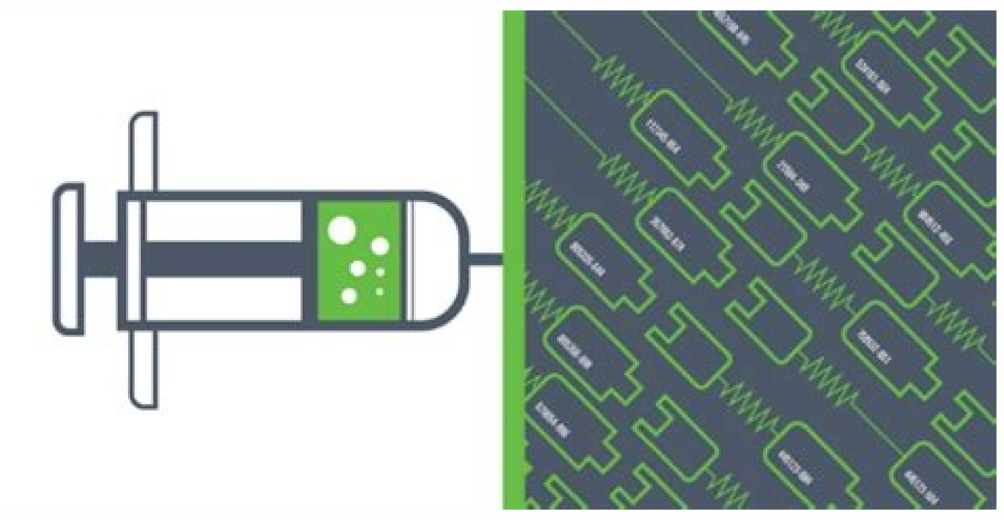
Kotlin dependency injection



Kotlin dependency injection spring. Kotlin dependency injection dagger. Kotlin dependency injection example. Kotlin dependency injection without framework. Kotlin dependency injection hilt. Kotlin dependency injection constructor. Kotlin dependency injection framework. Kotlin dependency injection library.

how to inject dependencies into a class using @Inject annotation for micronaut framework @Controller("/") class HelloController(val greetService:GreetService){ @Get("/hello") fun getMessage(){ greetService.greet } } class GreetService (val userRepo :UserRepo){ fun doSomething(val data:String){ userRepo.saveData(data) } } class UserRepo(val db:DbHandler){ fun saveData(val data){ db.save(data) } } as use @ Inject We can understand dependency injection using the following example. Let's say a child wants something from the fridge, so his parents ask the child what he needs and they can bring it to him. The parent does this so that the child does not collect unhealthy things, does not spoil, etc. Let's say the child wants to eat dinner and something to drink. His parents bring him a healthy lunch with orange juice from the fridge. As for Android, let's assume that each class needs some objects to function. So the class doesn't need to create these objects. These objects will be provided by the system to the class for it to function properly. This simple concept is called dependency injection android.car â â â â â -Engine In the example above, you can see that the class of the car depends on the class of the engine to run. So we can say that the class of the car depends on the class of the engine. This example has the following disadvantages. To write a unit test for a car class you need to create an engine object otherwise it won't work. the dependent motor class is coded. Suppose a different engine comes out in the future, so integration will be difficult. Here the car class controls the car and also creates an engine object. This is against the Single Responsibility Principle. Let's say there are constructors' changes in the engine class, so we have to make changes in the car class as well. This is not best practice, the class of the car should only change when its main function changes. When a car class is destroyed, a part of the engine is also destroyedThus, the life of the engine depends on the vehicle. In the real world, it does not happen that the engine is in order, but the car is damaged, and vice versa. We cannot reuse the engine object, it is created locally and used locally. Therefore, we cannot use the engine object with any other vehicle class. The system provided dependency counters in the following way to recognize the shortcomings of a hard-coded dependency. Here we are passing the dependency to the constructor, so this is called constructor injection. Constructor injection is most often used in large projects. We can now test the car class by passing in any mock object or implementation instead of the engine object. Suppose there is an electric motor in the future, so we can simply pass the electric motor object to the Car class, which will replace the motor object. Here, the car class has only one function - to drive a car. The engine object is not created. The service life of a motorcycle object does not depend on the vehicle class. The system creates an engine object elsewhere and passes it to the car class. We can reuse the system generated engine object with other classes. Here we are passing the dependency in the fields, so this is called field injection. We need to make sure you assign the object to an engine variable before we use it so we don't throw an error. By following the principles of dependency injection, you are laying the foundation for a good application architecture. Dependency Injection gives you the following benefits: Code reusability Easy refactoring Easy testing Dependency Injection Basics Before we dive into specific Android dependency injection, this page provides a more general overview of how dependency injection works. What is dependency injection? Classes often require references to other classes. For example, a car class might need a reference todegree. These required classes are called dependencies, and in this example the car class depends on the trigger of the engine class instance. A class can get the object it needs in three ways: The class creates the dependency it needs. In the example above, the car creates and initializes its