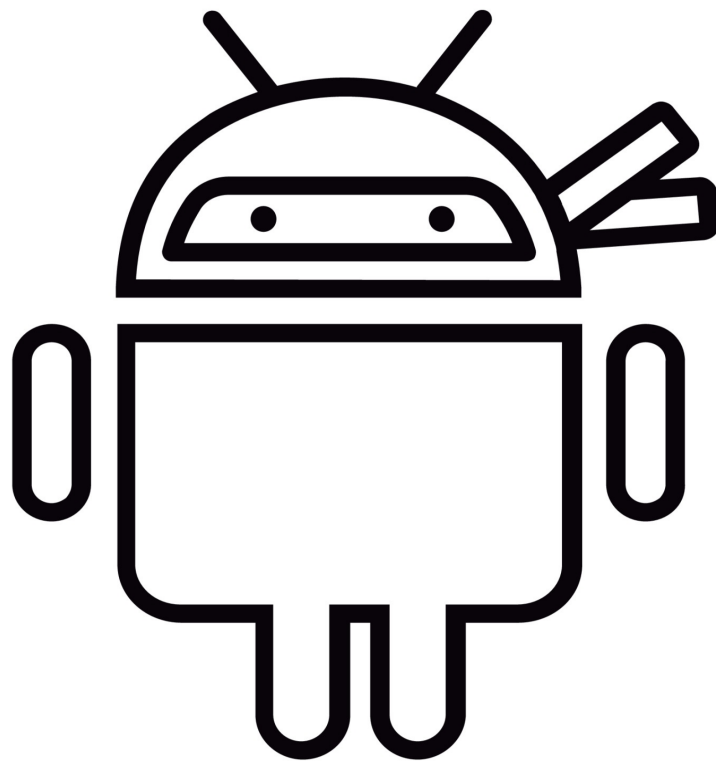


Técnicas de Desenvolvimento Android Avançado

Curso FJ-59



Sumário

1 Projeto Casa do Código	1
1.1 Listas com RecyclerView e o Padrão ViewHolder	1
1.2 Exercícios	17
1.3 Melhorando nosso código com ButterKnife	15
1.4 Exercícios	17
2 Detalhes do Livro	20
2.1 Desacoplando nosso código com o Pattern Delegate	24
2.2 Exercícios - Utilizando o Pattern Delegate	25
3 Dados Reais De um WebService	30
3.1 Fazendo a busca do Servidor	30
3.2 Melhorando nossas Requisições com Retrofit	34
3.3 Exercícios	53
3.4 Deixando a requisição otimizada com Broadcast Receiver	45
3.5 Facilitando o código com EventBus	51
3.6 Exercícios	53
3.7 Exercícios - Carregando imagens com Picasso	56
4 Trazendo Apenas as Informações que Queremos	58
4.1 Ensinando o Retrofit a Fazer Requisições Dinâmicas	58
4.2 Aplicando o Padrão EndlessList	59
4.3 Exercícios	62
5 Vendendo os livros	68
5.1 Exercícios	78
5.2 Desacoplando os objetos com Dagger	73
6 Testes Automatizados com Espresso	82
6.1 Exercícios	83
7 Utilizando Firebase	84

7.1 Firebase Authentication	84
7.2 Firebase Remote Config	95
7.3 Firebase Notification	101
7.4 Exercicios	111
7.5 Tratando Notificações Dentro Da Aplicação	104
7.6 Recebendo notificações do nosso servidor	107
8 Utilizando Banco de Dados	113
8.1 Colocando forma de pagamento	113
8.2 Exercícios	122

Versão: 21.2.6

PROJETO CASA DO CÓDIGO

Quando começamos a aprender a desenvolver para Android, costumamos aprender os conceitos mais básicos e logo criamos um aplicativo simples para testar os conhecimentos. A partir daí fica fácil testar novos recursos e expandir esse aplicativo com novas funcionalidades. Só que quando estamos iniciando ainda não temos aquela preocupação com boas práticas e também acabamos reinventando diversas soluções para problemas comuns. Por isso, nesse curso veremos algumas boas práticas de desenvolvimento de aplicativos para Android e as principais bibliotecas utilizadas hoje no mercado, tudo isso de forma prática e aplicada no desenvolvimento de um aplicativo do zero.

1.1 LISTAS COM RECYCLERVIEW E O PADRÃO VIEWHOLDER

Para começarmos nosso aprendizado, vamos supor que fomos contratados pela **Casa do Código** para desenvolver um aplicativo para os seus clientes onde eles possam consultar o catálogo de livros e realizar compras.

Dado esse cenário podemos imaginar que uma das telas mais importantes do nosso aplicativo seja a lista de livros disponíveis. Poderíamos começar montando essa tela mas ainda não teríamos como populá-la pois ainda não temos as classes necessárias para representar os livros! Vamos então começar definindo nossa classe mais importante do modelo que vai representar um `Livro` :

```
public class Livro {  
  
    private long id;  
    private String nome;  
    private String descricao;  
    private int numPaginas;  
    private String dataPublicacao;  
    private String ISBN;  
    private double valorFisico;  
    private double valorVirtual;  
    private double valorDoisJuntos;  
    private String urlFoto;  
    private List<Autor> autores;  
  
    // getters e setters  
  
}
```

Como temos uma lista de autores no livro, seria bom criarmos também uma classe para representar um `Autor` :

```

public class Autor {

    private long id;
    private String nome;
    private String biografia;
    private String urlFoto;

    // getters e setters

}

```

Agora que já conhecemos um pouco mais dos modelos, podemos começar a desenvolver o aplicativo em si. Como dissemos antes, seria legal exibir logo de início a listagem de todos os livros disponíveis utilizando, por exemplo, uma `ListView`. Vamos começar pelo *layout*:

```

<ListView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/lista_livros"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

```

Depois criamos uma `ListaLivrosActivity` usando o *layout* que acabamos de criar. Também já aproveitamos para preparar o `Adapter` que usaremos para popular a lista:

```

public class ListaLivrosActivity extends AppCompatActivity {

    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.lista);

        ListView lista = ( ListView ) findViewById(R.id.lista_livros);

        List<Livro> livros = // recupera listagem

        lista.setAdapter(new LivrosAdapter(livros));
    }

}

```

Para que o código acima funcione ainda precisamos implementar a classe `LivrosAdapter` usando como base a classe `BaseAdapter` do próprio Android:

```

public class LivrosAdapter extends BaseAdapter {

    private List<Livro> livros;

    public LivrosAdapter (List<Livro> livros) {
        this.livros = livros;
    }

    public int getCount() {
        return livros.size() ;
    }

    public Object getItem(int posicao) {
        return livros.get(posicao);
    }
}

```

```

    }

    public long getItemId(int posicao){
        return livros.get(posicao).getId();
    }

    public View getView(int posicao, View convertView, ViewGroup parent) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.livro_item, parent, false);

        // recupera o livro da posição
        // popula a view

        return view;
    }
}

```

Agora nossa *Activity* deve estar compilando sem nenhum problema. Mas o que será que aconteceria se nossa lista tivesse muitos livros? Dependendo da versão do Android e do dispositivo usado, daria pra perceber que ele apresenta um pouco de **lentidão** e alguns **travamentos** quando estamos rolando a lista. Por que será que isso acontece?

Quando executamos nosso aplicativo, para a lista ser exibida, ela passa pelo *adapter*, que irá converter cada item da nossa `List<Livro>` numa *View* que será exibida na tela. Para fazer esse procedimento, a lista irá perguntar inicialmente ao *adapter* quantos itens ele possui usando o método `getCount()` para que ela consiga se planejar para inserir cada item na tela.

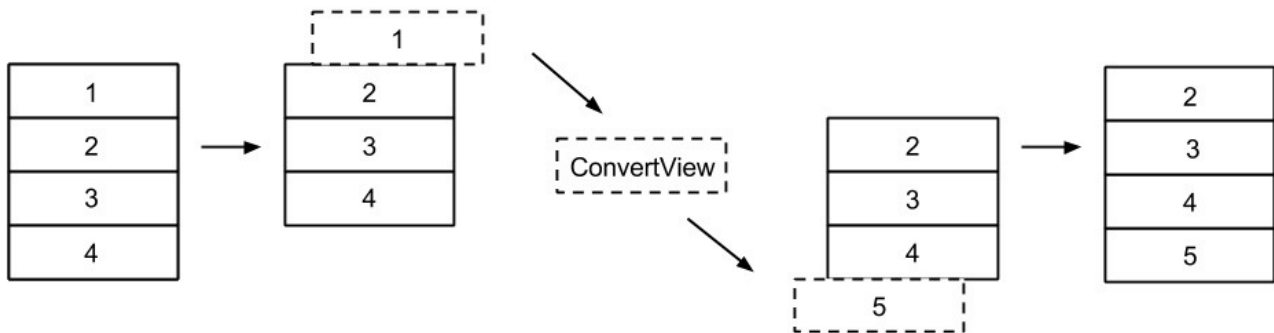
No próximo passo, ela vai fazer a conversão dos livros em *views* utilizando o método `getView()`. Para isso a `ListView` irá perguntar ao *adapter* qual é o *view* correspondente a cada posição, por esse motivo o método `getView()` recebe um inteiro como primeiro parâmetro.

O segundo parâmetro do `getView()` é um `ViewGroup` que representa a *view* que irá guardar os itens da lista. Esse `ViewGroup` geralmente é utilizado ao inflar a *view* de cada item da lista e é a *view* que será considerada para os itens quando eles fizerem alguma referência a um `match_parent` em seus *layouts*, por exemplo.

Mas o que será que representa a *View* que recebemos como terceiro parâmetro do `getView()` ?

Esse parâmetro tem muito a ver com o problema de lentidão na rolagem que comentamos. Vamos supor que a nossa lista possui 10000 livros. Quando a `ListView` precisar ser mostrada na tela ela vai conversar com o *Adapter* e criar (inflar) as *views* correspondentes aos livros. Mas quantas *views* seriam suficientes para montar a tela inicial do aplicativo? Como a tela tem um espaço limitado, não precisaríamos e nem conseguiríamos mostrar os 10000 livros, uns 7 ou 8 já seriam suficientes nesse momento inicial. E quando precisaríamos criar as *views* dos demais livros? Somente quando rolamos a lista e precisamos mostrar esses outros livros! E o que acontece com os livros que foram rolados para fora da tela? A princípio não precisamos mantê-los na memória, pode ser que o usuário nem volte a vê-los. Nesse caso, a `ListView` tem um comportamento bem bacana que é reutilizar essas *views* que foram

roladas para fora da tela para que a lista não precise ficar inflando novos itens toda vez que a lista é rolada. Então, o terceiro parâmetro do `getView()` é justamente uma *view* que já foi inflada e que agora não está mais visível e que pode ser reaproveitada para exibir um novo livro, por exemplo.



Para utilizar esse recursos de reaproveitamento das *views* na `ListView`, só precisamos fazer apenas uma pequena alteração no código do nosso *adapter*:

```
public class LivrosAdapter extends BaseAdapter {

    // código anterior

    public View getView(int posicao, View convertView, ViewGroup parent) {

        View view;

        // Não tem view para reaproveitar, tem que inflar uma nova
        if (convertView == null) {

            view = LayoutInflater.from(parent.getContext()).inflate(R.layout.livro_item, parent, false);

        } else { // Opa! Aqui tem view para reaproveitar!

            view = convertView ;

        }

        // recupera o livro da posição
        // popula a view

        return view;

    }

}
```

Perceba que precisamos verificar se a *convertView* está ou não **nula**, pois inicialmente a lista ainda não possui uma *View* pronta para reaproveitar e por esse motivo nós inflamos o suficiente para exibir na tela e quando ela já possuir todas prontas, ela apenas nos fornece aquela que pode ser reaproveitada. Dessa forma, a rolagem da nossa listagem ficará bem mais rápida e muito mais agradável para o usuário.

Demos uma boa melhoria em nosso código, que resultou numa ótima performance para o aplicativo, contudo nosso código ainda continua fazendo muito trabalho, vamos dar uma olhada no

código :

```
public class LivrosAdapter extends BaseAdapter {

    public View getView(int posicao, View convertView, ViewGroup parent) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.livro_item, parent, false);

        if (convertView == null) {

            view = LayoutInflater.from(parent.getContext()).inflate(R.layout.livro_item, parent, false);

        } else {

            view = convertView ;

        }

        // recupera o livro da posição

        ImageView foto = (ImageView) view.findViewById(R.id.foto);
        TextView nome = (TextView) view.findViewById(R.id.nome);
        TextView descricao = (TextView) view.findViewById(R.id.descricao);
        TextView autor = (TextView) view.findViewById(R.id.autor);
        Button comprar = (Button) view.findViewById(R.id.comprar);

        // popula a view com livro

        return view;
    }
}
```

Repare que para cada item, estamos indo, **em tempo de execução**, no layout e buscando a `View` pelo `id`. Esse procedimento é bem custoso, pois ele busca as `views` do `layout` do item hierarquicamente do pai para cada filho. Inicialmente, para as primeiras `Views` que serão criadas faz sentido irmos buscar no layout os seus respectivos componentes. Entretanto, a partir do momento que estamos utilizando uma `View` que já foi criada e agora está sendo reaproveitada, não faz sentido irmos novamente ao layout buscar, pois já fizemos isso previamente. Seria interessante reaproveitarmos, além da própria `View`, as referências para as `views` que já foram buscadas anteriormente.

Uma forma de resolver esse problema é criar uma classe onde você armazene as `views` buscadas anteriormente e guardar uma instância dessa classe dentro da própria `View` do item da lista e no momento que ele é reaproveitada, recuperar o objeto com as `views` para usufruir dele no momento de popular os componentes:

```
public class LivrosAdapter extends BaseAdapter {

    // outros métodos

    public View getView(int posicao, View convertView, ViewGroup parent) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.livro_item, parent, false);
    }
}
```



```

// Cria um objeto para guardar as views
Objeto objeto;

if (convertView == null) {

    view = LayoutInflater.from(parent.getContext()).inflate(R.layout.livro_item, parent, false
);

    // Aqui passamos a view para o Objeto buscar e guardar as views
    objeto = new Objeto(view);

    // E aqui guardamos o objeto com as views no item
    view.setTag(objeto);
} else {

    view = convertView;

    // Reaproveitando uma view! Então pega o objeto do próprio item
    objeto = (Objeto) view.getTag();

}

// recupera o livro da posição

// Perceba que aqui só usamos as views do objeto
ImageView foto = objeto.foto;
TextView nome = objeto.nome;
TextView descricao = objeto.descricao;
TextView autor = objeto.autor;
Button comprar = objeto.comprar;

// popula a view com livro

return view;
}

// Essa classe é o objeto que vai guardar as views
class Objeto {

    ImageView foto;
    TextView nome;
    TextView descricao;
    TextView autor;
    Button comprar;

    // Fazemos todos os findViewById no construtor
    public Objeto(View view) {
        foto = (ImageView) view.findViewById(R.id.foto);
        nome = (TextView) view.findViewById(R.id.nome);
        descricao = (TextView) view.findViewById(R.id.descricao);
        autor = (TextView) view.findViewById(R.id.autor);
        comprar = (Button) view.findViewById(R.id.comprar);
    }

}
}

```

Essa forma de resolver este problema é uma solução bem comum que se tornou um *design pattern* conhecido como **ViewHolder**. Agora podemos assegurar que nossa lista está bem estruturada,

reaproveitando a *convertView* e também a busca dos componentes utilizando o *design pattern ViewHolder*.

Resolvemos esse problema de maneira elegante e padronizada, mas e se precisarmos criar uma nova lista em uma nova tela do nosso aplicativo? Vamos ter que lembrar e aplicar todos os conceitos que acabamos de ver! Eventualmente podemos nos esquecer de aplicar e ver nossa lista com lentidão novamente. Pra evitar isso, a Google disponibilizou a *RecyclerView*, um outro tipo de *view* para lista onde este comportamento já está totalmente **encapsulado**. Vamos ver o que muda ao usar o *RecyclerView*.

A primeira coisa a fazer é incluir a dependência da biblioteca em nosso projeto. Para isso, basta adicionar a seguinte linha no arquivo *build.gradle* do projeto:

```
compile 'com.android.support:recyclerview-v7:XX.XX.XX'
```

Agora vamos modificar o *layout* da nossa *Activity*:

```
<android.support.v7.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/lista_livros"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Precisamos passar o nome completo da classe da *view* pois este componente é uma dependência externa. Agora precisamos alterar o código da *Activity* para também se referenciar ao *RecyclerView*:

```
public class ListaLivrosActivity extends AppCompatActivity {

    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.lista);

        RecyclerView lista = ( RecyclerView ) findViewById(R.id.lista_livros);

        List<Livro> livros = // recupera listagem

        lista.setAdapter(new LivrosAdapter(livros));
    }
}
```

Agora vamos precisar mudar o tipo de *Adapter* que estamos utilizando pois o *RecyclerView* trabalha com um *RecyclerView.Adapter*. Nesse caso, vamos criar um novo *_adapter* do zero:

```
public class LivrosAdapter extends RecyclerView.Adapter {

}
```

Da mesma forma que éramos obrigados a sobreescrever alguns métodos da classe *BaseAdapter*, quando usamos *RecyclerView.Adapter* também será necessário implementarmos alguns métodos:

```
public class LivrosAdapter extends RecyclerView.Adapter {
```

```

public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
}

public void onBindViewHolder(RecyclerView.ViewHolder viewHolder, int position) {
}

public int getItemCount() {
}
}

```

Da mesma forma que o `ListView`, precisamos deixar o *adapter* informado sobre a quantidade de itens que irão para tela implementando o método `getItemCount()`. Vamos precisar da lista de livros no nosso *adapter* então vamos passá-la como parâmetro no construtor:

```

public class LivrosAdapter extends RecyclerView.Adapter {

    private List<Livro> lista;

    public LivrosAdapter(List<Livro> lista) {
        this.lista = lista;
    }

    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    }

    public void onBindViewHolder(RecyclerView.ViewHolder viewHolder, int position) {
    }

    public int getItemCount() {
        return lista.size();
    }
}

```

Agora precisamos criar a `View` que será exibida e reaproveitada. Para isso, temos o método `onCreateViewHolder()` que será responsável por instanciar uma nova *view* e armazená-la dentro de um `ViewHolder`. Repare que desta vez já somos obrigados a implementar este *design pattern*. Vamos iniciar criando a *view*:

```

public class LivrosAdapter extends RecyclerView.Adapter {

    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.livro_item, parent, false);

        return new ViewHolder(view);
    }

    // demais métodos
}

```

```
}
```

Nesse instante estamos com um problema de compilação no retorno do método, pois a classe `ViewHolder` ainda não existe. Para solucionar isto, vamos criar essa classe:

```
public class LivrosAdapter extends RecyclerView.Adapter {  
  
    // resto do código  
  
    class ViewHolder extends RecyclerView.ViewHolder {  
  
        public ViewHolder(View view) {  
            super(view);  
        }  
  
    }  
  
}
```

Agora precisamos apenas colocar o comportamento que o *design pattern* `ViewHolder` precisa, que será armazenar a busca para cada *view* do item. Como a biblioteca do `RecyclerView` já está **encapsulando** alguns comportamentos, não precisaremos vincular o nosso `ViewHolder` com a *view* criada, pois ele já faz isto por nós:

```
public class LivrosAdapter extends RecyclerView.Adapter {  
  
    // resto do código  
  
    class ViewHolder extends RecyclerView.ViewHolder {  
  
        ImageView foto;  
        TextView nome;  
        TextView descricao;  
        TextView autor;  
        Button comprar;  
  
        public ViewHolder(View view) {  
            super(view);  
  
            foto = (ImageView) view.findViewById(R.id.foto);  
            nome = (TextView) view.findViewById(R.id.nome);  
            descricao = (TextView) view.findViewById(R.id.descricao);  
            autor = (TextView) view.findViewById(R.id.autor);  
            comprar = (Button) view.findViewById(R.id.comprar);  
        }  
  
    }  
  
}
```

Criamos a *view* do item e o vinculamos com o `ViewHolder` mas ainda não populamos nenhum dos seus componentes. Para realizarmos esse procedimento, temos mais o método `onBindViewHolder()`. Esse método é invocado automaticamente para popular cada item da lista e recebe como parâmetro o `ViewHolder` e também a posição do item. Com esses parâmetros conseguimos popular nossos itens:

```
public class LivrosAdapter extends RecyclerView.Adapter {  
  
    // resto do código
```

```

public void onBindViewHolder(RecyclerView.ViewHolder viewHolder, int position) {

    ViewHolder holder = (ViewHolder) viewHolder;

    Livro livro = lista.get(position);

    holder.nome.setText(livro.getNome());

    // demais componentes

}
}

```

Como recebemos um `ViewHolder` genérico, precisamos fazer um *cast* para o nosso `ViewHolder` específico pois é ele que conhece corretamente todos os componentes do nosso item.

Com o *adapter* pronto, precisamos agora nos atentar a um outro detalhe que não existia na `ListView`. No Android, possuímos componentes diferentes para representar as diferentes formas de exibir os itens de uma lista. Podemos exibir itens em grade, como no **Instagram**, utilizando uma `GridView` ou podemos exibir numa lista comum, como a lista de conversas do **WhatsApp** ou **Telegram**, utilizando uma `ListView`. O que aconteceria se utilizássemos o `GridView` e agora precisássemos migrar para uma `ListView`? Nesse caso, precisaríamos fazer mudanças nos *layouts* e também nas classes que faziam referência ao `GridView`.

O `RecyclerView` além de resolver todos os problemas que citamos anteriormente, também veio para solucionar o problema de trocar a forma de exibição, alterando apenas uma linha de código. Para informarmos ao `RecyclerView` qual é o tipo de exibição que queremos, usamos o método `setLayoutManager()` que irá receber um objeto do tipo `LayoutManager`, que é uma **interface**. Temos 3 implementações possíveis:

- `LinearLayoutManager` - posiciona cada item linearmente, sendo horizontalmente ou verticalmente. **Ex.: WhatsApp**
- `GridLayoutManager` - posiciona cada item em um grid. **Ex.: Instagram**
- `StaggeredGridLayoutManager` - posiciona cada item de forma escalonada dentro de um grip. **Ex.: Keep (Google)**

Como queremos apresentar uma lista de livros, vamos preferir a forma mais tradicional de lista por isso usaremos o `LinearLayoutManager`.

