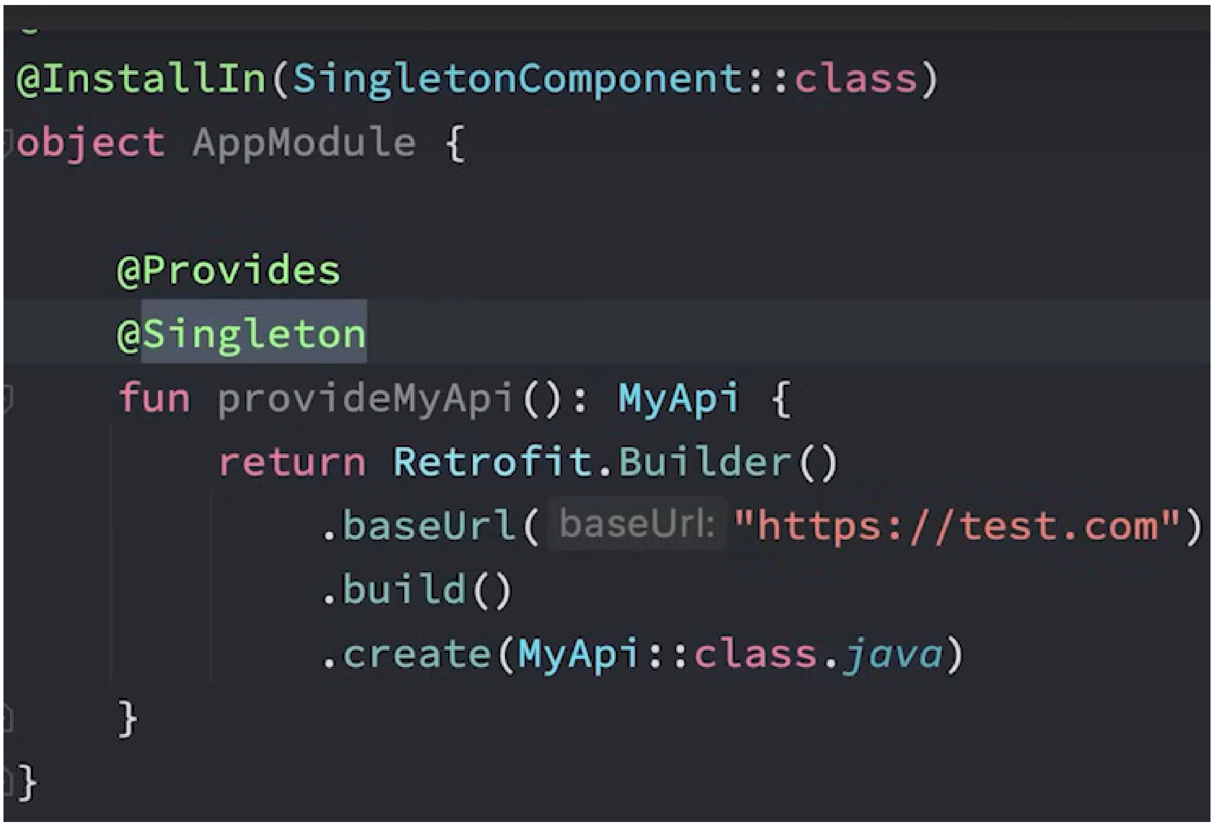
```
class MyRepositoryImpl(
    private val api: MyApi
): MyRepository {

    override suspend fun doNetworkCall() {

    }
}
```

Now we have to inject the MyApi dependency, we will do that by creating a module to tell Dagger-Hilt how to do it.

Creating a module

You should have different modules for different porpuses, for example a broadcasting module, an auth module and modules of specific scopes (for fragments, activities, etc.).

Inside a package called “di” create the object AppModule.