own engine instance. Get it from somewhere else. Some Android APIs, such as context getters and getSystemService(), work this way. Pass it as a parameter. An application can expose these dependencies when creating a class or pass them to functions that require individual dependencies. In the example above, the Car constructor receives an Engine parameter. The third option is dependency injection! With this approach, you take the class dependencies and provide them instead of downloading from the class instance. Here is an example. Without dependency injection, the representation of a car creating its own engine dependency looks like this: class Car { private val engine = Engine() fun start() { engine.start() } } fun main(args: Array) { val auto = Auto ( ) car.start() } class Car { private Motor motor = new Motor(); public void start() { engine.start(); } } class MyApplication { public static void main(String[] args) { Car = new Car(); auto.start(); } } This is not an example of dependency injection because the car class constructs its own engine. This can be problematic because: Car and engine are tightly coupled - a car instance uses the same engine type and cannot easily be subclassed or have alternative implementations. If the car made its own engine, you would have to make two types of cars instead of just reusing the same car for both gas and electric engines. High engine dependency makes testing difficult. The car uses a real engine case, which avoids double testing to modify the engine for different test cases. What does the dependency injection code look like?Each Car instance that creates its own Engine object after initialization receives an Engine object as a parameter in its constructor: class Car(private val engine: Engine) { fun start() { engine.start() } } fun main(args: Array) { val engine = Motor() val car = Car(motor) car.start() } class Car {private final Engine engine; public car (Engine engine) { this.engine = engine; } public void start() { engine.start(); } } class MyApp { public static void main(String[] args) { Engine engine = new Engine(); auto auto = new car (engine); AutoStart(); } } The main function uses Auto. Because Car depends on Engine, the application creates an instance of Engine and then uses it to create an instance of Car. Advantages of this DI-based approach: Vehicle reusability. You can switch to different engine-to-car implementations. For example, you can define a new engine subclass named ElectricEngine for use in Car. If you are using DI, all you have to do is pass an instance of the updated ElectricEngine subclass and Car will continue to work without any further modifications. Simple vehicle check. You can go through test pairs and try different scenarios. For example, you can create a test engine twin called FakeEngine and customize it for different tests. There are two ways to inject dependencies in Android: Constructor injection. This is the method described above. You pass class dependencies to your constructor. Field injection (or setter injection). Some classes of the Android platform, e.g. Things like activities and fragments are generated by the system, so no constructor injection is possible. The embed field is used to create dependencies after the class is created. The code would look like this: class Car { latinit var engine: Engine fun start() { engine.start() } } fun main(args: Array) { val car = Car() car.engine = Engine() car. start() } class Auto { private engine Engine; public void setEngine(Engine-Engine) { this.engine = engine; } Publicitystart() { engine.start(); } } class MyApp { public static void main(String[] args) { Car car = new Car(); car.setEngine(new Engine()); autostart(); } } Note. Dependency injection is based on the principle of inverse control, where shared code controls the execution of specific code. Automatic Dependency Injection In the previous example, you created, deployed, and managed dependencies for different classes yourself without relying on a library. This is called manual dependency injection or manual dependency injection. The car example only had one dependency, but more dependencies and classes can make adding dependencies manually more tedious. There are also several problems with manual dependency entry. For large applications, getting all the dependencies and including them correctly can require a lot of boilerplate code. In a layered architecture, all the dependencies of the lower layers must be provided to create an object for the upper layer. For