Vamos injetar?

A primeira coisa que devemos fazer se queremos utilizar o Dagger é importar as duas dependências. Então, no arquivo **build.gradle** (Módulo), adicionamos:

dependencies {  
 ...  
 compile 'com.google.dagger:dagger-compiler:2.5'  
 apt 'com.google.dagger:dagger:2.5'  
 ...  
}

Aqui temos a API do Dagger (segunda linha) e a dependência que será responsável por processar tudo em tempo de compilação (primeira linha).

Só que para tudo isso funcionar, temos que colocar um plugin no projeto, o **android-apt**. Então, no arquivo **build.gradle** (Projeto), adicionamos:

dependencies {  
 ...  
 classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'  
 ...  
}

Como um plugin foi adicionado, precisamos dizer que queremos utilizá-lo, adicionando a seguinte linha no início do arquivo **build.gradle** (Módulo):

apply plugin: 'com.neenbedankt.android-apt'

Com tudo isso feito, clicamos em ***Sync Now*** para que as dependências sejam baixadas e configuradas.

## **Injeção com Dagger**

Para não ficarmos sempre dependentes de um objeto, precisamos fazer alguns passos, o primeiro deles e mais importante é deixar de se preocupar de onde está vindo o ChatService, para isso nós o injetaremos:

// MainActivity.java  
  
@Inject  
public ChatService chatService;

**OBSERVAÇÃO:** Para o Dagger conseguir fazer a injeção, o atributo não pode ser privado.

Agora precisamos falar como esse @Inject deve se comportar e para isso há uma classe especialista em fornecer objetos que serão injetados. Essas classes são conhecidas como **módulos** e para fazermos que nossa classe seja um módulo, basta utilizarmos a anotação **@Module**. Logo, vamos criar a classe **ChatModule** no pacote br.com.caelum.ichat.module, anotá-la e mover o método **getChatService()** para dentro dela:

@Module  
public class ChatModule {  
  
 public ChatService getChatService() {  
 Retrofit retrofit = new Retrofit.Builder()  
 .baseUrl("http://192.168.0.208:8080/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .build();  
  
 ChatService chatService = retrofit.create(ChatService.class);  
  
 return chatService;  
 }  
  
}

Agora nossa classe tem que saber como fornecer quem será injetado. Usamos a anotação @Provides para falar que nosso método irá fornecer o tipo esperado:

@Module  
public class ChatModule {  
  
 @Provides  
 public ChatService getChatService() {  
 Retrofit retrofit = new Retrofit.Builder()  
 .baseUrl("http://192.168.0.208:8080/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .build();  
  
 ChatService chatService = retrofit.create(ChatService.class);  
  
 return chatService;  
 }  
  
}

Mas ainda temos uma pequena parte de configuração que serve para deixarmos claro ao Dagger onde ele terá que trabalhar.

## **Onde o Dagger tem que trabalhar?**

Onde será que precisaremos do ChatService? Onde anotamos com @Inject, certo? Só que para evitar que o Dagger passe por todas as classes do nosso projeto à procura dessa anotação, nós podemos dar uma dica para ele. Podemos criar uma interface para definir onde e que objetos serão injetados. Ela será responsável por todos os módulos do projeto.

O Dagger chama esse tipo de classe de **componente**. Logo, vamos criar a interface **ChatComponent**no pacote br.com.caelum.ichat.component

@Component(modules=ChatModule.class)  
public interface ChatComponent {  
  
 void inject(MainActivity activity);  
  
}

Agora, da mesma forma que fizemos com o Retrofit, teremos que fazer com nosso Component seja implementado por alguma classe, para que de fato os objetos sejam injetados. O melhor disso tudo é que não temos que nos preocupar com quase nada! O Dagger é bem inteligente, em tempo de compilação ele gerará a classe para nós.

Para isso já temos a classe **ChatApplication**, que por enquanto está vazia. Então vamos criar o método **getComponent()**, que retornará uma instância de **ChatComponent**:

public class ChatApplication extends Application {  
  
 private ChatComponent component;  
  
 public ChatComponent getComponent() {  
 return component;  
 }  
}

Mas como instanciaremos o ChatComponent. Para isso, é bem simples, a classe que o Dagger gera para nós tem o comportamento de um *Builder*:

public class ChatApplication extends Application {  
  
 private ChatComponent component;  
  
 public ChatComponent getComponent() {  
 component = DaggerChatComponent.builder().build();  
 return component;  
 }  
}

Será que precisamos, toda vez que chamarmos o método getComponent(), instanciar um component novo? Não, podemos instanciar na hora da **criação da aplicação**, sobrescrevendo o método **onCreate()**:

public class ChatApplication extends Application {  
  
 private ChatComponent component;  
  
 @Override  
 public void onCreate() {  
 component = DaggerChatComponent.builder().build();  
 }  
  
 public ChatComponent getComponent() {  
 return component;  
 }  
}

E agora para fazermos as injeções, basta utilizar o componente que está dentro da nossa aplicação:

// MainActivity.java  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.activity\_main);  
  
 ChatApplication app = (ChatApplication) getApplication();  
 component = app.getComponent();  
 component.inject(this);  
  
 // restante do código omitido  
}

Desta forma deixamos nossos objetos complexos muito mais desacoplados, sem nos preocuparmos com dependências durante nosso código. Assim podem ser feitas algumas mudanças, entretanto não irão impactar nosso código.

Por último, para o **ChatApplication** ser instanciado, precisamos declará-lo dentro da tag **<application>**, no arquivo **manifests/AndroidManifest.xml**:

<application  
 android:name=".app.ChatApplication"  
 ...

Podemos testar agora, com a API rodando, e ver que o envio e o recebimento de mensagens está funcionando corretamente.