MASTER DEPENDENCY INJECTION

FOR ANDROID USING

DAGGER

by Mahmoud Ramadan

Master Dependency Injection for Android Using Dagger 2

learn Dagger 2 with Kotlin Step by Step

Mahmoud Ramadan

This book is for sale at http://leanpub.com/dagger2_for_android

This version was published on 2019-10-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my mom , my wife and my daughter

Contents

Getting Started To Dagger 2 With Kotlin

Dagger is one of the most popular Dependency Injection Libraries for Java Programming Language and for Android App Development also. Because of the popularity of Kotlin as a new official language for developing Android Apps, I will use Kotlin in this book so I hope you are familiar with it. After finishing this chapter you will be able to understand the following:

- What is Dependency Injection(DI)
- Why we need DI
- Introduction to Dagger 2 library
- Understanding the types of DI
- Understanding Dagger Component

Prerequisites

To be able to get most out of this chapter you will need:

- Android Studio 3.2.1 or higher
- Emulator or real device
- Kotlin Basics
- Intermediate level as Android Developer

Clone the Repo from Github[1]

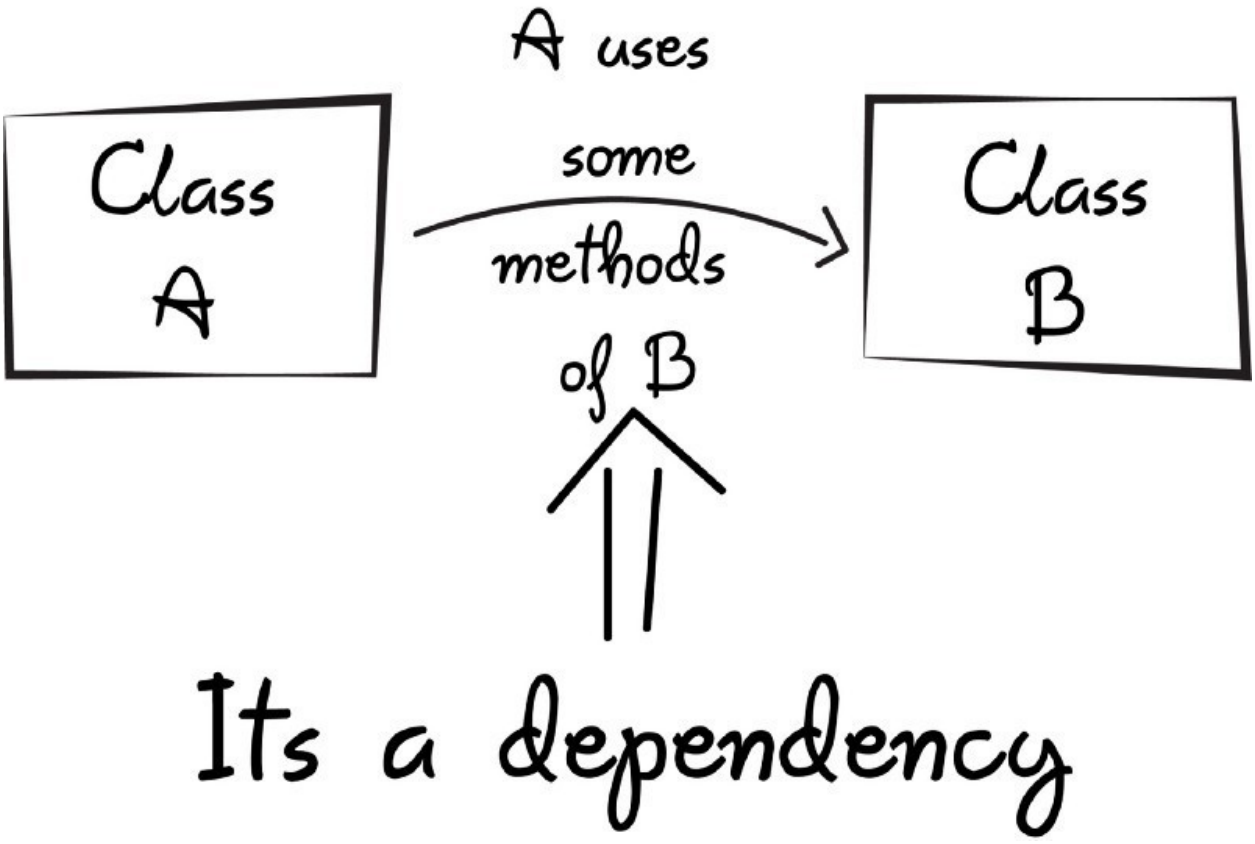
What is Dependency Injection(DI)

> In software engineering, dependency injection is a technique whereby one object supplies the dependencies of another object. A "dependency" is an object that can be used, for example as a service (wikipedia)

I know you may not understand this definition correctly so let me explain more in detail. In the beginning, I need you to understand the dependency meaning before understanding DI. Let me give you an example form our real life when we say you rely on transportation system to reach to your work this means you are dependent on the transportation system, you need them and without transportation you can not be able to go to your work so in this case you are a dependent (Client) and transportation is your dependency (Service).

[1] https://github.com/mrabelwahed/Dagger2Course

Simple DI Example



di

Now let us go back to our software Domain when we have class A that uses some methods form another class B so class A is a client and class B is a service. So DI creates your dependencies in someplace and returns your dependencies to you when you need them it is like magic.

