Clean Architecture & Dependency Injection in Android

Software Studio 2022

Q1. Who is the target audience for this lab?

- You want to learn an architecture guideline to **better collaborate with your team**. (share work, debug)
- You have a basic understanding of *Architecture (data/domain/presentation)* and *Dependency Injection* from given videos or other resources.
- We won't cover much about the interaction between UI and ViewModels (we will cover it next week)

Q2. Why should we learn clean architecture? Isn't MVVM already the most famous architecture?

A. The purpose of the clean architecture is making the app **scalable** . A good architecture comes with several different aspects:

- How easy it is to extend the architecture with new features
- How easy it is to test the app
- How long does a new team member need to actually understand what the project is about

Q3. I have seen the official tutorial videos. What else can I expect to learn from this lab?

A. **This lab will give you a deeper understanding of the concepts you learnt from the videos**. We will guide you through tracing codes of a small project that implemented with clean architecture.

Last thing before we dive into today's topic

Most of the content in this lab is from this YouTube video (LINK). If you prefer to learn in English, you could go watch the video instead. It may be longer but is well explained.

1. Clean Architecture

Quick review of the concept of clean architecture

Clean Architecture

- Data Layer (Data Source / Repository)
- Domain Layer (Usecase / Entity)
- Presentation Layer (ViewModel/UI)

Clean Architecture - Data Layer

1. Exposing data and receiving events
2. Source of truth
3. Immutability
4. Threading and error handling

```
class NewsRepository (
  val localNewsDataSource: LocalNewsDataSource,
  val remoteNewsDataSource: RemoteNewsDataSource
) {
  suspend fun fetchNews() : List<Article> {
    try {
      val news = remoteNewsDataSource.fetchNews()
      localNewsDataSource.updateNews(news)
    }.catch (e: RemoteDataSourceNotAvailableException) {
      Log.d("NewsRepository", "Connection failed, using local data source")
    }
    return localNewsDataSource.fetchNews()
  }
}
```

Clean Architecture - Domain Layer

1. Simple
2. Lightweight
3. Immutable

```
class GetLatestNewsWithAuthorsUseCase(
  private val newsRepository: NewsRepository,
  private val authorsRepository: AuthorsRepository,
  private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default
) {
  suspend operator fun invoke(): List<ArticleWithAuthor> =
    withContext(defaultDispatcher) {
      val news = newsRepository.fetchLatestNews()
      return buildList {
        news.forEach { article ->
          val author = authorsRepository.getAuthor(article.authorId)
          add(ArticleWithAuthor(article, author))
        }
      }
    }
}
```

Clean Architecture - Presentation Layer

1. Define UI state
2. Production of UI state
3. Expose UI state
4. Consume UI state

```
sealed interface HomeUiState {
  val isLoading : Boolean
  val errorMessages: List<Int>
  val searchInput: String
  ...
}

class HomeViewModel(...) : ViewModel() {
  ...
  val uiState: StateFlow<HomeUiState> =
  fun refreshPosts() {
    _uiState.update { it.copy(isLoading = true)}

    viewModelScope.launch {
      val result = postsRepository.getPostsFeed()
      _uiState.update {
        when (result) {
          is Result.Success -> {
            it.copy(postsFeed = result.data, isLoading = false)
          }
          is Result.Error -> {
            val errorMessages = it.errorMessages + R.string.load_error
            it.copy(errorMessages = errorMessages, isLoading = false)
          }
        }
      }
    }
  }
}
```

Steps to trace the code

1. Clone the project from this GitHub Link.
2. Checkout to branch *app*.
3. Try to trace the code (without actually run the app) and answer the following questions:

Basic

1. What data are included in a Note?
2. What services does this app provide?
3. Where is the code for "sorting notes"?
4. What user can do to interact with the app on the Main Page?

Advanced

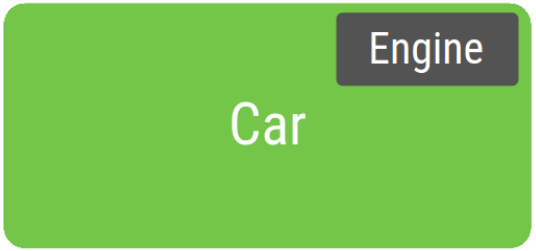
1. Currently the app is using local database for data storage. If we decide to switch to remote server instead, which parts should be adjusted and how? (list all)
2. If we want to use fake data instead of using real data source for testing, which parts should be adjusted and how?
3. If we click the delete button to delete a note but the process failed (the note is still in database), will the UI show the note?

