E as dependências?

*Começando deste ponto? Você pode fazer o* [*DOWNLOAD*](https://s3.amazonaws.com/caelum-online-public/android-chat/stages/capitulo-4.zip) *do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.*

No capítulo anterior, fizemos uso do **Retrofit** para fazer requisições à API do chat. E para fazer essas requisições, o **Retrofit** cria uma classe, em tempo de execução, abstraindo assim toda a infraestrutura e os detalhes necessários para criar uma requisição HTTP.

Basta informar à ele, a URL de base:

new Retrofit.Builder()  
 .baseUrl("http://192.168.0.208:8080/")

E informar também qual biblioteca queremos usar para fazer o *Marshal*/*Unmarshal* de JSON:

new Retrofit.Builder()  
 .baseUrl("http://192.168.0.208:8080/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .build();

Com isso configurado, podemos pedir para ele criar o serviço baseado em um contrato:

chatService = retrofit.create(ChatService.class);

Como precisamos executar esse código antes de começar a fazer as requisições, colocamos todo o código no método onCreate da nossa Activity. O que significa que teremos algo assim:

// MainActivity.java  
  
public void onCreate(Bundle savedInstanceState) {   
 // Restante omitido ...  
  
 Retrofit retrofit = new Retrofit.Builder()  
 .baseUrl("http://192.168.0.208:8080/")  
 .addConverterFactory(GsonConverterFactory.create())  
 .build();  
  
 chatService = retrofit.create(ChatService.class);  
  
 // Restante omitido ...  
  
}

## **Um serviço por aplicação**

Precisamos que o ChatService seja acessado de qualquer tela e queremos mantê-lo vivo enquanto durar a aplicação. Para isso, o Android possui uma classe que é instanciada quando iniciamos a aplicação e morre quando o aplicativo é finalizado. A classe se chama **Application**.

Logo, basta criarmos uma classe que herda dessa classe Application. Vamos criar a classe **ChatApplication** no pacote br.com.caelum.ichat.app:

public class ChatApplication extends Application {  
  
}

E podemos criar um método chamado **getChatService()**, para isolarmos a criação do serviço:

// ChatApplication.java  
  
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;  
}

E dessa forma, podemos agora usar o ChatService na MainActivity:

// MainActivity.java  
  
public void onCreate(Bundle savedInstanceState) {  
  
 // Restante omitido  
  
 ChatApplication app = (ChatApplication) getApplication();  
 this.chatService = app.getChatService();  
  
 // Restante omitido  
}

## **O problema do acoplamento**

Vamos pensar que estamos utilizando o nosso ChatService nessa tela e, a medida em que o aplicativo cresce, teremos outras telas onde gostaríamos de mostrar o chat e por consequência precisaremos usar o ChatService.

Para conseguirmos recuperar o ChatService, estamos sempre fazendo a seguinte linha:

ChatApplication app = (ChatApplication) getApplication();  
this.chatService = app.getChatService();

E se agora tivéssemos que passar algum parâmetro para dentro do método getChatService()? Teríamos que passar em cada classe que utiliza o serviço e atualizar o método, passando o parâmetro indicado. Toda vez em que realizarmos qualquer alteração nesse método, teremos que alterar todas as classes que a utilizam, para manter o projeto funcionando.

Esse é um problema bem comum, onde ficamos altamente presos a qualquer mudança (alto acoplamento). Para resolver esse problema, existem muitos frameworks que auxiliam no momento em que nos encontramos nessa situação, o Java possui o CDI e Spring, que fazem exatamente isso para nós, contudo para o utilizarmos no Android, seria algo muito pesado, já que ele faz um grande uso de Reflection. Contudo, a necessidade foi ficando maior e alguns frameworks surgiram para o mundo Android, o Dagger da Square, foi muito eficiente durante muito tempo, ele realizava o mesmo papel que o CDI e não utilizava tanto Reflection. O Google achou bem interessante e resolveu investir neste projeto, com isso foi anunciado o Dagger 2, que não conta com nenhum tipo de Reflection. Então é ele que utilizaremos no nosso projeto, o **Dagger 2**.