example, to build a real car, you may need an engine, transmission, chassis, and other parts. and the engine, in turn, needs cylinders and spark plugs. If you can't create dependencies before distributing them, such as using lazy initializers or setting objects to application threads, you must write and maintain your own container (or dependency graph) that manages the runtime of your dependencies. . in mind. There are libraries that solve this problem by automating the process of creating and deploying dependencies. They can be divided into two categories: reflection-based solutions that add dependencies at runtime. Static solutions that generate code for linking dependencies at compile time. Dagger is a popular Java, Kotlin and Android dependency injection library maintained by Google. Dagger makes it easy to use DI in your application by creating and managing the dependency graph for you. It offers completely static anddependencies to solve many of the design and performance problems of thought-based solutions like Guice. Alternative to Dependency Injection An alternative to dependency injection is to use a service locator. The service locator design pattern also improves the separation of classes from specific dependencies. You create a class known as a service finder that creates and stores dependencies and then provides those dependencies on demand. object ServiceLocator { fun getEngine(): Engine = Engine() } class Car { private val engine = ServiceLocator.getEngine() fun start() { engine.start() } } fun main(args: Array) { val car = Car () car.start() } class ServiceLocator { private static ServiceLocator instance = null; private ServiceLocator() {} public static ServiceLocator getInstance() { if (instance == null) { synchronized(ServiceLocator.class) { instance = new ServiceLocator(); } } return instance; } public Engine getEngine() { return new Engine(); } } class Car { private Engine = ServiceLocator.getInstance().getEngine(); public void start() { engine.start(); } } class MyApplication { public static void main(String[] args) { Car = new Car(); car.start(); } } The service locator pattern differs from dependency injection in the way it uses elements. Using the service locator model, classes have the ability to control and request the input of objects; with dependency injection, the application takes control and injects the required objects ahead of time. Compared to Dependency Injection: Aggregating the dependencies required by the service locator makes it difficult to test the code because all tests must interact with the same global service locator. Dependencies are hardcoded into the class implementation, not the API surface. Therefore, it is more difficult to understand from the outside what the class needs. As a result, changes to the dependencies available in the car or service finder can cause runtime errors or tests that cause link failures. Lifecycle Managementit's harder if you want to target something other than the lifetime of the whole application. Use Hilt in your android app. Hilt is Jetpack's recommended dependency injection library for Android. Hilt defines a standard way to perform dependency injection in your application by providing containers for every Android class in your project and automatically managing their life cycles. Handle is based on the popular Dagger DI library to take advantage of the compilation correctness, runtime performance, scalability, and Android Studio support that Dagger offers. For more information about the handle, see Dependency Injection with Hilt. Conclusion Dependency injection gives your application the following benefits: Class reuse and dependency separation: It is easier to replace dependency implementations. Code reuse is improved through inversion of control, and classes no longer have control over the creation of their dependencies, but instead work with any configuration. Simple refactoring: Dependencies become an auditable part of the API surface, allowing them to be checked at object creation or compile time, rather than hiding them as implementation details. It's easy to check: the class doesn't manage its dependencies. So when testing them, you can pass in different implementations and test all your different cases. To fully understand the benefits of dependency injection, you should try it manually in your application, as shown in Manual Dependency Injection. Additional Resources For more information about dependency injection, see the following additional resources. samples samples