Why we need DI

This is a good question in fact and I am here today to give you some thoughts I think it will convince you to use DI. I know you may face hard times when you try to implement DI in your project to be specific implement Dagger 2 in your Android Application but let me ask you a question and be honest with yourself, the question is **Did you really understand the DI and How Dagger 2 works for example?**

Now I will tell you why I need to use DI in the following points:

- Resolve the tight coupling between components

- Reuse some components and modules
- Optimize the heap memory using scopes
- Architect your app into separated components
- Facilitate the unit testing

Understanding DI with Simple Example

The idea is very simple we need to create a login module in our app where the user can log in through username and password for example and if it is successful we will save the username in shared preference. Let's do the following:

- Create a new Android Studio Project with Kotlin support
- create a new Package called di
- create a new class called Login Manager inside di package
- create a new class called Local Store inside di package

```
class LocalStore {}
```

- create a new class called ApiService inside di package

```
class ApiService {}
```

- Add a reference from LocalStore and ApiService in Login Manager class

```
1  class LoginManager {
2      val localStore = LocalStore()
3      val apiService = ApiService()
4  }
```

Now we need to pass the username and password for API and also save Token in shared preference so we will modify our classes to be like the following:

```
class APiService (val username : String , val password: String ){}
```

```
class LocalStore(val token : String){}
```

Now let us go back to our LoginManager class you will notice the error and this happens because you make changes on its dependencies LocalStore and ApiService classes. To fix this issue you need to modify your LoginManager to be like this

```
class LoginManager(val username : String  , val password: String ,val token : String\
 ){
   val localStore = LocalStore(token)
    val apiService =ApiService(username , password)
}
```

Now we fixed the issue **congrats** but Do you think this is the best solution for your problem? I think **NO** because if I need to change anything in the login manager's dependencies I need to change the login manager itself and this is called **TIGH COUPLING.**
So the simple fix for this issue is to pass the dependencies into login manager's constructor like this

```
class LoginManager(localStore , apiService){}
```

Understanding the Types of DI

we have three types of dependency injection

- Constructor Injection
- Field Injection
- Method Injection

Introduction to Dagger 2 library

Dagger 2 is one of the famous DI libraries among the Android Community since it is developed by Google. It is a forked project from Dagger 1 which introduced by Square. Dagger 2 does its work on Compile time effectively using Annotation Processing. Dagger has a lot of interesting annotations like Component, Subcomponent, Module, Scope, Bind. As I said before, Dagger is a Java DI framework so you can use it for enabling DI for your Java APPS also for Android but recently we have the Dagger-Android version. It is optimized for Android APPS in particular and it saves a lot of boilerplate code but I would like to pick up the pure Dagger first.

Notice: we are going to use Dagger 2 not Dagger2-android

Add Dagger to your Android Application

Let's go to Android Studio then open **Build.gradle** for app module and add the following lines:

```
    api 'com.google.dagger:dagger:2.22'
    kapt 'com.google.dagger:dagger-compiler:2.22'
```

and add kapt plugin on the top of the same file like this

```
apply plugin: 'kotlin-kapt'
```

Then sync and congrats you did it.

Dagger 2 Component

Before Talking about Dagger 2 Component I need to go back to the previous example to show you why we need a component.

Okay, let us open our MainActivity.kt

```
package com.daggerudemy

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import com.daggerudemy.di.ApiService
import com.daggerudemy.di.LocalStore
import com.daggerudemy.di.LoginManager
import com.daggerudemy.di.component.DaggerLoginComponent
import javax.inject.Inject
import kotlin.math.log

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Let's instantiate an object from our LoginManager and its dependencies and see what happens

```
package com.daggerudemy

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import com.daggerudemy.di.ApiService
import com.daggerudemy.di.LocalStore
import com.daggerudemy.di.LoginManager
import com.daggerudemy.di.component.DaggerLoginComponent
import javax.inject.Inject
import kotlin.math.log

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

```
        val localStore = LocalStore ("bdjdhjdhfhfdjhd87878787d78d7")
        val apiService = ApiService("ramadan","123")
        val manager = LoginManager(localStore , apiStore)
    }
}
```

if you look carefuly you will notice a boilerplate code and if you need to modify the local store to hold another object for example you will need also to create it first then pass it to the local store and this is a hassle. I just need instance form login manager and that's so we need something to abstract this creation and manage dependencies efficiently and here Component comes to the stage let's give it a big hand
The component is just a bridge between the client which is MainActivity in this case and the service which is LoginManager and its function just to give the requester what needs when he asks.