2. Dependency Injection

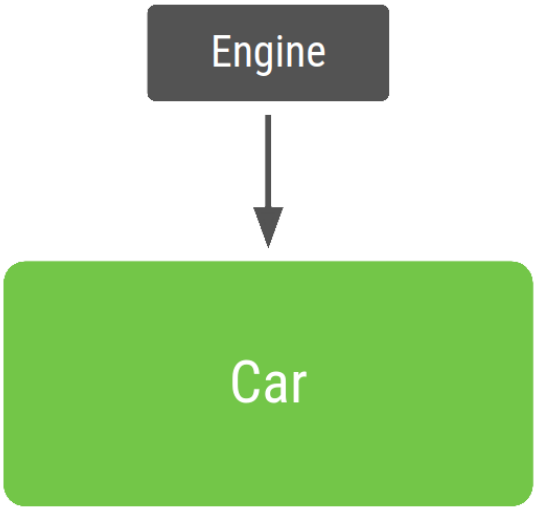
- 2.1. What is dependency injection
- 2.2. Manual dependency injection
- 2.3. Dependency injection with Hilt

2.1 What is dependency injection

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

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



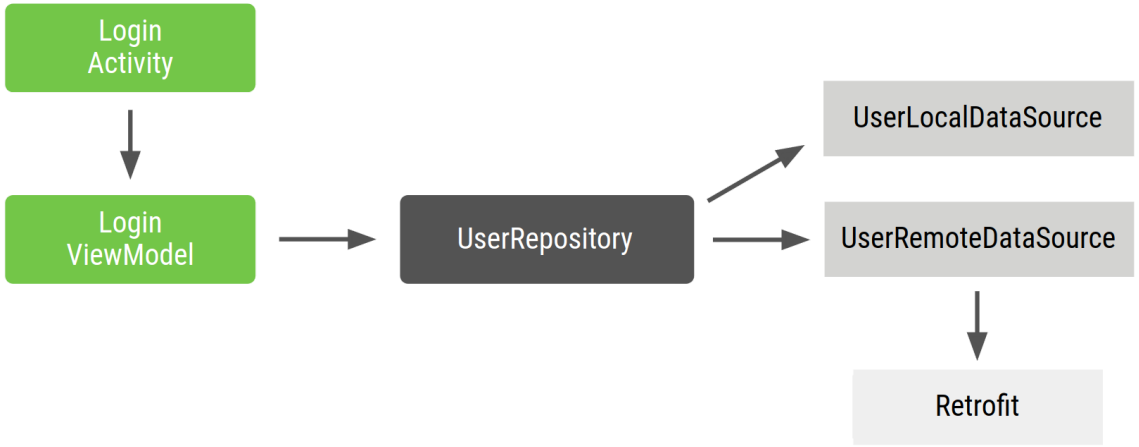
```
class Car(private val engine: Engine) {
    fun start() {
        engine.start()
    }
}

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



2.2 Manual Dependency Injection

- Basic
- Manage in containers
- Manage in application flows

Basics of manual DI




```
class UserRepository(
    private val localDataSource: UserLocalDataSource,
    private val remoteDataSource: UserRemoteDataSource
) { ... }

class UserLocalDataSource { ... }
class UserRemoteDataSource(
    private val loginService: LoginRetrofitService
) { ... }
```

```
class LoginActivity: Activity() {

    private lateinit var loginViewModel: LoginViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // In order to satisfy the dependencies of LoginViewModel, you have to also
        // satisfy the dependencies of all of its dependencies recursively.
        // First, create retrofit which is the dependency of UserRemoteDataSource
        val retrofit = Retrofit.Builder()
            .baseUrl("https://example.com")
            .build()
            .create(LoginService::class.java)

        // Then, satisfy the dependencies of UserRepository
        val remoteDataSource = UserRemoteDataSource(retrofit)
        val localDataSource = UserLocalDataSource()

        // Now you can create an instance of UserRepository that LoginViewModel needs
        val userRepository = UserRepository(localDataSource, remoteDataSource)

        // Lastly, create an instance of LoginViewModel with userRepository
        loginViewModel = LoginViewModel(userRepository)
    }
}
```

Issues with current approach:

- A lot of boilerplate code. Code duplication if we want to create another instance of `LoginViewModel`
- Dependenies have to be declared in order

Managing dependencies with a container