```

public class ListaLivrosActivity extends AppCompatActivity {

    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.lista);

        RecyclerView lista = ( RecyclerView ) findViewById(R.id.lista_livros);

        List<Livro> livros = // recupera listagem

```

```

        lista.setAdapter(new LivrosAdapter(livros));

        LayoutManager layoutManager = new LinearLayoutManager(this);

        lista.setLayoutManager(layoutManager);
    }
}

```

1.2 EXERCÍCIOS

1. Siga as instruções do instrutor para importar o projeto inicial do curso.
2. Com o projeto pronto podemos começar a desenvolver algumas de suas funcionalidades, a primeira será implementar a listagem de livros. Vamos criar o *layout*, que terá apenas o `RecyclerView`, chame-o de *fragment_lista_livros.xml*:

```

<android.support.v7.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/lista_livros"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

```

3. Com o *layout* criado vamos referenciar ele à uma tela, para isso crie um `Fragment` e o chame de **ListaLivrosFragment** e coloque no pacote `br.com.caelum.casadocodigo.fragment`:

```

public class ListaLivrosFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
        Bundle savedInstanceState) {

        View view = inflater.inflate(R.layout.fragment_lista_livros, container, false);

        return view;
    }
}

```

4. Agora precisamos recuperar o `RecyclerView` para podermos manipulá-lo na tela:

```

public class ListaLivroFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
        Bundle savedInstanceState) {

        View view = inflater.inflate(R.layout.lista_livros_fragment, container, false);

        List<Livro> livros = new ArrayList<>();
        for (int i = 0; i < 6; i++) {
            Autor autor = new Autor();
            autor.setNome("Autor " + i);
            Livro livro =

```

```

        new Livro("Livro " + i, "Descricao " + i, Arrays.asList(autor));
        livros.add(livro);
    }

    RecyclerView recyclerView = (RecyclerView) view.findViewById(R.id.lista_livros);
    recyclerView.setAdapter(new LivroAdapter(livros));

    return view;
}
}

```

5. Agora que já temos uma listagem, ainda que **hardcoded**, conseguimos exibir no emulador, precisaremos apenas adaptar nosso objeto do tipo `Livro` para uma `View`, para isso vamos criar uma classe especialista nessa conversão. Crie a classe `LivroAdapter` no pacote `br.com.caelum.casadocodigo.adapter`:

```

public class LivroAdapter {

}

```

Para que nossa classe seja um *adapter* precisamos herdar de `RecyclerView.Adapter`:

```

public class LivroAdapter extends RecyclerView.Adapter {

}

```

No momento em que herdamos, o *Android Studio* nos alerta que precisamos sobrescrever alguns métodos. Use o atalho `Alt + Enter` e faça com que ele implemente os métodos que precisamos:

```

public class LivroAdapter extends RecyclerView.Adapter {

    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        return null;
    }

    @Override
    public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {

    }

    @Override
    public int getItemCount() {
        return 0;
    }

}

```

Em nossa `ListaLivroFragment` ainda estamos com um problema de compilação. Utilizando o atalho `F2` para ir para linha que está com erro e para corrigir, utilize novamente o atalho `Alt + Enter` para criar um construtor que receba uma lista de livros. Guarde essa lista que recebemos por parâmetro em um atributo:

```
public class LivroAdapter extends RecyclerView.Adapter {

    private List<Livro> livros;

    public LivroAdapter (List<Livro> livros) {
        this.livros = livros;
    }

    // resto do código

}
```

Agora que temos a referência da lista conseguimos terminar de implementar nosso `Adapter`. Começaremos com o método `getItemCount()` que retorna o **tamanho total de nossa lista**.

Como nossa lista será de itens intercalados, precisamos alertar o `Adapter` para que ele saiba como deve se portar, para isso iremos sobrescrever mais um método que será responsável por deixar o `Adapter` ciente.

```
public class LivroAdapter extends RecyclerView.Adapter {

    // resto do código

    @Override
    public int getItemCount() {
        return livros.size();
    }

    @Override
    public int getItemViewType(int position) {
        return position % 2;
    }

}
```

Precisamos inflar as `Views` agora, para isso faremos uso do método `onCreateViewHolder()` :

```
public class LivroAdapter extends RecyclerView.Adapter {

    // resto do código

    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        int tipoDeLayout = R.layout.item_livro_par;
        if (viewType % 2 != 0) {
            tipoDeLayout = R.layout.item_livro_impar;
        }
        View view = LayoutInflater.from(parent.getContext()).inflate(tipoDeLayout, parent, false);
        ;
        return new ViewHolder(view);
    }

}
```

Agora é necessário criarmos ambos os layouts, para isso usaremos mais uma biblioteca o `CardView` para isso vá ao gradle e adicione a linha :

compile 'com.android.support:cardview-v7:24.2.1'

Estamos retornando `null` em um dos métodos. Para resolvermos esse problema vamos implementar o retorno correto, que é um `ViewHolder`. Para isso, iremos criar uma classe que represente o **nosso** `ViewHolder` e que já faça as buscas das *Views* do item da lista:

```
public class LivroAdapter extends RecyclerView.Adapter {

    // resto do código

    class ViewHolder extends RecyclerView.ViewHolder {
        TextView nome;
        ImageView foto;

        public ViewHolder(View view) {
            super(view);

            nome = (TextView) view.findViewById(R.id.item_livro_nome);
            foto = (ImageView) view.findViewById(R.id.item_livro_foto);
        }
    }
}
```

Precisamos apenas popular os itens agora. Para isso, implementaremos o método `onBindViewHolder()` no *adapter*:

```
@Override
public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {
    ViewHolder viewHolder = (ViewHolder) holder;
    Livro livro = livros.get(position);
    viewHolder.nome.setText(livro.getNome());
}
```

6. Nosso `RecyclerView`, além do *adapter* também precisa saber qual é a maneira que ele deve ser exibido na tela, qual vai ser o padrão adotado, o deixaremos como uma lista vertical:

```
public class ListaLivroFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
        @Nullable Bundle savedInstanceState) {

        // mantenha o código anterior e adicione a linha abaixo

        recyclerView.setLayoutManager(new LinearLayoutManager(getContext()));

        return view;
    }
}
```

7. Precisamos referenciar nosso `Fragment` em nossa `Activity`, para isso abra a classe `MainActivity` que está no pacote `br.com.caelum.casadocodigo.activity` e faça a troca do

FrameLayout pelo nosso Fragment :

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
        transaction.replace(R.id.frame_principal, new ListaLivrosFragment());
        transaction.commit();
    }
}
```

Rode o aplicativo e veja o resultado !

1.3 MELHORANDO NOSSO CÓDIGO COM BUTTERKNIFE

Nosso código está funcionando e está bem encapsulado, contudo na medida em que formos desenvolvendo telas mais complexas, por exemplo os detalhes do livro, onde temos que exibir todas as informações que o livro possui, nosso código não terá uma legibilidade tão boa, já que ficará repleto de `findViewById` e várias classes anônimas para cada `Listener` .

Esse problema de legibilidade é algo que vem incomodando bastante os desenvolvedores porque atrapalha a legibilidade e manutenção do código. Por esse motivo, há várias bibliotecas que tentam sanar isto, por exemplo **DataBinding**, **ButterKnife**, **RoboGuice**, entre outras. Usaremos em nosso aplicativo o **ButterKnife**, que trará alguns recursos bem interessantes além da facilidade de uso.

Vamos resolver o primeiro problema que citamos que foi a quantidade de `findViewById` que estamos fazendo em nosso código. Para isso, só precisaremos indicar para o 'Butterknife' quais os `_ids_` das `_views_` que queremos buscar e quais são os atributos onde essas `_views_` ficarão armazenadas. Com o Butterknife, temos uma forma fácil de fazer isso que é pelo uso de *Annotations*.

Como queremos *injetar* uma `View` , usaremos a *annotation* `@BindView` com o `id` da `view` que queremos buscar como parâmetro:

```
public class UmaActivity extends AppCompatActivity {

    @BindView(R.id.txt)
    TextView txt;

    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.uma_activity);

        txt.setText("Recuperada pelo ButterKnife");
    }
}
```

```
}  
  
}
```

Está tudo muito bom, mas no momento que mandamos executar esse código, levamos uma `Exception` desagradável, conhecida como `NullPointerException`, na linha que estamos setando o texto em nossa `View`. Isso ocorre pelo fato de não termos especificado para o **ButterKnife** onde ele deve fazer a busca daquele componente. Para fazer isso, basta invocarmos o método `bind()` da classe `ButterKnife` passando a referência da `Activity` como parâmetro:

```
public class UmaActivity extends AppCompatActivity {  
  
    @BindView(R.id.txt)  
    TextView txt;  
  
    public void onCreate(Bundle bundle) {  
        super.onCreate(bundle);  
        setContentView(R.layout.uma_activity);  
  
        ButterKnife.bind(this);  
  
        txt.setText("Recuperada pelo ButterKnife");  
    }  
}
```

Devemos invocar o método `bind()` sempre depois de termos vinculado o xml com a `Activity` ou `Fragment`, já que no momento que o método `bind()` é chamado, ele fará a busca dos componentes anotados. Caso não haja um layout vinculado, ele atribuirá as referências como nulas, que em tempo de execução levaria nosso aplicativo a quebrar quando formos manipular tais componentes.

Quando estamos manipulando componentes dentro de um `Fragment` precisamos tomar um pouco mais de cuidado, pois temos que fazer algumas alterações bem sutis no método `bind()`.

```
public class UmFragment extends Fragment {  
  
    @BindView(R.id.txt)  
    TextView txt;  
  
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,  
                             Bundle savedInstanceState) {  
  
        View view = inflater.inflate(R.layout.um_layout, container, false);  
  
        txt.setText("ButterKnife dentro de um Fragment :D");  
  
        return view;  
    }  
  
}
```

Da mesma forma que anteriormente, nosso código produzirá uma `NullPointerException`, pois ele não conhece qual é o layout onde deve realizar a busca. Por isso, precisaremos chamar o método `bind()`, desta vez um pouco diferente, já que o **ciclo de vida** do `Fragment` funciona de outra forma. Então passaremos a referência para o `Fragment` e além disso temos que passar a `View` que possui o layout onde a busca será executada:

```
public class UmFragment extends Fragment {

    @BindView(R.id.txt)
    TextView txt;

    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
                             Bundle savedInstanceState) {

        View view = inflater.inflate(R.layout.um_layout, container, false);

        ButterKnife.bind(this, view);

        txt.setText("ButterKnife dentro de um Fragment :D");

        return view;
    }
}
```

CONHECENDO MAIS SOBRE O BUTTERKNIFE

Além de injeção de `View`, podemos injetar outros recursos do Android. No curso estudaremos algumas dessas funcionalidades, mas se você quiser ver todas elas, pode olhar a documentação oficial do **ButterKnife**:

<http://jakewharton.github.io/butterknife/>

Para trazermos a dependência do `ButterKnife` para nosso aplicativo temos que adicionar duas linhas no `build.gradle`:

```
dependencies {
    compile 'com.jakewharton:butterknife:8.2.1'
    apt 'com.jakewharton:butterknife-compiler:8.2.1'
}
```

1.4 EXERCÍCIOS

1. Para começar a utilizar o *ButterKnife* precisamos trazer a dependência para o projeto. Para isso abra o arquivo `build.gradle` do módulo e adicione essas linhas dentro do contexto de dependências:

```
dependencies {
    compile 'com.jakewharton:butterknife:8.2.1'
    apt 'com.jakewharton:butterknife-compiler:8.2.1'
```

```
}
```

2. Nesse instante devemos estar com um problema falando que a função *apt* não está indexada, para isso precisaremos adicionar ao gradle de onde ele irá aprender a executar tal comportamento, para isso abra o build.gradle do projeto e adicione essa linha :

```
dependencies {  
    // essa linha já aparece para você  
    classpath 'com.android.tools.build:gradle:2.1.2'  
  
    // adicione esta  
    classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'  
}
```

3. Precisamos ainda fazer o *import* para conseguirmos utilizar a função do dentro no gradle, para isso abra o arquivo build.gradle do módulo e adicione o import :

```
// você já tem essa linha  
apply plugin: 'com.android.application'  
  
// adicione essa linha !  
apply plugin: 'android-apt'
```

4. Agora vamos abrir nosso `LivroAdapter` e começar a utilizar o *ButterKnife*:

```
class ViewHolder extends RecyclerView.ViewHolder {  
  
    @BindView(R.id.imagem_livro_item)  
    ImageView fotoLivro;  
  
    @BindView(R.id.nome_livro_item)  
    TextView nomeLivro;  
  
    public ViewHolder(View itemView) {  
        super(itemView);  
  
        ButterKnife.bind(this, itemView);  
    }  
}
```

5. Precisamos apenas melhorar o código do nosso `Fragment` , portanto abra a classe `ListaLivrosFragment` e usufrua do *ButterKnife*:

```
public class ListaLivroFragment extends Fragment {  
  
    // Vamos adicionar o atributo abaixo  
    @BindView(R.id.lista_livros)  
    RecyclerView recycler;  
  
    @Override  
    public View onCreateView(LayoutInflater inflater,  
        @Nullable ViewGroup container, Bundle savedInstanceState) {  
  
        View view = inflater.inflate(R.layout.lista_livros_fragment, container, false);  
  
        // Aqui fazemos o bind usando o butterknife  

```

```
        ButterKnife.bind(this, view);

        // resto do código

        return view;
    }
}
```

Rode novamente o projeto !

DETALHES DO LIVRO

A tela principal do nosso aplicativo já está pronta e conseguimos visualizar todo o catálogo disponível de livros. Eventualmente, um usuário pode se interessar por algum livro e querer ver mais informações antes de efetuar a compra. No nosso sistema, ainda não temos nenhuma forma de exibir mais informações, por isso agora vamos criar uma nova tela para mostrar todos os detalhes adicionais de um livro.

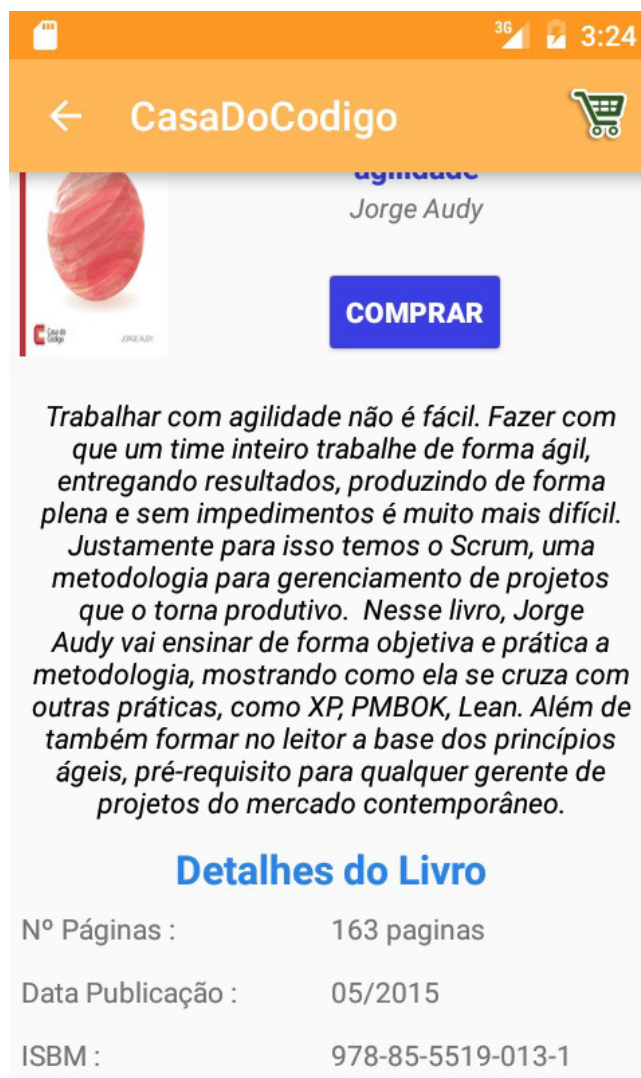


Figura 2.1: Tela Detalhes

Da mesma forma que fizemos com a listagem, manteremos a tela de detalhes em um `Fragment` também. O layout dessa nova tela é um pouco mais complexo então na criação do classe teremos um bom ganho utilizando o `ButterKnife` :

```
public class DetalhesLivroFragment extends Fragment {

    @BindView(R.id.detalhes_livro_foto)
    ImageView foto;

    @BindView(R.id.detalhes_livro_nome)
    TextView nome;

    @BindView(R.id.detalhes_livro_autores)
    TextView autores;

    @BindView(R.id.detalhes_livro_descricao)
    TextView descricao;

    @BindView(R.id.detalhes_livro_num_paginas)
    TextView numPaginas;

    @BindView(R.id.detalhes_livro_isbn)
    TextView isbn;

    @BindView(R.id.detalhes_livro_data_publicacao)
    TextView dataPublicacao;

    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_detalhes_livro, container, false);
        ButterKnife.bind(this, view);

        Bundle arguments = getArguments();
        Livro livro = (Livro) arguments.getSerializable("livro");
        populaCamposCom(livro);

        return view;
    }
}
```

Agora que já criamos nosso `Fragment` temos que ter uma forma de deixar o usuário navegar da listagem de livros para a tela de detalhes. Primeiro vamos precisar tratar o clique na lista de livros e depois passar o `livro` que foi selecionado para o `DetalhesLivroFragment` . Mas se formos em busca de algum método em nosso `RecyclerView` que trate a seleção do item pelo *click*, não iremos encontrar nenhum.

Esta é uma das diferenças que encontramos quando usamos o `RecyclerView` . Como não tem um método para registrar um *listener* para todos os itens da lista, vamos ter que registrar os *listeners* individualmente para cada item da lista. Nessa caso, onde temos acesso aos itens da nossa lista? No nosso `LivrosAdapter` , mais especificamente nos `ViewHolder` do `LivrosAdapter` . Podemos então registrar os *listeners* nessa classe:


```

class ViewHolder extends RecyclerView.ViewHolder {

    @BindView(R.id.item_livro_foto)
    ImageView fotoLivro;

    @BindView(R.id.item_livro_nome)
    TextView nomeLivro;

    public ViewHolder(View itemView) {
        super(itemView);

        ButterKnife.bind(this, itemView);

        itemView.setOnClickListener(new OnClickListener() {

            @Override
            public void onClick(View view) {

                // recupera o livro e troca o fragment

            }

        });
    }
}

```

Estamos colocando toda a regra que queremos que nossos itens tenham, dentro do construtor do `ViewHolder`. Pode parecer algo incorreto, contudo quando estamos trabalhando com `RecyclerView` devemos fazer desta forma. Para tornar nosso código mais limpo e organizado, faremos uso do `ButterKnife` utilizando apenas uma **Annotation**:

```

class ViewHolder extends RecyclerView.ViewHolder {
    @BindView(R.id.item_livro_nome)
    TextView nome;

    @BindView(R.id.item_livro_foto)
    ImageView foto;

    public ViewHolder(View view) {
        super(view);

        ButterKnife.bind(this, view);
    }

    @OnClick(R.id.item_livro)
    public void clickItem() {

        // recupera o livro e troca o fragment

    }
}

```

Essa **Annotation** irá, em tempo de execução, gerar o código referente ao `Listener` para nós. Esse comportamento facilitará muito no momento em que formos registrar nossos eventos, além de manter nosso código mais limpo. Veremos mais adiante que além de facilitar para a *Interface*

OnClickListener , teremos outras **Annotations**, para tratar os eventos mais comuns.

Agora precisamos implementar de fato o *click* do item. Primeiro precisamos recuperar o *livro* , que é uma tarefa bem fácil de realizarmos. Como possuímos a lista como atributo em nosso *Adapter* fica bem fácil recuperarmos o item que foi selecionado:

```
@OnClick(R.id.item_livro)
public void clickItem() {

    Livro livro = livros.get(posicao);
    // alterar fragment

}
```

Agora precisamos obter a posição do item que foi clicado, para isso o próprio *Adapter* nos ajudará. Para recuperarmos a posição correspondente da lista, invocaremos o método `getAdapterPosition()` :

```
@OnClick(R.id.item_livro)
public void clickItem() {

    int posicao = getAdapterPosition();
    Livro livro = livros.get(posicao);
    // alterar fragment

}
```

Agora como realmente temos a referência para o *Livro* daquela posição podemos enfim manipular a tela e trocar o *Fragment* .

