

Dependency Injection in Android



Java™ Education
and Technology Services

Invest In Yourself,
Develop Your Career

Course Duration and Evaluation

- **Duration (12 hours)**
 - 2 Lectures (6 hours)
 - 2 Labs (6 hours)
- **Evaluation Criteria**
 - 100% on Lab activities.

What is Dependency?

Introduction

Dependencies (Needed objects)

- Car class might need a reference to an Engine class
- Car class is dependent on having an instance of the Engine class to run.



```
data class Engine(val manufacturer: String,  
                 val power: Int)
```

```
data class Car (val engine: Engine,  
               val numberOfWheels: Int)
```

Providing Dependencies

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

1. The class constructs the dependency it needs.

```
class Car { val engine: Engine = Engine( ) }
```

2. Grab it from somewhere else.

Some Android APIs, such as **Context** getters and **getSystemService()** work this way.

3. Have it supplied as a parameter to a method or a constructor.

```
class Car (val engine: Engine)
```

Let's discuss those solutions!



1. The Class Creates Its Dependencies

- This can be problematic because of:
 1. Tight coupling.
 2. Hard dependency makes testing more difficult.
 3. Losing the Single responsibility for the class.

This approach can't be considered as a dependency injection



2. Grab Dependencies from Somewhere

- By making the class depends ask for its dependencies from another class. This approach is called Service Locator
- Service Locator improves decoupling of classes from concrete dependencies.
- You create a class known as the *service locator* that creates and stores dependencies and then provides those dependencies on demand.

Still, Service locator is not a dependency injection too.



3. Dependencies supplied as a parameter.

With this approach you take the dependencies of a class and provide them rather than having the class instance obtain them itself.

Dependency injection!

The benefits of this DI-based approach are:

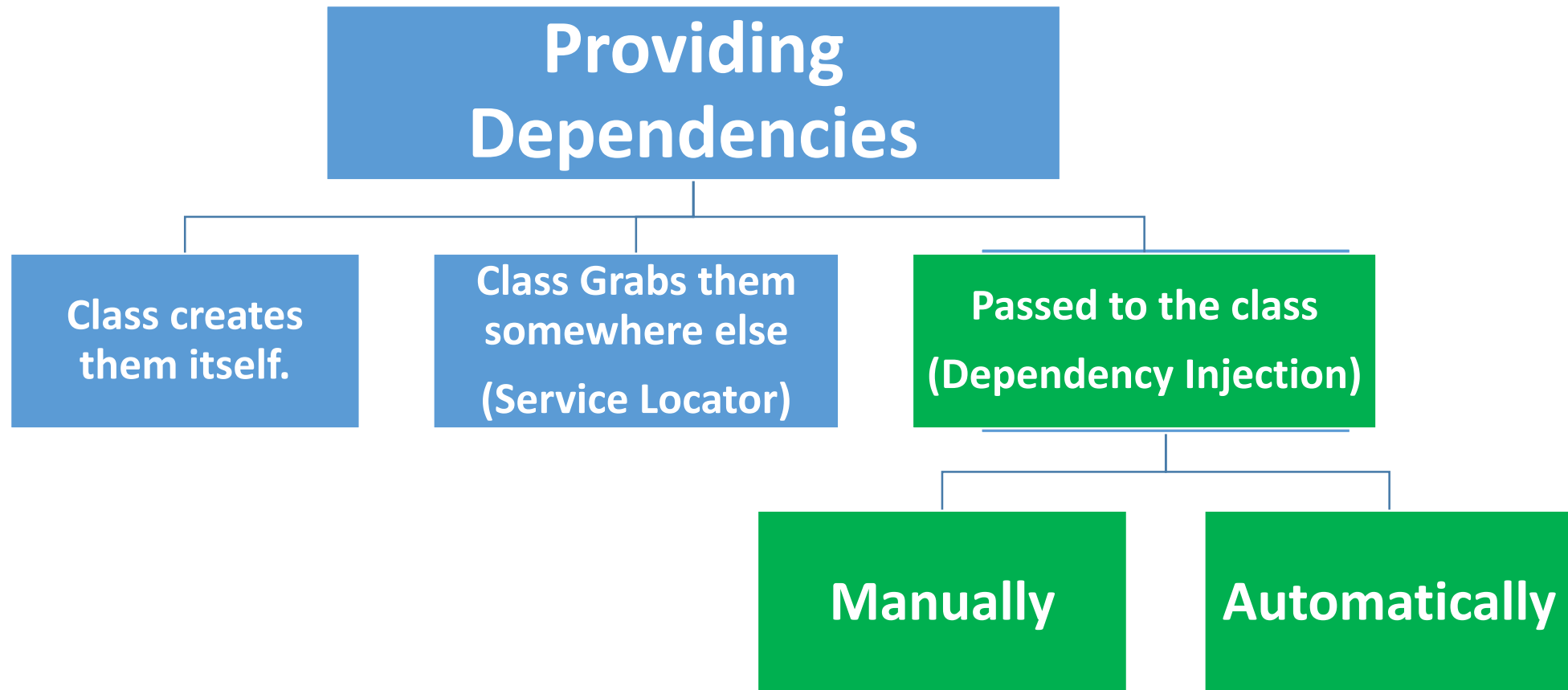
1. Reusability: You can pass in different implementations the required dependencies. And your code still works without any further changes.
2. Easy testing: You can pass in test doubles to test your different scenarios.



Dependency Injection in Android

- There are **two** major ways to do dependency injection in Android:
 1. **Constructor Injection.** This is the way described above. You pass the dependencies of a class to its constructor.
 2. **Field Injection (or Setter Injection).** Certain Android framework classes such as activities and fragments are instantiated by the system, so constructor injection is not possible. With field injection, dependencies are instantiated after the class is created.