```
// Container of objects shared across the whole app
class AppContainer {

    // Since you want to expose userRepository out of the container, you need to satisfy
    // its dependencies as you did before
    private val retrofit = Retrofit.Builder()
                            .baseUrl("https://example.com")
                            .build()
                            .create(LoginService::class.java)

    private val remoteDataSource = UserRemoteDataSource(retrofit)
    private val localDataSource = UserLocalDataSource()

    // userRepository is not private; it'll be exposed
    val userRepository = UserRepository(localDataSource, remoteDataSource)
}
```

```
// Custom Application class that needs to be specified
// in the AndroidManifest.xml file
class MyApplication : Application() {

    // Instance of AppContainer that will be used by all the Activities of the app
    val appContainer = AppContainer()
}
```

```
class LoginActivity: Activity() {

    private lateinit var loginViewModel: LoginViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Gets userRepository from the instance of AppContainer in Application
        val appContainer = (application as MyApplication).appContainer
        loginViewModel = LoginViewModel(appContainer.userRepository)
    }
}
```

Issues with current approach:

- Cannot have objects to just live in the scope of different flow
- Dependenies have to be declared in order

Managing dependencies in application flows

```
class LoginContainer(val userRepository: UserRepository) {

    val loginData = LoginUserData()

    val loginViewModelFactory = LoginViewModelFactory(userRepository)
}

// AppContainer contains LoginContainer now
class AppContainer {
    ...
    val userRepository = UserRepository(localDataSource, remoteDataSource)

    // LoginContainer will be null when the user is NOT in the login flow
    var loginContainer: LoginContainer? = null
}
```

```
class LoginActivity: Activity() {

    private lateinit var loginViewModel: LoginViewModel
    private lateinit var loginData: LoginUserData
    private lateinit var appContainer: AppContainer


    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        appContainer = (application as MyApplication).appContainer

        // Login flow has started. Populate loginContainer in AppContainer
        appContainer.loginContainer = LoginContainer(appContainer.userRepository)

        loginViewModel = appContainer.loginContainer.loginViewModelFactory.create()
        loginData = appContainer.loginContainer.loginData
    }