Para efetuarmos a troca, precisaremos manipular o *FragmentManager* :

```
@OnClick(R.id.item_livro)
public void clickItem() {

    int posicao = getAdapterPosition();
    Livro livro = livros.get(posicao);

    Bundle arguments = new Bundle();
    arguments.putExtra("livro", livro);

    DetalheLivroFragment detalhes = new DetalheLivroFragment();
    detalhes.setArguments(arguments);

    MainActivity activity = (MainActivity) getActivity();
    FragmentTransaction tx = activity.getFragmentManager().beginTransaction();

    tx.replace(R.id.frame, detalhes);

    tx.commit();

}
```

Agora quando estamos realizando o *click* sobre qualquer item da lista, estamos trocando o *Fragment* .

Quando usamos `Fragment`, a ideia é termos um pedaço de tela com comportamento e layout já definidos, dessa forma conseguimos o reaproveitá-lo em diversos outros locais. Por exemplo, poderíamos reaproveitar esse nosso `Fragment` em uma tela um pouco maior onde além de exibir a listagem também exibimos ao lado os detalhes do livro selecionado. O aconteceria nessa tela quando clicássemos em um item da lista? Seria legal que só a parte dos detalhes fosse atualizada mas da forma como está, nosso código está fazendo a troca de um *fragment* específico com o id `frame`.

2.1 DESACOPLANDO NOSSO CÓDIGO COM O PATTERN DELEGATE

Eventualmente nos pegamos numa situação semelhante a que nos encontramos em nossa aplicação, onde temos objetos totalmente acoplados. Vamos começar a melhorar um pouco nosso código, para desacoplar esse comportamento. Vamos iniciar, fazendo com que nosso `Fragment` apenas avise a `Activity` que um item foi selecionado sem saber exatamente o que será feito com ele:

```
@OnClick(R.id.item_livro)
public void clickItem() {

    int posicao = getAdapterPosition();
    Livro livro = livros.get(posicao);

    MainActivity activity = (MainActivity) getActivity();

    activity.lidaComLivroSelecionado(livro);
}
```

Dessa maneira, nosso `fragment` não conhece mais o comportamento do clique na lista e apenas avisa a `Activity` que um item foi selecionado. Nessa caso, dizemos que estamos delegando o tratamento do evento de clique para a `Activity`.

Mas agora, vamos supor que precisamos reaproveitar esse mesmo *fragment* com uma *activity* diferente. O que aconteceria? Ainda estamos acoplados a `MainActivity`! Perceba que no código temos uma referência e um *casting* para `MainActivity` e para trocar de `Activity` precisaríamos alterar esses dois pontos no `_fragment`.

Para solucionarmos mais esse problema, vamos deixar a classe mais genérica para conseguirmos desacoplar mais ainda nosso código:

```
@OnClick(R.id.item_livro)
public void clickItem() {

    int posicao = getAdapterPosition();
    Livro livro = livros.get(posicao);

    ObjetoGenerico objeto = (ObjetoGenerico) getActivity();

    objeto.lidaComLivroSelecionado(livro);
}
```

Agora precisamos **garantir** que esse `ObjetoGenerico` possua o método `lidaComLivroSelecioneado()`. Para isso, usaremos um recurso muito poderoso e útil que é o **Polimorfismo**. Começamos criando uma *Interface* que declara este método:

```
public interface ObjetoGenerico {  
  
    public void lidaComLivroSelecioneado(Livro livro);  
  
}
```

Agora basta que as `Activities` que queiram aproveitar o nosso *fragment* implementem a *interface* `ObjetoGenerico`. Dessa forma, deixamos nosso objeto sem nenhum acoplamento com qualquer `Activity`. Essa maneira de resolver este tipo de problema é bem comum, tanto que se tornou um *pattern* bem conhecido, chamado de **Delegate**. Vamos ver o que muda:

```
public interface LivrosDelegate {  
  
    public void lidaComLivroSelecioneado(Livro livro);  
  
}  
  
class ViewHolder extends RecyclerView.ViewHolder {  
  
    // resto do código  
  
    @OnClick(R.id.item_livro)  
    public void clickItem() {  
  
        int posicao = getAdapterPosition();  
        Livro livro = livros.get(posicao);  
  
        LivrosDelegate delegate = (LivrosDelegate) getActivity();  
  
        delegate.lidaComLivroSelecioneado(livro);  
  
    }  
  
public class MainActivity extends AppCompatActivity implements LivrosDelegate {  
  
    // métodos anteriores  
  
    public void lidaComLivroSelecioneado(Livro livro){  
  
        // faz troca de fragment  
  
    }  
  
}
```

2.2 EXERCÍCIOS - UTILIZANDO O PATTERN DELEGATE

1. Agora vamos tratar um pouco do clique de cada item da listagem, para facilitar nossa vida, usaremos uma `Annotation` do **ButterKnife**, que irá deixar nosso código mais limpo e com mais qualidade, para isso abra nosso `adapter` e dentro de nosso `ViewHolder` adicione esse método :

```

class ViewHolder extends RecyclerView.ViewHolder {
    @BindView(R.id.item_livro_nome)
    TextView nome;

    @BindView(R.id.item_livro_foto)
    ImageView foto;

    public ViewHolder(View view) {
        super(view);

        ButterKnife.bind(this, view);
    }

    @OnClick(R.id.item_livro)
    public void clickItem() {
        // ainda iremos implementar o que o click faz
    }
}

```

2. Vamos deixar nosso código arquitetado para não sofrermos com alto acoplamento, para isso faremos uso do *pattern Delegate*, crie a interface `LivrosDelegate` no pacote `delegate`:

```

public interface LivrosDelegate {

    public void lidaComLivroSelecionado(Livro livro);

}

```

3. Faça com que nossa `MainActivity` seja obrigatória a ter o método `lidaComLivroSelecionado()`:

```

public class MainActivity extends AppCompatActivity implements LivrosDelegate {

    // Código anterior

    @Override
    public void lidaComLivroSelecionado(Livro livro) {
        Toast.makeText(this, "Livro selecionado: " + livro.getNome(), Toast.LENGTH_LONG).show();
    }
}

```

4. Agora temos que fazer nosso `Adapter` delegar o clique do item, para isso faremos uso do nosso `delegate`:

```

@OnClick(R.id.item_livro)
public void clickItem() {
    Livro livro = livros.get(getAdapterPosition());
    LivrosDelegate delegate = (LivrosDelegate) itemView.getContext();
    delegate.lidaComLivroSelecionado(livro);
}

```

Rode o aplicativo e veja o `Toast` ser exibido.

5. Agora que vimos que nosso `Delegate` está funcionando e vimos ele funcionar, podemos fazer a troca dos `Fragments`, para isso crie a classe `DetalheLivroFragment` no pacote de `fragment`.

```
``` java
```

```
public class DetalheLivroFragment extends Fragment {
```

```
}
```

```
```
```

1. Para podermos manipular a tela, necessitaremos de um *layout* portanto vamos utilizar um layout já pronto que importamos logo após criarmos o projeto no começo do curso. Aproveite para dar uma olhada nesse layout chamado `fragment_detalhes_livro.xml` .
2. Agora com nosso *layout* já definido podemos vincular ele ao nosso `Fragment` :

```
public class DetalhesLivroFragment extends Fragment {
```

```
    @Nullable
```

```
    @Override
```

```
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
```

```
        View view = inflater.inflate(R.layout.fragment_detalhes_livro, container, false);
```

```
        return view;
```

```
    }
```

```
}
```

3. Precisamos recuperar todas as views, fazendo `findViewById()` , mas para essa tela teríamos um trabalho bem grande. Vamos continuar tirando proveito máximo do **ButterKnife** utilizando o conceito que já vimos ainda há pouco, a injeção de Views :

```
public class DetalhesLivroFragment extends Fragment {
```

```
    @BindView(R.id.detalhes_livro_foto)
```

```
    ImageView foto;
```

```
    @BindView(R.id.detalhes_livro_nome)
```

```
    TextView nome;
```

```
    // faça o restante dos bind views para os demais atributos
```

```
    @Nullable
```

```
    @Override
```

```
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
```

```
        View view = inflater.inflate(R.layout.fragment_detalhes_livro, container, false);
```

```
        ButterKnife.bind(this, view);
```

```
        return view;
```

```
    }
```

```
}
```

4. Com o `Fragment` pronto, podemos executar a troca, para isso vamos alterar o método `lidaComLivroSelecionado()` da nossa `MainActivity` :

```

public void lidaComLivroSelecionado(Livro livro) {
    FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
    DetalhesLivroFragment detalhesLivroFragment = criaDetalhesCom(livro);
    transaction.replace(R.id.frame_principal, detalhesLivroFragment);
    transaction.addToBackStack(null);
    transaction.commit();
}

```

5. Nesse ponto estamos com um pequeno problema de compilação, pois ainda não temos o método `criaDetalhesCom()`. Vamos criar este método na própria `MainActivity`:

```

private DetalhesLivroFragment criaDetalhesCom(Livro livro) {
    Bundle bundle = new Bundle();
    bundle.putSerializable("livro", livro);
    DetalhesLivroFragment detalhesLivroFragment = new DetalhesLivroFragment();
    detalhesLivroFragment.setArguments(bundle);
    return detalhesLivroFragment;
}

```

6. Caso seja necessário, implemente a interface `Serializable` na classe `Livro` pois estamos colocando-a dentro de um `Bundle` que só aceita objetos que implementem essa interface.
7. Agora precisamos abrir o `Bundle` que passamos para poder popular os campos do nosso `DetalheLivroFragment`:

```

public class DetalhesLivroFragment extends Fragment {

    // views injetadas

    private Livro livro;

    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_detalhes_livro, container, false);
        ButterKnife.bind(this, view);

        Bundle arguments = getArguments();
        livro = (Livro) arguments.getSerializable("livro");
        populaCamposCom(livro);

        return view;
    }

    private void populaCamposCom(Livro livro){

        // popule as views injetadas com o livro recebido por parâmetro

    }
}

```

8. O código ainda não deve estar compilando porque ainda precisamos implementar o método `populaCamposCom()` no `DetalhesLivroFragment`:

```

private void populaCamposCom(Livro livro) {
    nome.setText(livro.getNome());
}

```

```

String listaDeAutores = "";
for (Autor autor : livro.getAutores()) {
    if (!listaDeAutores.isEmpty()) {
        listaDeAutores += ", ";
    }
    listaDeAutores += autor.getNome();
}
autores.setText(listaDeAutores);

descricao.setText(livro.getDescricao());
numPaginas.setText(String.valueOf(livro.getNumPaginas()));
isbn.setText(livro.getISBN());
dataPublicacao.setText(livro.getDataPublicacao());

String textoComprarFisico = String.format("Comprar Livro Fisico - R$ %.2f", livro.getValorFisico());
botaoComprarFisico.setText(textoComprarFisico);

String textoComprarEbook = String.format("Comprar E-book - R$ %.2f", livro.getValorVirtual());
;
botaoComprarEbook.setText(textoComprarEbook);

String textoComprarAmbos = String.format("Comprar Ambos - R$ %.2f", livro.getValorDoisJuntos());
botaoComprarAmbos.setText(textoComprarAmbos);
}

```

Rode novamente o aplicativo e selecione um livro para ver como ficou a tela de detalhes!



DADOS REAIS DE UM WEBSERVICE

Até o momento todos os dados do nosso aplicativo estão *hardcoded*, só para manipularmos testes e deixarmos as funções mais básicas operacionais, visando um **MVP** (*minimum viable product*), suficiente para entregar um software de qualidade.

3.1 FAZENDO A BUSCA DO SERVIDOR

Para melhorarmos nosso produto, iremos manipular dados reais, da própria Casa do Código. Para podermos conversar com nosso servidor sem a preocupação de travar a `Thread` principal e receber uma `NetworkOnMainThreadException`, faremos nossa requisição em uma `Thread` secundária:

```
public class MainActivity extends AppCompatActivity {

    @BindView(R.id.lista)
    ListView lista;

    @Override
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.main);
        ButterKnife.bind(this);

        new Thread(new Runnable() {

            public void run(){
                List<Livro> livros = new Webservice().getLivros();

                NossoAdapter adapter = new NossoAdapter(livros);
                lista.setAdapter(adapter);
            }

        }).start();
    }
}
```

Com esse nosso código, nós receberíamos outra `Exception` referente a estarmos tentando atualizar a lista de livros fora da `Thread` principal. Para solucionarmos isto, temos que fazer com que a parte onde manipulamos a tela seja executada na `Thread` principal. Para isso, a própria `Activity` nos ajuda, possuindo um método que permite agendar a execução de instruções diretamente na `Thread` principal.

```

public class MainActivity extends AppCompatActivity {

    @BindView(R.id.lista)
    ListView lista;

    @Override
    public void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.main);
        ButterKnife.bind(this);

        new Thread(new Runnable() {

            public void run(){
                List<Livro> livros = new Webservice().getLivros();

                runOnUiThread(new Runnable() {

                    public void run() {
                        NossoAdapter adapter = new NossoAdapter(livros);
                        lista.setAdapter(adapter);
                    }
                });
            }

        }).start();
    }
}

```

Dessa maneira quando nosso código for executado, ele buscará os dados fora Thread principal e quando conseguir a resposta, mandará os dados para a Thread principal para tratar a resposta. Contudo ao analisarmos este código que acabamos de gerar, temos alguns problemas. Inicialmente, se quisermos dar manutenção é algo bem complexo, por estarmos tratando duas Threads em poucas linhas de código. Além disso, a legibilidade do código piora bastante com o uso das classes anônimas.

Fazer requisições igual a esta que fizemos ainda há pouco, na grande maioria dos aplicativos de mercado é bem comum mas ficar manipulando Threads na mão, não é uma prática muito recomendada já que precisamos de uma atenção redobrada para não nos perdermos entre as Threads . Por esse motivo, temos uma classe nativa do *Android* que nos ajudará a fazer a requisição e tratar sua resposta nos locais corretos. O nome dessa classe é *AsyncTask* .

```

public class BuscaLivrosTask extends AsyncTask <Object, Object, Object> {

}

```

Para termos uma tarefa sendo executada numa Thread secundária só precisamos sobrescrever o método `doInBackground` da classe *AsyncTask* :

```

public class BuscaLivrosTask extends AsyncTask < Object, Object, Object> {

```

```

@Override
public Object doInBackground(Object... params) {

    // aqui teremos que fazer a tarefa que seria executada na thread secundária

    return null;
}
}

```

Por trás dos panos, o método `doInBackground()` abrirá uma nova `Thread` para fazer todo o processamento pesado e assim que terminar ele precisa de alguma forma entregar o resultado da tarefa para a `Thread` principal. A forma que temos de fazer isso é devolvendo o resultado no método `doInBackground` :

```

public class BuscaLivrosTask extends AsyncTask < Void, Object, List<Livro> > {

    @Override
    public List<Livro> doInBackground(Object... params) {

        WebService ws = new WebService();

        List<Livro> livros = ws.getLivros();

        return livros;
    }

}

```

Necessitamos apenas cuidar do retorno da tarefa que foi executada, para isso temos que ter um comportamento similar ao que fazemos ao manipular as `Threads` na mão onde ao terminar voltávamos para a `Thread` principal e fazíamos o tratamento do resultado. Para termos o mesmo comportamento, a classe `AsyncTask` possui o método `onPostExecute` que é executado logo após o fim do `doInBackground` , com a diferença de que esse método é executado na `Thread` principal. Um outro detalhe importante é que esse método recebe como parâmetro o mesmo resultado devolvido ao fim do `doInBackground` . Desse modo, conseguimos então receber o resultado da tarefa na `Thread` principal:

```

public class BuscaLivrosTask extends AsyncTask < Void, Object, List<Livro> > {

    @Override
    public List<Livro> doInBackground(Void... params) {

        WebService ws = new WebService();

        List<Livro> livros = ws.getLivros();

        return livros;
    }

    @Override
    public void onPostExecute(List<Livro> livros) {

        // como atualizar a tela agora ?
    }
}

```

```

}
}

```

Agora que estamos recebendo o resultado na `Thread` principal, só precisamos colocá-lo dentro de nosso `ListView`. Para isso, precisaremos ter uma referência para a *view* dentro da `AsyncTask`. Uma possibilidade seria trazer uma referência da própria `Activity` que nos dará acesso ao `ListView`:

```

public class BuscaLivrosTask extends AsyncTask < Void, Object, List<Livro> > {

    private MainActivity activity;

    public BuscaLivrosTask(MainActivity activity) {
        this.activity = activity;
    }

    @Override
    public List<Livro> doInBackground(Void... params) {

        WebService ws = new WebService();

        List<Livro> livros = ws.getLivros();

        return livros;
    }

    @Override
    public void onPostExecute(List<Livro> livros) {

        ListView lista = (ListView) activity.findViewById(R.id.lista);

        Adapter adapter = new Adapter(livros);

        lista.setAdapter(adapter);
    }
}

```

Com isso, a requisição será executada e nossa lista atualizada. Contudo vamos olhar com mais atenção esse pedaço de código:

```

@Override
public void onPostExecute(List<Livro> livros) {

    ListView lista = (ListView) activity.findViewById(R.id.lista);

    Adapter adapter = new Adapter(livros);

    lista.setAdapter(adapter);
}

```

Nossa `AsyncTask` além de fazer a requisição está manipulando a tela, que não é sua responsabilidade! Além disso, ela faz referência a um *id* que não necessariamente pode estar presente em todas as *activities* onde essa *task* pode ser usada. Para resolvermos isso é bem simples, basta fazer com que nossa `AsyncTask` apenas avise alguém que a tarefa foi terminada:

```

@Override
public void onPostExecute(List<Livro> livros) {

    activity.atualizaTela(livros);

}

```

Dessa forma, estamos utilizando um conceito bem bacana que é a separação de responsabilidades entre as classes, onde cada uma tem suas especialidades. Nesse caso, estamos deixando que a `Activity` atualize a tela, que é o seu papel, e nossa `AsyncTask` está fazendo a requisição e avisando apenas que terminou, que é a sua função.

Tudo parece estar certo agora, contudo ainda temos mais um problema. O que acontece se quisermos reaproveitar essa tarefa em outra tela? Não iríamos conseguir porque estamos acoplados com a `MainActivity`, a mesma coisa que aconteceu quando fomos trocar nossos `Fragments`. Aqui resolveremos da mesma maneira, utilizando o *pattern* Delegate:

```

public class BuscaLivrosTask extends AsyncTask < Void, Object, List<Livro> > {

    private Delegate delegate;

    public BuscaLivrosTask(Delegate delegate) {
        this.delegate = delegate;
    }

    @Override
    public List<Livro> doInBackground(Void... params) {

        WebService ws = new WebService();

        List<Livro> livros = ws.getLivros();

        return livros;
    }

    @Override
    public void onPostExecute(List<Livro> livros) {

        delegate.lidaComResposta(livros);
    }

}

```

3.2 MELHORANDO NOSSAS REQUISIÇÕES COM RETROFIT

Algo bem comum é termos que trafegar os dados através de requisições, independente da arquitetura que esteja o servidor, teremos que abrir uma conexão, e como já vimos, para cada requisição que fazemos que temos que criar uma `AsyncTask`. Caso tenhamos um sistema um pouco maior, onde para cada tela tenhamos que fazer uma nova requisição, nosso sistema ficará repleto de classes, o que dará um certo trabalho para dar manutenção.

Precisamos dar um jeito de melhorar a nossa vida como desenvolvedor, para isso faremos uso de uma biblioteca bem famosa e reconhecida de mercado que consegue facilitar bastante a vida do desenvolvedor, conhecida como **Retrofit**.