How to create a Component

to create a component in Dagger 2 you need to define a new interface and your methods but you must annotate this interface with @Component.Let's add new package called component under di package in our project. Under this package, I will create a new interface called LoginComponent and I will add @Component annotation on the top of this interface like the following:

```
@Component
interface LoginComponent {
    fun getLoginManager() :LoginManager
}
```

Now let's make constructor Injection for our LoginManager class and add login method within it like the following:

```
package com.daggerudemy.di
import android.util.Log
import javax.inject.Inject

class LoginManager  @Inject constructor(private val localStore: LocalStore , private\
 val apiService: ApiService){
  fun login(username : String , pass:String){
        Log.d("LoginManager","login($username , $pass)")
       val token = apiService.authenticate(username,pass)
        localStore.saveUserToken(token)
    }

}
```

`@Inject` annotation means I need to create an instance from this class by passing objects to its constructor from another place. Now we need to make constructor Injection for login manager's dependencies also like the following:

```
package com.daggerudemy.di
import android.util.Log
import javax.inject.Inject
class LocalStore  @Inject constructor(){
    fun saveUserToken(token: String) {
        Log.d("LocalStore","saveUserToken($token)")
    }
}
```

```
package com.daggerudemy.di
import android.util.Log
import javax.inject.Inject

class ApiService @Inject constructor() {

 fun authenticate(username: String, pass: String): String {
        Log.d("ApiService", "authenticate($username , $pass)")
        return "wxydldklkd78dsnjuudiiudf"
    }
}
```

Now go to Build menu in the toolbar and click on MakeProject, Dagger begins to create your dependencies in the background and your component now is ready to use. Let's go back to the MainActivity class and remove the previous code and make an instance from your new login component like the following:

```
package com.daggerudemy

import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import com.daggerudemy.di.ApiService
import com.daggerudemy.di.LocalStore
import com.daggerudemy.di.LoginManager
import com.daggerudemy.di.component.DaggerLoginComponent
import javax.inject.Inject
import kotlin.math.log

class MainActivity : AppCompatActivity() {
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
         super.onCreate(savedInstanceState)
         setContentView(R.layout.activity_main)
           val loginComponent = DaggerLoginComponent.create()
           loginComponent.getLoginManager().login("ramadan","123")
     }
}
```

Now run the app and see the log

Logcat

```
2019-10-06 13:18:06.491 7261-7261/com.daggerudemy D/LoginManager: login(ramadan , 12\
3)
2019-10-06 13:18:06.491 7261-7261/com.daggerudemy D/ApiService: authenticate(ramadan\
 , 123)
2019-10-06 13:18:06.491 7261-7261/com.daggerudemy D/LocalStore: saveUserToken(wxydld\
klkd78dsnjuudiiudf)
Now let's see the generated code for our component to understand DI well.
```

Now Let's see the generated code for login component

```
// Generated by Dagger (https://google.github.io/dagger).
package com.daggerudemy.di.component;

import com.daggerudemy.di.ApiService;
import com.daggerudemy.di.LocalStore;
import com.daggerudemy.di.LoginManager;

public final class DaggerLoginComponent implements LoginComponent {
  private DaggerLoginComponent() {}

  public static Builder builder() {
    return new Builder();
  }

  public static LoginComponent create() {
    return new Builder().build();
  }

  @Override
  public LoginManager getLoginManager() {
    return new LoginManager(new LocalStore(), new ApiService());
```

```
22    }
23
24    public static final class Builder {
25      private Builder() {}
26
27      public LoginComponent build() {
28        return new DaggerLoginComponent();
29      }
30    }
31  }
```

As you can see the Java generated DaggerLoginComponnet implements the LoginComponnet Interface and instantiate instance from LoginManager and there is Builder class with build method which returns an object from DaggerLoginComponnet, Simple isn't it?. Now let's improve the readability of our code to be like this

```
1  package com.daggerudemy
2
3  import android.os.Bundle
4  import android.support.v7.app.AppCompatActivity
5  import com.daggerudemy.di.LoginManager
6  import com.daggerudemy.di.component.DaggerLoginComponent
7
8  class MainActivity : AppCompatActivity() {
9
10  private val loginManager: LoginManager? = null
11
12 override fun onCreate(savedInstanceState: Bundle?) {
13         super.onCreate(savedInstanceState)
14         setContentView(R.layout.activity_main)
15         val loginComponent = DaggerLoginComponent.create()
16         loginManager?.login("ramadan", "123")
17     }
18 }
```

It is a simple enhancement we just make private nullable instance from LoginManager and make safe call login on it right, but the power here is we are able to make private instance from our class and this may be very important for your use case, unlike Field Injection.

Field Injection

Until now we are dealing with Constructor Injection so what about the second type of DI (Field Injection).Well, Field Injection is very useful in some situations like when we deal with third

party classes and we do not own them like Picasso, Okhttp,..etc or classes that implement some interface. Let's see Field Injection in Action right now. Please go to your MainActivity.kt and LoginComponent.kt and make the following changes

```
package com.daggerudemy

import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import com.daggerudemy.di.LoginManager
import com.daggerudemy.di.component.DaggerLoginComponent
import javax.inject.Inject

class MainActivity : AppCompatActivity() {

 @Inject
    lateinit var loginManager: LoginManager

override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        val loginComponent = DaggerLoginComponent.create()
        loginComponent.inject(this)
       loginManager.login("ramadan", "123")
    }
}
```

```
package com.daggerudemy.di.component

import com.daggerudemy.MainActivity
import dagger.Component

@Component
interface LoginComponent {
    fun inject(mainActivity: MainActivity)
}
```

As you can see we changed the private nullable instance of LoginManager to be public lateinit var with @Inject annotation because we can not inject private instance in Field Injection and this is one of Field Injection's Problems. Also, we modified the LoginComponent through adding a new method called inject that takes the client or the MainActivity as a client and it inject the login manager object in the activity in a simple way.

Notice: make sure to make the instance public and access the instance after calling inject(this) method.