To Wrap Up



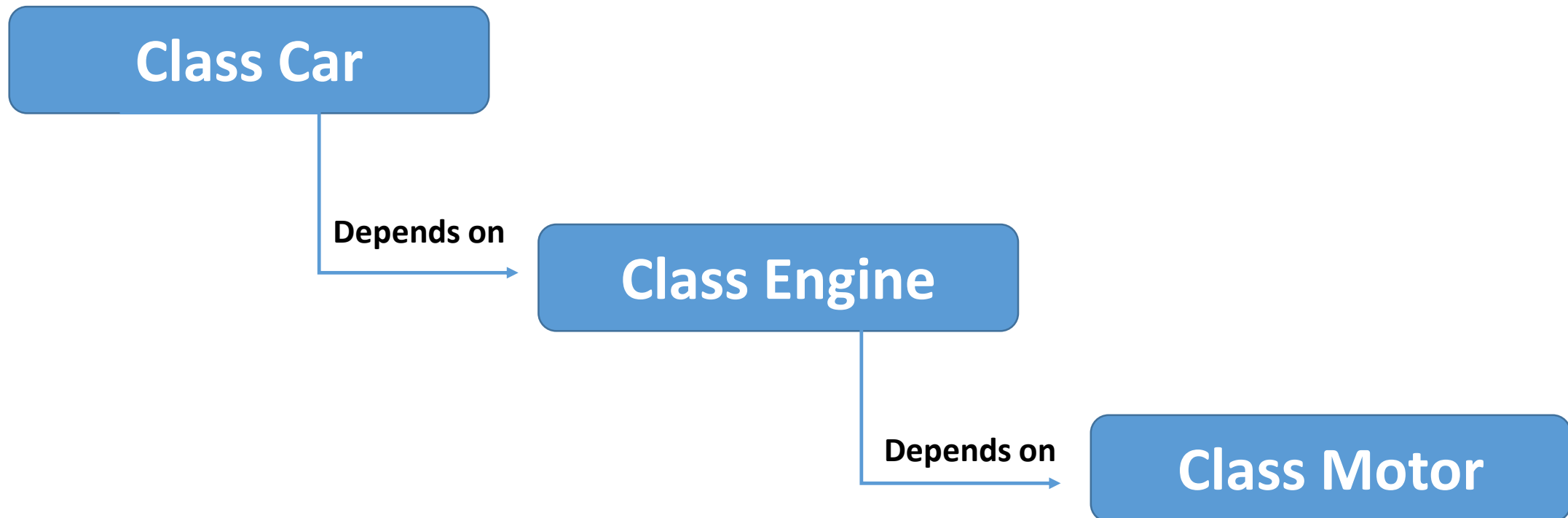
Service Locator

Introduction to Service Locator

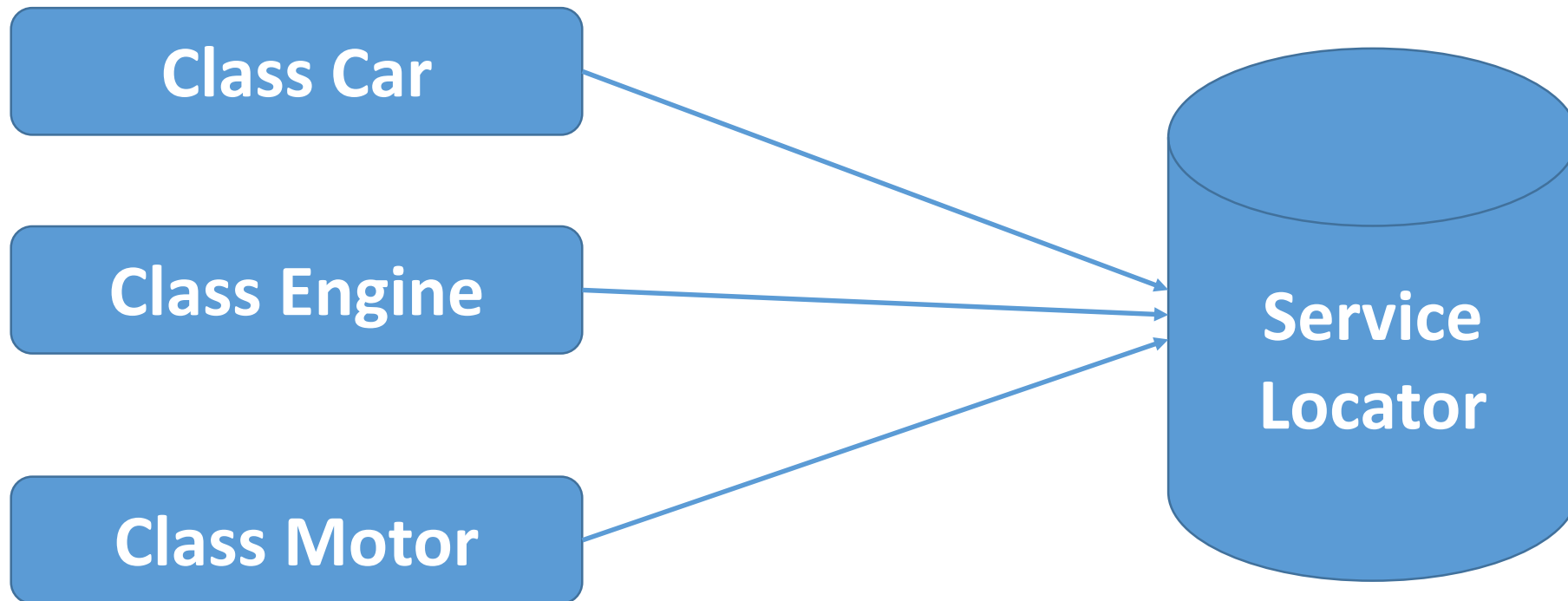
The main idea of the Service Locator is to create a registry containing all the dependencies and get components from it whenever we need them.

Classes have control and ask for objects to be injected; with dependency injection, the app has control and proactively injects the required objects.

Dependency Acyclic Graph (DAG)



Service Locator DAG



Service Locator Example

```
object ServiceLocator {  
    fun getEngine(): Engine = Engine()  
}  
  
class Car {  
    private val engine = ServiceLocator.getEngine()  
  
    fun start() {  
        engine.start()  
    }  
}
```


Movie App using Service Locator

Determine the Needed Dependencies

```
enum class DEPENDENCY{  
    NETWORK,  
    DATABASE,  
    REPOSITORY,  
    ALL_MOVIE_FACTORY,  
    FAV_MOVIE_FACTORY  
}
```

Movies ServiceLocator

```
interface ServiceLocator{
    fun <T> getDependency(required: DEPENDENCY): T
}

class ServiceLocatorImpl (private val ctx: Context) : ServiceLocator{
    override fun <T> getDependency(required: DEPENDENCY): T{
        return when(required){
            DEPENDENCY.NETWORK-> MoviesRemoteDataSourceImpl.getInstance() as T
            DEPENDENCY.DATABASE -> MoviesLocalDataSourceImpl(ctx) as T
            DEPENDENCY.REPOSITORY -> MoviesRepositoryImpl.getInstance(serviceLocator: this) as T
            DEPENDENCY.ALL_MOVIE_FACTORY -> AllMovieViewModelFactory(serviceLocator: this) as T
            DEPENDENCY.FAV_MOVIE_FACTORY -> FavMoviesViewModelFactory(serviceLocator: this) as T
            else -> throw Exception("Can't find this dependency")
        }
    }
}
```

Refactor Classes

- Now, all used classes will depend on this single ServiceLocator.

```
class MoviesRepositoryImpl private constructor(
    serviceLocator: ServiceLocator,
) : MoviesRepository {
    private var moviesRemoteDataSource: MoviesRemoteDataSource = serviceLocator.getDependency(DEPENDENCY.NETWORK)
    private var moviesLocalDataSource: MoviesLocalDataSource = serviceLocator.getDependency(DEPENDENCY.DATABASE)
    companion object {...}

    override suspend fun getAllMovies(): List<Movie> {...}

    override suspend fun getStoredMovies(): List<Movie> {...}

    override suspend fun insertMovie(movie: Movie) {...}

    override suspend fun deleteMovie(movie: Movie) {...}
}
```

ViewModelFactory using ServiceLocator

```
class AllMovieViewModelFactory(private val serviceLocator: ServiceLocator) : ViewModelProvider.Factory {  
    override fun <T : ViewModel> create(modelClass: Class<T>): T {  
        return if (modelClass.isAssignableFrom(AllMoviesViewModel::class.java)) {  
            AllMoviesViewModel(serviceLocator) as T  
        } else {  
            throw IllegalArgumentException("ViewModel Class not found")  
        }  
    }  
}  
  
class FavMoviesViewModelFactory(private val serviceLocator: ServiceLocator) : ViewModelProvider.Factory {  
    override fun <T : ViewModel> create(modelClass: Class<T>): T {  
        return if (modelClass.isAssignableFrom(FavMoviesViewModel::class.java)) {  
            FavMoviesViewModel(serviceLocator) as T  
        } else {  
            throw IllegalArgumentException("ViewModel Class not found")  
        }  
    }  
}
```

The Application class to the rescue



Application Class in Android

Application class is a base class for maintaining global application state.

You can provide your own implementation by creating a subclass and specifying the fully-qualified name of this subclass as the `"android:name"` attribute in your `AndroidManifest.xml`'s `<application>` tag.

The Application class, or your subclass of the Application class, is instantiated before any other class when the process for your application/package is created.

You Might Extend Application class

- Specialized tasks that need to run before the creation of your first activity
- Global initialization that needs to be shared across all components (crash reporting, persistence)
- Static methods for easy access to static immutable data such as a shared network client object

As our service locator will be used by all components we can subclass Application class and load the service locator there

Using the Service Locator

- Subclass the Application class
- Inside your subclass override the onCreate()
- Inside the onCreate(), create the instance of the service locator that you need for your program.