Para começarmos a usar essa lib é bem simples, a primeira coisa que precisamos fazer é declarar qual é a requisição que queremos fazer e o que ela deve retornar. Para isso temos que criar uma *interface* para declarar esses dois passos:

```
public interface ListaLivrosService {  
  
    @GET("listarLivros?indicePrimeiroLivro=0&qtdLivros=20")  
    Call<List<Livro>> listaLivros();  
  
}
```

Criamos um método que fará uma chamada (`Call`) que retornará uma lista de livros, e deixamos bem explícito que estamos realizando um `Get` para aquela URL já com seus parâmetros de busca. Agora que vamos utilizar esse **Service**, iremos alterar nosso `WebService` :

```
public class WebService {  
  
    public List<Livro> getLivros() {  
  
        // como utilizar a interface ?  
    }  
  
}
```

Como só especificamos a interface do nosso serviço, precisamos de alguém que consiga construir uma implementação para ela. Além disso, também precisamos dizer em algum lugar qual vai ser o endereço onde iremos fazer o `Get` . A classe que vai nos ajudar a fazer isso é a `Retrofit` que é o *client* do Retrofit. Para construí-la, vamos usar um `Builder` que existe dentro da própria classe:

```
public class WebService {  
  
    public List<Livro> getLivros() {  
  
        Retrofit client = new Retrofit.Builder()  
            .baseUrl("www.enderecodoserver.com.br/")  
            .build();  
    }  
  
}
```

Agora que já possuímos o `client`, podemos utilizá-lo para criar o nosso serviço baseado na interface que definimos anteriormente. Para isso, usaremos o método `create()` que irá receber a interface e devolver uma implementação dessa interface:

```

public class WebService {

    public List<Livro> getLivros() {

        Retrofit client = new Retrofit.Builder()
            .baseUrl("www.enderecodoserver.com.br/")
            .build();

        ListaLivrosService service = client.create(ListaLivrosService.class);

    }

}

```

Agora com o objeto criado podemos invocar nosso método para recuperarmos a listagem:

```

public class WebService {

    public List<Livro> getLivros() {

        Retrofit client = new Retrofit.Builder()
            .baseUrl("www.enderecodoserver.com.br/")
            .build();

        ListaLivrosService service = client.create(ListaLivrosService.class);

        Call<List<Livro>> call = listaLivrosService.listaLivros();

    }

}

```

E com esse objeto do tipo `Call` podemos fazer a chamada para o servidor, utilizando o método `execute()` que retornará a resposta do servidor :

```

public class WebService {

    public List<Livro> getLivros() {

        Retrofit client = new Retrofit.Builder()
            .baseUrl("www.enderecodoserver.com.br/")
            .build();

        ListaLivrosService service = client.create(ListaLivrosService.class);

        Call<List<Livro>> call = listaLivrosService.listaLivros();

        Response<List<Livro>> response = call.execute();

    }

}

```

Finalmente, podemos pegar nossa listagem que está no corpo da resposta. Nesse caso, podemos

recuperar o corpo usando o método `body()` :

```
public class WebService {

    public List<Livro> getLivros() {

        Retrofit client = new Retrofit.Builder()
            .baseUrl("www.enderecodoserver.com.br/")
            .build();

        ListaLivrosService service = client.create(ListaLivrosService.class);

        Call<List<Livro>> call = listaLivrosService.listaLivros();

        Response<List<Livro>> response = call.execute();

        List<Livro> livros = response.body();

        return livros;
    }
}
```

Agora não precisamos mais usar `AsyncTask` para enviar nossa requisição, além de não estarmos mais precisando escrever diversas linha manipulando a requisição na mão. Podemos chamar nosso `WebService` diretamente na nossa `Activity` :

```
public class MainActivity extends AppCompatActivity {

    @Override
    public void onCreate(Bundle bundle) {
        // super e setContentView

        ListView lista = // findViewById
        List<Livros> livros = new WebService().getLivros();

        // adapter
    }
}
```

Está bem mais prático agora! Não estamos nos preocupando mais com a `Thread` secundária. Entretanto, no momento que executamos nosso código recebemos uma `Exception` que não queríamos : `NetworkOnMainThreadException` !

Vamos entender por que levamos essa `Exception` novamente. Quando utilizamos o método `execute()` estamos fazendo todo processamento na `Thread` principal, o que estávamos tentando não fazer. Para resolver esse problema é bem simples, basta não utilizarmos esse método. No seu lugar, usaremos outro bem similar que é o `enqueue()` , que abrirá automaticamente uma `Thread` secundária.


```

public class WebService {

    public List<Livro> getLivros() {

        Retrofit client = new Retrofit.Builder()
            .baseUrl("www.enderecodoserver.com.br/")
            .build();

        ListaLivrosService service = client.create(ListaLivrosService.class);

        Call<List<Livro>> call = listaLivrosService.listaLivros();

        call.enqueue();

        return ???;

    }

}

```

Ao usarmos este método resolvemos o problema de rodar na `Thread` principal, mas ele não nos retorna nada, como recuperaremos as informações? Como estamos fazendo a requisição em uma `Thread` secundária, precisamos ser avisados quando essa requisição terminou e a resposta já está pronto. Para isso, temos que passar um `Callback` de conexão, que será um método onde estaremos preparados para tratar se tudo deu certo e também se deu algum problema nessa requisição.

```

public class WebService {

    public List<Livro> getLivros() {

        Retrofit client = new Retrofit.Builder()
            .baseUrl("www.enderecodoserver.com.br/")
            .build();

        ListaLivrosService service = client.create(ListaLivrosService.class);

        Call<List<Livro>> call = listaLivrosService.listaLivros();

        call.enqueue(new Callback<List<Livro>>() {
            @Override
            public void onResponse(Call<List<Livro>> call, Response<List<Livro>> response) {

                // deu tudo certo! podemos tratar a resposta

            }

            @Override
            public void onFailure(Call<List<Livro>> call, Throwable t) {

                // oops! aconteceu algum problema

            }
        });

        return ???;

    }

}

```

```

    }
}

```

Caso dê errado podemos pegar o problema e dar um certo tratamento, como exibir uma mensagem ou tentar exibir uma lista em cache. Então podemos melhorar ainda mais o nosso código, fazendo que seja delegada a responsabilidade da resposta, logo não precisaríamos de um retorno e dentro do `Callback` só delegamos a ação.

```

public class Webservice {

    public void getLivros(LivrosDelegate delegate) {

        Retrofit client = new Retrofit.Builder()
            .baseUrl("www.enderecodoserver.com.br/")
            .build();

        ListaLivrosService service = client.create(ListaLivrosService.class);

        Call<List<Livro>> call = listaLivrosService.listaLivros();

        call.enqueue(new Callback<List<Livro>>() {
            @Override
            public void onResponse(Call<List<Livro>> call, Response<List<Livro>> response) {

                delegate.lidaComSucesso(response.body());
            }

            @Override
            public void onFailure(Call<List<Livro>> call, Throwable erro) {

                delegate.lidaComProblema(erro);
            }
        });
    }
}

```

Uma outra pergunta que pode surgir é como o `Retrofit` consegue trazer diretamente uma lista de livros na resposta sendo que o servidor retorna um *json*? Para ele fazer essa mágica para nós, temos que ensinar ele a fazer a conversão dos dados. A responsabilidade de executar essa tarefa é de um `Converter`. Ele será responsável por transformar a resposta de um formato específico (por exemplo: XML, JSON, etc) em objetos. Usaremos então mais um método no momento em que criamos nosso objeto do tipo `Retrofit` para especificar qual o `Converter` que queremos utilizar:

```

public void getLivros(LivrosDelegate delegate) {

    Retrofit client = new Retrofit.Builder()
        .baseUrl("www.enderecodoserver.com.br/")
        .addConverterFactory(???)
        .build();
}

```

}

Retrofit

Para usarmos o Retrofit em nossa aplicação precisamos adicionar a seguinte linha em nosso gradle :

```
compile 'com.squareup.retrofit2:retrofit:2.1.0'
```

Além de conseguirmos o usar em projetos Android também é possível utilizarmos em qualquer projeto que seja em Java !

Resta passarmos qual vai ser o `Converter` a ser utilizado. Na própria documentação do Retrofit temos as escolhas possíveis e além disso qual a dependência do Gradle para importar cada um desses `Converter` em nosso projeto:

- Gson: `com.squareup.retrofit2:converter-gson`
- Jackson: `com.squareup.retrofit2:converter-jackson`
- Moshi: `com.squareup.retrofit2:converter-moshi`
- Protobuf: `com.squareup.retrofit2:converter-protobuf`
- Wire: `com.squareup.retrofit2:converter-wire`
- Simple XML: `com.squareup.retrofit2:converter-simplexml`
- Scalars (primitives, boxed, and String): `com.squareup.retrofit2:converter-scalars`

Depois de termos importado qualquer um desses `Converter` , basta passar a `ConverterFactory` correspondente para o método `addConverterFactory()` :

```
public void getLivros(LivrosDelegate delegate) {  
  
    Retrofit client = new Retrofit.Builder()  
        .baseUrl("www.enderecodoserver.com.br/")  
        .addConverterFactory(GsonConverterFactory.create())  
        .build();  
  
}
```

Todas as demais opções também tem o mesmo método de criação!

CRIANDO UM CONVERTER FACTORY

É bem comum as opções padrões não conseguirem criar corretamente os objetos que estão sendo trafegados. Como usaremos o Retrofit ? Temos a opção de criar o nosso `ConverterFactory` , para isso basta criarmos uma classe e estender de `Converter.Factory` e sobrescrever os métodos.

E dessa forma basta instanciar o nosso `ConverterFactory` para realizar a conversão dos dados !

3.3 EXERCÍCIOS

1. Vamos criar uma classe onde iremos encapsular nossa requisição cria a classe `WebClient` no pacote `server` :

```
public class WebClient{  
}
```

Precisamos definir o que queremos fazer agora, para isso vamos criar o método `getLivros()` :

```
public class WebClient{  
    public void getLivros() {  
        // teremos que recuperar a listagem de livros aqui  
    }  
}
```

2. Para fazermos nossa requisição usaremos o `Retrofit` precisamos trazer a dependência para dentro de nosso projeto, para isso utilize o atalho `Ctrl + shift + alt + s` e da mesma forma que fizemos com `ButterKnife` , procure por `retrofit` e selecione `com.squareup.retrofit2:retrofit` .

3. Como vamos trabalhar com requisições, precisaremos adicionar a permissão de internet em nosso aplicativo. Edite o seu `AndroidManifest.xml` e adicione a linha abaixo antes da tag `<application>` :

```
<uses-permission android:name="android.permission.INTERNET" />
```

4. Vamos definir qual requisição queremos fazer, para isso crie uma interface `LivrosService` no pacote `server` e defina que o método retorne a listagem, além disso defina qual é o tipo de requisição que faremos e em qual url devemos bater:

```
public interface LivrosService {

    @GET("listarLivros?indicePrimeiroLivro=0&qtdLivros=20")
    Call<List<Livro>> listaLivros();

}
```

5. Agora precisamos criar o client do Retrofit que consiga fazer as requisições da nossa aplicação até o servidor:

```
public class WebClient{

    private static final String SERVER_URL = "http://cdcmob.herokuapp.com/";

    public void getLivros() {

        Retrofit client = new Retrofit.Builder()
            .baseUrl(SERVER_URL)
            .addConverterFactory(new LivroServiceConverterFactory())
            .build();

        LivrosService service = client.create(LivrosService.class);

    }

}
```

Com o service em mão podemos disparar o evento, que deve ser executado em uma Thread secundária. Para isso faremos uso do método `enqueue()` :

```
public class WebClient{

    private static final String SERVER_URL = "http://cdcmob.herokuapp.com/";

    public void getLivros() {

        Retrofit client = new Retrofit.Builder()
            .baseUrl(SERVER_URL)
            .build();

        LivrosService service = client.create(LivrosService.class);

        Call<List<Livro>> call = service.listaLivros();

        call.enqueue(???);

    }

}
```

Agora para deixarmos nosso código compilando precisamos definir um `Callback` para essa requisição, onde teremos que tratar tanto o sucesso quanto qualquer problema :

```
public class WebClient{

    public void getLivros() {
```

```

// resto do código

call.enqueue(new Callback<List<Livro>>() {
    @Override
    public void onResponse(Call<List<Livro>> call, Response<List<Livro>> response) {

    }

    @Override
    public void onFailure(Call<List<Livro>> call, Throwable t) {

    }
});

}

}

```

Agora temos que *delegar* a responsabilidade dessa requisição, para isso precisaremos ter um Delegate. Nesse caso, vamos pedir o delegate no construtor da nossa classe WebClient :

```

public WebClient(LivrosDelegate delegate) {
    this.delegate = delegate;
}

```

Vamos implementar o retorno do Callback :

```

call.enqueue(new Callback<List<Livro>>() {
    @Override
    public void onResponse(Call<List<Livro>> call, Response<List<Livro>> response) {
        delegate.lidaComSucesso(response.body());
    }

    @Override
    public void onFailure(Call<List<Livro>> call, Throwable t) {
        delegate.lidaComErro(t);
    }
});

```

6. Adicione os métodos lidaComSucesso e lidaComErro na interface LivrosDelegate :

```

public interface LivrosDelegate {

    public void lidaComLivroSelecionado(Livro livro);

    public void lidaComSucesso(List<Livro> livros);

    public void lidaComErro(Throwable erro);

}

```

7. Agora vamos implementar os novos métodos do delegate na MainActivity :

```

public class MainActivity extends AppCompatActivity implements LivrosDelegate {
    // resto do código

    @Override

```

```

    public lidaComSucesso(List<Livro> livros) {
        listaLivrosFragment.populaListaCom(livros);
    }

    @Override
    public lidaComErro(Throwable erro) {
        Toast.makeText(this, "Não foi possível carregar os livros...", Toast.LENGTH_SHORT).show();
    }
}

```

8. O método `lidaComSucesso` vai apresentar um erro pois o fragment da lista de livros não foi definido como atributo da `MainActivity`. Vamos adicionar esse atributo e populá-lo no `onCreate`:

```

public class MainActivity extends AppCompatActivity implements LivrosDelegate {

    // outros atributos

    private ListaLivrosFragment listaLivrosFragment;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
        listaLivrosFragment = new ListaLivrosFragment();
        transaction.replace(R.id.frame_principal, listaLivrosFragment);
        transaction.commit();
    }
}

```

9. Agora precisamos adaptar o `ListaLivrosFragment` para deixar de gerar os livros no `onCreateView`, apenas inicializar o `Adapter` e receber a lista de livros da `MainActivity` no método `populaListaCom`:

```

public class ListaLivrosFragment extends Fragment {

    private List<Livro> livros = new ArrayList<>();

    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_lista_livros, container, false);

        ButterKnife.bind(this, view);

        recyclerView.setAdapter(new LivroAdapter(livros));
        recyclerView.setLayoutManager(new LinearLayoutManager(getContext()));

        return view;
    }

    public void populaListaCom(List<Livro> livros) {
        this.livros.clear();
        this.livros.addAll(livros);
        recyclerView.getAdapter().notifyDataSetChanged();
    }
}

```

```
}
```

10. Finalmente, faça a chamada de nosso `WebClient` no método 'onCreate da MainActivity':

```
public class MainActivity extends AppCompatActivity implements LivrosDelegate {

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        // resto do código

        new WebClient(this).getLivros();
    }

}
```

11. Rode a aplicação e veja se ela consegue carregar a lista de livros do servidor!

3.4 DEIXANDO A REQUISIÇÃO OTIMIZADA COM BROADCAST RECEIVER

Já melhoramos bastante nosso código, evitando o uso demasiado de classes para darmos manutenção e reduzimos o uso de `AsyncTask` s com o `Retrofit` .

Agora temos que pensar em outras coisas que podem estar ocorrendo com nosso sistema, vamos analisar a seguinte :

Iniciamos nosso aplicativo, dessa forma ele fará uma requisição ao nosso servidor, entretanto a internet do usuário é bem lenta, ou está com problema de sinal. Neste cenário a requisição demoraria mais para ser enviada. Enquanto ele espera que a requisição, o usuário acaba rotacionando o aparelho, com isso sabemos que o `Android` irá fazer alguns comportamentos do seu **ciclo de vida**, mas sabemos que a referência para aquela `Activity` será destruída e será criada uma nova para a tela em paisagem.

Vamos analisar o nosso código :

```
public class MainActivity extends AppCompatActivity {

    @Override
    public void onCreate(Bundle bundle) {
        // super e setContentView

        ListView lista = //
        List<Livros> livros = new WebService().getLivros();

        //adapter
    }

}
```

Então no momento que a tela for rotacionada, o `Android` irá destruir e criar uma nova `Activity` ,

que ao passar pelo método `onCreate()` irá fazer uma nova requisição.

Estamos com alguns problemas de referência, pois o momento em que nossa `Activity` é destruída, ela está a disposição do `GarbageCollector` para ser retirada da memória, o que fato pode ocorrer a qualquer instante. E nossa requisição, que está sendo executada pelo nosso serviço ainda está viva, está sendo processada, ou seja, não está disponível para ser coletada ainda. Quando o nosso primeiro serviço terminar de executar a requisição o que acontece?

```
call.enqueue(new Callback<List<Livro>>() {
    @Override
    public void onResponse(Call<List<Livro>> call, Response<List<Livro>> response) {
        delegate.lidaComSucesso(response.body());
    }

    @Override
    public void onFailure(Call<List<Livro>> call, Throwable t) {
        delegate.lidaComErro(t);
    }
});
```

Ele dará a responsabilidade de lidar com a resposta para o `delegate`, mas como essa referência não é mais a `Activity` que está na tela, o que irá acontecer é que tentaremos atualizar uma `Activity` que não mais está sendo gerenciada pelo Android. Nesse caso, dependendo da operação que estamos tentando realizar podemos causar uma `Exception`. Então como podemos evitar esse tipo de problema?

O grande problema que estamos enfrentando é estarmos com a referência direta para o `Delegate` que pode estar morto. Para resolver isso, podíamos reaproveitar a busca na nova `Activity`, que será o novo `Delegate`, mas como iremos atualizar a referência em nosso serviço?

Como não sabemos em qual momento a resposta chegará, podemos utilizar o comportamento padrão do Android, como o recebimento de mensagens SMS, e deixar alguém responsável por ouvir esse evento usando um `BroadcastReceiver`.

```
public class ReceptorDoServico extends BroadcastReceiver {

    public void onReceive(Context ctx, Intent data ) {

        // aqui trataremos a resposta do server

    }

}
```

Mas e agora, como faremos para ativar esse receptor e passar a resposta da requisição para ele?

É bem simples, poderíamos fazer um *broadcast* da resposta para o sistema operacional ouvir. Contudo, não faria sentido deixar o sistema operacional disparar esse *broadcast* para todas as aplicações instaladas, isso seria um grande problema de vulnerabilidade. Para não termos esse problema, faremos

uso de um recurso bem interessante do Android que é fazer o *broadcast* ser interno, pertencente apenas a nossa aplicação. Para isso, faremos uso da classe `LocalBroadcastManager` que irá disparar um *broadcast* apenas para o nosso aplicativo!

Então no momento em que a requisição nos der o resultado, temos que fazer o *broadcast*, desacoplando o nosso `delegate` :

```
call.enqueue(new Callback<List<Livro>>() {
    @Override
    public void onResponse(Call<List<Livro>> call, Response<List<Livro>> response) {
        ReceptorDoServico.disparaSucesso(response.body());
    }

    @Override
    public void onFailure(Call<List<Livro>> call, Throwable t) {
        ReceptorDoServico.disparaErro(t);
    }
});
```

Agora precisamos definir esses dois métodos que criamos em nossa requisição, para que façam o broadcast para nós, disparando tanto o erro quanto a própria listagem:

```
public class ReceptorDoServico extends BroadcastReceiver {

    public static void disparaSucesso(List<Livro> livros) {

    }

    public static void disparaErro(Throwable erro){

    }

    public void onReceive(Context ctx, Intent data ) {

        // em breve faremos algo aqui
    }

}
```

Para dispararmos um *broadcast*, precisaremos de uma instância de `LocalBroadcastManager` . Para a obtermos, basta invocarmos o método `getInstance()` :

```
public static void disparaSucesso(List<Livro> livros) {

    LocalBroadcastManager localBroadcast = LocalBroadcastManager.getInstance();

}
```

Porém para conseguirmos uma instância, já que iremos usar um recurso do Android , é necessário

passar um `Context` para que possa ser enviado o *broadcast*, por esse motivo passaremos o `Context` do próprio `delegate` que estará ouvindo esse evento:

```
public class ReceptorDoServico extends BroadcastReceiver {

    private LivrosDelegate delegate;

    public static void disparaSucesso(List<Livro> livros) {

        LocalBroadcastManager localBroadcast = LocalBroadcastManager.getInstance(delegate.getContext());

    }

    public static void disparaErro(Throwable erro){

    }

    public void onReceive(Context ctx, Intent data ) {

        // em breve faremos algo aqui

    }

}
```

Agora precisamos disparar o *broadcast* com a lista de livros usando o método `sendBroadcast()`. Mas precisamos associar a lista de livros com o broadcast. Como fazemos para trafegar informações de uma `Activity` para outra no Android? Utilizamos `Intents` e aqui não será diferente. Passaremos como parâmetro uma `Intent` para ser enviada:

```
public class ReceptorDoServico extends BroadcastReceiver {

    private LivrosDelegate delegate;

    public static void disparaSucesso(List<Livro> livros) {

        LocalBroadcastManager localBroadcast = LocalBroadcastManager.getInstance(delegate.getContext());

        Intent intent = new Intent();
        intent.putExtra("livros", livros);

        localBroadcast.sendBroadcast(intent);

    }

    // demais métodos

}
```

Quando instanciamos uma `Intent`, geralmente passamos em seu construtor o contexto no qual estamos e para qual `Activity` queremos ir. Uma outra possibilidade é fazer o uso das `Intents` *Implicitas*, onde geralmente passamos constantes para direcionarmos nosso aplicativo para outra aplicação.