Now let's see the generated Java code of Login Component after the recent modification to be able to understand the DI well.

```
1  DaggerLoginComponent.java
2  // Generated by Dagger (https://google.github.io/dagger).
3  package com.daggerudemy.di.component;
4
5  import com.daggerudemy.MainActivity;
6  import com.daggerudemy.MainActivity_MembersInjector;
7  import com.daggerudemy.di.ApiService;
8  import com.daggerudemy.di.LocalStore;
9  import com.daggerudemy.di.LoginManager;
10
11 public final class DaggerLoginComponent implements LoginComponent {
12   private DaggerLoginComponent() {}
13
14   public static Builder builder() {
15     return new Builder();
16   }
17
18   public static LoginComponent create() {
19     return new Builder().build();
20   }
21
22   private LoginManager getLoginManager() {
23     return new LoginManager(new LocalStore(), new ApiService());
24   }
25
26   @Override
27   public void inject(MainActivity mainActivity) {
28     injectMainActivity(mainActivity);
29   }
30
31   private MainActivity injectMainActivity(MainActivity instance) {
32     MainActivity_MembersInjector.injectLoginManager(instance, getLoginManager());
33     return instance;
34   }
35
36   public static final class Builder {
37     private Builder() {}
38     public LoginComponent build() {
39       return new DaggerLoginComponent();
40     }
```

```
  }
}
```

As you can see the newly generated code looks like the same previous one but there is a little change where we have a method called injectMainActivity(mainActivity) like the following:

```
private MainActivity injectMainActivity(MainActivity instance) {
  MainActivity_MembersInjector.injectLoginManager(instance,getLoginManager());
    return instance;
  }
```

This method will be responsible for injecting LoginManager Instance in the MainActivity.kt by using MainActivity_MembersInjector class

```
MainActivity_MembersInjector.java
// Generated by Dagger (https://google.github.io/dagger).
package com.daggerudemy;

import com.daggerudemy.di.LoginManager;
import dagger.MembersInjector;
import javax.inject.Provider;

public final class MainActivity_MembersInjector implements MembersInjector<MainActiv\
ity> {
  private final Provider<LoginManager> loginManagerProvider;

  public MainActivity_MembersInjector(Provider<LoginManager> loginManagerProvider) {
    this.loginManagerProvider = loginManagerProvider;
  }

  public static MembersInjector<MainActivity> create(Provider<LoginManager> loginMan\
agerProvider) {
    return new MainActivity_MembersInjector(loginManagerProvider);
  }

  @Override
  public void injectMembers(MainActivity instance) {
    injectLoginManager(instance, loginManagerProvider.get());
  }

  public static void injectLoginManager(MainActivity instance, LoginManager loginMan\
ager) {
```

```
    instance.loginManager = loginManager;
  }
}
```

MainActivity_MembersInjector class implements an interface called MembersInjector.java which has a method called void injectMembers(T instance).

```
MembersInjector.java
/*
 * Copyright (C) 2012 The Dagger Authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package dagger;

/**
 * Injects dependencies into the fields and methods on instances of type {@code T}. \
Ignores the
 * presence or absence of an injectable constructor.
 *
 * @param <T> type to inject members of
 *
 * @since 2.0 (since 1.0 without the provision that {@link #injectMembers} cannot ac\
cept
 *      {@code null})
 */
public interface MembersInjector<T> {

  /**
   * Injects dependencies into the fields and methods of {@code instance}. Ignores t\
he presence or
   * absence of an injectable constructor.
```

```
37     *
38     * <p>Whenever a {@link Component} creates an instance, it performs this injection\
39  automatically
40     * (after first performing constructor injection), so if you're able to let the co\
41 mponent create
42     * all your objects for you, you'll never need to use this method.
43     *
44     * @param instance into which members are to be injected
45     * @throws NullPointerException if {@code instance} is {@code null}
46     */
47   void injectMembers(T instance);
48 }
```

so in a simple way, you just set the login manager instance to The MainActivity Instance through injectMembers method.

Conclusion

In this chapter, we talked about the Dependency Injection and why it is very vital for your application architecture, also we learned the different types of DI and we started to learn Dagger 2 Library and to understand the component concept. I the next tutorial we will talk about more concepts in Dagger 2 Like Module, Scopes, SubComponnet deeply.

Dagger 2 Method Injection, Module, Custom Scope

Dagger is one of the most popular Dependency Injection Libraries for Java Programming Language and for Android App Development also.After finishing this chapter you will be able to understand the following

- What is Method Injection
- Understanding Dagger 2 Module
- Understanding Custom Scopes

Prerequisites

To be able to get most out of this chapter you will need:

- Android Studio 3.2.1 or higher
- Emulator of phone
- Kotlin Basics
- Intermediate level as Android Developer

Clone Repo From Github[2]

Method Injection

Method Injection is the third type of DI and it is very helpful in some scenarios like when we need to enable caching for our login manager. Let's assume we have Config class that responsible for enabling and Disabling all features in your App like enable/disable caching So let's create a new class called Config.kt

[2] https://github.com/mrabelwahed/Dagger2Course

```
package com.daggerudemy.di
import android.util.Log
import javax.inject.Inject

class Config @Inject constructor(){
  var isCachEnabled = false
  fun enableCache(loginManager: LoginManager){
        isCachEnabled = true
        Log.d("Config","${this.isCachEnabled}")
    }
}
```