Using the Service Locator

```
class MyApplication : Application() {  
  
    lateinit var serviceLocator: ServiceLocator  
  
    override fun onCreate() {  
        super.onCreate()  
        serviceLocator = ServiceLocatorImpl(ctx: this)  
    }  
}
```

Accessing ServiceLocator from Activity

```
class AllMoviesActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            AllMoviesScreen(  
                ViewModelProvider(owner: this, AllMoviesFactory(  
                    (application as MovieApplication).serviceLocator)  
                )[AllMoviesViewModel::class.java]  
            )  
        }  
    }  
}
```

Manual Dependency Injection

Introduction

- Android's recommended app architecture encourages dividing your code into classes to benefit from separation of concerns, a principle where each class of the hierarchy has a single defined responsibility.
- This leads to more, smaller classes that need to be connected together to fulfill each other's dependencies.
- The dependencies between classes can be represented as a graph, in which each class is connected to the classes it depends on. The representation of all your classes and their dependencies makes up the *application graph*.

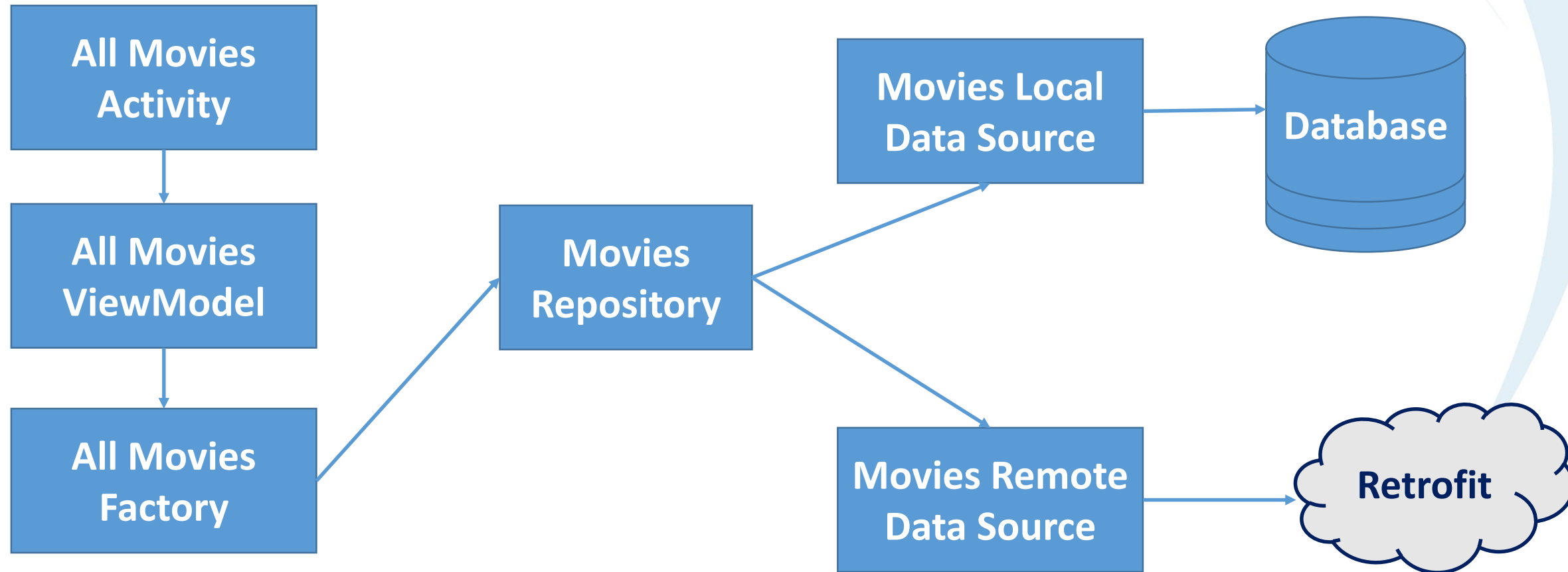
Dependency Graph

- When class A (ViewModel) depends on class B (Repository), there's a line that points from A to B representing that dependency.
- Dependency injection helps make these connections and enables you to swap out implementations for testing. For example, when testing a ViewModel that depends on a repository, you can pass different implementations of Repository with either fakes or mocks to test the different cases.

Use case – Movie App

- Consider a flow to be a group of screens in your app that correspond to a feature. For example Showing all movies from a network.
- When covering a Showing all movies from a network flow for a typical our Android app, the AllMovieActivity depends on AllMoviesViewModel, which in turn depends on MoviesRepository. Then MoviesRepository depends on a MoviesLocalDataSource which depends on RoomDatabase and a MoviesRemoteDataSource, which in turn depends on a Retrofit service.

All Movies' Feature Dependency Graph



AllMoviesActivity

```
class AllMovieActivity: Activity() {  
  
    private lateinit var viewModel: AllMoviesViewModel  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        // In order to satisfy the dependencies of AllMoviesViewModel, you have to also  
        // satisfy the dependencies of all of its dependencies recursively.  
        // First, satisfy the dependencies of UserRepository  
        val remoteDataSource = MoviesRemoteDataSourceImpl()  
        val localDataSource = MoviesLocalDataSourceImpl(this)  
  
        // Now you can create an instance of UserRepository that LoginViewModel needs  
        val moviesRepository = MoviesRepository(localDataSource, remoteDataSource)  
        val factory = AllMovieViewModelFactory(moviesRepository)  
  
        // Lastly, create an instance of LoginViewModel with userRepository  
        viewModel = ViewModelProvider(this, factory).get(AllMoviesViewModel::class.java)  
    }  
}
```

Troubles in this approach

1. There's a lot of boilerplate code. If you wanted to create another instance of `ViewModel` in another part of the code, you'd have code duplication.
2. Dependencies have to be declared in order. You have to instantiate `MoviesRepository` before `AllMoviesViewModel` in order to create it.
3. It's difficult to reuse objects. If you wanted to reuse `UserRepository` across multiple features, you'd have to make it follow the singleton pattern. The singleton pattern makes testing more difficult because all tests share the same singleton instance.

Managing dependencies with a container

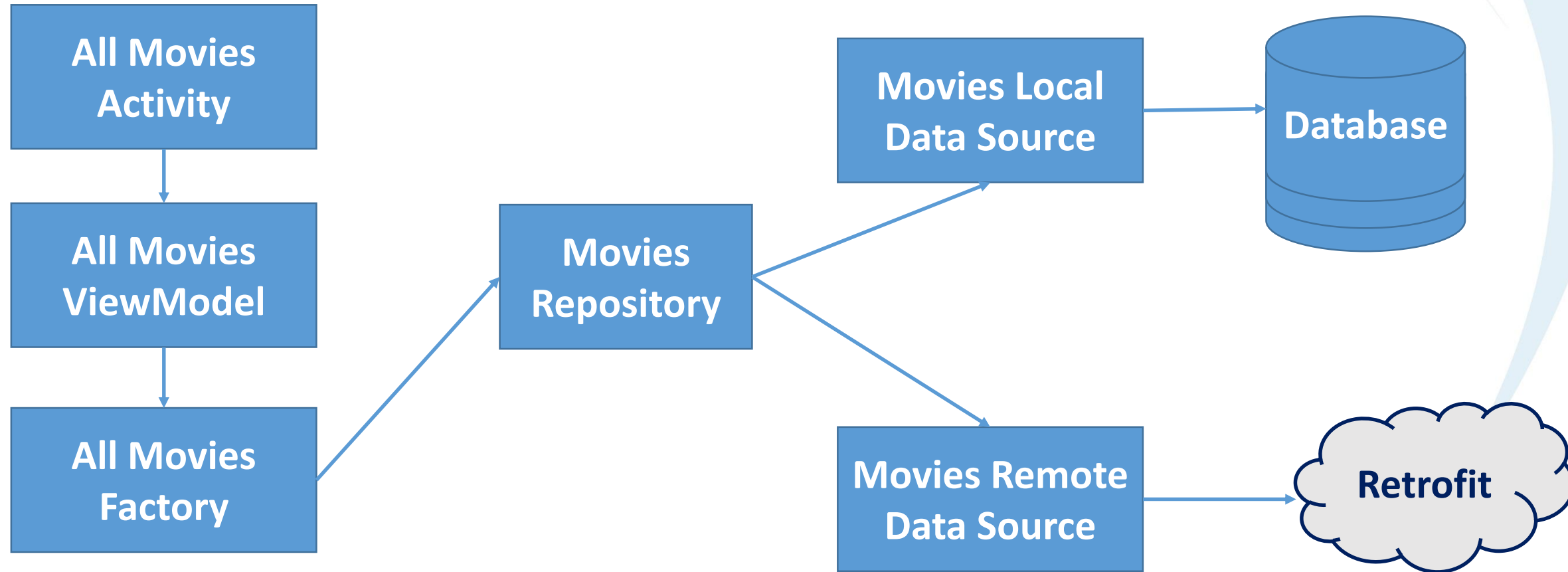
To solve the issue of reusing objects, you can create your own dependencies container class that you use to get dependencies.