No nosso caso, apenas queremos que ela continue na nossa aplicação e que caia em nosso `BroadcastReceiver`, para isso, iremos **criar** nossa própria `Intent` `Implicita`. Para fazermos isso, basta definirmos qual é o código de identificação dela, similar a isto :

```
public class ReceptorDoServico extends BroadcastReceiver {

    private LivrosDelegate delegate;

    public static void disparaSucesso(List<Livro> livros) {

        LocalBroadcastManager localBroadcast = LocalBroadcastManager.getInstance(delegate.getContext());

        Intent intent = new Intent("BananasDePijama");
        intent.putExtra("livros", livros);

        localBroadcast.sendBroadcast(intent);

    }

    // demais métodos
}
```

Agora precisamos fazer a mesma coisa para o método `disparaErro` :

```
public class ReceptorDoServico extends BroadcastReceiver {

    private LivrosDelegate delegate;

    public static void disparaErro(Throwable erro){

        LocalBroadcastManager localBroadcast = LocalBroadcastManager.getInstance(delegate.getContext());

        Intent intent = new Intent("BananasDePijama");
        intent.putExtra("erro", erro);

        localBroadcast.sendBroadcast(intent);

    }

    // demais métodos
}
```

Agora ambas irão disparar um *broadcast*, mas como iremos tratar o resultado de cada uma, já que no método `onReceive()` estamos recebendo a `Intent` que é enviada pelo `LocalBroadcastManager` ?

```
public void onReceive(Context ctx, Intent data) {

    // como tratar os diferentes casos ?
}
```

Temos diversas formas: podíamos verificar se ela possui alguma tag específica ou podíamos recuperar as duas e ver qual não está nula. Nós faremos de uma maneira mais elegante. Enviaremos junto com a *intent* um outro parâmetro avisando se a requisição deu certo ou não:

```

public class ReceptorDoServico extends BroadcastReceiver {

    public static void disparaSucesso(List<Livro> livros) {

        intent.putExtra("deuCerto", true);

    }

    public static void disparaErro(Throwable erro){

        intent.putExtra("deuCerto", false);

    }

}

```

Agora para tratarmos a resposta é bem mais fácil, buscamos apenas essa nova tag e já validamos o resultado, fazendo o tratamento correto para a condição que queremos:

```

public void onReceive(Context ctx, Intent data) {

    boolean deuCerto = data.getBooleanExtra("deuCerto", false);

    if(deuCerto){
        delegate.lidaComSucesso((List<Livro>) data.getSerializableExtra("livros"));
    } else {
        delegate.lidaComErro((Throwable) data.getSerializableExtra("erro"));
    }

}

```

Agora precisamos dar um jeito de termos o delegate e além disso atualizar o delegate quando a referência morrer. Toda vez que nossa Activity for criada vamos registrá-la como alguém interessada em ouvir o *broadcast* da listagem dos livros. Quando a Activity for destruída, removemos o registro dela para o *broadcast*. Dessa forma, somente a Activity que estiver ativa na tela do aplicativo ficará responsável por tratar a resposta da requisição. Vamos ver as alterações necessárias no código:

```

public class ReceptorDoServico extends BroadcastReceiver {

    private Delegate delegate;

    public static void começaOuvir(Delegate delegate){
        this.delegate = delegate;

        LocalBroadcastManager
            .getInstance(delegate.getContext())
            .registerReceiver(this, new IntentFilter("BananasDePijama"));
    }

    public static void paraDeOuvir(Delegate delegate){
        LocalBroadcastManager
            .getInstance(delegate.getContext())
            .unregisterReceiver(this);
    }

}

```

```
}
```

E em nossa Activity :

```
public class MainActivity extends AppCompatActivity implements LivrosDelegate {  
  
    public void onCreate(Bundle bundle) {  
        // código anterior  
  
        ReceptorDoServico.comecaOuvir(this);  
    }  
  
    public void onStop(){  
        super.onStop();  
  
        ReceptorDoServico.paraDeOuvir(this);  
    }  
}
```

Dessa forma deixamos nosso código bem mais flexível e desacoplado. Além de ser uma maneira bem elegante de tratar requisições.

3.5 FACILITANDO O CÓDIGO COM EVENTBUS

Acabamos de aprender algo muito útil para tratarmos requisições, que é a utilização do pattern `delegate` para desacoplarmos o código que recebe a resposta do código que faz o tratamento. Deixamos o tratamento da resposta desacoplada de qualquer cenário, bastando que quem quisesse receber a resposta fosse um `Delegate`.

Vimos que além de resolvermos o problema do `Delegate`, também podíamos ter problemas com a referência. Vimos que ainda que tenhamos desacoplado o uso do nosso `WebService`, ele ficava "preso" com quem o chamou. Sendo assim, no momento em que a resposta do servidor chega e é passada para a referência do `Delegate` morto, nosso aplicativo posso quebrar.

Para solucionarmos essa quebra por causa da referência, vimos uma forma de usar o do `Broadcast` do próprio `Android` sem deixarmos nossos dados expostos para todos aplicativos. Por isso fizemos uso do `LocalBroadcastManager`, que nos permitia fazer broadcasts internos, mantendo os dados de nossa aplicação seguros.

Para consumir esses dados, nós tínhamos que avisar que nosso `delegate` queria ouvir o evento, para isso registramos nosso `Delegate` como ouvinte do *broadcast*, e toda vez que ele era disparado, nosso `Delegate` estava a espera para fazer qualquer tratamento. Além disso, caso a referência morresse, nós apenas parávamos de ouvir o evento e então registrávamos o novo `Delegate` como ouvinte do *broadcast*.

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 seria um pouco

grande e trabalhosa.

Já entendemos a vantagem de possuir o `BroadcastReceiver` para tratar os eventos para gente, mas a manutenção a longo prazo pode ser ruim. Por esse motivo, existem outras formas simplificar isso, deixar esse comportamento encapsulado apenas para utilizarmos. Todo esse conceito ganhou o nome de **Eventbus**.

Existem várias implementações diferentes para **Eventbus**. Por exemplo, o *Otto*, pertencente à Squareup, a mesma que nos fornece o `Retrofit`. Usaremos em nosso projeto uma outra implementação, fornecida pela Greenrobot, que é a mais utilizada no mercado de trabalho.

A primeira coisa que temos que fazer é registrar nossa `Activity` como ouvinte do evento e da mesma forma como fazíamos com o *broadcast*, temos que parar de ouvir antes de nossa referência morrer. Temos uma classe especialista para fazer esse procedimento para nós, a própria classe `Eventbus`. Para pegarmos um objeto desse tipo, há o método `getDefault()`, que retorna uma referência que será única durante toda a aplicação. Agora vamos ver como começar a ouvir e também parar de ouvir os eventos que o `Eventbus` irá disparar:

```
public class MainActivity extends AppCompatActivity {

    public void onStart(){
        super.onStart();
        // vamos começar a ouvir o evento, por isso passamos o this
        EventBus.getDefault().register(this);
    }

    public void onStop(){
        super.onStop();
        // deixando claro que não queremos mais ouvir o evento
        EventBus.getDefault().unregister(this);
    }

}
```

Agora precisamos dar um jeito de enviar a resposta que recebemos do servidor, para que o próprio **EventBus** consiga disparar isso para quem está ouvindo. É bem simples fazermos isso, só precisamos chamar o comportamento que fará com que a resposta do servidor seja disparada. Para isso, faremos uso do método `post()`, que precisará receber por parâmetro os dados que serão trafegados. Vamos começar criando uma classe que possua os dados que queremos enviar:

```
public class LivroEvent {

    private final List<Livros> livros;

    public LivroEvent(List<Livros> livros) {
        this.livros = livros;
    }

}
```

Agora basta fazer o disparo do evento :

```
call.enqueue(new Callback<List<Livro>>() {
    @Override
    public void onResponse(Call<List<Livro>> call, Response<List<Livro>> response) {
        EventBus.getDefault().post(new LivroEvent(response.body()));
    }

    @Override
    public void onFailure(Call<List<Livro>> call, Throwable t) {
        EventBus.getDefault().post(new ThrowableEvent(t));
    }
});
```

Falta apenas conseguir capturar esse *post* que estamos realizando e seria bem interessante se conseguirmos reaproveitar o comportamento que deixamos com os métodos do *delegate* :

```
public void lidaComResposta(List<Livro> livros){
    //pega a lista e joga no adapter
}
```

O **EventBus** consegue nos ajudar a manter esse nosso código sem que precisemos mudar nada. Para isso, basta informarmos que este método é quem está interessado em receber o evento. Tudo que temos a fazer é utilizar uma anotação no método para que o recebimento do evento dispare a execução do método:

```
@Subscribe
public void lidaComResposta(List<Livro> livros){
    //pega a lista e joga no adapter
}
```

A única coisa que temos que alterar é o parâmetro que nosso método recebe, pois nosso *post* está enviando um *LivroEvent* portanto o método que quer tratar esse evento precisa estar esperando o mesmo objeto que está sendo enviado:

```
@Subscribe
public void lidaComResposta(LivroEvent event){
    List<Livro> livros = event.livros;
    //pega a lista e joga no adapter
}
```

Dessa forma, conseguimos deixar nosso código muito mais elegante e sem grandes problemas de manutenção.

3.6 EXERCÍCIOS

1. Vamos começar adicionar a dependência do *EventBus* . Utilize o atalho *Ctrl + shift + alt + s* , navegue até a aba de dependências, adicione uma nova lib. Procure por *greenrobot* e selecione *org.greenrobot:eventbus* .

2. Agora precisamos criar uma classe que vai representar o evento que vamos enviar com o EventBus :

```
public class LivroEvent {  
  
    private final List<Livro> livros;  
  
    public LivroEvent(List<Livro> livros) {  
        this.livros = livros;  
    }  
  
    public List<Livro> getLivros() {  
        return livros;  
    }  
  
}
```

3. Na classe WebClient , vamos alterar o código onde antes invocávamos o delegate pelo código de envio do EventBus :

```
public class WebClient {  
  
    // resto do código  
  
    public void getLivros() {  
  
        // resto do código  
  
        call.enqueue(new Callback<List<Livro>>() {  
            @Override  
            public void onResponse(Call<List<Livro>> call, Response<List<Livro>> response) {  
                EventBus.getDefault().post(new LivroEvent(response.body()));  
            }  
  
            @Override  
            public void onFailure(Call<List<Livro>> call, Throwable t) {  
                EventBus.getDefault().post(t);  
            }  
        });  
  
    }  
  
}
```

4. Aproveite para remover todas referências ao Delegate na classe WebClient incluindo o construtor.
5. Já estamos enviando nossos eventos mas precisamos fazer nossa MainActivity tratá-los. Vamos adaptar nossos método lidaComSucesso e lidaComErro para receberem os eventos do EventBus usando a anotação @Subscribe :

```
public class MainActivity extends AppCompatActivity implements LivrosDelegate {  
  
    // resto do código  
  
    @Subscribe  
    public void lidaComSucesso(LivroEvent livroEvent) {
```

```

        listaLivrosFragment.populaListaCom(livroEvent.getLivros());
    }

    @Subscribe
    public void lidaComErro(Throwable erro) {
        Toast.makeText(this, "Não foi possível carregar os livros...", Toast.LENGTH_SHORT).show();
    }
}

```

6. Como agora esses métodos não estão mais vindo do `LivrosDelegate`, remova esses métodos (`lidaComSucesso` e `lidaComErro`) da interface `LivrosDelegate` para que a `MainActivity` volte a compilar.
7. Para finalizar, registre a `MainActivity` no `EventBus` quando ela for criada e desregistre a mesma do `EventBus` quando a `MainActivity` for destruída. Para isso, você vai precisar usar os métodos do ciclo de vida correspondentes. Além disso, remova o parâmetro `this` da linha onde instanciamos o `WebClient` já que agora ele não precisa mais do `Delegate`:

```

public class MainActivity extends AppCompatActivity implements LivrosDelegate {

    // resto do código

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        // resto do código

        new WebClient().getLivros();

        EventBus.getDefault().register(this);
    }

    @Override
    protected void onDestroy() {

        EventBus.getDefault().unregister(this);
    }
}

```

8. Teste seu aplicativo novamente e verifique se ele consegue carregar os livros normalmente!

PICASSO - CARREGANDO IMAGENS DE FORMA SIMPLES

Nosso catálogo já mostra os livros mas ainda não estamos carregando as imagens das capas. Poderíamos fazer o código para carregar essas imagens na mão, utilizando múltiplas requisições e várias `AsyncTask` mas esse código além de complicado ficaria muito sujeito a erros e bastante repetitivo.

Como o carregamento de imagens é um problema bastante comum, podemos utilizar uma biblioteca que resolva o problema de forma mais simples. Uma boa escolha é a biblioteca Picasso. Com poucas

instruções, podemos disparar o carregamento de múltiplas imagens sem termos que nos preocupar com o gerenciamento das requisições e de suas *threads*.

Por exemplo, se quiséssemos carregar uma imagem utilizando Picasso, fariamos o seguinte código:

```
ImageView foto = (ImageView) findViewById(R.id.foto);
Picasso.with(this).load(urlDaFoto).placeholder(R.drawable.livro_generico).into(foto);
```

O método `load(...)` serve para especificar a url da imagem a ser carregada, o `placeholder(...)` para indicar a imagem que deve ser exibida até que o carregamento seja finalizado e o `into(...)` para informar qual a View onde a imagem deve ser carregada.

3.7 EXERCÍCIOS - CARREGANDO IMAGENS COM PICASSO

1. Para podermos carregar as imagens dos livros usaremos a biblioteca Picasso. Precisamos trazer a dependência para dentro de nosso projeto, para isso utilize o atalho `Ctrl + shift + alt + s` e da mesma forma que fizemos com `ButterKnife`, procure por `picasso` e selecione `com.squareup.picasso:picasso:2.5.2`.
2. Modifique o método `onBindViewHolder(...)` da classe `LivroAdapter` para fazer o carregamento da imagem dos livros utilizando a biblioteca Picasso. Siga o código abaixo como referência:

```
public class LivroAdapter extends RecyclerView.Adapter {

    // outros atributos e métodos

    @Override
    public void onBindViewHolder(RecyclerView.ViewHolder holder, int position) {

        // código anterior

        Picasso.with(viewHolder.foto.getContext())
            .load(livro.getUrlFoto())
            .placeholder(R.drawable.livro)
            .into(viewHolder.foto);
    }
}
```

3. Modifique também o método `populaListaCom(...)` da classe `DetalhesLivroFragment` para fazer o mesmo. Siga o código abaixo:

```
public class DetalhesLivroFragment extends Fragment {

    // outros atributos e métodos

    private void populaCamposCom(Livro livro) {

        // código que seta os outros campos

        Picasso.with(getContext()).load(livro.getUrlFoto()).placeholder(R.drawable.livro).into(foto);
    }
}
```

}

TRAZENDO APENAS AS INFORMAÇÕES QUE QUEREMOS

4.1 ENSINANDO O RETROFIT A FAZER REQUISIÇÕES DINÂMICAS

Nosso aplicativo já está exibindo uma lista que vem do nosso servidor mas no momento esta lista está apresentando uma quantidade fixa de livros. Quando o usuário terminar de percorrer esses itens, será necessário fazer uma nova requisição para mostrar mais livros. Vamos dar uma olhada no serviço que está trazendo os livros iniciais:

```
public interface ListaLivrosService {  
  
    @GET("listarLivros?indicePrimeiroLivro=0&qtdLivros=20")  
    Call<List<Livro>> listaLivros();  
  
}
```

Podemos ver que a quantidade de livros está limitada aos 20 primeiros livros. Algo que podíamos fazer é trazer todos os itens do servidor de uma vez mas se fizéssemos isso, estaríamos trafegando uma grande quantidade de dados que nesse exato momento não é necessária. Além disso, temos que pensar que podem ser bem mais do que apenas 100 livros e que processar tudo isso de uma vez, mesmo nos celulares mais potentes, poderia levar muito tempo ou até mesmo fazer que o aplicativo quebrasse por ficar sem memória para processar tudo.

Algo que podíamos fazer e que é bem comum em grandes aplicativos é trazer apenas uma pequena quantidade de livros de cada vez e depois trazer um pouco mais toda vez que o usuário terminar de visualizar os primeiros itens apresentados.

Para diminuir a quantidade de livros trazidos na requisição basta alterarmos o valor que passamos via parâmetro na requisição. Dessa forma, se quiséssemos trazer 10 livros, por exemplo, poderíamos alterar nossa serviço como abaixo:

```
public interface ListaLivrosService {  
  
    @GET("listarLivros?indicePrimeiroLivro=0&qtdLivros=10")  
    Call<List<Livro>> listaLivros();  
  
}
```

Agora nosso problema é outro: o que fazer quando o usuário terminar de visualizar esses 10

primeiros itens? Nesse caso, precisaríamos fazer uma nova requisição alterando o índice do primeiro livro. Mas como os parâmetros estão definidos diretamente na anotação do Retrofit, não conseguiríamos trocar esses parâmetros dinamicamente enquanto rodamos o aplicativo.

Como existe essa necessidade, o próprio `Retrofit` já nos ajuda bastando avisarmos o `Retrofit` que ele deve realizar a busca com os parâmetros de maneira dinâmica. Para isso, vamos alterar a assinatura do método da nossa serviço para receber os parâmetros da requisição, algo desse gênero:

```
public interface ListaLivrosService {  
  
    @GET("listarLivros?indicePrimeiroLivro=0&qtdLivros=10")  
    Call<List<Livro>> listarLivros(int indice, int qtd);  
  
}
```

Agora que já estamos recebendo os valores, precisamos passá-los para a requisição, para que ela seja executada de forma dinâmica. Tudo que precisamos fazer é deixar claro que os parâmetros que recebemos no método serão utilizados na requisição e quais são os campos que serão utilizados. Para isso, usaremos a anotação `@Query` do `Retrofit`:

```
public interface ListaLivrosService {  
  
    @GET("listarLivros")  
    Call<List<Livro>> listarLivros(@Query("indicePrimeiroLivro") int indice,  
                                  @Query("qtdLivros") int qtd);  
  
}
```

Agora o próprio `Retrofit` já vai se encarregar de criar a *url* da requisição com base nos parâmetros do método `listarLivros` que foram anotados com `@Query`.

4.2 APLICANDO O PADRÃO ENDLESSLIST

Com a nossa requisição mais dinâmica, podemos brincar um pouco e seguir o padrão de grandes aplicações que trabalham com listas com uma grande quantidade de itens, como o **Facebook**, por exemplo. No momento em que entramos no aplicativo, ele traz uma pequena quantidade de posts do servidor e quando o usuário chega próximo ao fim da lista, ele vai trazendo mais posts para que o usuário continue na aplicação, dando a impressão que o *feed* de notícias é **infinito**.

Esse comportamento recebeu o nome de `EndlessList`. Para implementá-lo em nossa lista de livros, reaproveitando o comportamento que fizemos para deixar as requisições dinâmicas, temos que verificar onde o usuário está na nossa listagem e dependendo do item que ele esteja vendo, temos que buscar mais itens para serem exibidos. Para fazer isso, temos que ficar ouvindo o *scroll* da listagem, para ver qual é o melhor momento para fazer a nova requisição:

```
@Override
```

```

    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.lista_livros_fragment, container, false);
        ButterKnife.bind(this, view);

        LinearLayoutManager manager = new LinearLayoutManager(getContext());
        listaLivros.setLayoutManager(manager);

        listaLivros.addOnScrollListener(new EndlessList());

        return view;
    }

```

Nesse momento estamos com problema de compilação já que a classe `EndlessList` não existe ainda, vamos criar essa classe e para que ela atenda ao parâmetro do método `addOnScrollListener()`, temos que fazer a classe estender de `RecyclerView.OnScrollListener` e como é uma classe abstrata somos obrigados a sobrescrever o método `onScrolled()` que ficará ouvindo o *scroll*:

```

public abstract class EndlessList extends RecyclerView.OnScrollListener {

    @Override
    public void onScrolled(RecyclerView recyclerView, int qtdScrollHorizontal, int qtdScrollVertical)
    {

        // logo implementaremos

    }

}

```

O método `onScrolled` recebe três parâmetros: o primeiro é o próprio `RecyclerView` ao qual o listener está associado, seguido de dois inteiros que representam a quantidade de scroll horizontal e vertical que foi dado no componente. Agora precisaremos checar alguns requisitos para podermos pegar novos itens em uma requisição.