Let's consume this class in our Login Manager Like this

```
package com.daggerudemy.di

import android.util.Log
import javax.inject.Inject

class LoginManager  @Inject constructor(private val localStore: LocalStore , private\
 val apiService: ApiService){
   fun login(username : String , pass:String){
       Log.d("LoginManager","login($username , $pass)")
       val token = apiService.authenticate(username,pass)
       localStore.saveUserToken(token)
    }
    @Inject
     fun enableCache(config: Config){
      Log.d("LoginManger","${config.isCachEnabled}")
      config.enableCache(this)
     }
}
```

Notice: Method Injection happens after constructor Injection

Now Let's go to MainActivity.kt and add a new variable of Config class like the following

`@Inject lateinit var config:Config`

Then we need to call enable cache Method on the login Manager like the following
‘
loginManager.enableCache(config)‘

now let's run the app and see the log to test the method injection is working well

Logcat
2019-10-11 14:13:48.41013470-13470/com.daggerudemy D/LoginManger: false
2019-10-11 14:13:48.41013470-13470/com.daggerudemy D/Config: true
As you can see the injection of login manager through the method injection is working well

Dagger 2 Module

The Module is one of awesome features in Dagger 2 because it is an optimized solution for Handling third party dependencies like Retrofit, Okhttp, Picasso,..etc. Also, you can reuse these modules in your apps you just need to define them once and you can copy these modules and use them in your different apps.

I will assume I do not own the Local Store class and it is developed by another developer as a library for example and let's apply module concept but the question now is

How to define a new Dagger 2 module?

First, let's create a new package called module then define a new class called LocalStoreModule.kt

Then ass `@Module` Annotation on the top of it then add a new function called provideLocalStore which returns new Instance of LocalStore() and do not forget to add @Provides annotation on the top of the method

```
1  package com.daggerudemy.di.module
2  mport com.daggerudemy.di.LocalStore
3  import dagger.Module
4  import dagger.Provides
5
6  @Module
7  class LocalStoreModule {
8      @Provides
9      fun provideLocalStore() = LocalStore()
10 }
```

Then we need to change Local Store class to be like this to demonstrate the Module concept

```
package com.daggerudemy.di

import android.util.Log

class LocalStore {
  fun saveUserToken(token: String) {
        Log.d("LocalStore", "saveUserToken($token)")
    }
}
```

we removed the constructor Injection for the class only, perfect Now let's go to our login component and add the following

```
package com.daggerudemy.di.component
import com.daggerudemy.MainActivity
import com.daggerudemy.di.module.LocalStoreModule
import dagger.Component

@Component(modules = [LocalStoreModule::class])
interface LoginComponent {
    fun inject(mainActivity: MainActivity)
}
```

we link the component with a new module called LocalStoreModule where Login Componenet can access the Local store dependencies and provide them to the client when it needs them. let's run to see if our app still behave as previous or not

Logcat
2019-10-11 14:56:04.73914555-14555/com.daggerudemy D/Config: true
2019-10-11 14:56:04.73914555-14555/com.daggerudemy D/LoginManager:login(ramadan , 123)
2019-10-11 14:56:04.74014555-14555/com.daggerudemy D/ApiService: authenticate(ramadan , 123)
2019-10-11 14:56:04.74014555-14555/com.daggerudemy D/LocalStore:
saveUserToken(wxydldklkd78dsnjuudiiudf)
2019-10-11 14:56:04.74014555-14555/com.daggerudemy D/LoginManger: false
2019-10-11 14:56:04.740
14555-14555/com.daggerudemy D/Config: true

Perfect it is working as the previous

Generated Code for our Login component after adding the Local store Module

```
1  // Generated by Dagger (https://google.github.io/dagger)
2  package com.daggerudemy.di.component;
3  import com.daggerudemy.MainActivity;
4  import com.daggerudemy.MainActivity_MembersInjector;
5  import com.daggerudemy.di.ApiService;
6  import com.daggerudemy.di.Config;
7  import com.daggerudemy.di.LoginManager;
8  import com.daggerudemy.di.LoginManager_Factory;
9  import com.daggerudemy.di.LoginManager_MembersInjector;
10 import com.daggerudemy.di.module.LocalStoreModule;
11 import com.daggerudemy.di.module.LocalStoreModule_ProvideLocalStoreFactory;
12 import dagger.internal.Preconditions;
13 public final class DaggerLoginComponent implements LoginComponent {
14 private final LocalStoreModule localStoreModule;
15
16 private DaggerLoginComponent(LocalStoreModule localStoreModuleParam) {
17  this.localStoreModule = localStoreModuleParam;
18 }
19 public static Builder builder() {
20  return new Builder();
21 }
22
23  public static LoginComponent create() {
24   return new Builder().build();
25 }
26  private LoginManager getLoginManager() {
27  return injectLoginManager(
28  LoginManager_Factory.newInstance<>(
29 LocalStoreModule_ProvideLocalStoreFactory.provideLocalStore(localStoreModule),
30  new ApiService())); }
31  @Override
32  public void inject(MainActivity mainActivity) {
33  injectMainActivity(mainActivity); }
34  private LoginManager injectLoginManager(LoginManager instance) {
35  LoginManager_MembersInjector.injectEnableCache(instance, new Config());
36 return instance;
37  }
38 private MainActivity injectMainActivity(MainActivity instance) { MainActivity_Member\
39 sInjector.injectLoginManager(instance, getLoginManager());
40 MainActivity_MembersInjector.<em>injectConfig</em>(instance, new Config());
41  return instance;
42 }
43  public static final class Builder {
```

```
 private LocalStoreModule localStoreModule;
 private Builder() {}
 public Builder localStoreModule(LocalStoreModule localStoreModule) {
 this.localStoreModule = Preconditions.checkNotNull(localStoreModule);
 return this;
 }
public LoginComponent build() {
if (localStoreModule == null) {
this.localStoreModule = new LocalStoreModule();
 }
return new DaggerLoginComponent(localStoreModule);
 }
 }
}
```