All instances provided by this container can be public.

```
class AppContainer {  
    // Since you want to expose moviesRepository out of the container, you need to satisfy  
    // its dependencies as you did before  
    val remoteDataSource = MoviesRemoteDataSource()  
    val localDataSource = MoviesLocalDataSource(dao)  
  
    // Then, create the moviesRepository  
    val moviesRepository = MoviesRepository(localDataSource, remoteDataSource)  
}
```

Movie App using Manual Dependency Injection

All Movies' Feature DAG



Refactoring DataSources - Local

```
class MoviesLocalDataSource(private val dao: MoviesDao) : LocalDataSource {  
    override suspend fun getAllMovies(): List<Movie> {  
        return dao.getAllFavoriteMovies()  
    }  
  
    override suspend fun insertMovie(movie: Movie): Long {  
        return dao.insertMovie(movie)  
    }  
  
    override suspend fun deleteMovie(movie: Movie?): Int {  
        return if(movie!=null)  
            dao.deleteMovie(movie)  
        else  
            -1  
    }  
}
```

Refactoring DataSources - Remote

```
class MoviesRemoteDataSource(private val service: MoviesAPIService) : RemoteDataSource {  
    override suspend fun getAllMovies(): List<Movie>? {  
        return service.getAllMovies().body()?.results  
    }  
}
```

Creating Movies' App Container

- Determining all the needed dependencies to be exposed we can make an Interface like the following

```
interface AppContainer{  
    val moviesRemoteDataSource: MoviesRemoteDataSource  
    val moviesLocalDataSource: MoviesLocalDataSource  
    val repository : RepositoryInterface  
    val favFactory : FavMoviesViewModelFactory  
    val allFactory : AllMovieViewModelFactory  
}
```


Implementing the App Container

```
class AppContainerImpl(context: Context): AppContainer{

    override val moviesRemoteDataSource: MoviesRemoteDataSource by lazy {
        val service = RetrofitHelper.getInstance().create(MovieService::class.java)
        MoviesRemoteDataSourceImpl(service) ^lazy
    }

    override val moviesLocalDataSource: MoviesLocalDataSource by lazy {
        val db: AppDataBase = AppDataBase.getInstance(context)
        val dao = db.movieDAO()
        MoviesLocalDataSourceImpl(dao) ^lazy
    }

    override val repository: RepositoryInterface by lazy {
        Repository(moviesRemoteDataSource, moviesLocalDataSource)
    }

    override val favFactory: FavMoviesViewModelFactory by lazy {
        FavMoviesViewModelFactory(repository)
    }

    override val allFactory: AllMovieViewModelFactory by lazy {
        AllMovieViewModelFactory(repository)
    }
}
```

Injecting the App Container

- Inside your application class

```
class MyApplication : Application() {  
  
    lateinit var appContainer: AppContainer  
  
    override fun onCreate() {  
        super.onCreate()  
        appContainer = AppContainerImpl(context: this)  
    }  
}
```

Using the Dependencies

```
class AllMoviesActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        val application = application as MyApplication  
        setContent {  
            val factory = application.appContainer.allFactory  
            AllMoviesScreen(  
                ViewModelProvider(owner: this, factory)[AllMoviesViewModel::class.java]  
            )  
        }  
    }  
}
```

Let's Wrap up

Providing Dependencies

Class creates its own dependency	Service Locator	Manual Dependency Injection
Hard to test due to tight coupling	Hard to test because all objects interact with the same global service locator	Easy to test as we can swap the real implementation
Harder to know what a class needs from the outside.	Harder to know what a class needs from the outside.	Easier to know the dependencies of the class
Changes require all revisiting your class and edit everywhere the dependency is used.	Changes might cause runtime errors.	Changes isn't a heavy task. It could be done easily

Labs

1. Refactor the Products App architected with MVVM applying the Service Locator pattern
2. Refactor the Products App architected with MVVM applying the Manual Dependency Injection

For Both of your labs, don't forget to: refactor both of Local and Remote data sources to depend on Dao or Retrofit Service.

Automated dependency injection




More dependencies and classes can make manual injection of dependencies more tedious.

Manual dependency injection also presents several problems: 

1. For big apps, taking all the dependencies and connecting them correctly can require a large amount of boilerplate code.
2. Management of the lifetimes of your dependencies in memory.

Dagger - Hilt

Dagger

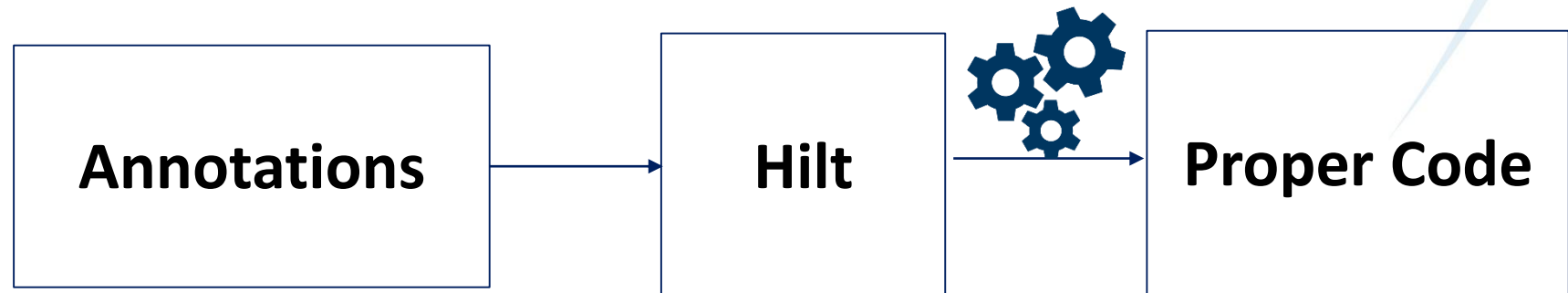
- Developers at Square  created Dagger-1 in 2012 as a dependency injection library
- It is a code generation tool.
- Before Dagger developers used other tools like:
 - PicoContainer  PicoContainer
 - Google Guice 
- They were hard to use and performed poorly
- The main reason for the poor performance was: the use of reflection at runtime

Introduction

- Hilt is a dependency injection library for Android that reduces the boilerplate of doing manual dependency injection in your project.
- Hilt provides a standard way to use DI in your application by providing containers for every Android class in your project and managing their lifecycles automatically.
- Hilt is built on top of the popular DI library **Dagger** to benefit from the compile-time correctness, runtime performance, scalability, and Android Studio support that Dagger provides.

Introduction Cont'd

- Hilt will help us generating the app container that we have done previously in the Manual dependency injection. With enhancement to the code
- Think of Hilt as a tool that does generate the desired code based on the rules that you will be telling it using the annotations
- Looks familiar?



Why using Hilt

- Reduced boilerplate
- Decoupled build dependencies
- Simplified configuration
- Improved testing
- Standardized components

Adding Hilt to Android Applications

1. In your build.gradle(project) add to your plugin section

```
id("com.google.devtools.ksp") version "2.0.21-1.0.27" apply false  
id("com.google.dagger.hilt.android") version "2.51.1" apply false
```

Adding Hilt to build.gradle (Module)

```
plugins {  
    id("com.google.devtools.ksp")  
    id("com.google.dagger.hilt.android")  
}  
dependencies {  
    //Hilt  
    implementation("com.google.dagger:hilt-android:2.51.1")  
    ksp("com.google.dagger:hilt-android-compiler:2.51.1")  
}
```

Hilt application class

All apps that use Hilt must contain an Application class that is annotated with **@HiltAndroidApp**.

