**Arquitetura desacoplada com eventos à moda antiga - LocalBradcastManager**

*Começando deste ponto? Você pode fazer o* [*DOWNLOAD*](https://s3.amazonaws.com/caelum-online-public/android-chat/stages/capitulo-7.zip) *do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.*

## **Acoplamento com a MainActivity e o Padrão Delegate**

Quando iniciamos o aplicativo, uma das primeiras coisas que ele faz é começar a ouvir as mensagens na API. Nesse momento, a requisição fica travada aguardando por mensanges, como já vimos no primeiro capítulo desse treinamento. Quando a resposta chega, entra em ação a classe OuvirMensagensCallback que é uma classe especialista em **delegar** à **MainActivity** a tarefa de colocar a mensagem na tela:

public class OuvirMensagensCallback implements Callback<Mensagem> {  
  
 private MainActivity activity;  
  
 public OuvirMensagensCallback(MainActivity activity) {  
 this.activity = activity;  
 }  
  
 @Override  
 public void onResponse(Call<Mensagem> call, Response<Mensagem> response) {  
 if(response.isSuccessful()) {  
 Mensagem mensagem = response.body();  
  
 activity.colocaNaLista(mensagem);  
 }  
  
 }  
  
 @Override  
 public void onFailure(Call<Mensagem> call, Throwable t) {  
 activity.ouvirMensagem();  
 }  
}

Podemos ver que quem de fato coloca a mensagem na tela é a MainActivity, através do método colocaNaLista(mensagem). Isso que estamos fazendo tem um nome bonito e se chama ***Delegation Pattern***.

O problema é que do jeito que implementamos, apenas a MainActivity poderá receber as mensagens usando esse *callback*. Se quisermos deixar nosso código mais flexível e permitir que outras telas também possam receber mensagens, precisamos diminuir esse acoplamento através de uma interface.

## **Trabalhando com eventos com LocalBroadcast**

Uma alternativa seria disparar um evento toda vez que recebemos uma mensagem e dessa forma, deixamos aberto pra qualquer componente do Android receber essa mensagem caso esteja interessado.

Se você fez o treinamento de Android da Alura, sabe que ao receber um SMS ou quando você liga seu Android, ele dispara um evento para todo **BroadcastReceiver**que esteja interessado possa tratar.

No nosso caso, não queremos expor nossas mensagens para todo o sistema operacional, mas apenas para os ***receivers*** da nossa aplicação. Para isso, vamos usar o **LocalBroadcastManager**.

Na classe OuvirMensagensCallback vamos disparar o evento:

// OuvirMensagensCallback.java  
  
@Override  
public void onResponse(Call<Mensagem> call, Response<Mensagem> response) {  
 if(response.isSuccessful()) {  
 Mensagem mensagem = response.body();  
  
 Intent intent = new Intent("nova\_mensagem");  
 intent.putExtra("mensagem", mensagem);  
  
 // O método 'getInstance' recebe um Context !!  
 LocalBroadcastManager localBroadcastManager = LocalBroadcastManager.getInstance(activity);  
  
 localBroadcastManager.sendBroadcast(intent);  
 }  
  
}

Como o método **LocalBroadcastManager.getInstance(..)** recebe um **Context**, podemos generalizar a nossa classe, já que agora não precisamos necessariamente receber uma **Activity**, e sim um **Context**. A classe ficará assim:

public class OuvirMensagensCallback implements Callback<Mensagem> {  
  
 private Context context;  
  
 public OuvirMensagensCallback(Context context) {  
 this.context = context;  
 }  
  
 @Override  
 public void onResponse(Call<Mensagem> call, Response<Mensagem> response) {  
 if(response.isSuccessful()) {  
 Mensagem mensagem = response.body();  
  
 Intent intent = new Intent("nova\_mensagem");  
 intent.putExtra("mensagem", mensagem);  
  
 // O método 'getInstance' recebe um Context !!  
 LocalBroadcastManager localBroadcastManager = LocalBroadcastManager.getInstance(context);  
  
 localBroadcastManager.sendBroadcast(intent);  
 }  
  
 }  
  
 @Override  
 public void onFailure(Call<Mensagem> call, Throwable t) {  
  
 }  
}

E para o **intent.putExtra(..)** funcionar, nossa classe **Mensagem** deve implementar **Serializable**:

public class Mensagem implements Serializable {

E na MainActivity, precisamos criar o **BroadcastReceiver** que irá tratar o evento. Para isso, vamos criar um novo atributo:

// MainActivity.java   
  
private BroadcastReceiver receiver = new BroadcastReceiver() {  
 @Override  
 public void onReceive(Context context, Intent intent) {  
 Mensagem mensagem = (Mensagem) intent.getSerializableExtra("mensagem");  
  
 colocaNaLista(mensagem);  
  
 }  
};

Porém apenas o callback não é o suficiente, precisamos registrá-lo ao iniciar a aplicação:

// MainActivity.java  
  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.chat\_activity);  
  
 ButterKnife.bind(this);  
  
 picasso.with(this).load("https://api.adorable.io/avatars/285/" + idDoCliente + ".png").into(avatar);  
  
 ChatApplication app = (ChatApplication) getApplication();  
 component = app.getComponent();  
 component.inject(this);  
  
 mensagens = new ArrayList<>();  
  
 MensagemAdapter adapter = new MensagemAdapter(idDoCliente, mensagens, this);  
 listaDeMensagens.setAdapter(adapter);  
  
 ouvirMensagem();  
  
 LocalBroadcastManager localBroadcastManager = LocalBroadcastManager.getInstance(this);  
 localBroadcastManager.registerReceiver(receiver, new IntentFilter("nova\_mensagem"));  
  
}

Agora imagina que o usuário vire a tela do dispositivo, o que acontece com a MainActivity quando ele faz isso? Ela é recriada! O problema é que quando ele chamar o onCreate da MainActivity, ele fará de novo o registro e teremos dois observadores para o mesmo evento. Para solucionar isso, vamos desregistrar o nosso receiver no método **onStop()**. Então vamos sobrescrevê-lo:

@Override  
protected void onStop() {  
 super.onStop();  
  
 LocalBroadcastManager localBroadcastManager = LocalBroadcastManager.getInstance(this);  
 localBroadcastManager.unregisterReceiver(receiver);  
}

Ao rodar a aplicação, tudo continua funcionando.

Fizemos isso para receber apenas um evento, agora imagine numa aplicação um pouco maior, onde teremos diversas consultas ao servidor, quantas classes teríamos que ter? A manutenção, ainda que resolvesse diversos problemas, seria um pouco grande e trabalhosa.

Já entendemos a vantagem de possuir o **BroadcastReceiver** para disparar os eventos para gente, mas a manutenção a longo prazo pode ser ruim, por esse motivo virão que há uma forma de simplificar isso, deixar esse comportamento encapsulado apenas para utilizarmos, todo esse conceito ganhou o nome de **Eventbus**, e é ele que veremos a partir do próximo vídeo :)