As you can see you can notice some changes like DaggerLoginComponent takes localStroeModule as a parameter because it depends on it. The second change is injectMainActivity

This method has some changes it injects two instance from the login manager and config for our main activity And also if we give getLoginManager a close look we can notice

```
private LoginManager getLoginManager() {
  return injectLoginManager(LoginManager_Factory.newInstance(
LocalStoreModule_ProvideLocalStoreFactory.provideLocalStore(localStoreModule),new Ap\
iService()));
}
```

And `LoginManager_Factory Implements Factory<LoginManager>`

Second Usecase for Module
Let assume that the Api service is an interface and I need to make different implementation for this interface like I need to make separate Login services for our login module So we will make some changes in API service class

Firstful we will change it to be interface then we will create a new class called LoginService that implements ApiService Like this

```
interface ApiService {
    fun authenticate(username: String, pass: String): String
}
```

```
package com.daggerudemy
import android.util.Log
import com.daggerudemy.di.ApiService
import javax.inject.Inject

class LoginService  @Inject constructor() : ApiService {
  override fun authenticate(username: String, pass: String): String {
       Log.d("ApiService", "authenticate($username , $pass)")
       return "wxydldklkd78dsnjuudiiudf"}
}
```

let's connect the login component with the new login service module

```
package com.daggerudemy.di.component
import com.daggerudemy.MainActivity
import com.daggerudemy.di.module.LocalStoreModule
import com.daggerudemy.di.module.LoginServiceModule
import dagger.Component

@Component(modules = [LocalStoreModule::class , LoginServiceModule::class])
interface LoginComponent {
  fun inject(mainActivity: MainActivity)
}
```

if you run the application you can find the same behavior, perfect. Now if you open the Dagger-LoginComponent to see the generated code you will find some changes like DaggerLoginComponent now depends on a new object which is LoginService Module

Now let's go back to our LoginServiceModule. You can see there is a clear redundancy like in provideLoginService Method, the passing parameter is the same as the return So we can make little improvement which is @Bind. You can use @Bind with an abstract method to do the same behavior of @provides but with more optimization like this

```
package com.daggerudemy.di.module
import com.daggerudemy.LoginService
import com.daggerudemy.di.ApiService
import dagger.Binds
import dagger.Module
import dagger.Provides

@Module
abstract  class LoginServiceModule  {
     @Binds
```

```
11      abstract  fun bindLoginService(loginService: LoginService) :ApiService
12  }
```

we made some refactoring here, firstful we changed the class to be abstract class with an abstract method called bindLoginService instead of provideLoginMethod and with @Binds as annotation

Introduction to Dagger2 Scopes
The Scope is the lifetime of the object in memory so if you do not need to recreate instance again and again you need to define a custom scope for it . One of most popular scopes in Dagger is Singelton and this scope provides shared objects through the whole app so we need to reuse this idea for custom scope.

Why Scopes?

Scope provides us a localization and this is very important for layered architecture like clean architecture where where we have 3 layers presentation for handling UI, Domain Layer for managing the business and Data Layer for managing the different data sources. So we can make different scopes for each layer and this is very important for the separation of concerns and scalability.

The question now how can I define a custom scope?

Okay, let's create an example without scope first to see the benefits when we add a custom scope. The idea is very simple we will create app component to give us an instance from App Logger With incremental index like the following:

```
1  package com.daggerudemy.di
2
3   class AppLogger (val value:String) {
4   }
```

then App Module

```
1  package com.daggerudemy.di.module
2   import com.daggerudemy.di.AppLogger
3   import dagger.Module
4   import dagger.Provides
5
6   @Module
7   class AppModule {
8       var index = 0
9     @Provides
10      fun getAppLogger():AppLogger {
```

```
11          index++
12          return  AppLogger("index = $index")
13      }
14  }
```

then link the app component with app module

```
1  package com.daggerudemy.di.component
2   import com.daggerudemy.di.AppLogger
3   import com.daggerudemy.di.module.AppModule
4   import dagger.Component
5
6  @Component(modules = [AppModule::class])
7   interface AppComponent {
8       fun getAppLogger():AppLogger
9   }
```

```
1  package com.daggerudemy
2
3   import android.app.Application
4   import android.util.Log
5   import com.daggerudemy.di.component.DaggerAppComponent
6   import com.daggerudemy.di.module.AppModule
7
8   class App: Application() {
9       override fun onCreate() {
10          super.onCreate()
11          val appComponent =
12        DaggerAppComponent.builder().appModule(AppModule()).build()
13          Log.d("App",appComponent.getAppLogger().value)
14          Log.d("App",appComponent.getAppLogger().value)
15          Log.d("App",appComponent.getAppLogger().value)
16      }
17  }
```

now let's run and see the result

Logcat
2019-10-12 12:52:52.0723115-3115/com.daggerudemy D/App: index = 1
2019-10-12 12:52:52.0723115-3115/com.daggerudemy D/App: index = 2
2019-10-12 12:52:52.0733115-3115/com.daggerudemy D/App: index = 3
Now you can see the result which has 3 different values for the index which means three different instantiations for the App logger class and this is not memory optimization now how can we solve this situation here custom scope comes to the scene please give a big hand forDagger2 Custom scope