@HiltAndroidApp triggers Hilt's code generation, including a base class for your application that serves as the application-level dependency container.

This generated Hilt component is attached to the Application object's lifecycle and provides dependencies to it. Additionally, it is the parent component of the app, which means that other components can access the dependencies that it provides.

Providing Dependencies

- Constructor Injection: Main injecting method to provide dependencies. It's used when dealing with a source code that you own in your project. So Hilt owns access to the constructor
- Field Injection: The second way to make DI in Android used with classes that we usually don't create ourselves. For example, Activity that we are not able to create, instead it's created using `PackageManager` for you

Define Hilt bindings

- **Binding:** is what Hilt needs to know how to provide instances of the necessary dependencies from the corresponding component. A binding contains the information necessary to provide instances of a type as a dependency.
- One way to provide binding information to Hilt is constructor injection. Use the `@Inject` annotation on the constructor of a class to tell Hilt how to provide instances of that class

@Inject

- By annotating the constructor of a class with `@Inject`. That tells hilt how to provide instances from this class.
- And when annotating an Android class with **`@AndroidEntryPoint`** hilt injects an instance of that type into a class.

Example

```
class Helper @Inject constructor ( ){  
    fun doSomething( ){...}  
}
```

@AndroidEntryPoint

```
class MainActivity : ComponentActivity {  
    @Inject lateinit var helper: Helper  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent{ }  
        helper.doSomething() //used the created instance by hilt  
    }  
}
```

Injecting Dependencies in Android classes

Once Hilt is set up in your Application class and an application-level component is available, Hilt can provide dependencies to other Android classes that have the `@AndroidEntryPoint` annotation.

`@AndroidEntryPoint`

```
class ExampleActivity : ComponentActivity() { ... }
```

If you annotate an Android class with `@AndroidEntryPoint`, then you also must annotate Android classes that depend on it. For example, if you annotate a fragment, then you must also annotate any activities where you use that fragment.

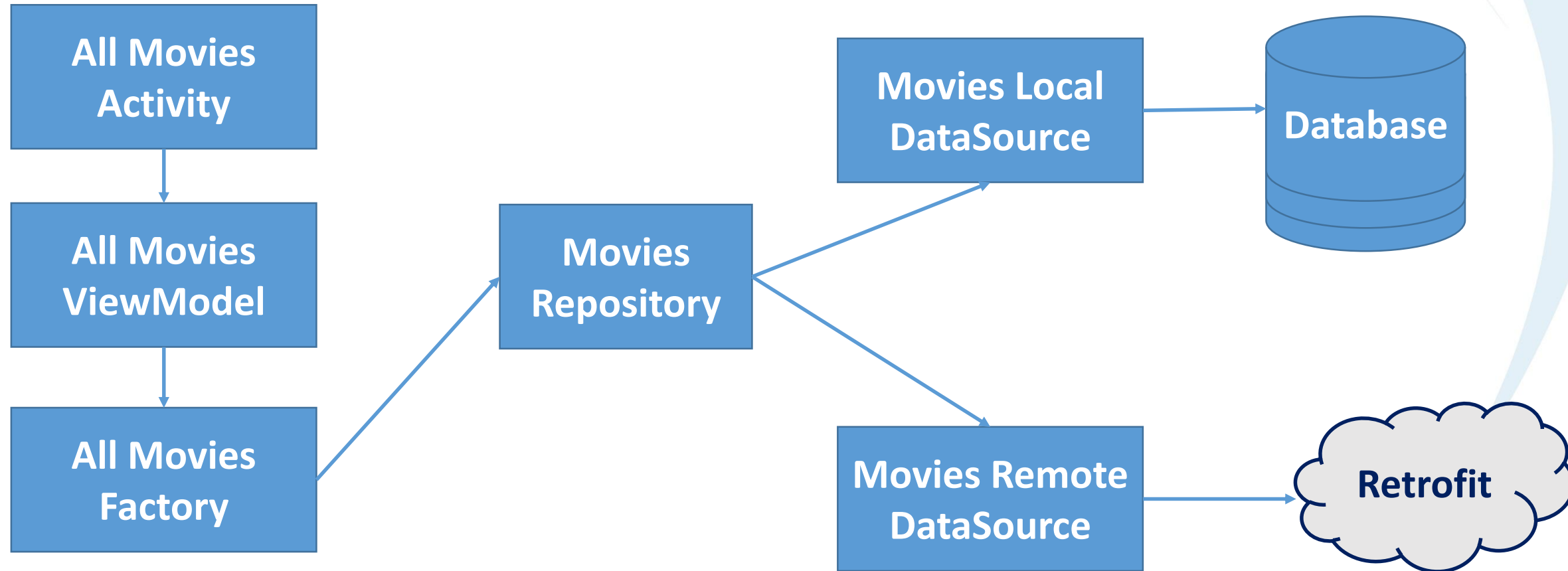
Supported Classes in Hilt

Hilt currently supports the following Android classes:

Android Class	Used Annotation
Application	<code>@HiltAndroidApp</code>
ViewModel	<code>@HiltViewModel</code>
Activity	<code>@AndroidEntryPoint</code>
Fragment	<code>@AndroidEntryPoint</code>
View	<code>@AndroidEntryPoint</code>
Service	<code>@AndroidEntryPoint</code>
Broadcast Receiver	<code>@AndroidEntryPoint</code>

Movies App. Using Hilt

Use Case - All Movies' Feature



Use Case - All Movies' Feature

- When you think about the dependency graph you can find that we have a transitive dependencies: Room database & Retrofit
- Both of these dependencies are classes that we can't have access to their constructor. So how we should add the `@inject` ?
- There should be another way to tell Hilt how it can provide objects from these classes.

Therefore, Modules are there for help!

Hilt modules

- Modules are used to add bindings to Hilt, or in other words, to tell Hilt how to provide instances of different types.
- In Hilt modules, you can include bindings for types that cannot be constructor-injected such as:
 1. Interfaces using **@Binds**
 2. Classes that are not contained in your project. An example of this is `Retrofit` or `Gson` - you need to use its builder to create an instance. Using **@Provides**

Hilt Modules Cont'd

- A Hilt module is a class annotated with **@Module** and **@InstallIn**
- **@Module** tells Hilt that this is a module
- **@InstallIn(SomeComponent)** tells Hilt the containers where the bindings are available by specifying a Hilt component. You can think of a Hilt component as a container.

@Provides

- Inside Hilt's Module. We can annotate a function with **@Provides** in Hilt modules to tell Hilt how to provide types that cannot be constructor injected.
- The function body of a function that is annotated with @Provides will be executed every time Hilt needs to provide an instance of that type.
- The return type of the @Provides-annotated function tells Hilt the binding type, the type that the function provides instances of..
- The function parameters are the dependencies of that type.

@Provides – Cont'd

```
@Module
@InstallIn(SingletonComponent::class)
public class FirstModuleProvider {

    @Provides
    fun provideRetrofit (): Retrofit {
        return Retrofit.Builder().baseUrl("https://api.themoviedb.org/3/discover/")
            .addConverterFactory(GsonConverterFactory.create()).build()    }

    @Provides
    fun provideMovieService (retrofit: Retrofit): MoviesAPIService{
        return retrofit.create(MoviesAPIService::class.java)    }

    @Provides
    fun provideDatabase(@ApplicationContext context: Context): MoviesDataBase {
        return MoviesDataBase.getInstance(context)    }

    @Provides
    fun provideMovieDao(db: MoviesDataBase): MoviesDao{ return db.getMovieDao() }
}
```

@Binds

- Inside Hilt's Module. We can annotate an abstract function with **@Binds** in Hilt modules to tell Hilt how to provide interface types that cannot be constructor injected.
- The return type of the @Binds-annotated function tells Hilt the binding type, the type that the function provides instances of.
- The function parameters are the class implementation of that type.

