Dependencies & DI

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

Motivation

Much of the code that you write will depend on other code, and managing dependencies between modules is an important part of your design. As we've discussed, you want to carefully control access to encourage high cohesion and loose loose coupling between classes.

This situation occurs with any module that we might use. e.g.,

- you might split your code into modules, which then have explicit dependencies to manage access between them,
- the Kotlin standard libraries represent a common dependency,
- you will likely import third-party libraries to provide functionality e.g., Compose libraries for user interfaces, or the Ktor libraries for networking.

We want to control access, but also enable legitimate uses where classes need to reference other classes [1].

Motivation

Imagine a `Car` class that needs a reference to an `Engine` class. We could say that the `Car` class is *dependent* on having an instance of the `Engine` class.

There are three ways for a class to get an object it needs:

1. The class constructs the dependency it needs e.g., `Car` creates and initializes its own instance of `Engine`.
2. The class gets the reference from somewhere else. Some Android APIs e.g., Context getters and `getSystemService()`, work this way, by constructing and managing a class reference for you.
3. Have the class reference provided to the class that needs it. This reference can be provided when the class is constructed i.e. constructor injection, or be passed them in to functions that need them i.e. method injection.

This third option, providing the reference, is dependency injection (DI). Let's explore that a little bit.

Dependency Injection (DI)

Introduction

Constructor injection is easy, once you recognize it:

```
class Car(private val engine: Engine) {
    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val engine = Engine()
    val car = Car(engine)
    car.start()
}
```

Compared to the other ways of handling dependencies, DI has advantages:

- Dependencies are explicit and required.
- They are `immutable` (using `val`)
- DI is much easier to test (i.e. you control the dependencies)
- There is a clear dependency graph

Koin (DI) Framework

For simple applications, manual DI is a reasonable approach, but dependencies can get quite complex as your application grows. A dependency injection framework attempts to simplify dependency management for large projects.

Advantages of a DI solution:

- Handles large numbers of dependencies,
- Can address ordering issues e.g., transitive dependendicies.
- Centralized configuration.
- Simplifies your code.

Examples of Kotlin DI frameworks include:

- hilt is Google's recommended DI framework for Android-only solutions.
- koin is a KMP library that works cross-platform.
- metro is a KMP library that is gaining traction.

We'll use Koin so that we can share our DI code across Android, Desktop and iOS targets.

Example

Manual DI

Manual DI looks like this. We need to carefully instantiate components in our top-level method, and then pass them into variously constructors.

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

        // Manually creating the entire dependency graph
        val database = Database()
        val apiClient = ApiClient()
        val userRepo = UserRepo(database, apiClient)
        val authRepo = AuthRepo(database, apiClient)
        val userService = UserService(userRepo, authRepo)
        val viewModel = UserViewModel(userService)

        // Finally can use viewModel ...
    }
}
```

Example

Container

One problem that manual DI doesn't address is service location. We need to have references available everywhere.

```
object AppContainer {
    private val database by lazy { Database() }
    private val apiClient by lazy { ApiClient() }

    val userRepo by lazy { UserRepo(database, apiClient) }
    val authRepo by lazy { AuthRepo(database, apiClient) }

    fun createUserViewModel() = UserViewModel(
        UserService(userRepo, authRepo)
    )
}

class MainActivity : AppCompatActivity() {
    // AppContainer singleton provides global access to components
    private val viewModel = AppContainer.createUserViewModel()
}
```

Example

Koin

Koin addresses these issues by instantiating components for you as needed, and managing access to them.

```
// Define dependencies once
val appModule = module {
    single<Database>()
    single<ApiClient>()
    single<UserRepo>()
    single<AuthRepo>()
    single<UserService>()
    viewModel<UserViewModel>()
}
```

Example

```
// Start Koin once
class MyApplication : Application() {
    override fun onCreate() {
        startKoin {
            modules(appModule)
        }
    }
}

// Use references anywhere
class MainActivity : AppCompatActivity() {
    private val viewModel: UserViewModel by viewModel()
}

// You can also just inject dependencies as needed
class UserService : KoinComponent {
    val repo: UserRepo by inject()  // "Pulling" dependency
    //  ...
}
```

Bibliography

[1] M. Fowler, “Inversion of Control Containers and the Dependency Injection pattern.” [Online]. Available: https://martinfowler.com/articles/injection.html