13

Dependency Injection with Dagger, Hilt, and Koin

Activity 13.01 – Injected repositories

Solution

Perform the following steps to complete the activity:

1. Create a new Android Studio project with an **Empty** activity.
2. Let's start by adding the versions of the libraries that we will need to `gradle/libs.versions.toml`:

   ```
   [versions]
   …
   ksp = "2.0.21-1.0.25"
   hilt = "2.56.2"
   viewModelCompose = "2.8.7"
   retrofit="2.9.0"
   retrofitGson="2.6.2"
   gson="2.10.1"
   ```

3. Now, let's add the right plugins and libraries in the same file:

   ```
   [libraries]
   …
   androidx-viewmodel-compose = { group = "androidx.lifecycle",
   ```

```
    name = "lifecycle-viewmodel-compose",
        version.ref = "viewModelCompose" }
hilt-android = { group = "com.google.dagger",
    name = "hilt-android", version.ref = "hilt" }
hilt-android-compiler = { group = "com.google.dagger",
    name = "hilt-android-compiler", version.ref = "hilt" }
hilt-android-testing = { group = "com.google.dagger",
    name = "hilt-android-testing", version.ref = "hilt" }
squareup-retrofit = { group = "com.squareup.retrofit2",
    name = "retrofit", version.ref = "retrofit" }
squareup-retrofit-gson = { group = "com.squareup.retrofit2",
    name = "converter-gson", version.ref = "retrofitGson" }
gson = { group = "com.google.code.gson", name = "gson",
    version.ref = "gson" }

[plugins]
…
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
hilt = { id = "com.google.dagger.hilt.android",
    version.ref = "hilt" }
```

4. Next, add the `ksp` and `hilt` plugins to the root `build.gradle.kts` file, but do not apply them:

```
plugins {
    …
    alias(libs.plugins.ksp) apply false
    alias(libs.plugins.hilt) apply false
}
```

5. Next, add the `ksp` and `hilt` plugins to `app/build.gradle.kts`:

```
plugins {
    …
    alias(libs.plugins.ksp)
    alias(libs.plugins.hilt)
}
```

6. In app/build.gradle.kts, configure the Compose version with the following code snippet:

```
android {
    composeOptions {
        kotlinCompilerExtensionVersion = "1.5.4"
    }
}
```

7. Now, let's add the dependencies to app/build.gradle.kts:

```
dependencies {
    …
    implementation(libs.androidx.viewmodel.compose)
    implementation(libs.hilt.android)
    ksp(libs.hilt.android.compiler)
    implementation(libs.squareup.retrofit)
    implementation(libs.squareup.retrofit.gson)
    implementation(libs.gson)
    …
    androidTestImplementation(
        libs.hilt.android.testing
    )
}
```

8. In the app module, create a package called api.
9. In the api package, create a class called Post that represents the Post JSON object received from the internet:

```
data class Post(
    @SerializedName("id") val id: Long,
    @SerializedName("userId") val userId: Long,
    @SerializedName("title") val title: String,
    @SerializedName("body") val body: String
)
```

10. In the api package, create an interface called PostService that will be used to fetch data from the internet:

```
interface PostService {

    @GET("posts")
    suspend fun getPosts(): List<Post>
}
```

11. Create a `repository` package:
12. In the `repository` package, create a `PostRepository` interface that will contain a method that fetches a list of `Post` objects:

```
interface PostRepository {

    suspend fun getPosts(): List<Post>
}
```

13. Create an implementation of `PostRepository` that will use `PostService` to fetch the list of `Post` objects:

```
class PostRepositoryImpl(
    private val postService: PostService
) : PostRepository {
    override suspend fun getPosts(): List<Post> {
        return postService.getPosts()
    }
}
```

14. Create a `PostViewModel` class and, inside this class, define a `PostUi` data class that will have a title and a body:

```
class PostViewModel : ViewModel() {
    …
    data class PostUi(
        val title: String = "",
        val body: String = ""
    )
}
```

15. Inside the `PostViewModel` class, create a `State` class that will hold the list of `PostUi` objects:

```
class PostViewModel : ViewModel() {
    …
    data class State(
        val posts: List<PostUi> = emptyList()
    )
    …
}
```

16. Modify `PostViewModel` by adding the `HiltViewModel` annotation and then define the `StateFlow` properties, which are meant to hold the list of posts. Then, add the `PostRepository` dependency and load the list of posts:

```
@HiltViewModel
class PostViewModel @Inject constructor(
    private val postRepository: PostRepository
) : ViewModel() {

    private val _state =
        MutableStateFlow<State>(State())
    val state: StateFlow<State> = _state

    init {
        viewModelScope.launch {
            _state.emit(
                State(
                    postRepository.getPosts().map {
                        PostUi(
                            title = it.title,
                            body = it.body
                        )
            }))
        }
    }
    …
}
```