@Binds – Cont'd

```
@Module
@InstallIn(SingletonComponent::class)
abstract class SecondModule {

    @Binds
    abstract fun bindLocalDataSource(local: MoviesLocalDataSource): LocalDataSource

    @Binds
    abstract fun bindRemoteDataSource(remote: MoviesRemoteDataSource): RemoteDataSource

    @Binds
    @Singleton
    abstract fun bindRepository(repo: MoviesRepositoryImpl): MoviesRepository
}
```

Using the Bindings

Inside the ViewModel:

```
@HiltViewModel
class AllMoviesViewModel @Inject constructor(
    private val repo: MoviesRepository) : ViewModel() {

    private val mutableMovies: MutableLiveData<List<Movie>> = MutableLiveData()
    val movies: LiveData<List<Movie>> = mutableMovies

    private val mutableMessage: MutableLiveData<String> = MutableLiveData()
    val message: LiveData<String> = mutableMessage

    fun getMovies() {...}

    fun addToFavorites(movie: Movie?) {...}
}
```

Using ViewModel with Hilt

- Hilt is the recommended solution for dependency injection in Android apps, and works seamlessly with Compose.
- To use Hilt in your composable all you have to do is to call the `viewModel()` function
- The `viewModel()` function automatically uses the ViewModel that Hilt constructs with the `@HiltViewModel` annotation.

Using ViewModel with Hilt

@Composable

```
fun AllMoviesScreen( viewModel: AllMoviesViewModel= viewModel() ) {  
  
    viewModel.getMovies()  
    val moviesState = viewModel.movies.observeAsState()  
    val messageState = viewModel.message.observeAsState()  
    val snackBarHostState = remember { SnackBarHostState() }  
    Scaffold(  
        snackBarHost = { SnackBarHost(snackBarHostState) }  
    ) {  
        ...  
    }  
}
```

Activity Code

Inside the AllMovieActivity:

```
@AndroidEntryPoint
class AllMoviesActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView {
            AllMoviesScreen( )
        }
    }
}
```

Defining Multiple Modules

There are multiple reasons why we should create a new module:

- For better organization, a module's name should convey the type of information it provides.
- Every module is installed in its own scope so that it can be well managed by Hilt
- Hilt Modules cannot contain both `@Binds` and `@Provides` annotations in the same class.

Components

- A ***container*** is a class which is in charge of providing dependencies in your codebase and knows how to create instances of other types in your app. It manages the graph of dependencies required to provide those instances by creating them and managing their lifecycle.
- A container exposes methods to get instances of the types it provides. Those methods can return a new instance every time or always return the same instance. If the method always provides the same instance, we say that the type is ***scoped*** to the container.

@InstallIn Components

Hilt Component	Injector for
SingletonComponent	Application
ViewModelComponent	ViewModel
ActivityComponent	Activity
FragmentComponent	Fragment
ViewComponent	View
ServiceComponent	Service

Scoping instances to containers

By default, all bindings in Hilt are *unscoped*. This means that each time your app requests the binding, Hilt creates a new instance of the needed type.

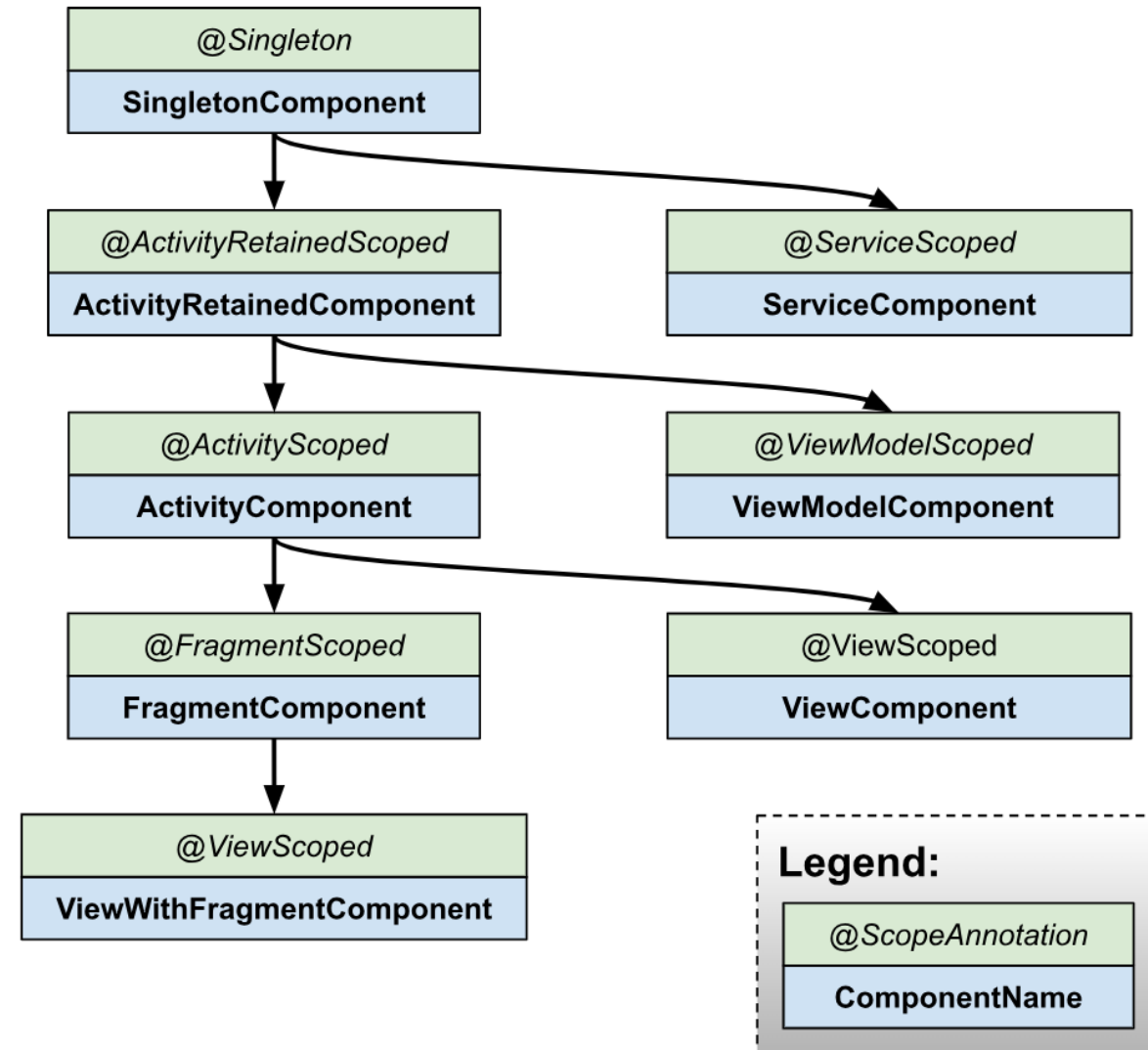
We can use annotations to scope instances to containers. As Hilt can produce different containers that have different lifecycles, there are different annotations that scope to those containers.

Hilt provides some pre-defined scopes that can be used in our code.

Scoping in Hilt

Android Class	Generated Component	Scope
Application	SingleComponent	@Singleton
Activity	ActivityRetainedComponent	@ActivityRetainedScoped
ViewModel	ViewModelComponent	@ViewModelScoped
Activity	ActivityComponent	@ActivityScoped
Fragment	FragmentComponent	@FragmentScoped
View	ViewComponent	@ViewScoped
Service	ServiceComponent	@ServiceScoped