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

    @Provides
    @Singleton
    fun provideMyApi(): MyApi {
        return Retrofit.Builder()
            .baseUrl( baseUrl: "https://test.com")
            .build()
            .create(MyApi::class.java)
    }
}
```

Now whenever we inject a MyApi dependency Hilt will look in its modules and find such instance and pass it to the constructor.

The first annotation, before @InstallIn is @Module

The @Singleton annotation is the Scope, we will only have one dependency in the whole life of the application. If we didn't use it in the example before, in the case that we injected the dependency in two places, there would be two MyApis instances living the same as the application does.

The @InstallIn annotation decides about the lifetime of our dependencies in the module:

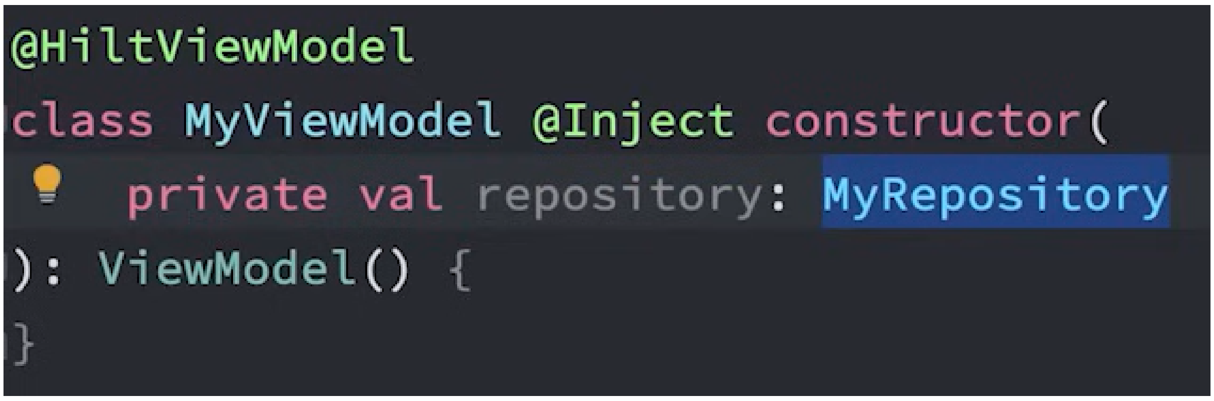
@InstallIn(SingletonComponent::class) means that the module will live as long as the whole application does.

@InstallIn(ActivityComponent::class) means that the module will live as long as the activity in which is injected does.

Injecting into ViewModels

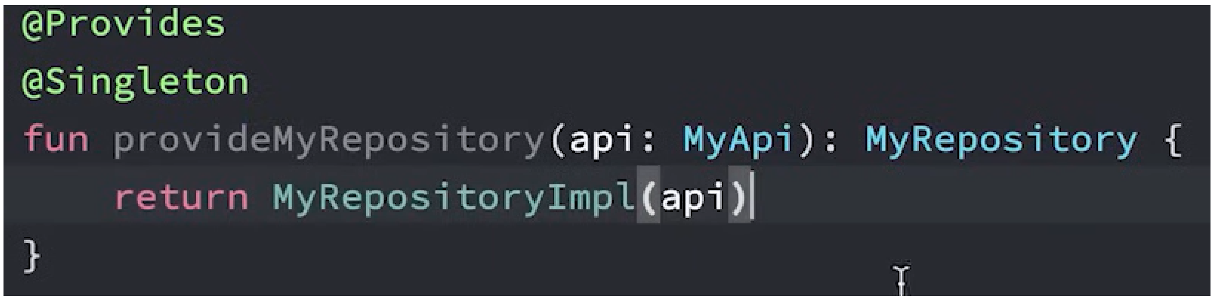
Creating the ViewModel needs a factory, that it is why we is difficult to inject in a ViewModel.

Now create the "MyViewModel" class:



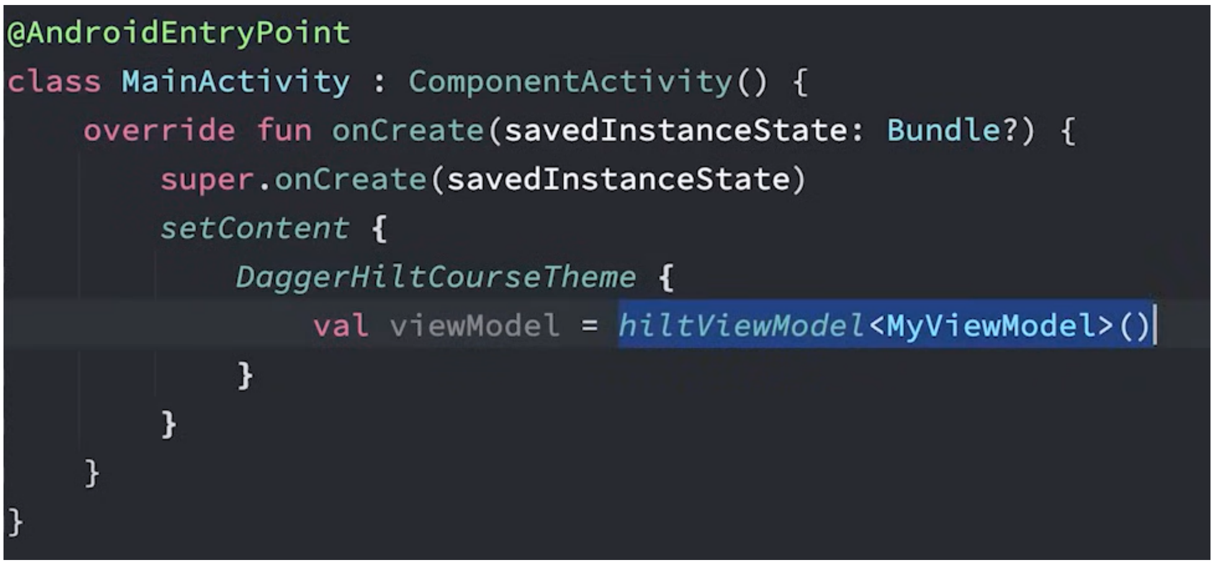
```
@HiltViewModel
class MyViewModel @Inject constructor(
    private val repository: MyRepository
): ViewModel() {
}
```

Define the dependency injection for the MyRepository variable in the Hilt Module:



```
@Provides
@Singleton
fun provideMyRepository(api: MyApi): MyRepository {
    return MyRepositoryImpl(api)
}
```

Call the ViewModel from the MainActivity:



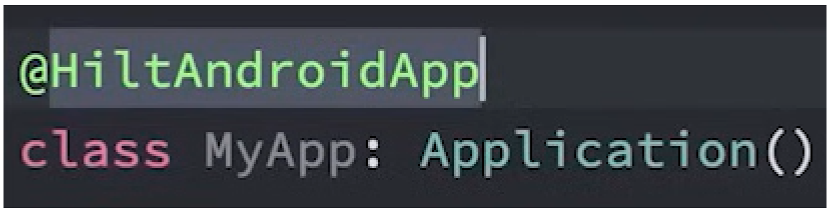
```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            DaggerHiltCourseTheme {
                val viewModel = hiltViewModel<MyViewModel>()
            }
        }
    }
}
```

The @AndroidEntyPoint annotation is needed because we are injecting the dependency of the viewModel in an android component class.

Creating the application class

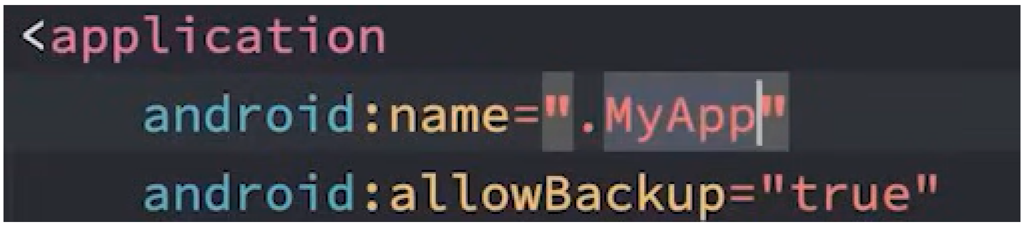
We need it for example if we wanted to inject a Context.

Create the class MyApp:



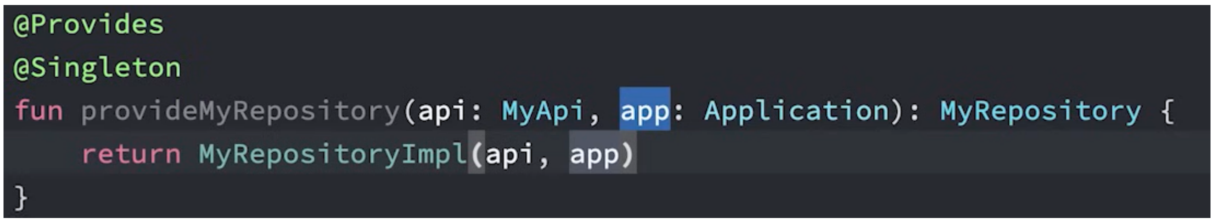
```kotlin
@HiltAndroidApp
class MyApp: Application()
```

Register it in the Manifest file:



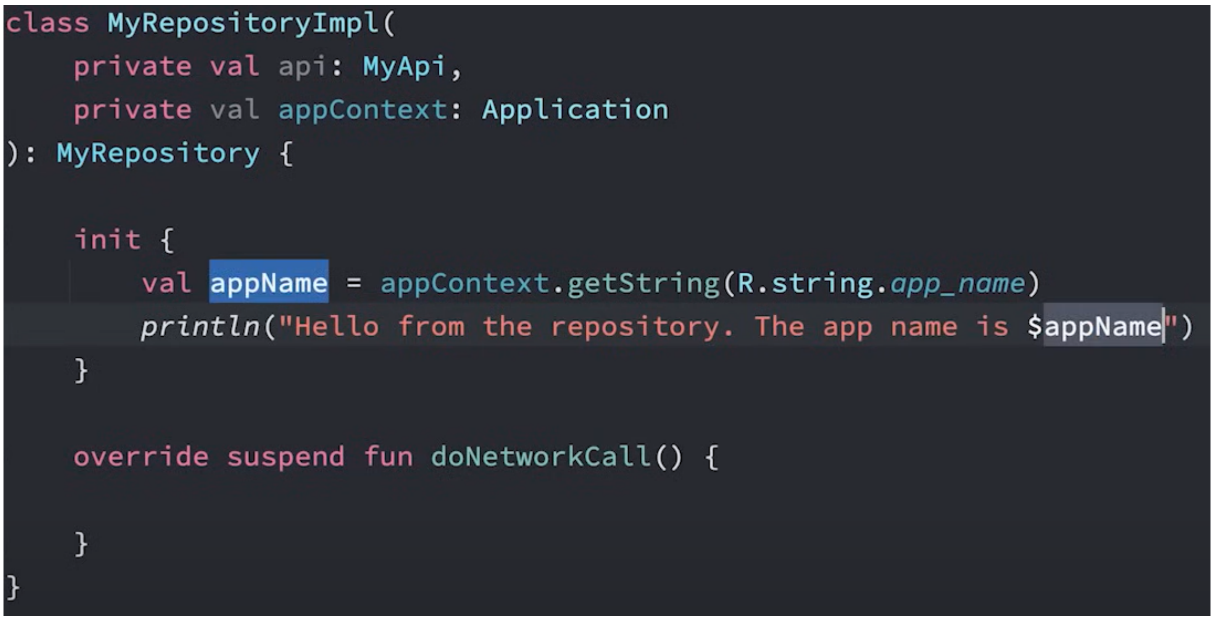
```xml
<application
    android:name=".MyApp"
    android:allowBackup="true"
```

Now you can inject the app:



```kotlin
@Provides
@Singleton
fun provideMyRepository(api: MyApi, app: Application): MyRepository {
    return MyRepositoryImpl(api, app)
}
```

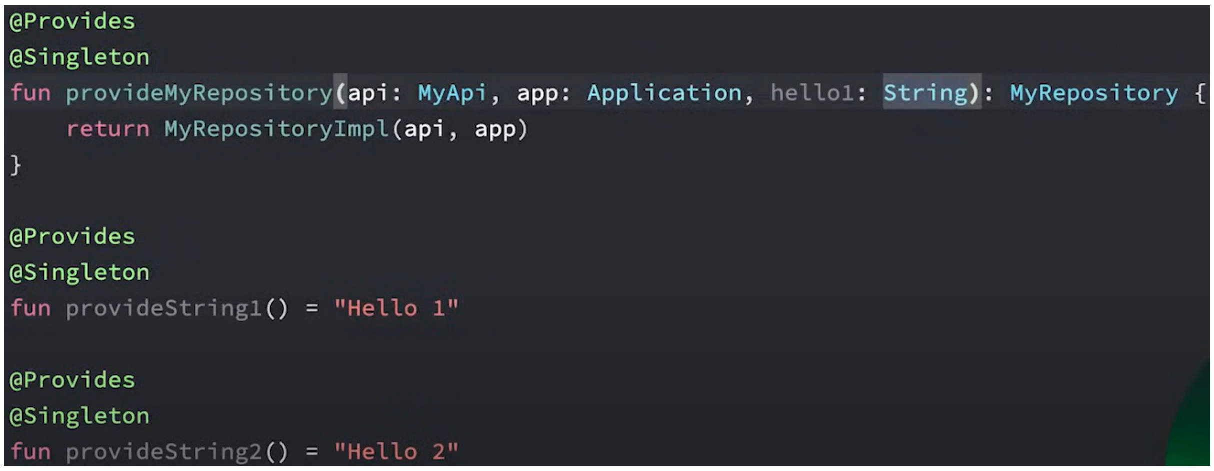
Now for example if we wanted to get a string in MyRepository we would do:



```kotlin
class MyRepositoryImpl(
    private val api: MyApi,
    private val appContext: Application
): MyRepository {

    init {
        val appName = appContext.getString(R.string.app_name)
        println("Hello from the repository. The app name is $appName")
    }

    override suspend fun doNetworkCall() {

    }
}
```

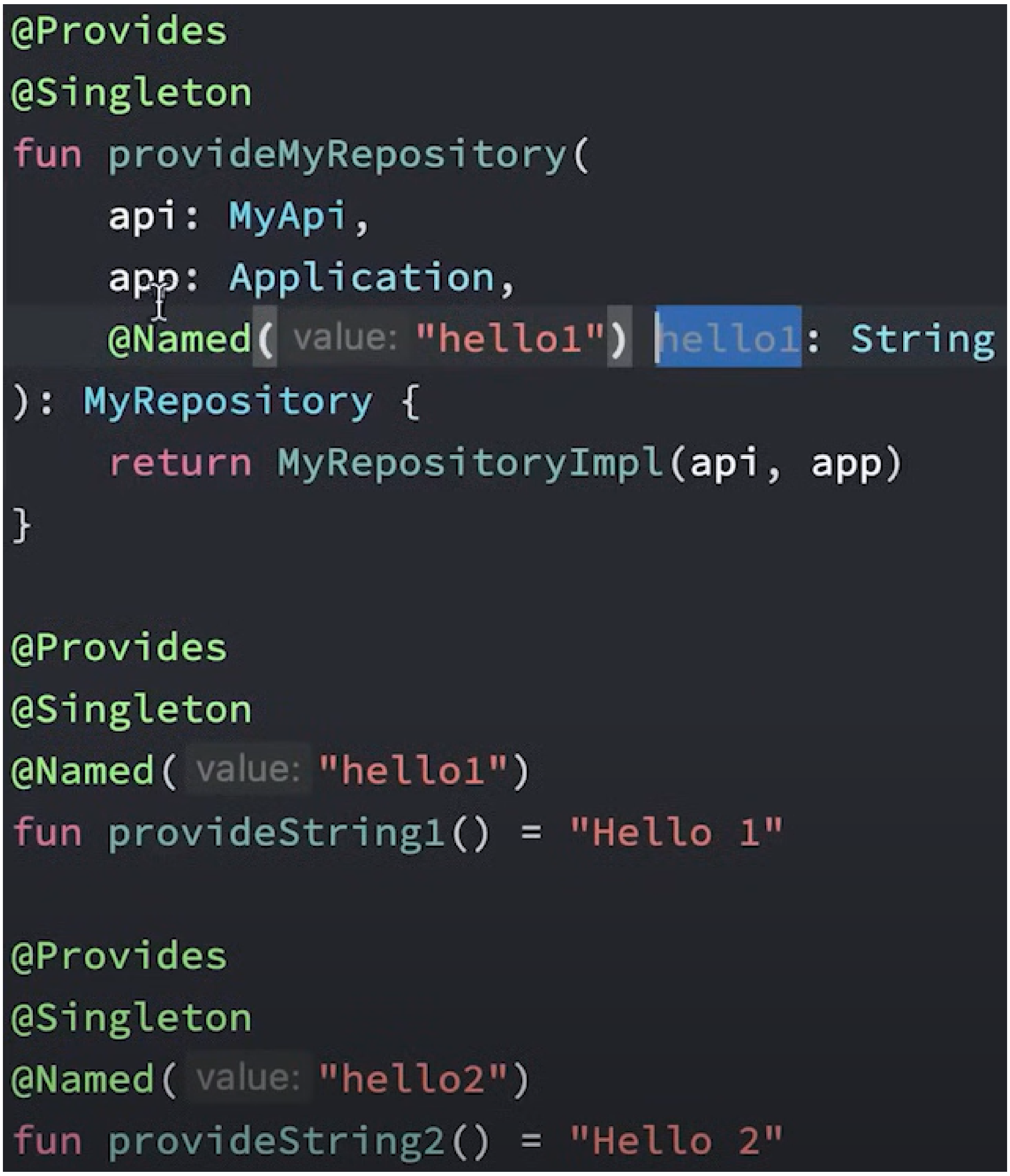
Two dependencies of the same type

For example, two provides functions that both provide a String:



```kotlin
@Provides
@Singleton
fun provideMyRepository(api: MyApi, app: Application, hello1: String): MyRepository {
    return MyRepositoryImpl(api, app)
}

@Provides
@Singleton
fun provideString1() = "Hello 1"