Para conseguirmos validar quando devemos fazer uma nova busca, vamos precisar de vários dados que podem ser fornecidos pelo `LayoutManager` da nossa `RecyclerView`. Os dados que vamos pegar são: a quantidade total de itens da nossa lista, a quantidade de itens visíveis e o índice do primeiro item visível:

```

public abstract class EndlessList extends RecyclerView.OnScrollListener {

    private int quantidadeTotalItens;
    private int quantidadeItensVisiveis;
    private int primeiroItemVisivel;
    private LinearLayoutManager layoutManager;

    @Override
    public void onScrolled(RecyclerView recyclerView, int qtdScrollHorizontal, int qtdScrollVertical)
    {

        super.onScrolled(recyclerView, dx, dy);

        layoutManager = (LinearLayoutManager) recyclerView.getLayoutManager();
    }
}

```

```

        quantidadeTotalItens = layoutManager.getItemCount();
        quantidadeItensVisiveis = recyclerView.getChildCount();
        primeiroItemVisivel = layoutManager.findFirstVisibleItemPosition();
    }
}

```

Inicialmente, nossa lista vai estar vazia e já estaremos fazendo uma requisição para obter os dados. Toda vez que a lista for rolada precisamos saber se ocorreu alguma mudança na quantidade de itens na lista significando que novos itens foram carregados. Para controlar isso, vamos criar um outro atributo que será o total de itens anterior, caso esteja menor que quantidade atual, significa que foram carregados novos itens e aí temos que checar se está no momento de buscar novos:

```

public abstract class EndlessList extends RecyclerView.OnScrollListener {

    // todos os atributos anteriores

    private int totalAnterior = 0;

    @Override
    public void onScrolled(RecyclerView recyclerView, int qtdScrollHorizontal, int qtdScrollVertical)
    {
        // código anterior

        if (quantidadeTotalItens > totalAnterior) {
            totalAnterior = quantidadeTotalItens;
        }
    }
}

```

Também precisamos tomar cuidado para não ficarmos disparando várias requisições quando rolamos a lista. Vamos controlar mais esse caso criando um atributo que indicará se estamos carregando novos itens:

```

public abstract class EndlessList extends RecyclerView.OnScrollListener {

    // todos os atributos anteriores

    private boolean carregando = true;

    @Override
    public void onScrolled(RecyclerView recyclerView, int qtdScrollHorizontal, int qtdScrollVertical)
    {
        // código anterior

        if (carregando) {
            if (quantidadeTotalItens > totalAnterior) {
                carregando = false;
                totalAnterior = quantidadeTotalItens;
            }
        }
    }
}

```


Agora com todas as informações precisamos definir um limite de rolagem para quando devemos carregar mais itens. Isso deve acontecer quando o usuário estiver chegando no final da lista mas se esperarmos ele alcançar o final, o usuário precisará esperar o carregamento dos novos itens. Podemos antecipar um pouco o carregamento para melhorar a experiência do usuário. Sabemos que a lista de tem quantidadeTotalItens itens carregados e em uma tela conseguimos visualizar quantidadeItensVisiveis então podemos carregar mais itens quando o primeiroItemVisivel ultrapassar esse limite! Mais ainda, podemos disparar o carregamento um pouquinho antes, por exemplo, uns 5 itens antes desse limite. Vamos ver como tudo isso fica no código:

```
@Override
public void onScrolled(RecyclerView recyclerView, int dx, int dy) {

    // código anterior

    int indiceLimiteParaCarregar = quantidadeTotalItens - quantidadeItensVisiveis - 5;

    if (!carregando && primeiroItemVisivel >= indiceLimiteParaCarregar) {
        carregaMaisItens();
        carregando = true;
    }
}
```

Está faltando apenas definirmos o método carregaMaisItens() que será um método abstrato, para que possamos sempre reaproveitar essa mesma classe em outros projetos, sendo que como será carregado novos itens vai depender de projeto por projeto:

```
public abstract void carregaMaisItens();
```

Com isso temos apenas que implementar esse método no momento que adicionamos o *listener* na lista, dessa forma :

```
@Override
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.lista_livros_fragment, container, false);
    ButterKnife.bind(this, view);

    LinearLayoutManager manager = new LinearLayoutManager(getContext());
    listaLivros.setLayoutManager(manager);

    listaLivros.addOnScrollListener(new EndlessList() {
        @Override
        public void carregaMaisItens() {
            Snackbar.make(listaLivros, "Carregando mais itens", Snackbar.LENGTH_SHORT).show();
            webClient.retornaLivroDoServidor(livros.size(), 10);
        }
    });

    return view;
}
```

4.3 EXERCÍCIOS

1. Prepare a sua lista de livros para que ela avise quando ocorrer alguma rolagem. Isso vai permitir fazer o controle de quando buscar novos itens para exibir.

Para isso, crie uma classe `EndlessListListener` herdando a classe `RecyclerView.OnScrollListener` e associe o mesmo com a `RecyclerView` que representa a lista de livros no método `populaListaCom(...)` da classe `ListaLivrosFragment`.

Seu código deve ficar parecido com o abaixo:

```
public class ListaLivrosFragment extends Fragment {  
  
    // resto do código  
  
    public void populaListaCom(List<Livro> livros) {  
        // código anterior  
  
        recyclerView.addOnScrollListener(new EndlessListListener());  
    }  
}
```

2. Agora implemente o método `onScrolled(...)` na classe `EndlessListListener`. Este método será invocado sempre que a lista for rolada. Aproveite e mude o nome do parâmetro `dx` para `qtdScrollHorizontal` e do parâmetro `dy` para `qtdScrollVertical`:

```
public class EndlessListListener extends RecyclerView.OnScrollListener {  
  
    @Override  
    public void onScrolled(RecyclerView recyclerView, int qtdScrollHorizontal, int qtdScrollVertical) {  
        super.onScrolled(recyclerView, qtdScrollHorizontal, qtdScrollVertical);  
    }  
}
```

3. Para implementar o comportamento da %endless list%, vamos precisar recuperar algumas informações da lista. O primeiro passo é conseguir uma referência para o `LinearLayoutManager` da lista pedindo para a `recyclerView` que recebemos como parâmetro no método `onScrolled(...)`. Você pode utilizar o método `getLayoutManager(...)` da `recyclerView` mas você vai precisar fazer um *casting* para `LinearLayoutManager` pois esse método devolve apenas um `LayoutManager`:

```
@Override  
public void onScrolled(RecyclerView recyclerView, int qtdScrollHorizontal, int qtdScrollVertical)  
{  
    super.onScrolled(recyclerView, qtdScrollHorizontal, qtdScrollVertical);  
  
    LinearLayoutManager layoutManager = (LinearLayoutManager) recyclerView.getLayoutManager();  
}
```

4. Agora crie os atributos `quantidadeTotalItens` e `primeiroItemVisivel` na classe `EndlessListListener`. No método `onScrolled(...)`, popule esse dois atributos usando os

métodos `getItemCount(...)` e `findFirstVisibleItemPosition(...)` do `LayoutManager` . Use o código abaixo como referência:

```
// declaração dos atributos

@Override
public void onScrolled(RecyclerView recyclerView, int qtdScrollHorizontal, int qtdScrollVertical)
{
    // código anterior

    quantidadeTotalItens = layoutManager.getItemCount();
    primeiroItemVisivel = layoutManager.findFirstVisibleItemPosition();
}
```

5. Crie mais um atributo `quantidadeItensVisiveis` e popule esse atributo chamando o método `getChildCount(...)` da `recyclerView` . Aproveite e crie também um atributo booleano carregando que deve ser inicializado com `true` para indicar que inicialmente a lista está carregando novos itens. Depois usaremos esse atributo para controlar quando tivermos uma requisição em andamento.

```
// declaração dos atributos

@Override
public void onScrolled(RecyclerView recyclerView, int qtdScrollHorizontal, int qtdScrollVertical)
{
    // código anterior

    quantidadeItensVisiveis = recyclerView.getChildCount();
}
```

6. Implemente o comportamento de carregamento de novos itens no método `onScrolled(...)` . Primeiro crie uma variável local para indicar o índice limite de rolagem para disparar o carregamento dos itens.

```
int indiceLimiteParaCarregar = quantidadeTotalItens - quantidadeItensVisiveis - 5;
```

7. Agora faça uma verificação para descobrir se o primeiro item visível já passou do limite de carregamento. Além disso, certifique-se também de que a lista não está carregando novos itens nesse momento. O código ficará parecido com o abaixo:

```
@Override
public void onScrolled(RecyclerView recyclerView, int dx, int dy) {

    // código anterior

    if (!carregando && primeiroItemVisivel >= indiceLimiteParaCarregar) {
        carregaMaisItens(); // ainda vamos criar esse método
        carregando = true;
    }
}
```

8. Agora que você já cuidou do disparo das novas requisições, precisamos verificar toda vez que a lista for rolada se os novos itens já chegaram. Primeiro, crie um novo atributo `totalAnterior` e

inicialize esse atributo com 0 .

9. Faça uma nova verificação no método `onScrolled(...)` para checar se já existe uma requisição em andamento (usando o atributo `carregando`) e se existir verifique se a quantidade de itens na lista é maior que a quantidade de itens que haviam antes. Caso isso seja verdade, então atualize o `totalAnterior` com a nova quantidade de itens e marque que não estamos mais carregando novos itens. Seu código final do método `onScrolled(...)` deve ficar parecido com o abaixo:

```
public class EndlessListListener extends RecyclerView.OnScrollListener {

    private int quantidadeTotalItens;
    private int primeiroItemVisivel;
    private int quantidadeItensVisiveis;
    private boolean carregando = true;
    private int totalAnterior = 0;

    @Override
    public void onScrolled(RecyclerView recyclerView, int qtdScrollHorizontal, int qtdScrollVertical) {
        super.onScrolled(recyclerView, qtdScrollHorizontal, qtdScrollVertical);

        LinearLayoutManager layoutManager = (LinearLayoutManager) recyclerView.getLayoutManager();

        quantidadeTotalItens = layoutManager.getItemCount();
        primeiroItemVisivel = layoutManager.findFirstVisibleItemPosition();
        quantidadeItensVisiveis = recyclerView.getChildCount();

        int limiteLimiteParaCarregar = quantidadeTotalItens - quantidadeItensVisiveis - 5;

        if (carregando) {
            if (quantidadeTotalItens > totalAnterior) {
                totalAnterior = quantidadeTotalItens;
                carregando = false;
            }
        }
        if (!carregando && primeiroItemVisivel >= limiteLimiteParaCarregar) {
            carregaMaisItens();
            carregando = true;
        }
    }
}
```

10. Declare o método `carregaMaisItens(...)` na classe `EndlessListListener` e faça com que ele seja um método abstrato (lembre de tornar a classe abstrata também). Dessa forma, podemos aproveitar o comportamento da `%endless list%` e customizar somente o método `carregaMaisItens(...)` quando precisarmos reutilizar a classe em outros projetos.

```
public abstract class EndlessListListener extends RecyclerView.OnScrollListener {

    public abstract void carregaMaisItens();

}
```

11. Implemente o método `carregaMaisItens(...)` no momento em que instanciamos o `EndlessListListener` na classe `ListaLivrosFragment`. Na implementação você deve utilizar o `WebClient` para carregar os livros. Use o código abaixo como referência:

```

public class ListaLivrosFragment extends Fragment {

    // resto do código

    public View onCreateView(...) {

        // código anterior

        recyclerView.addOnScrollListener(new EndlessListListener() {
            @Override
            public void carregaMaisItens() {
                new WebClient().getLivros();
            }
        });
    }
}

```

12. Da forma como resolvemos no item anterior, estamos buscando todos os livros novamente. O ideal seria buscar apenas uma pequena quantidade de livros a mais. Para fazer essa alteração, modifique a classe `LivrosService` para que o método `listaLivros` receba o índice do primeiro livro e a quantidade de livros a serem buscados como parâmetros. O resultado deve ficar parecido com o código a seguir:

```

public interface LivrosService {

    @GET("listarLivros")
    Call<List<Livro>> listaLivros(@Query("indicePrimeiroLivro") int indicePrimeiroLivro,
        @Query("qtdLivros") int qtdLivros);

}

```

13. Modifique o método `getLivros(...)` da classe `WebClient` para que ela também receba os mesmos parâmetros (`indicePrimeiroLivro` e `qtdLivros`). Dentro do método, repasse esses parâmetros para a chamada do método `listaLivros(...)` que fazemos ao `service` :

```

public class WebClient {

    public void getLivros(int indicePrimeiroLivro, int qtdLivros) {

        // código anterior

        LivrosService service = client.create(LivrosService.class);
        Call<List<Livro>> call = service.listaLivros(indicePrimeiroLivro, qtdLivros);

        // resto do código

    }

}

```

14. Volte à classe `ListaLivrosFragment` e na implementação do `carregaMaisItens(...)` , adicione os 2 parâmetros que o método `listaLivros(...)` pede. O índice do primeiro livro a ser buscado vai ser igual a quantidade de livros que já temos na lista e a quantidade de livros a serem buscados

pode ficar fixada em 10. Para pegar a quantidade livros na lista você pode usar o método `size(...)` na lista de livros mas para fazer isso será necessário declarar o parâmetro `livros` do método `populaListaCom(...)` como `final`. Também apague a linha que limpava a lista de livros já que agora só queremos adicionar os novos livros que chegaram. Veja o código alterado:

```
public void populaListaCom(final List<Livro> livros) {
    // this.livros.clear(); <-- apague esta linha
    this.livros.addAll(livros);
    recyclerView.getAdapter().notifyDataSetChanged();

    recyclerView.addOnScrollListener(new EndlessListListener() {
        @Override
        public void carregaMaisItens() {
            new WebClient().getLivros(livros.size(), 10);
        }
    });
}
```

15. Finalmente, volte à classe `MainActivity` e adicione os parâmetros à chamada inicial ao `WebClient`. O índice do primeiro livro inicialmente é 0 e a quantidade de livros a serem buscados pode ser fixada novamente em 10:

```
public class MainActivity extends AppCompatActivity implements LivrosDelegate {

    @Override
    protected void onCreate(Bundle savedInstanceState) {

        // código anterior

        new WebClient().getLivros(0, 10);

        // resto do código
    }
}
```

16. Teste a sua aplicação e verifique se ela está carregando novos livros conforme você faz a rolagem da lista!

VENDENDO OS LIVROS

Agora que já temos o catálogo dos livros e a tela de detalhes, precisamos adicionar a funcionalidade de compra dos livros para o usuário. O processo que vamos implementar é bastante similar às lojas virtuais que encontramos na internet: primeiro o usuário adiciona todos os livros que quer comprar em um carrinho e depois finaliza a compra.



Figura 5.1: Venda do Livro

Poderíamos então começar modelando a classe `Carrinho` mas antes disso precisamos de uma

informação adicional. A Casa do Código vende seus livros em três modalidades:

- Livro físico
- Livro virtual
- Ambos livros

Então, quando formos colocar um livro no carrinho, precisamos salvar a informação do livro e também qual a modalidade escolhida. Por esse motivo, vamos criar uma classe auxiliar que vai representar cada `Item` que pode ser colocado no `Carrinho` :

```
public class Item {  
  
    private Livro livro;  
    private TipoDeCompra tipoDeCompra;  
  
    // getters e setters  
  
}
```

Precisamos definir a classe `TipoDeCompra` também, que é apenas um `enum` :

```
public enum TipoDeCompra {  
  
    VIRTUAL,  
    FISICO,  
    JUNTOS  
  
}
```

Agora falta apenas definirmos a class `Carrinho` que vai guardar a lista de `Item` s escolhidos pelo usuário. Além disso, vamos implementar todos os comportamentos de manipulação dos itens na própria classe para que os dados estejam seguros:

```
public class Carrinho implements Serializable {  
  
    private List<Item> itens = new ArrayList<>();  
  
    public void adiciona(Item item) {  
        itens.add(item);  
    }  
  
    public void remove(Item item) {  
        itens.remove(item);  
    }  
  
    public void limpa() {  
        itens.clear();  
    }  
  
    public List<Item> getItens() {  
        return Collections.unmodifiableList(itens);  
    }  
}
```



Além dessas classes de modelo, vamos precisar de uma nova tela para mostrar quais livros estão no carrinho. Além disso, temos que dar uma opção para que o usuário consiga acessar essa nova tela. Vamos preparar essa estrutura no próximo exercício.

5.1 EXERCÍCIOS

1. Crie uma nova activity `CarrinhoActivity` e altere o layout dessa activity para que ela utilize o layout `activity_carrinho.xml` que importamos no começo do curso, não esqueça de alterar o `AndroidManifest` registrando a nova activity.
2. Utilize o **ButterKnife** para injetar a `RecyclerView` com id `lista_itens_carrinho`. Faça o mesmo para o `TextView` com id `valor_carrinho`. Lembre-se de invocar o método `bind(...)` do **ButterKnife** no `onCreate(...)` da activity para que ele consiga injetar as referências para esses componentes. Use o código abaixo como referência:

```
public class CarrinhoActivity extends AppCompatActivity {

    @BindView(R.id.lista_itens_carrinho)
    RecyclerView listaItens;

    @BindView(R.id.valor_carrinho)
    TextView valorTotal;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_carrinho);
        ButterKnife.bind(this);
    }
}
```

3. Apesar de não termos como colocar itens no carrinho, vamos preparar nossa tela para que ela consiga exibir os itens futuramente. Declare um atributo do tipo `Carrinho` na `CarrinhoActivity`. Agora, implemente o método `carregaLista(...)` que deve usar o `ItemsAdapter` como adapter da `listaItens` para carregar os itens na lista. Não esqueça de setar também o `LinearLayoutManager` na `listaItens` com o método `setLayoutManager(...)`. Finalmente, invoque o método `carregaLista(...)` no método `onResume(...)` da `CarrinhoActivity`. Veja as alterações no código:

```
public class CarrinhoActivity extends AppCompatActivity {

    // código anterior

    private Carrinho carrinho = new Carrinho();

    public void carregaLista() {
        listaItens.setAdapter(new ItemsAdapter(carrinho.getItems(), this));
        listaItens.setLayoutManager(new LinearLayoutManager(this));
    }
}
```

```

@Override
protected void onResume() {
    super.onResume();
    carregaLista();
}
}

```

4. Também precisamos calcular o valor total da compra. Para isso, no método `carregaLista(...)`, percorra os itens do carrinho e calcule a soma de todos os itens. Atualize a view `valorTotal` com a soma calculada. Seu código deve ficar parecido com o abaixo:

```

public void carregaLista() {
    listaItens.setAdapter(new ItensAdapter(carrinho.getItens(), this));
    listaItens.setLayoutManager(new LinearLayoutManager(this));

    double total = 0;
    for (Item item : carrinho.getItens()) {
        total += item.getValor();
    }
    valorTotal.setText("R$ " + total);
}

```

5. Crie um menu `main_menu.xml` para a `MainActivity` contendo um único item com id `vai_para_carrinho` e com o ícone `carrinho_vazio`. Seu xml do menu deve ficar parecido com o abaixo:

```

<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/vai_para_carrinho"
        android:icon="@drawable/carrinho_vazio"
        android:title="Carrinho"
        app:showAsAction="always" />

</menu>

```

6. Modifique a `MainActivity` para que ela crie um menu a partir desse xml e implemente o código que faz a mudança de tela da `MainActivity` para a `CarrinhoActivity` ao clicar no item `vai_para_carrinho` do menu. As mudanças no código da sua activity devem ficar parecidas com as abaixo:

```

public class MainActivity extends AppCompatActivity {

    // código anterior

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main_menu, menu);
        return true;
    }

    @Override

```

```

        public boolean onOptionsItemSelected(MenuItem item) {
            if (item.getItemId() == R.id.vai_para_carrinho) {
                Intent vaiParaCarrinho = new Intent(this, CarrinhoActivity.class);
                startActivity(vaiParaCarrinho);
            }
        }
    }
}

```

7. Teste seu aplicativo e verifique se o menu está funcionando e você consegue acessar a tela do carrinho.

INTEGRANDO O CARRINHO COM O APLICATIVO

Nosso carrinho precisa ser acessado a partir de qualquer tela do nosso aplicativo e além disso precisa existir enquanto o aplicativo não for fechado pelo usuário. No mundo web temos um comportamento bem similar quando guardamos informações na própria sessão do usuário. No `Android`, podemos simular esse comportamento criando uma classe para representar a nossa **Aplicação**. A classe `Application` do `Android` é iniciada a partir do momento que você inicia o aplicativo e só é destruída quando o usuário mata a aplicação na mão ou quando o `Android` necessita de memória e encerra a aplicação.

Para podermos utilizar esse conceito, teremos que criar uma nova classe para representar nosso aplicativo e herdar de `Application`:

```

public class CasaDoCodigoApplication extends Application {

}

```

Agora vamos declarar o `Carrinho` dentro da classe `CasaDoCodigoApplication` para que ele fique disponível durante todo o tempo de vida da aplicação:

```

public class CasaDoCodigoApplication extends Application {

    private Carrinho carrinho = new Carrinho();

    // getter
}

```

Agora podemos criar uma nova tela onde veremos todos os livros que o usuário adicionou ao carrinho, exibindo as informações como valor, tipo do livro, valor total da compra e uma opção para finalizar a compra. O resultado seria algo parecido com:



Figura 5.2: Carrinho de Compras

CONFIGURANDO A NOSSA CLASSE APPLICATION

Para conseguirmos utilizar esse novo conceito, temos que deixar o `Android` ciente que não queremos usar a classe `'Application'` definida por padrão. Para isso, temos que fazer uma pequena configuração no **AndroidManifest.xml** :

```
android:name=".application.CasaDoCodigoApplication"
<!-- demais configurações -->
```

5.2 DESACOPLANDO OS OBJETOS COM DAGGER

Quando queremos utilizar o carrinho de compras em nossa `Activity` temos que fazer algo parecido com isso :

```

public class CarrinhoActivity extends AppCompatActivity {

    public void onCreate(Bundle bundle){
        // super e content view

        Carrinho carrinho = ((CasaDoCodigoApplication) getApplication()).getCarrinho();

        List<Itens> itens = carrinho.getItens();

        // adapter e demais views
    }
}

```

Agora para adicionarmos um livro tínhamos algo parecido com isto :

```

Carrinho carrinho = ((CasaDoCodigoApplication) getApplication()).getCarrinho();

carrinho.adiciona(item);

```

Vamos pensar que estamos utilizando o nosso carrinho nos dois `Fragments` que possuímos até o momento e na própria tela do carrinho. Temos três pontos onde ele é necessário e na medida que nosso aplicativo for crescendo, teremos mais telas e pode ser necessário acessar o carrinho a partir dessas telas também.

Para conseguirmos recuperar o carrinho, estamos sempre escrevendo o seguinte código:

```

Carrinho carrinho = ((CasaDoCodigoApplication) getApplication()).getCarrinho();

```

E se agora tivéssemos que passar algum parâmetro para dentro do método `getCarrinho()` ? Teríamos que passar em cada classe que utiliza o carrinho e atualizar o método, passando o parâmetro indicado. Isso acontece porque a responsabilidade de conseguir o carrinho está com cada classe que precisa dele. Como as nossas classes dependem do carrinho para funcionar, podemos passar a responsabilidade de fornecer essa dependência para algum outra classe ou para quem faz uso das nossas *activities*. Esse é o princípio da **inversão de controle**, onde a classe pede todas as suas dependências e alguém precisa fornecê-las. As dependências então podem ser fornecidas por quem instancia a nossa classe passando as dependências como parâmetro ou então atribuindo-as usando *setters*. Esse procedimento é chamado de **injeção de dependência**.