Component Hierarchy

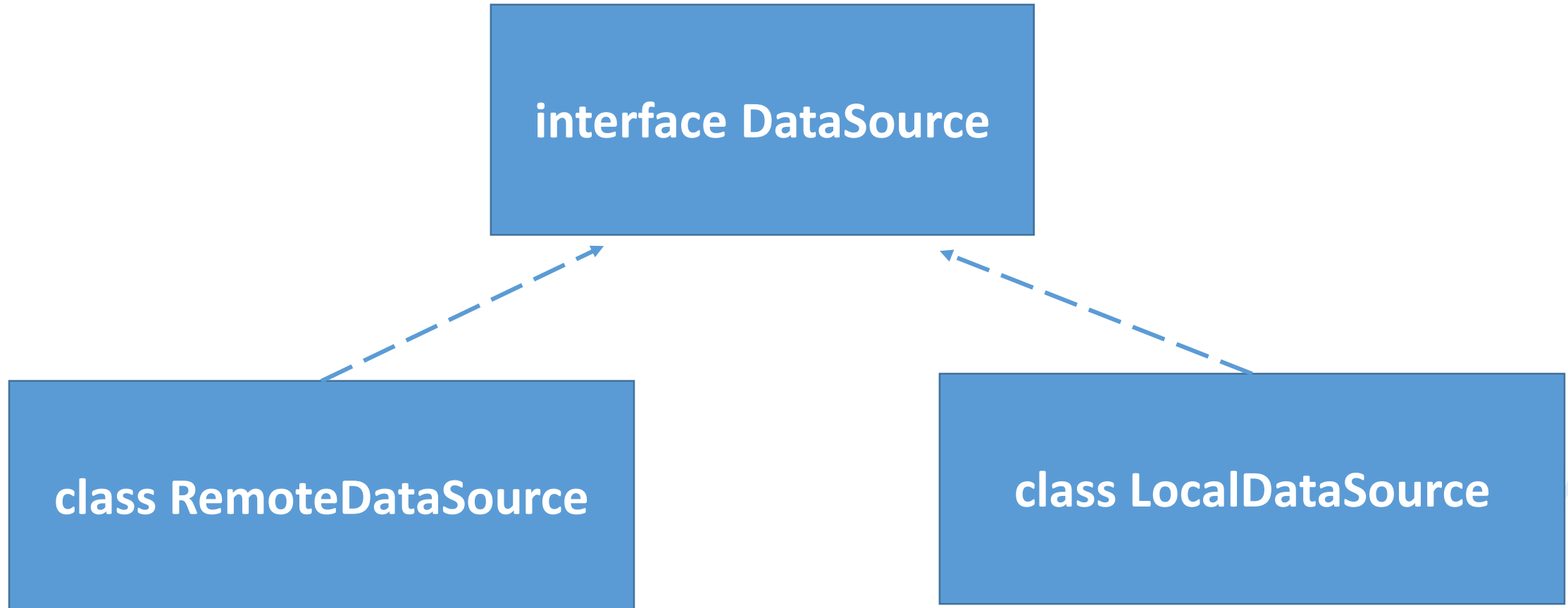


Provide Multiple Bindings for the Same Type

In cases where you need Hilt to provide different implementations of the same type as dependencies, you must provide Hilt with multiple bindings. You can define multiple bindings for the same type with qualifiers.

A qualifier is an annotation that you use to identify a specific binding for a type when that type has multiple bindings defined.

Provide Multiple Bindings for the Same Type



Provide Multiple Bindings for the Same Type

1. Define the qualifiers that you will use to annotate the **@Binds** or **@Provides** methods:

@Qualifier

annotation class RemoteDataSource

@Qualifier

annotation class LocalDataSource

Provide Multiple Bindings for the Same Type

2. Use the qualifier by adding them to your Module

```
@InstallIn(SingletonComponent::class)
@Module
abstract class RemoteModule {
    @RemoteDataSource
    @Singleton
    @Binds
    abstract fun bindRemoteDataSource(impl: RemoteDataSourceImpl): DataSource
}

@InstallIn(ActivityComponent::class)
@Module
abstract class LocalModule {
    @LocalDataSource
    @ActivityScoped
    @Binds
    abstract fun bindLocalDataSource(impl: LocalDataSourceImpl): DataSource
}
```

Provide Multiple Bindings for the Same Type

3. In your Activity| Fragment use the @Inject annotation as normal.
Also, add the qualifier

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

    @RemoteDataSource
    @Inject lateinit var source: DataSource
    ...
}
```

Predefined qualifiers in Hilt

- Hilt provides some predefined qualifiers. For example, as you might need the Context class from either the application or the activity, Hilt provides the `@ApplicationContext` and `@ActivityContext` qualifiers.
- Remember?

`@Provides`

```
fun provideMovieDao(@ApplicationContext appContext: Context): MovieDAO {  
    return AppDataBase.getInstance(appContext).movieDAO()  
}
```

To Wrap Up – Hilt

In order to use hilt inside an android application, we have to

1. Add the dependencies to both of the build.gradle files
2. Subclass the Application class
3. Annotate the Application subclass with **@HiltAndroidApp**
4. Annotate your android class with the proper annotation
5. Provide the dependency via:
 1. Constructor: by using **@Inject** before the constructor if you own this class.
 2. Module: If you don't own the class and hence you need using **@Provides** & **@Binds** annotations

@Provides vs. @Binds

@Provides	@Binds
Creating and providing instances of dependencies, allowing you to write custom creation logic within the method.	Does not handle object creation itself but establishes a relationship between an interface and its implementation.
Can be used to provide instances of any type, including interfaces, classes, or other objects.	specifically used for binding an interface to its implementation
Methods can be defined within regular classes.	methods must be declared as abstract within an abstract class or interface
need for explicit instance creation	more concise code as it focuses solely on binding the interface to its implementation

Koin



koin

Introduction

A framework to help you build any kind of Kotlin & Kotlin Multiplatform application:

- Android mobile
- Multiplatform apps

Koin is developed by [Kotzilla](https://kotzilla.com)  **kotzilla** and open-source contributors.

Why using Koin?

- Simplify your Dependency Injection infrastructure with smart API
- Kotlin DSL easy to read, easy to use, to let you write any kind of application
- Provides different kind of integration from Android ecoysystem

Getting Started with Koin

1. **Gradle Setup:** In your `build.gradle(app)` file add Koin's dependency for Android
2. Prepare your feature's scenario [user story]. And resolve the dependency acyclic graph (Components needed).
3. Create a Koin module using the `module` function.
4. Create the needed components inside your module.
5. Inject Dependencies in Android.
6. Start Koin in your Application class.

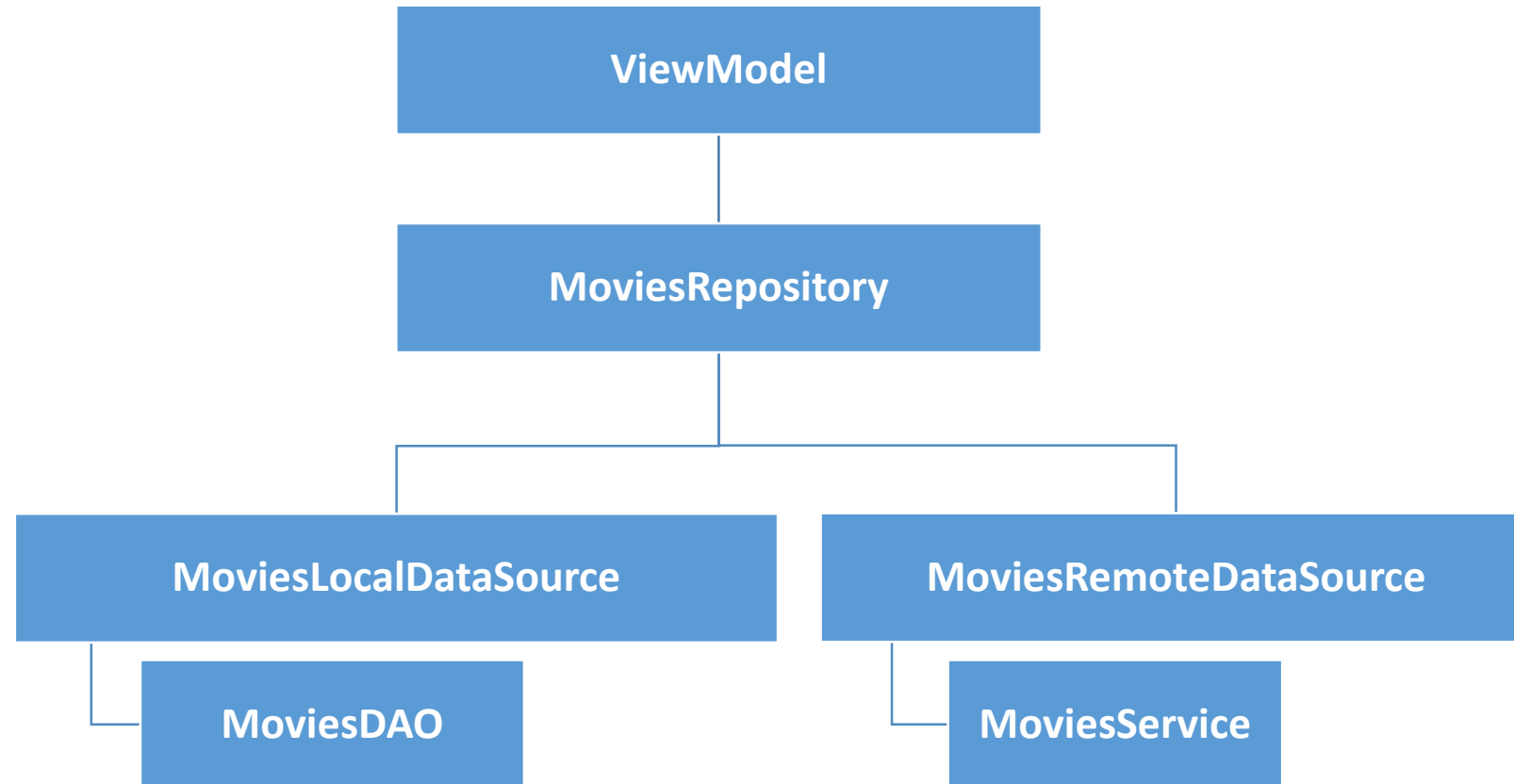
1. Adding Gradle dependencies.