@Provides
@Singleton
fun provideString2() = "Hello 2"
```

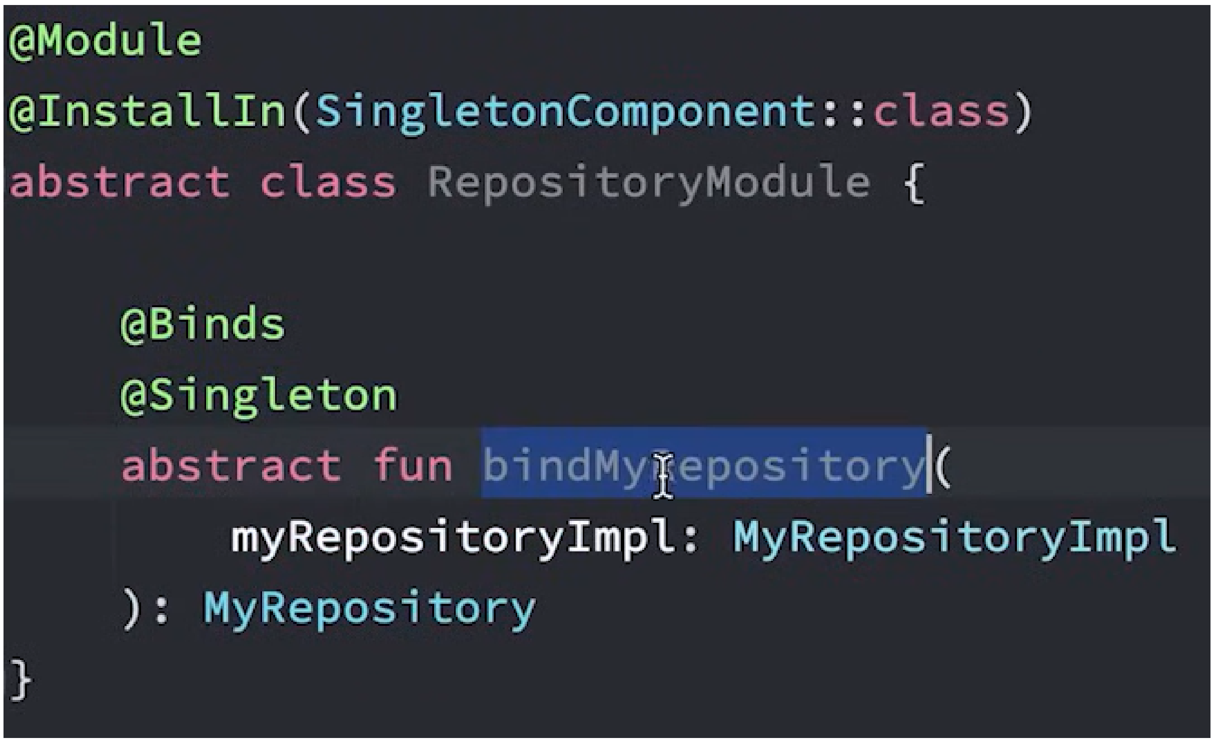
We can use the @Named annotation to distinguish between them:



```kotlin
@Provides
@Singleton
fun provideMyRepository(
    api: MyApi,
    app: Application,
    @Named( value: "hello1") hello1: String
): MyRepository {
    return MyRepositoryImpl(api, app)
}

@Provides
@Singleton
@Named( value: "hello1")
fun provideString1() = "Hello 1"