Para resolver esse problema, existem muitos *frameworks*, por exemplo, no Java temos a especificação **CDI** que faz esse papel. Geralmente, os frameworks de injeção de dependência fazem bastante uso de *reflection* mas no Android esse uso não é recomendado. Ainda assim, vários *frameworks* surgiram para o mundo Android, o **Dagger** da **SquareUp** foi muito eficiente durante muito tempo, ele realizava o mesmo papel que o **CDI** e não utilizava tanto *Reflection*. A **Google** achou esse projeto interessante e resolveu investir nele, com isso foi lançado o **Dagger 2** que não conta com nenhum tipo de *Reflection*.

PARA SABER MAIS : DAGGER 2

Todos os recursos do Dagger além de serem muito úteis no mundo Android podem ser muito bem aproveitados em qualquer projeto Java.

Para saber mais acerca desse projeto olhe a documentação que se encontra aqui:

<http://google.github.io/dagger/>

Quando usamos o Dagger em nossos projetos, não precisamos mais nos preocupar em com obter uma dependência. Podemos apenas declarar um atributo do tipo da dependência e pedir para que o Dagger injete esse atributo:

```
public class CarrinhoActivity extends AppCompatActivity {  
  
    @Inject  
    Carrinho carrinho;  
  
}
```

Também podemos obter o mesmo resultado fazendo a injeção via *setter*:

```
public class CarrinhoActivity extends AppCompatActivity {  
  
    private Carrinho carrinho;  
  
    @Inject  
    void setCarrinho(Carrinho carrinho) {  
        this.carrinho = carrinho;  
    }  
  
}
```

A diferença é que na segunda abordagem mantemos o atributo `carrinho` encapsulado enquanto na primeira o atributo pode ser acessado por classes no mesmo pacote do `Carrinho`.

Por enquanto, apenas marcamos nosso atributo como uma dependência que deve ser injetada pelo Dagger. Agora precisamos pedir para que o Dagger injete todas dependências do nosso objeto. Para que ele consiga fazer isso vamos precisar definir em algum lugar como construir ou obter as dependências. No Dagger, fazemos isso criando um módulo que basicamente é uma classe com métodos que produzem as nossas dependências. Nesse caso, vamos criar então uma classe `CasaDoCodigoModule` e anotar essa classe com `@Module` para indicar para o Dagger que essa classe pode ser utilizada por ele para produzir dependências:

```
@Module  
public class CasaDoCodigoModule {
```

```
}
```

Agora nossa classe tem que saber como fornecer os objetos que serão injetados, usamos a anotação `@Provides` para indicar que nosso método irá fornecer o tipo esperado. Além disso, vamos anotar esse método com `@Singleton` para indicar ao Dagger que o `Carrinho` fornecido por ele deve ser o mesmo para qualquer componente do nosso aplicativo e que ele não precisa ficar criando novas instâncias do `Carrinho` para cada tela:

```
@Module
public class CasaDoCodigoModule {

    @Provides
    @Singleton
    public Carrinho getCarrinho() {
        return new Carrinho();
    }
}
```

Já indicamos ao Dagger quais objetos devem ser injetados e também como fornecê-los mas precisamos lembrar que para ler as anotações o Dagger precisaria utilizar o recurso de *Reflection* que já sabemos que ele não usa. Se ele não usa *Reflection*, então o Dagger tem que ser capaz de gerar o código que faz a injeção das dependências no momento em que vamos iniciar a compilação do nosso aplicativo.

Por exemplo, vamos precisar injetar o `Carrinho` em nossa `CarrinhoActivity` chamando algum método do Dagger para fazer isso. O problema é que esse método de injeção não existe! Ele vai depender do que precisa ser injetado e o código desse método precisa ser gerado antes da compilação para não depender da *Reflection*.

Esse problema foi resolvido no Dagger 2 definindo-se uma interface onde indicamos para o Dagger todos os objetos que poderão sofrer injeção. Nessa interface vamos especificar quais são os módulos que serão utilizados para fornecer as dependências e também vamos definir um método `inject(...)` recebendo como parâmetro o objeto que precisa de injeção. Vamos ver como fica o código:

```
@Singleton
@Component(modules = CasaDoCodigoModule.class)
public interface CasaDoCodigoComponent {

    void inject(CarrinhoActivity activity);
}
```

Agora precisamos fazer com que o Dagger crie automaticamente uma classe que implemente essa interface com esse método `inject(...)`. Quando fizermos o build do nosso aplicativo e existir pelo menos um `@Component` do Dagger, automaticamente será gerada uma classe com o nome `DaggerCasaDoCodigoComponent` (prefixo `Dagger` e sufixo nome do componente que criamos). Essa classe fornece um `Builder` capaz de gerar uma implementação da nossa interface que podemos utilizar

em qualquer parte da nossa aplicação para fazer a injeção das dependências. Como vamos precisar fazer a injeção em vários pontos do aplicativo, vamos construir essa implementação na nossa classe `CasaDoCodigoApplication` e criar um *getter* para facilitar seu uso:

```
public class CasaDoCodigoApplication extends Application {

    private CasaDoCodigoComponent component;

    @Override
    public void onCreate() {
        super.onCreate();

        component = DaggerCasaDoCodigoComponent
            .builder()
            .build();
    }

    public CasaDoCodigoComponent getComponent() {
        return component;
    }
}
```

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

```
public class CarrinhoActivity extends AppCompatActivity {

    @Inject
    Carrinho carrinho;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        CasaDoCodigoApplication application = (CasaDoCodigoApplication) getApplication();
        CasaDoCodigoComponent component = application.getComponent();
        component.inject(this);

        // A partir daqui podemos usar o carrinho sem problemas!
    }
}
```

Desta forma deixamos nossos objetos complexos muito mais desacoplados, sem nos preocuparmos em como conseguir as dependências. Um outro exemplo de uso seria fornecer todos os serviços do *Retrofit* que estamos utilizando até o momento com a injeção de dependências. Apenas dizemos que nossa `Activity` precisa de um serviço e o nosso módulo fica responsável por fornecer esse serviço invocando os métodos do *Retrofit*.

Para saber mais

CONFIGURAÇÃO DO DAGGER 2

Como qualquer biblioteca externa, temos que importá-la :

```
compile 'com.google.dagger:dagger:2.8'
```

Da mesma forma que o **ButterKnife**, temos que trazer o gerador de código do Dagger, portanto será necessário adicionar mais esta linha:

```
apt 'com.google.dagger:dagger-compiler:2.8'
```

Exercícios

1. Importe as bibliotecas do Dagger. Para isso abra o arquivo build.gradle do módulo e adicione essas linhas dentro do contexto de dependências :

```
dependencies {  
    compile 'com.google.dagger:dagger:2.8'  
    apt 'com.google.dagger:dagger-compiler:2.8'  
}
```

2. Defina um novo módulo do Dagger que ficará responsável por fornecer uma nova instância da classe `Carrinho` quando precisarmos injetá-lo em nossas telas. Para isso, crie uma classe `CasaDoCodigoModule`, anote a classe com `@Module` e implemente o método `getCarrinho()`. O método deve ser anotado com `@Provides` e `@Singleton` e deve devolver uma nova instância do `Carrinho`. Use o código abaixo como referência:

```
@Module  
public class CasaDoCodigoModule {  
  
    @Provides  
    @Singleton  
    public Carrinho getCarrinho() {  
        return new Carrinho();  
    }  
}
```

3. Agora defina o componente do Dagger onde definiremos quais classes onde será necessário utilizar a injeção de dependências. Como vamos precisar utilizar o `Carrinho` na activity `CarrinhoActivity` e no fragment `DetalhesLivroFragment`, vamos precisar informar isso ao Dagger. Para isso, crie uma **interface** `CasaDoCodigoComponent` e anote-a com `@Component(modules = CasaDoCodigoModule.class)`. Nessa interface defina os métodos

`inject(CarrinhoActivity activity)` e `inject(DetalhesLivroFragment fragment)` para indicarmos que vamos precisar injetar o `Carrinho` nesses objetos:

```
@Singleton
@Component(modules = CasaDoCodigoModule.class)
public interface CasaDoCodigoComponent {

    void inject(DetalhesLivroFragment fragment);
    void inject(CarrinhoActivity activity);

}
```

4. Para conseguir fazer a injeção das dependências vamos precisar de uma implementação do nosso componente do Dagger.

Crie a classe `CasaDoCodigoApplication` estendendo a classe `Application` do Android. Nessa classe, declare um atributo `component` do tipo `CasaDoCodigoComponent`. Para popular esse atributo, no método `onCreate(...)` instancie o `component` usando a classe gerada automaticamente pelo Dagger `DaggerCasaDoCodigoComponent`. Você pode utilizar o método `builder()` dessa classe e depois invocar o método `build()` no retorno do primeiro método. Se a classe ainda não existir, selecione a opção de menu **Build > Make Project** no Android Studio. Por fim, crie um *getter* para o atributo `component`.

```
public class CasaDoCodigoApplication extends Application {

    private CasaDoCodigoComponent component;

    @Override
    public void onCreate() {
        super.onCreate();

        component = DaggerCasaDoCodigoComponent.builder().build();
    }

    public CasaDoCodigoComponent getComponent() {
        return component;
    }

}
```

5. Associe a classe `CasaDoCodigoApplication` com a `application` definida no `AndroidManifest.xml`. Para isso, adicione o atributo `name` na tag `<application>`:

```
<application
    android:name=".CasaDoCodigoApplication"
    ... >
```

6. Agora temos que deixar o usuário adicionar novos itens no carrinho. Para isso, altere o `DetalhesLivroFragment` para que ele receba o `Carrinho` injetado pelo Dagger. Use o código abaixo como referência:

```
public class DetalhesLivroFragment extends Fragment {
```

```

// todos atributos anteriores

@Inject
Carrinho carrinho;

@Nullable
@Override
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
    @Nullable Bundle savedInstanceState) {

    // código anterior

    CasaDoCodigoApplication app = (CasaDoCodigoApplication) getActivity().getApplication();
    app.getComponent().inject(this);

    return view;
}

// resto da classe
}

```

- Use o **ButterKnife** para tratar o clique nos botões de comprar livro físico, e-book ou ambos. Quando o clique ocorrer em um desses botões, você deve instanciar um novo `Item` com o livro e tipo especificados e adicioná-lo ao carrinho. Mostre também um `Toast` para indicar que o livro foi adicionado. Veja o exemplo abaixo:

```

public class DetalhesLivroFragment extends Fragment {

    // resto da classe

    @OnClick(R.id.detalhes_livro_comprar_fisico)
    public void comprarFisico() {
        Toast.makeText(getActivity(), "Livro adicionado ao carrinho!", Toast.LENGTH_SHORT).show();
    }

    carrinho.adiciona(new Item(livro, TipoDeCompra.FISICO));

    // repita o mesmo com os outros dois botões
}

```

- Vamos fazer algo bem parecido agora na tela `CarrinhoActivity`. Altere o atributo `carrinho` para que ele não seja instanciado junto com a sua declaração. Remova também o modificador `private` e adicione a anotação `@Inject` para indicarmos que esse atributo será injetado pelo Dagger. Depois, invoque o componente da aplicação disparar a injeção de dependências pelo Dagger. Veja o código de exemplo:

```

public class CarrinhoActivity extends AppCompatActivity {

    // outros atributos

    @Inject
    Carrinho carrinho;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {

```

```
// código anterior

CasaDoCodigoApplication app = (CasaDoCodigoApplication) getApplication();
app.getComponent().inject(this);
}

// resto da classe
}
```

TESTES AUTOMATIZADOS COM ESPRESSO

Quanto tempo nós levamos para ver que nosso aplicativo está funcionando da maneira que esperamos ? Qual é o procedimento que fazemos para validar tudo ?

Temos que manualmente entrar no aplicativo e fazer a rota que o usuário faria, validando se os componentes estão aparecendo na ordem correta.

Isto é um procedimento muito caro, pois acabamos perdendo bastante tempo para subir o aplicativo no emulador e ainda temos que compartilhar a memória com outros processos que estão sendo executados no computador, caso tenhamos um celular para que possamos testar ganhamos um pouco mais de tempo, mas ainda assim temos que parar tudo para validar se o comportamento é igual ao esperado.

Este procedimento que fazemos tem um nome bem conhecido, **teste de aceitação**. Para conseguirmos fazer este tipo de teste ficar automatizado no android, temos um *framework* do Google chamado Espresso , que irá simular o usuário utilizando o aplicativo, para isso temos encontrar as *View* s na tela, fazemos isso através do método `onView()` , mas podemos ter inúmeras *View* s na tela, logo precisamos dar um jeito de pegar exatamente a que queremos, por isso precisamos passar um parâmetro do tipo `ViewMatchers` que servirá como filtro, temos diversas opções as mais comuns são :

- `withId` - passamos o id que deverá ser buscado no xml
- `withText` - passamos o texto que será buscado, caso ter dois ele quebra
- `withHint` - muito usado para testar formulários, buscando o campo com seu *placeholder*

Toda vez que realizamos uma busca, ele nos retorna um objeto do tipo `ViewInteraction` que é a representação da *View* , com este objeto nós conseguimos interagir, com ações : click, escrita, entre outros.

```
ViewInteraction btn = onView(  
    withId(R.id.btn));  
  
btn.perform( click() );
```

Se tentarmos executar o mesmo teste em uma lista, nosso teste falhará, pois no momento em que estiver executando os testes, nenhum objeto foi colocado dentro da listagem, nenhuma *View* foi inflada

ainda. Para solucionar este problema, usaremos outro método, `onData`, que sabe manipular a lista além de ficar atento aguardando as views serem adicionadas. Como podemos ter várias listas na mesma tela, precisamos ter um filtro mais sagaz, para isso usaremos filtros específicos também :

```
DataInteraction dataInteraction = onData(  
    allOf( is( instanceof( SEU_OBJETO_AQUI.class ) )  
);  
  
dataInteraction.atPosition(0)  
    .perform(click());
```

PARA SABER MAIS

Quando usamos *Recycler View*, instintivamente usaríamos o `onData` para executar nosso teste, mas infelizmente a implementação desse método não serve, e para fazermos nossos testes temos que usar o `onView`. Para fazermos nossos testes temos uma classe especialista para isso :

RecyclerViewActions

Para conhecer mais sobre os métodos e os usos, você pode consultar a documentação oficial do espresso :

<https://google.github.io/android-testing-support-library/docs/espresso/>

6.1 EXERCÍCIOS

1. Utilize o atalho `ctrl + shift + a` que irá ajudar a criar nosso teste, ele irá pegar o comportamento que queremos testar e já criará todo código de teste, utilizando as classes corretas. Monte alguns testes !

UTILIZANDO FIREBASE

7.1 FIREBASE AUTHENTICATION

A idéia do nosso aplicativo é realizar a venda dos nossos livros, para isso já estamos colocando no carrinho mas falta enviar as informações da compra para o servidor. Entretanto, para computarmos a venda daqueles livros é interessante saber quem está comprando. Podemos resolver isso enviando o e-mail do cliente juntamente com os livros que ele quer comprar:

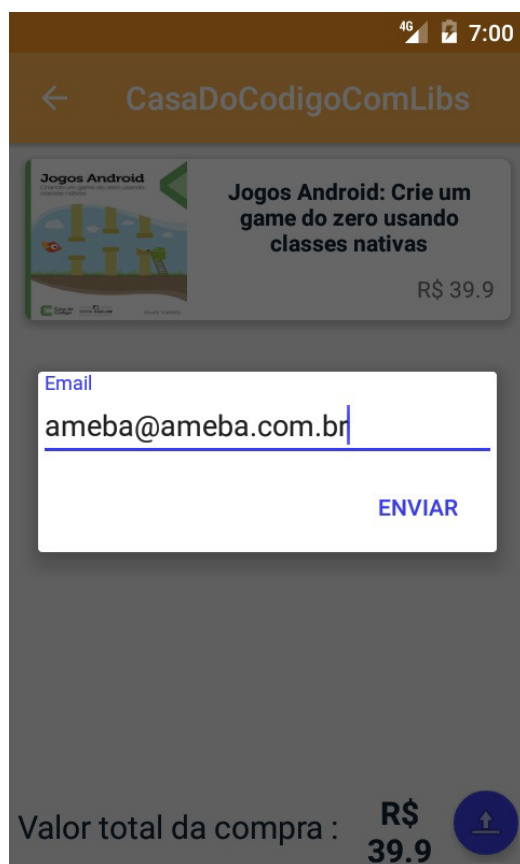


Figura 7.1: Finalização de compra

Se fizermos dessa forma, nosso problema estará resolvido, contudo podemos gerar outros que não esperávamos. Por exemplo, alguém pode fazer uma compra em nome de outro usuário. Uma forma de evitarmos isso seria exigir que cada usuário estivesse logado no aplicativo, para evitar qualquer inconsistência.

Podemos então criar uma tela de login para que nossos usuários possam se conectar devidamente em nosso aplicativo, poderia ser algo bem próximo desta tela:



Figura 7.2: Tela de login

A ação do botão *entrar* deve validar se as informações fornecidas nos campos condizem com as informações que existem no servidor, assim o usuário teria acesso ao restante da aplicação. Além de ter que verificar a cada vez que o usuário entra na aplicação se ele já está logado ou se devemos pedir novamente as informações de login. Fazer todo esse procedimento é bem trabalhoso e para cada aplicação teríamos que fazer o mesmo processo exatamente da mesma forma. Para facilitar esse procedimento a *Google* disponibiliza o serviço, **gratuito**, chamado de **Firebase Authentication**, que fará toda validação dos dados que enviaremos, desde que tenhamos esses dados na base de dados da Google. O bacana é que com esse serviço já conseguimos dar um passo para realizar o login com outros serviços como *Facebook*, *Twitter*, *Github*, *Google*.

Para começarmos a utilizar o `Firebase` em nossa aplicação, a própria IDE nos auxilia. Para isso podemos acompanhar o tutorial que encontramos no Android Studio.

Em `Tools` temos a opção `Firebase`. Basta clicarmos nesta opção que abrirá um assistente exibindo todos os módulos disponíveis deste serviço. Falaremos sobre alguns deles mais tarde no curso. Por enquanto o que nos interessa é a questão de login portanto buscaremos por **Authentication**:

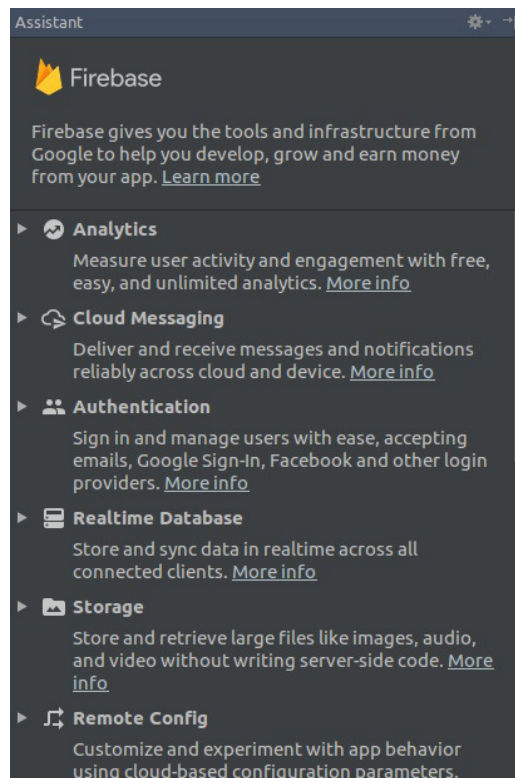


Figura 7.3: Assistente do Firebase

Ao clicarmos em **Authentication**, teremos a seguinte opção: Email and password authentication, que indica que autenticação do usuário será feita usando-se apenas o email e uma senha. Além disso, ele persistirá essas informações até que o usuário peça para se deslogar da aplicação.

Precisamos vincular nossa aplicação com o Firebase e no próprio assistente teremos o passo a passo necessário para isso e de forma bem simplificada, tudo isso com um simples clique:

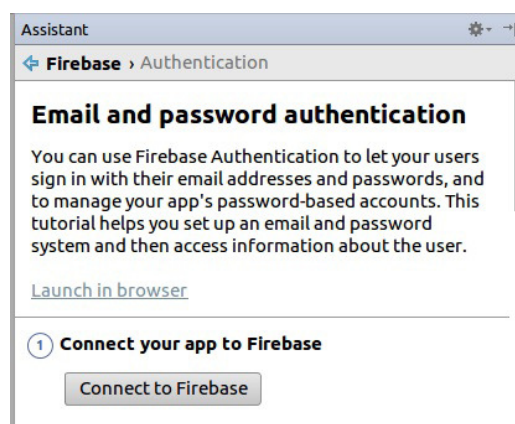


Figura 7.4: Conectando aplicativo com o Firebase

Para concluir o processo, precisamos vincular nossa conta de desenvolvedor Google ao Firebase e criar um projeto. Com esse clique ele apenas pedirá para nos conectarmos com a conta Google e ele

cuidará do resto.