```
dependencies {  
    // Koin for Android  
    implementation ("io.insert-koin:koin-android:$koin_version")  
  
    implementation("io.insert-koin:koin-androidx-compose:$koin_version")  
  
    implementation("io.insert-koin:koin-androidx-compose-navigation:$koin_version")  
  
}
```

You can find the dependencies [here](#)

2. Determine the Components Needed

For our Activities to work we need a ViewModel



3. Creating Koin Module

- A Koin module is a "space" to gather Koin definition. It's declared with the module function.
- Components doesn't have to be necessarily in the same module. A module is a logical space to help you organize your definitions, and can depend on definitions from other module. Definitions are lazy, and then are resolved only when a component is requesting it.
- To define a module use the module function for example:

```
val myFirstModule = module {  
    //your component goes here  
}
```

4. Create the Needed Components

Inside the module create your component by using these methods:

1. `single <Type>`: this function provides you with a Singleton that will live through your application
2. `factory<Type>`: this function creates a new instance every time an object is requested
3. `viewModel`: to help declare a ViewModel component and bind it to an Android Component lifecycle. (You **don't** need a ViewModelFactory)
4. `androidContext()`: to get a reference to the Context

Inside the module to get reference to any of the dependencies we can use the **get()** which will provide this component on our behalf

4. Create the Needed Components – Cont'd

```

val myModule = module {
    single<Retrofit> { Retrofit.Builder()
        .baseUrl("https://api.themoviedb.org/3/discover/")
        .addConverterFactory(GsonConverterFactory.create()).build()
    }
    single<MoviesAPIService> {
        get<Retrofit>().create(MoviesAPIService::class.java) }
    factory<RemoteDataSource> { MoviesRemoteDataSource(get()) }

    single<MoviesDataBase> { MoviesDataBase.getInstance(androidContext()) }
    single<MoviesDao> { get<MoviesDataBase>().getMovieDao() }
    factory<LocalDataSource> { MoviesLocalDataSource(get()) }

    single<MoviesRepository> {MoviesRepositoryImpl(get(), get()) }

    viewModel<AllMoviesViewModel> { AllMoviesViewModel(get()) }
    viewModel<FavMoviesViewModel> { FavMoviesViewModel(get()) }
}
  
```

5-a. Injecting into a @Composable

- While writing your composable function, you gain access to the following Koin API: **koinInject()**, to inject instance from Koin container
- For a module that declares a 'MyService' component:

```
val androidModule = module {  
    single { MyService() }  
}
```

- We can get service instance using **koinInject()** like that:

```
@Composable  
fun App() {  
    val myService = koinInject<MyService>()  
}
```

5-b. Injecting ViewModel for @Composable

- The same way you have access to classical single/factory instances, you gain access to the following Koin ViewModel API:
 1. **koinViewModel()** - inject ViewModel instance
 2. **koinNavViewModel()** - inject ViewModel instance + Navigation arguments data (if you are using Navigation API)

5. Injecting Dependencies in Android

- The AllMovieViewModel component will be created, resolving the MoviesRepository instance with it. To get it into our Composable, let's inject it with the **koinviewModel()** function:

@Composable

```
fun AllMoviesScreen(viewModel: AllMoviesViewModel = koinViewModel()) {
    viewModel.getMovies()
    val moviesState = viewModel.movies.observeAsState()
    val messageState = viewModel.message.observeAsState()
    val snackBarHostState = remember { SnackBarHostState() }
    Scaffold(
        snackBarHost = { SnackBarHost(snackBarHostState) }
    ) { contentPadding ->
        Column(
            modifier = Modifier(...), verticalArrangement = Arrangement.Center
        ) { ... }
    }
}
```

6. Start Koin in your Application class

We need to start Koin with our Android application. Just call the `startKoin()` function in the application's main entry point, our `Application` class

```
class MyApplication: Application() {  
    override fun onCreate() {  
        super.onCreate()  
  
        startKoin{ this: KoinApplication  
            androidContext(this@MyApplication)  
            modules(modules = arrayOf(modules, viewModels))  
        }  
    }  
}
```

MoviesApp using Koin

Injecting Dependencies

You inject your dependencies using:

- **by viewModel()** - lazy delegate property to inject a ViewModel into a property
- **getViewModel()** - directly get the ViewModel instance
- **by inject()** - lazy evaluated instance from Koin container
- **get()** - eager fetch instance from Koin container
- **by activityViewModel()** - lazy delegate property to inject shared ViewModel instance into a property
- **getActivityViewModel()** - directly get the shared ViewModel instance

Injecting Dependencies in Compose

When using compose, you can inject your dependencies using:

- **koinInject()** – injecting any object declared within your module
- **koinViewModel()** - inject ViewModel instance

Injecting the Dependencies- Examples

//To get any required dependency

```
val dependency: MyDependency = get ()  
val anotherDependency: MyDependency by inject()
```

//To get a viewModel

```
val viewModel: MyViewModel = getViewModel ()  
val anotherViewModel: MyViewModel by viewModel()
```

//To get a refernce to the shared viewModel inside your fragment

```
val viewModel : MyViewModel by activityViewModel()  
val anotherViewModel : MyViewModel = getActivityViewModel ()
```

Multiple Implementation

- You can specify a name to your definition, to help you distinguish two definitions about the same type
- Just request your definition with its name:

```
val myModule = module {  
    single<Service>(named("default")) { ServiceImpl1() }  
    single<Service>(named("test")) { ServiceImpl2() }  
}
```

```
val service : Service by inject(qualifier = named("default"))
```

- `get()` and `by inject()` functions let you specify a definition name if needed. This name is a qualifier produced by the `named()` function.

Hilt vs. Koin

Hilt	Koin
Compile time Injection Library, Dependencies will be injected at compile time using the annotationprocessor	Run time Injection Library, Dependencies will be provided when they are needed
Used for Injection for Android apps written in both Java and Kotlin	Used for Kotlin Injection with Applications like KMP
Follows the Dependency Injection	More likely to follow the Service locator Pattern
Using Annotation Processor to generate code	Using Reflections to generate code
Fast during Runtime	Slow during runtime
Slow during compile time	Fast during compile time

Labs

1. Refactor the Products App architected with MVVM applying Hilt
 - Refactor the data sources to depend on either the Dao or the ApiService
 - Create two Module one uses @Binds and another one that uses @Provide
2. Refactor the Products App architected with MVVM applying Koin.
 - Refactor the data sources to depend on either the Dao or the ApiService
 - Create multiple components using factory, single an viewModel and inject these modules in your Application.

Today's Required Badge



Hilt Codelab