CustomScope

Let's create a new package under di and called it scope and create a new scope called AppScope

package com.daggerudemy.di.scope
import javax.inject.Scope

```
 @Scope
 @Retention
 annotation class  AppScope
```

then add this scope to the app component and app module method to get instance from app logger like this

```
package com.daggerudemy.di.component
import com.daggerudemy.di.AppLogger
import com.daggerudemy.di.module.AppModule
import com.daggerudemy.di.scope.AppScope
import dagger.Component

@AppScope
@Component(modules = [AppModule::class])

interface AppComponent {
    fun getAppLogger():AppLogger
}
```

and like this

```
package com.daggerudemy.di.module

import com.daggerudemy.di.AppLogger
import com.daggerudemy.di.scope.AppScope
import dagger.Module
import dagger.Provides

@Module
class AppModule {
   var index = 0
   @Provides
   @AppScope
   fun getAppLogger():AppLogger {
```

```
14          index++
15          return  AppLogger("index = $index")
16      }
17
18  }
```

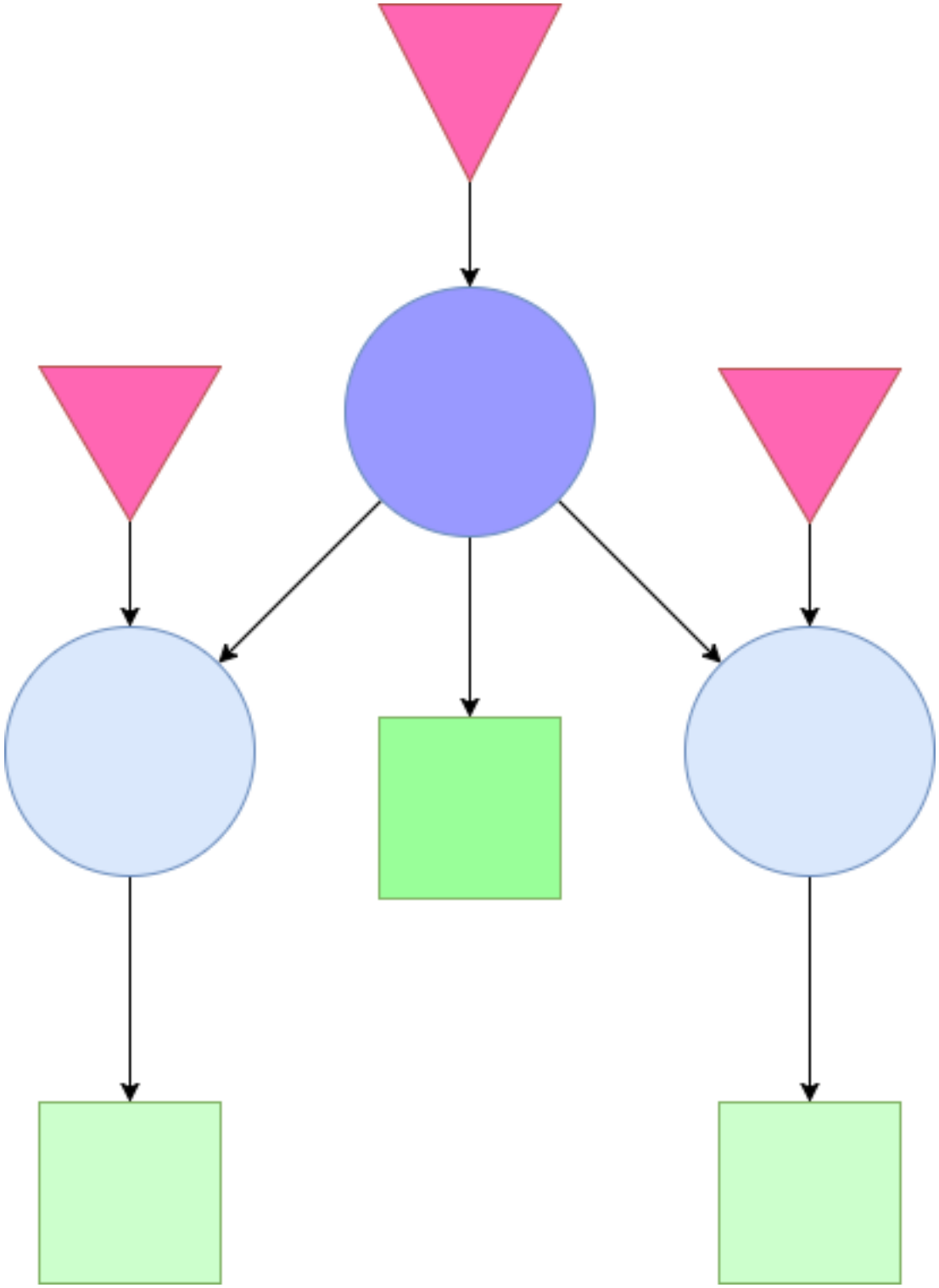
now let's run the app and see the result

LogCat
2019-10-12 13:10:58.5005605-5605/com.daggerudemy D/App: index = 1
2019-10-12 13:10:58.5005605-5605/com.daggerudemy D/App: index = 1

As you can see you can notice that the index has the same value which means dagger used the first instantiated object from logger and reuse it .