@Provides
@Singleton
@Named( value: "hello2")
fun provideString2() = "Hello 2"
```

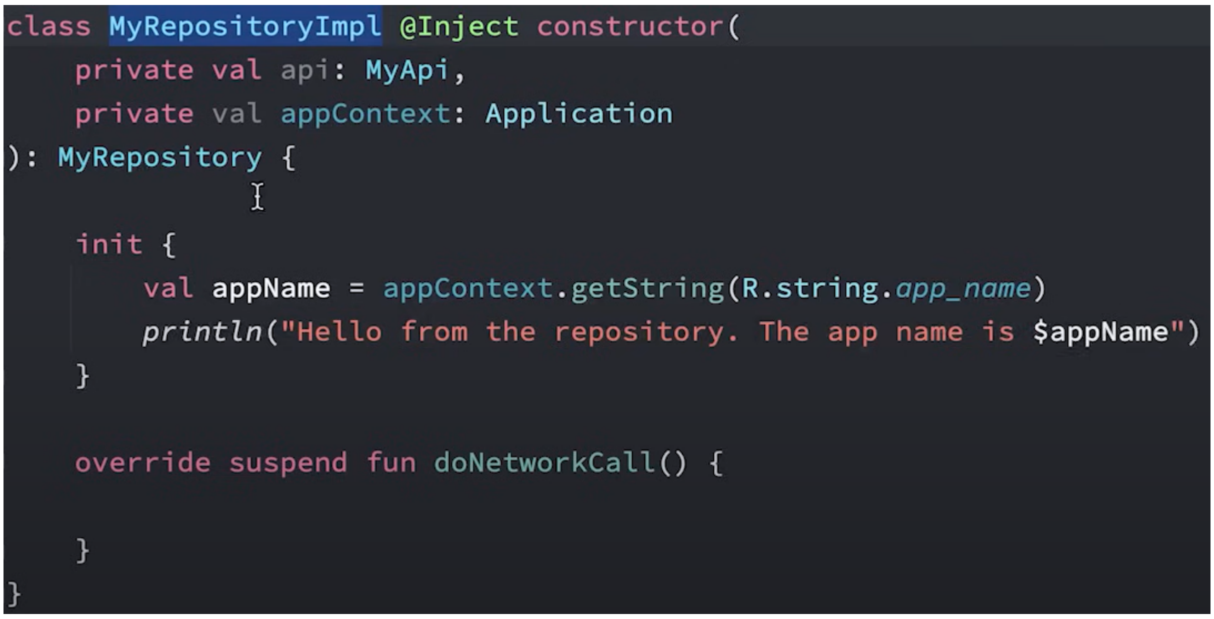
Binding abstactions

Create the abstract module in RepositoryModule in the “di” package:



```kotlin
@Module
@InstallIn(SingletonComponent::class)
abstract class RepositoryModule {

    @Binds
    @Singleton
    abstract fun bindMyRepository(
        myRepositoryImpl: MyRepositoryImpl
    ): MyRepository
}
```

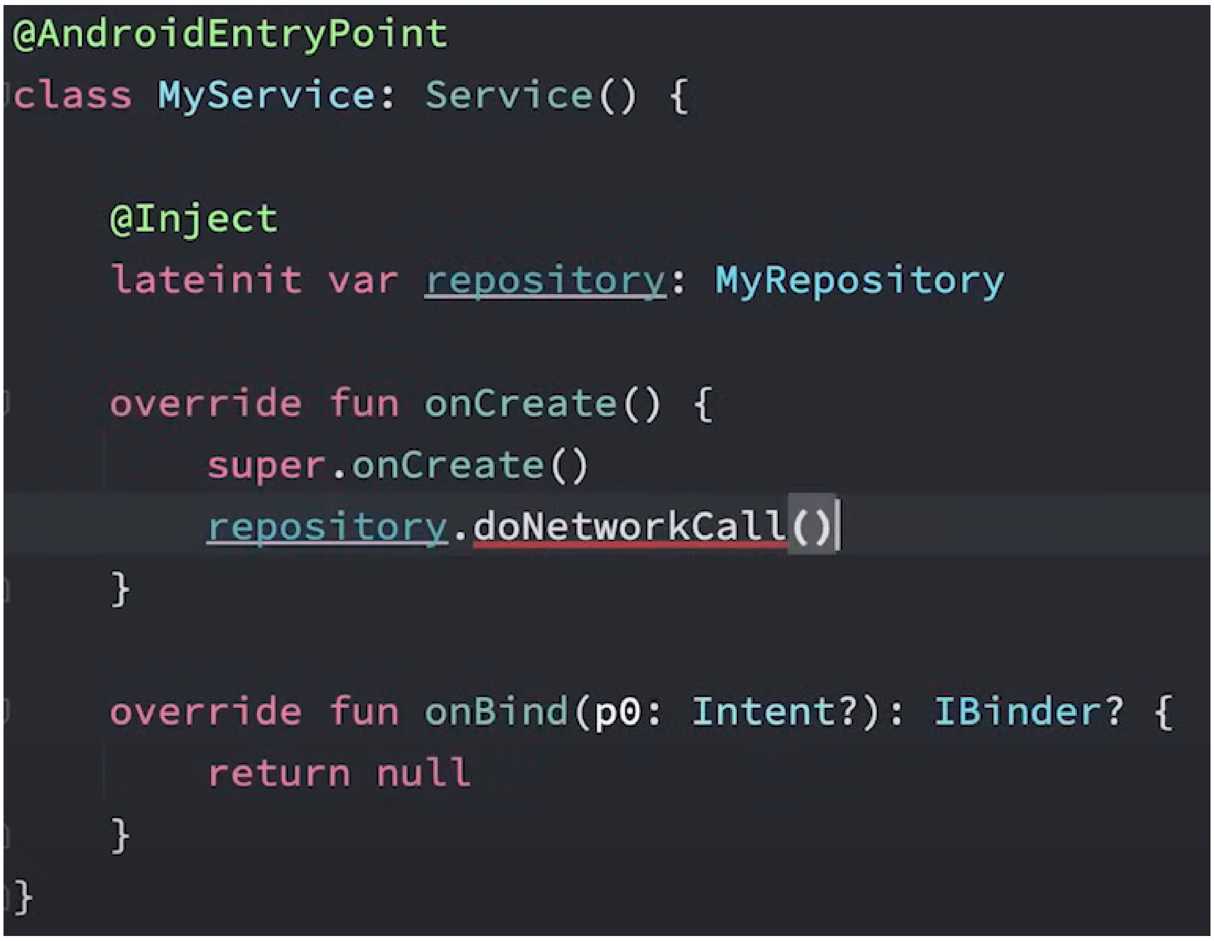
Annotate MyRepositoryImpl with @Inject annotation so that we don't need any provides function:



```kotlin
class MyRepositoryImpl @Inject constructor(
    private val api: MyApi,
    private val appContext: Application
): MyRepository {

    init {
        val appName = appContext.getString(R.string.app_name)
        println("Hello from the repository. The app name is $appName")
    }

    override suspend fun doNetworkCall() {

    }
}
```

Field injection

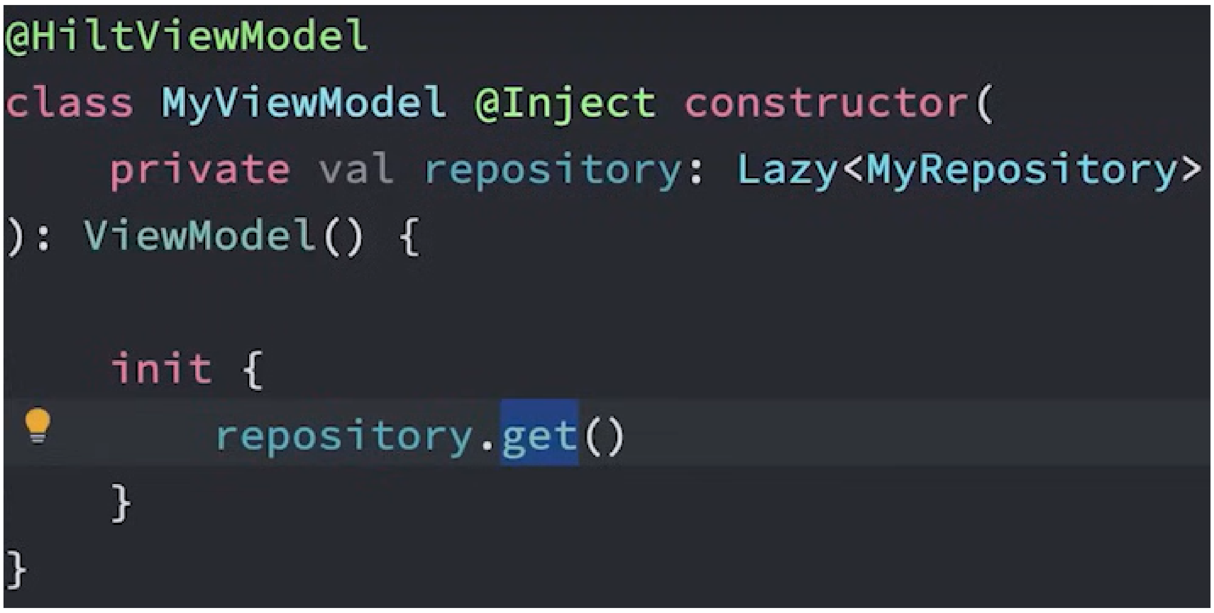
This would come in hand for example in a Service since services doesn't allow constructors so you can't use the inject constructor:



```kotlin
@AndroidEntryPoint
class MyService: Service() {

    @Inject
    lateinit var repository: MyRepository

    override fun onCreate() {
        super.onCreate()
        repository.doNetworkCall()
    }

    override fun onBind(p0: Intent?): IBinder? {
        return null
    }
}
```

Lazy injection

Injecting a dependency and delaying the creation of the object.



```kotlin
@HiltViewModel
class MyViewModel @Inject constructor(
    private val repository: Lazy<MyRepository>
): ViewModel() {

    init {
        repository.get()
    }
}
```

This would be useful for example if you had an auth module and you needed a token that is not known in compilation time. You would delay the creation to the moment where the user is logged in.