    override fun onDestroy() {
        // Login flow is finishing
        // Removing the instance of loginContainer in the AppContainer
        appContainer.loginContainer = null
        super.onDestroy()
    }
}
```

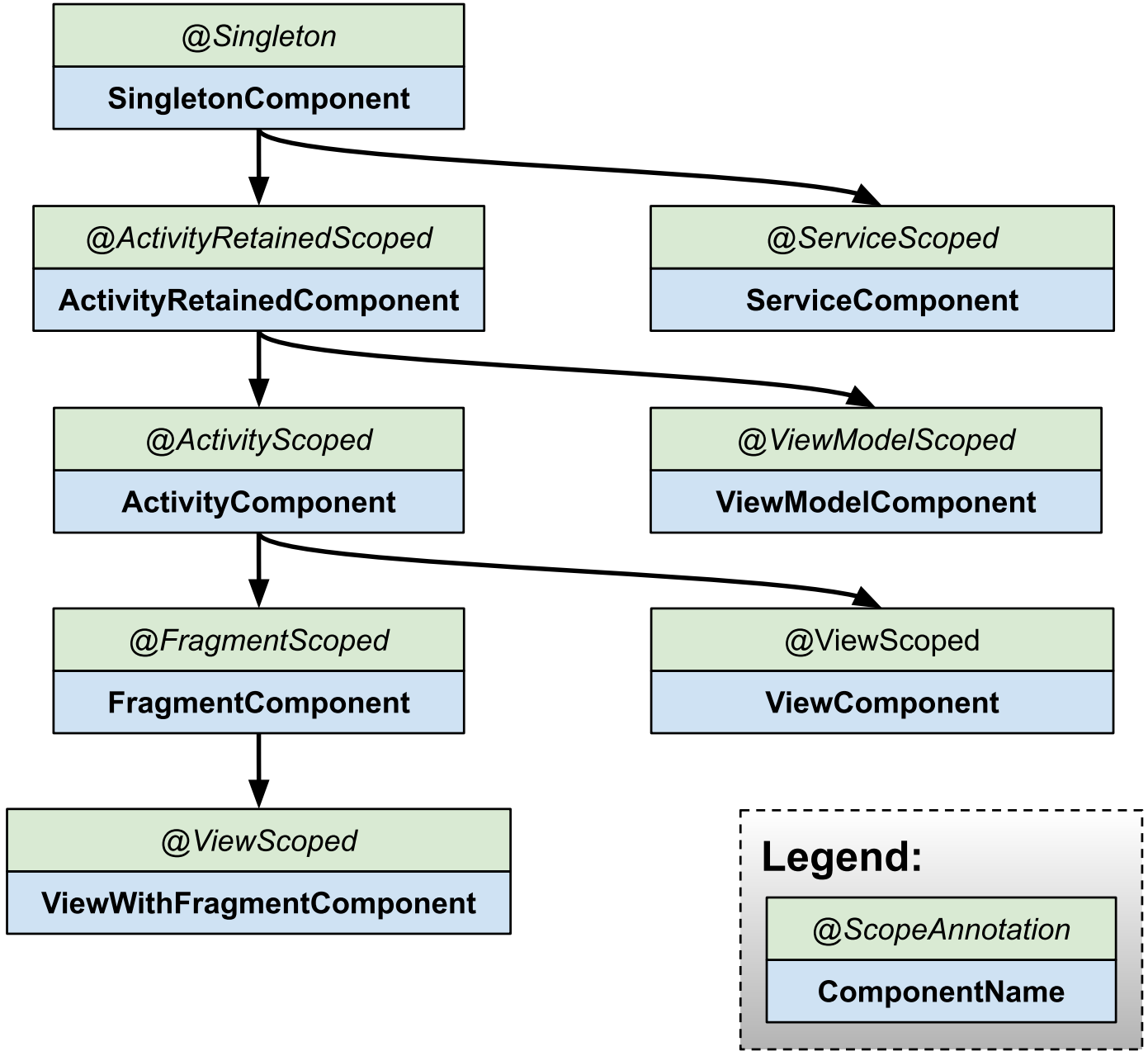
Issues with current approach:

- Have to handle the scope of container ourselves
- Dependenies have to be declared in order

2.3 Dependency injection with Hilt

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

    @Provides
    @Singleton
    fun provideNoteDatabase(app: Application): NoteDatabase {
        return Room.databaseBuilder(
            app,
            NoteDatabase::class.java,
            NoteDatabase.DATABASE_NAME
        ).build()
    }

    @Provides
    @Singleton
    fun provideNoteRepository(db: NoteDatabase): NoteRepository {
        return NoteRepositoryImpl(db.noteDao)
    }

    @Provides
    @Singleton
    fun provideNoteUseCases(repository: NoteRepository): NoteUseCases {
        return NoteUseCases(
            getNotes = GetNotes(repository),
            deleteNote = DeleteNote(repository),
            addNote = AddNote(repository),
            getNote = GetNote(repository)
        )
    }
}
```

```
@HiltViewModel
class NotesViewModel @Inject constructor(
    private val noteUseCases: NoteUseCases
) : ViewModel() {
```


@Singleton
SingletonComponent
@ActivityRetainedScoped
ActivityRetainedComponent
@ServiceScoped
ServiceComponent
@ActivityScoped
ActivityComponent
@ViewModelScoped
ViewModelComponent
@FragmentScoped
FragmentComponent
@ViewScoped
ViewComponent
@ViewScoped
ViewWithFragmentComponent
Legend:
@ScopeAnnotation
ComponentName

Answers to today's questions

Basic

1. What data are included in a Note?
   -> Check ***Model*** in Domain Layer
2. What services does this app provide?
   -> Check ***UseCase*** in Domain Layer
3. Where is the code for "sorting notes"?
   -> Check ***UseCase*** or ***ViewModel***
4. What user can do to interact with the app on the Main Page?
   -> Check ***UiState*** in Presentation Layer

Advanced

1. Currently the app is using local database for data storage. If we decide to switch to remote server instead, which parts should be adjusted and how? (list all)
   -> Add ***DataSource*** and update ***Repository***
2. If we want to use fake data instead of using real data source for testing, which parts should be adjusted and how?
   -> Add ***DataSource*** and add ***fake Repository***
3. If we click the delete button to delete a note but the process failed (the note is still in database), will the UI show the note?
   -> Yes, as the UI only shows the state that comes from a **single source or truth**

Discussion

- When should we use usecase?
  - Reduce complexity of UI layer
  - Avoid duplicatoin
  - Improve testability
- When should we use dependency injection?
  - Volatile dependencies (不穩定依賴性)
  - If you want to use Jetpack Compose

Conclusion

The goals of following the principle of clean architecture are:

- Make it easier to extend the architecture with new features
- Make it easier to maintain and test the app
- A new team member can easily understand what the project is doing and can start contribute soon