Subcomponent

Subcomponent is a type of component which dives from parent component and inherits its dependcies ,subcomponent can have one and only one parent component.



If you look carefully to the previous Diagram you will see the triangles are modules that provide dependencies and circles are componnets and squares places where they are injected.
The darker circle on the top is a parent component and lighter ones are the subcomponnets.

Subcomponent do the same like Component but there are some differences.To understand this let's take the following example,The idea is very simple we need to scale our app where we have login component that uses the objects in Appcomponent which is our base component so we can do this using component or subcomponent so let's start.

Scale using Component

We defined AppComponent before and it acts like the base component in our App ,Now we need to our login component to depend on AppComponent like the following:

```
1  package com.daggerudemy.di.component
2
3  import com.daggerudemy.MainActivity
4  import com.daggerudemy.di.module.LocalStoreModule
5  import com.daggerudemy.di.module.LoginServiceModule
6  import com.daggerudemy.di.scope.ActivityScope
7  import dagger.Component
8  @ActivityScope
9  @Component(dependencies =  [AppComponent::class], modules = [LocalStoreModule::class\
10  , LoginServiceModule::class])
11 interface LoginComponent {
12     fun inject(mainActivity: MainActivity)
13 }
```

you can see the login component depends on AppComponent and also there is new custom scope called **ActivityScope** which must be added because login component depends on scoped component which is AppComponent.

```
1  package com.daggerudemy.di.scope
2
3  import javax.inject.Scope
4
5  @Scope
6  @Retention
7  annotation class  ActivityScope
```

And you can use this in your MainActivity like the following

```
 package com.daggerudemy
import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import com.daggerudemy.di.Config
import com.daggerudemy.di.LoginManager
import com.daggerudemy.di.component.DaggerAppComponent
import com.daggerudemy.di.component.DaggerLoginComponent
import com.daggerudemy.di.module.LocalStoreModule
import javax.inject.Inject

class MainActivity : AppCompatActivity() {

    @Inject
    lateinit var loginManager: LoginManager

    @Inject
    lateinit var config: Config

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val appComponent = DaggerAppComponent.create()

        val loginComponent = DaggerLoginComponent.builder()
            .appComponent(appComponent)
            .localStoreModule(LocalStoreModule()).build()

        loginComponent.inject(this)

        loginManager.login("ramadan", "123")
        loginManager.enableCache(config)


    }
}
```

Scale using SubComponent

To do the same function using Subcomponent you need to use AppComponent directly and add refrence to your subcomponent ,now you can benefit from all objects in App module

```
package com.daggerudemy.di.component

import com.daggerudemy.di.AppLogger
import com.daggerudemy.di.module.AppModule
import com.daggerudemy.di.scope.AppScope
import dagger.Component

@AppScope
@Component(modules = [AppModule::class])
interface AppComponent {
    fun getAppLogger():AppLogger
    fun getLoginComponent():LoginComponent
}
```

```
package com.daggerudemy.di.component

import com.daggerudemy.MainActivity
import com.daggerudemy.di.module.LocalStoreModule
import com.daggerudemy.di.module.LoginServiceModule
import com.daggerudemy.di.scope.ActivityScope
import dagger.Component
import dagger.Subcomponent

@ActivityScope
@Subcomponent( modules = [LocalStoreModule::class , LoginServiceModule::class])
interface LoginComponent {
    fun inject(mainActivity: MainActivity)
}
```

```
package com.daggerudemy

import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import com.daggerudemy.di.Config
import com.daggerudemy.di.LoginManager
import com.daggerudemy.di.component.DaggerAppComponent
import javax.inject.Inject

class MainActivity : AppCompatActivity() {

    @Inject
```

```
13      lateinit var loginManager: LoginManager
14
15      @Inject
16      lateinit var config: Config
17
18      override fun onCreate(savedInstanceState: Bundle?) {
19          super.onCreate(savedInstanceState)
20          setContentView(R.layout.activity_main)
21
22          DaggerAppComponent.create().getLoginComponent().inject(this)
23
24          loginManager.login("ramadan", "123")
25          loginManager.enableCache(config)
26
27
28      }
29  }
```

finally, The main difference is the way of passing objects between the dependent modules. Using components gives a programmer more control over which objects may be used in the derivative components. Also important is the fact that we don't need to add new features to the base component each time, which fits very well with the idea of Open-Closed.

In the case of subcomponents, we immediately have access to all objects of the base component. In most cases, a better practice for our applications would be to use common components. This would make our code cleaner and more resistant to changes. We should use subcomponents in situations when both components are strongly related logically and when a derivative component uses most of the objects of the base component.

Conclusion:
In this chapter, we learned how to use the third way of dependency Injection (Method Injection) and we went deep in Dagger 2 Module and custom Scopes and how to use it.