17. In `MainActivity`, create the `PostScreen` function, which is meant to render the list of posts from the `State` object:

```
@Composable
fun PostScreen(
    state: PostViewModel.State,
    modifier: Modifier
) {
    LazyColumn(modifier = modifier.fillMaxSize()) {
        items(state.posts.size) {
            val index = it
            Column(modifier = Modifier.padding(4.dp))
            {
                Text(text = state.posts[index].title)
                Text(text = state.posts[index].body)
            }
        }

    }
}
```

18. In `MainActivity`, create the `Post` function, which will take the state from the `PostViewModel` object and use it for `PostScreen`:

```
@Composable
fun Post(
    postViewModel: PostViewModel,
    modifier: Modifier
) {
    PostScreen(
        state = postViewModel
            .state.collectAsState().value,
        modifier = modifier
    )
}
```

19. Modify `MainActivity` to add the `AndroidEntryPoint` annotation and the `setContent` function to call the `Post` function:

```
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?)
    {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            Activity1301Theme {
                Scaffold(
                    modifier = Modifier.fillMaxSize()
                ) { innerPadding ->
                    Post(
                        postViewModel = viewModel(),
                        modifier = Modifier
                            .padding(innerPadding)
                   )
                }
            }
        }
    }
}
```

20. Create a class called `MainApplication`, which will extend `Application`, and annotate it with `HiltAndroidApp`:

```
@HiltAndroidApp
class MainApplication : Application()
```

21. Modify `AndroidManifest.xml` to add the `INTERNET` permission:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest … >
    …
    <uses-permission
        android:name="android.permission.INTERNET"/>
    …
</manifest>
```

22. In `AndroidManifest.xml`, modify the `application` tag to add the `MainApplication` class defined earlier:

```
    <application
        android:name=".MainApplication"
        …
        >
```

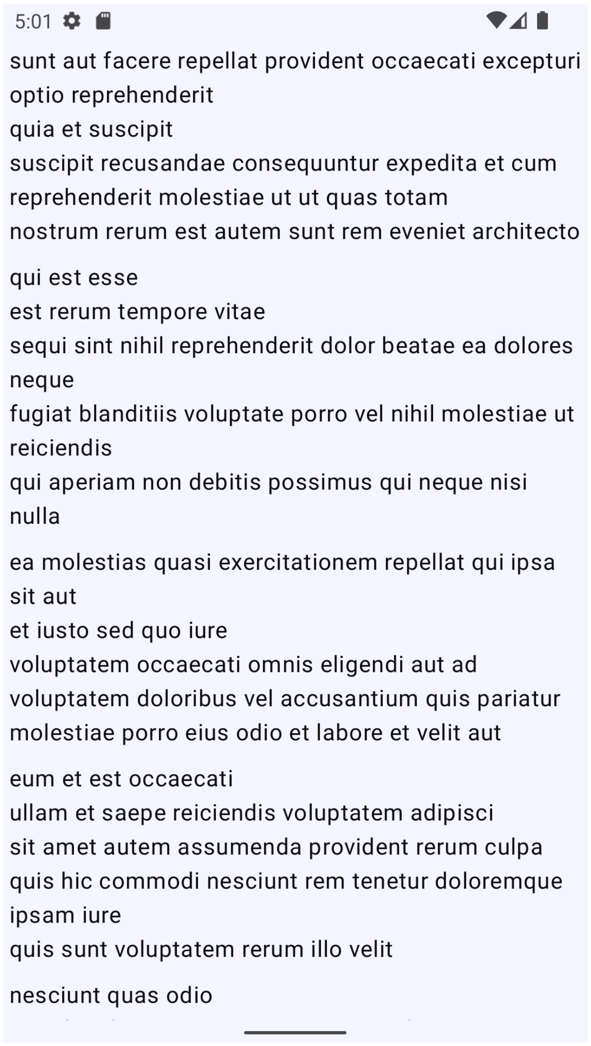
23. In the `api` package, create a class called `NetworkModule` that will hold the network-related dependencies:

```
@Module
@InstallIn(SingletonComponent::class)
class NetworkModule {

    @Singleton
    @Provides
    fun provideRetrofit(): Retrofit {
        return Retrofit.Builder()
            .baseUrl(
               "https://jsonplaceholder.typicode.com/"
            )
            .addConverterFactory(
                GsonConverterFactory.create()
            )
            .build()
    }

    @Singleton
    @Provides
    fun providePostService(retrofit: Retrofit):
        PostService
    {
        return retrofit.create<PostService>(
            PostService::class.java
        )
    }
}
```

24. In the `repository` package, create a class called `RepositoryModule` that will hold the `PostRepository` dependency:

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

    @Singleton
    @Provides
    fun providePostRepository(
        postService: PostService
    ): PostRepository {
        return PostRepositoryImpl(postService)
    }
}
```

If we run the app now, we should see the following screen:



5:01

sunt aut facere repellat provident occaecati excepturi optio reprehenderit
quia et suscipit
suscipit recusandae consequuntur expedita et cum
reprehenderit molestiae ut ut quas totam
nostrum rerum est autem sunt rem eveniet architecto

qui est esse
est rerum tempore vitae
sequi sint nihil reprehenderit dolor beatae ea dolores neque
fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis
qui aperiam non debitis possimus qui neque nisi nulla

ea molestias quasi exercitationem repellat qui ipsa sit aut
et iusto sed quo iure
voluptatem occaecati omnis eligendi aut ad
voluptatem doloribus vel accusantium quis pariatur
molestiae porro eius odio et labore et velit aut

eum et est occaecati
ullam et saepe reiciendis voluptatem adipisci
sit amet autem assumenda provident rerum culpa
quis hic commodi nesciunt rem tenetur doloremque ipsam iure
quis sunt voluptatem rerum illo velit

nesciunt quas odio

Figure 13.6 – Output of Activity 13.01

25. In the androidTest folder, create a DummyRepository class, which will hold fake data for each Post object:

```
class DummyRepository : PostRepository {

    override suspend fun getPosts(): List<Post> {
        return listOf(
            Post(1L, 1L, "Title 1", "Body 1"),
            Post(2L, 1L, "Title 2", "Body 2"),
            Post(3L, 1L, "Title 3", "Body 3")
        )
    }
}
```

26. In the androidTest folder, create a TestRepositoryModule class, which will replace the PostRepository instance from RepositoryModule with DummyRepository:

```
@Module
@TestInstallIn(
    components = [SingletonComponent::class],
    replaces = [RepositoryModule::class]
)
class TestRepositoryModule {

    @Singleton
    @Provides
    fun providePostRepository(): PostRepository {
        return DummyRepository()
    }
}
```

27. Create a class called MainActivityUiTest that will be annotated with HiltAndroidTest and use ComposeContentTestRule from createComposeRule and HiltAndroidRule:

```
@HiltAndroidTest
@RunWith(AndroidJUnit4::class)
class MainActivityUiTest {
    @get:Rule
    var hiltRule = HiltAndroidRule(this)

    @get:Rule
    val composeRule = createComposeRule()
}
```

28. In `MainActivityUiTest`, add a test method that will check that the content displayed on the screen is coming from `DummyRepository`:

```
@HiltAndroidTest
@RunWith(AndroidJUnit4::class)
class MainActivityUiTest {
    …
    @Test
    fun testDisplaysPosts() {
        val scenario =
            launch(MainActivity::class.java)
        scenario.moveToState(Lifecycle.State.RESUMED)
        composeRule
            .onNodeWithText("Title 1")
            .assertIsDisplayed()
        composeRule.onNodeWithText("Body 1")
            .assertIsDisplayed()
        composeRule
            .onNodeWithText("Title 2")
            .assertIsDisplayed()
        composeRule.onNodeWithText("Body 2")
            .assertIsDisplayed()
        composeRule
            .onNodeWithText("Title 3")
            .assertIsDisplayed()
        composeRule.onNodeWithText("Body 3")
            .assertIsDisplayed()
    }
}
```

29. Create a class called `HiltTestRunner` that will use `HiltTestApplication` as the application class to be initialized when the test is run:

```
class HiltTestRunner : AndroidJUnitRunner() {

    @Throws(Exception::class)
    override fun newApplication(
        cl: ClassLoader?,
        className: String?,
        context: Context?
    ): Application? {
        return super.newApplication(
            cl,
            HiltTestApplication::class.java.name,
            Context
        )
    }
}
```

30. Modify `app/build.gradle.kts` to point to the previously defined `HiltTestRunner` class:

```
android {
    …
    defaultConfig {
        …
        testInstrumentationRunner =
            "com.packt.android.HiltTestRunner"
    }
}
```

If we run `testDisplayPosts` from `MainActivityUiTest` at this point, our test should pass because we have successfully swapped `PostRepositoryImpl` with `DummyRepository`.