A segunda etapa é trazer todas as dependências necessárias para nosso aplicativo, no próprio assistente temos a opção para fazer o processo automaticamente:

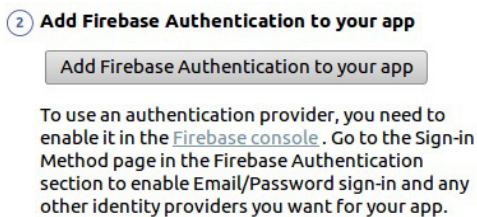


Figura 7.5: Configurando as dependências

Antes de começarmos a falar do código, temos apenas mais uma etapa que é deixar esse módulo ativo para o nosso projeto. Para isso, temos que ir ao console online do firebase, que podemos encontrar nesse link : *console.firebase.google.com*

No console teremos todas as configurações e ações que o Firebase fornece na barra do lado esquerdo. Lá buscaremos pelo módulo que queremos usar:

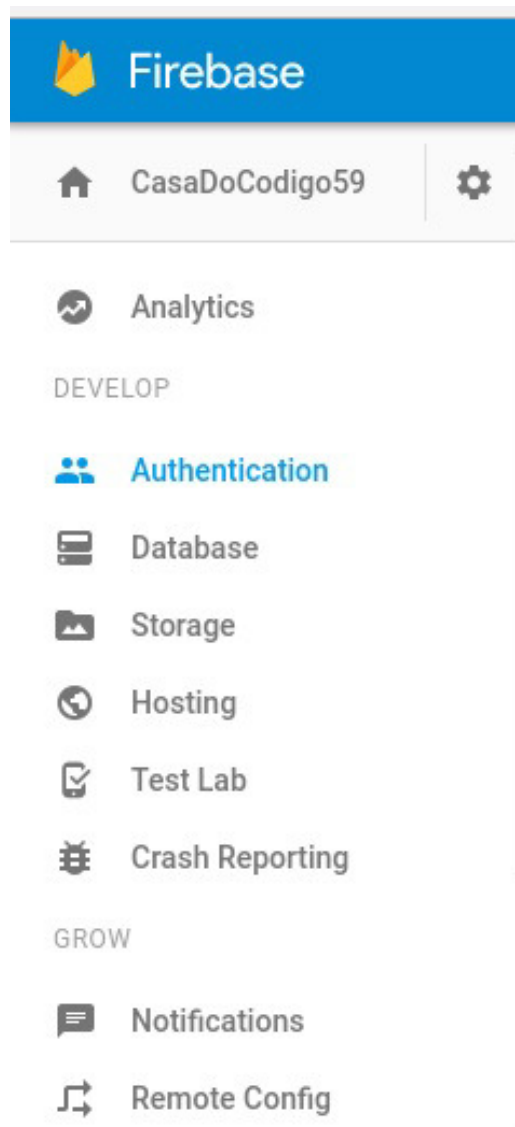


Figura 7.6: Serviços do Firebase no console

Teremos várias opções para optar mas inicialmente vamos habilitar apenas a autenticação através de login e senha. Repare que temos algumas outras escolhas para poder executar a autenticação:







Provedores de login	
Provedor	Status
 E-mail/senha	Ativado
 Google	Desativado
 Facebook	Desativado
 Twitter	Desativado
 GitHub	Desativado
 Anônimo	Desativado

Figura 7.7: Tipos de autenticação

Após termos feito toda configuração, podemos começar a mexer com código e realizar o login em nossa aplicação. Para realizarmos o gerenciamento da autenticação do usuário, o Firebase fornece um objeto que nos auxiliará: `FirebaseAuth`. Para conseguirmos um objeto deste tipo precisamos fazer o seguinte:

```
public class LoginActivity extends AppCompatActivity {

    private FirebaseAuth firebaseAuth;

    public void onCreate(Bundle bundle) {

        //código anterior

        firebaseAuth = FirebaseAuth.getInstance();

    }
}
```

Agora quando o usuário clicar no botão para entrar na aplicação temos apenas que usar este nosso objeto para fazer o login dele. Para isso, invocaremos o método `signInWithEmailAndPassword()`, onde passamos o email e senha respectivamente. Este método ficará responsável por fazer uma chamada *assíncrona* para o servidor de autenticação do Firebase. Como a chamada é assíncrona, precisamos ficar ouvindo seu retorno, por isso adicionaremos um listener para sermos notificados quando a chamada estiver completa. Para adicionar este listener temos o método `addOnCompleteListener()`, que recebe uma `Activity`, que servirá para passar o escopo do listener, e a implementação da interface `OnCompleteListener`, que representa o comportamento que deve ser executado quando a requisição terminar:

```
@OnClick(R.id.login_logar)
public void login() {

    String email = this.email.getText().toString().trim();
    String senha = this.senha.getText().toString().trim();

    if (!email.isEmpty() || !senha.isEmpty()) {

        firebaseAuth.signInWithEmailAndPassword(email, senha)
            .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
                @Override
                public void onComplete(@NonNull Task<AuthResult> task) {
                    Log.d(TAG, "signInWithEmail:onComplete:" + task.isSuccessful());

                    if (!task.isSuccessful()) {
                        Log.w(TAG, "signInWithEmail", task.getException());
                        Snackbar.make(LoginActivity.this.email, "Acesso não autorizado, verifique suas informações",
                                                                    Snackbar.LENGTH_SHORT).show();
                    }
                }
            })
    }
}
```

```

        });
    } else {
        Snackbar.make(this.senha, "Por favor complete todos os campos", Snackbar.LENGTH_SHORT)
            .show();
    }
}
}

```

Além de enviarmos a requisição de login, precisamos também ficarmos ouvindo as mudanças de estado com relação a autenticação, isto é, se o usuário realmente conseguiu se logar ou se houve algum erro. Para isso criaremos um listener que estará vinculado as ações do `FirebaseAuth`, para representar este listener temos a classe `FirebaseAuth.AuthStateListener`. Podemos criá-lo da mesma forma que fazemos com os listeners normais bastando instanciá-lo e já fornecer o comportamento que ele deverá executar:

```

private FirebaseAuth.AuthStateListener listener;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_login);
    ButterKnife.bind(this);

    firebaseAuth = FirebaseAuth.getInstance();

    listener = new FirebaseAuth.AuthStateListener() {
        @Override
        public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
            // comportamento do listener
        }
    };
}

```

Como esse listener servirá para validar se o usuário já realizou login, caso tenha tido sucesso devemos redirecionar ele para nossa tela principal da aplicação. Para sabermos se já foi realizado login alguma vez, o `FirebaseAuth` que realizou o login, poderá nos ajudar, pois podemos recuperar o usuário vigente através do método `getCurrentUser()`. Com isso podemos validar se temos ou não um usuário logado no sistema:

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_login);
    ButterKnife.bind(this);

    firebaseAuth = FirebaseAuth.getInstance();

    listener = new FirebaseAuth.AuthStateListener() {
        @Override
        public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
            FirebaseUser user = firebaseAuth.getCurrentUser();
        }
    };
}

```

```

        if (user != null) {
            // existe usuário
        }
    }
};

}

```

Caso possua um usuário não é necessário que ele se logue novamente, então podemos direcionar ele para o que interessa que é a próxima tela:

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_login);
    ButterKnife.bind(this);

    firebaseAuth = FirebaseAuth.getInstance();

    listener = new FirebaseAuth.AuthStateListener() {
        @Override
        public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
            FirebaseUser user = firebaseAuth.getCurrentUser();

            if (user != null) {
                vaiParaMain();
            }
        }
    };
};

}

```

Como esse listener pode ser invocado várias vezes pelo Firebase e o usuário pode já ter sido logado, precisamos validar se já caímos alguma vez nesse if e se sim temos que impedir que caia novamente e envie o usuário duas vezes para a tela principal do aplicativo. Vamos corrigir esse comportamento criando uma *flag* que validará isso por nós:

```

private boolean flagUsuarioLogado = false;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_login);
    ButterKnife.bind(this);

    firebaseAuth = FirebaseAuth.getInstance();

    listener = new FirebaseAuth.AuthStateListener() {
        @Override
        public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
            FirebaseUser user = firebaseAuth.getCurrentUser();

            if (user != null && !flagUsuarioLogado) {

```

```

        flagUsuarioLogado = true;
        vaiParaMain();
    }
};

}

```

Depois de termos configurado devidamente nosso listener, precisamos falar deixar ele vinculado com nosso `FirebaseAuth` com o método `addAuthStateListener()` :

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    // código anterior

    firebaseAuth.addAuthStateListener(listener);
}

```

PARA SABER MAIS

Caso queira ver como logar com as redes sociais usando o `Firebase` visite a documentação :

Facebook : https://firebase.google.com/docs/auth/android/facebook-login?utm_source=studio

Twitter : <https://firebase.google.com/docs/auth/android/twitter-login>

Google : https://firebase.google.com/docs/auth/android/google-signin?utm_source=studio

Github : <https://firebase.google.com/docs/auth/android/github-auth>

É interessante enviar as informações como tokens e ids que o `Firebase` gerencia para o servidor, para que possamos trabalhar de maneira mais isolada.

Exercícios

1. Vamos começar criando a tela de login. O layout dessa tela já está pronto e disponível no seu projeto então apenas crie a `LoginActivity` sem gerar um novo arquivo de layout.
2. Não vamos precisar da `ActionBar` nessa tela então vá até o `AndroidManifest.xml` e altere a `LoginActivity` para que ela utilize um tema que não possui `ActionBar` . Além disso, faça com que a `LoginActivity` seja a activity principal e limite a orientação dessa tela para o modo *portrait*. As tags no manifest devem ficar como abaixo:

```

<activity
    android:name=".activity.LoginActivity"
    android:screenOrientation="portrait"
    android:theme="@style/Theme.AppCompat.Light.NoActionBar">

```

```

<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>

```

3. Agora vamos ativar o módulo de autenticação do Firebase no projeto. Vá até o menu **Tools > Firebase**. Na aba lateral que abrirá em seguida, abra a seção **Authentication** e clique em **Email and password authentication**.
4. Clique em **Connect to Firebase** para se logar no Firebase e criar um novo projeto. Mude o nome do projeto para `CasaDoCodigo` e altere a região para `Brazil`. Em seguida, clique em **Connect to Firebase**. Aguarde até que o Android Studio confirme a criação do projeto.
5. Clique em **Add Firebase Authentication to your app** para adicionar as dependências do Firebase para o módulo de autenticação. Um pop-up será exibido mostrando as dependências a serem adicionadas e basta clicar em **Accept Changes** para efetivá-las no projeto.
6. Vamos preparar o código para disparar o evento de login. Primeiro utilize o ButterKnife para fazer o bind das views da tela de login. Em seguida, crie um atributo do tipo `FirebaseAuth` e no `onCreate(...)` inicialize este atributo chamando o método `Firebase.getInstance()`. Sua `LoginActivity` deve estar parecida com a abaixo:

```

public class LoginActivity extends AppCompatActivity {

    @BindView(R.id.login_email)
    private EditText campoEmail;

    @BindView(R.id.login_senha)
    private EditText campoSenha;

    private FirebaseAuth firebaseAuth;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login);
        ButterKnife.bind(this);

        firebaseAuth = FirebaseAuth.getInstance();
    }
}

```

7. Ao clicar no botão de logar, leia o valor dos campos de e-mail e senha e se ambos estiverem preenchidos invoque o método `signInWithEmailAndPassword(...)` do `firebaseAuth` passando os valores do e-mail e senha. Adicione um `OnCompleteListener(...)` e trate os eventos necessários. Siga a implementação abaixo como referência:

```

@OnClick(R.id.login_logar)
public void logar() {
    String email = campoEmail.getText().toString();
    String senha = campoSenha.getText().toString();

    if (!email.isEmpty() || !senha.isEmpty()) {

```



```

        firebaseAuth.signInWithEmailAndPassword(email, senha)
            .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
                @Override
                public void onComplete(@NonNull Task<AuthResult> task) {
                    if (!task.isSuccessful()) {
                        Snackbar.make(campoEmail, "Acesso não autorizado, verifique suas informações"
, Snackbar.LENGTH_SHORT).show();
                    }
                }
            });
    } else {
        Snackbar.make(campoSenha, "Por favor complete todos os campos", Snackbar.LENGTH_SHORT).sh
ow();
    }
}

```

8. Crie um atributo `listener` do tipo `AuthStateListener` e instancie esse objeto no `onCreate(...)` da `LoginActivity`.

```

private FirebaseAuth.AuthStateListener listener;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_login);
    ButterKnife.bind(this);

    firebaseAuth = FirebaseAuth.getInstance();

    listener = new FirebaseAuth.AuthStateListener() {
        @Override
        public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {

        }
    };
}

```

9. No método `onAuthStateChanged(...)`, verifique se o usuário conseguiu se logar com sucesso usando o método `firebaseAuth.getCurrentUser()`. Se ele devolver um `user` diferente de nulo, direcione o usuário para a `MainActivity`. Siga o código abaixo como referência:

```

private boolean flagUsuarioLogado = false;

@Override
protected void onCreate(Bundle savedInstanceState) {

    // código anterior

    listener = new FirebaseAuth.AuthStateListener() {
        @Override
        public void onAuthStateChanged(@NonNull FirebaseAuth firebaseAuth) {
            FirebaseUser user = firebaseAuth.getCurrentUser();

            if (user != null && !flagUsuarioLogado) {
                flagUsuarioLogado = true;
                Intent vaiParaMain = new Intent(LoginActivity.this, MainActivity.class);
                startActivity(vaiParaMain);
                finish();
            }
        }
    };
}

```

```

    }
};
}

```

10. Agora repita o procedimento do login mas para o botão de novo usuário. Agora você vai precisar um outro método bastante parecido com o `logar(...)` mas agora você vai usar o método `createUserWithEmailAndPassword(...)` para criar um novo usuário. O código do método deve ficar parecido com o abaixo:

```

@OnClick(R.id.login_novo)
public void novoUsuario() {
    String email = campoEmail.getText().toString();
    String senha = campoSenha.getText().toString();

    if (!email.isEmpty() || !senha.isEmpty()) {
        firebaseAuth.createUserWithEmailAndPassword(email, senha).addOnCompleteListener(this, new
        OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) {
                if (!task.isSuccessful()) {
                    Snackbar.make(campoEmail, "Acesso não autorizado, verifique suas informações"
                    ,
                        Snackbar.LENGTH_SHORT).show();
                }
            }
        });
    } else {
        Snackbar.make(campoSenha, "Por favor complete todos os campos", Snackbar.LENGTH_SHORT)
        .show();
    }
}

```

11. Adicione um menu na `MainActivity` com uma opção de deslogar. Ao clicar nessa opção, você deve invocar o método `signOut()` em uma instância do `FirebaseAuth` para deslogar o usuário. Além disso, você precisa finalizar a `MainActivity` e redirecionar o usuário para a `LoginActivity` novamente.
12. Teste a aplicação, crie um novo usuário e verifique se o login está funcionando!

7.2 FIREBASE REMOTE CONFIG

Algumas vezes queremos trabalhar com um layout, texto ou funcionalidade diferente do padrão e ver se o usuário está gostando mais da funcionalidade nova ou se prefere a antiga. Esse conceito, de forma bem simples, chamamos de **Teste A/B**. Podemos fazer a distinção com base em funis, onde restringiremos, por exemplo, o sexo do usuário, idade e até mesmo o idioma. Para utilizarmos esse conceito o `Firebase` nos fornece um recurso chamado `Remote Config` que facilitará nossa vida no momento de fazermos esses filtros. Para utilizarmos esse recurso em nossa aplicação, pensaremos na nossa lista, que hoje é uma listagem zebraada, e faremos um teste para um funil de usuários com a lista sequencial comum.

Para começarmos a utilizar esse recurso, usaremos novamente o assistente do `Firebase` integrado

no Android Studio, desta vez buscaremos por Remote Config :

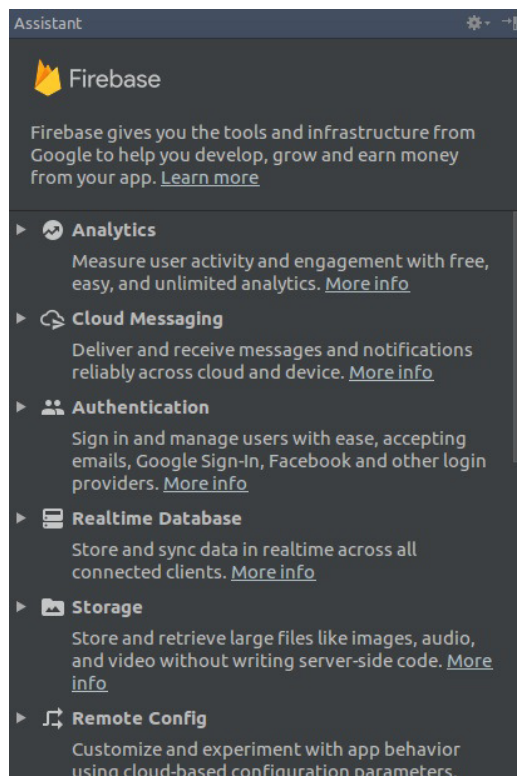


Figura 7.8: Assistente do Firebase

Novamente nos será pedido que vinculemos nossa aplicação ao Firebase , mas desta vez ele já utilizará a configuração que foi feita quando utilizamos o Firebase Authentication .

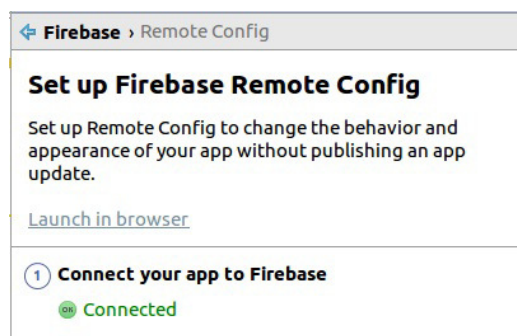


Figura 7.9: Conectado com o Firebase

Também é necessário trazer as dependências necessárias para o funcionamento do remote config, para isso basta clicarmos no botão para que tudo seja feito em apenas um clique!

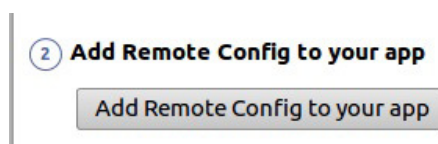


Figura 7.10: Configurando dependências

Depois de termos feito isso tudo, basta partirmos para a configuração em código. A primeira coisa que temos que fazer é obter um objeto que nos ajudará a realizar toda configuração, este objeto é do tipo `FirebaseRemoteConfig`. Para obtermos um objeto desse tipo basta invocarmos o método estático `getInstance()`:

```
FirebaseRemoteConfig remoteConfig = FirebaseRemoteConfig.getInstance();
```

Agora precisamos configurar esse nosso `FirebaseRemoteConfig` e para isso precisaremos definir quais serão as variáveis que poderemos alterar para fazermos nossos testes. Começaremos definindo os valores padrões da aplicação para essas variáveis, para que mais tarde possamos alterar eles com base nos funis. A definição deve ser feita em um arquivo xml que deve ser armazenado na pasta `res/xml/`. Esse arquivo será basicamente de marcação possuindo a chave e o valor, devendo respeitar a estrutura do firebase:

```
<defaultsMap>
  <entry>
    <key>list_type_single_item</key>
    <value>>false</value>
  </entry>
</defaultsMap>
```

Para cada valor que será alterado entre os funis devemos adicionar uma nova tag `<entry>` juntamente com a chave e o seu respectivo valor. Com o xml definido, temos que o referenciar o nosso `FirebaseRemoteConfig` para que os valores padrões estejam disponíveis para serem utilizados. Para isso, usaremos o método `setDefaults()`:

```
FirebaseRemoteConfig remoteConfig = FirebaseRemoteConfig.getInstance();
remoteConfig.setDefaults(R.xml.remote_config_defaults);
```

Agora precisamos definir quais são os funis e quais serão os valores respectivos para cada um deles. Isso será feito dentro do próprio console do Firebase como na imagem a seguir:

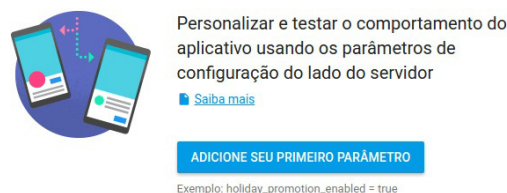


Figura 7.11: Configurando o Remote Config no console

Basta falarmos que queremos adicionar nosso primeiro parâmetro que ele abrirá um tela onde devemos passar o nome do parâmetro e qual é o seu valor padrão. Esse nome e o valor devem ser os mesmos que temos em nosso arquivo xml:

Figura 7.12: Adicionando um parâmetro do remote config

Agora precisamos criar qual será a variação do valor padrão. Para isso ainda no console precisamos definir o valor para condição. Fazemos isso clicando em **Adicionar valor para a condição** e teremos uma painel parecido com este onde serão definidos os funis que poderão ser definidos com base na versão do sistema operacional, versão que está instalado no celular, divisão meio a meio, por região, por idioma do celular entre outros (além de funis definidos através do **Firebase Analytics**). Após darmos o nome de nossa condição, adicionamos uma cor para identificá-la e por fim qual será o valor desejado para o funil escolhido.

Figura 7.13: Definindo condição no remote config

É sempre importante assim que terminarmos de fazer a configuração, **atualizar** e além disso também **publicar alterações** no próprio console!

Agora precisamos deixar o aplicativo pronto para ficar ouvindo e buscando as informações do **Firestore**, para isso temos um método que já realiza essa busca por nós. Usaremos o método `fetch()` que recebe por parâmetro o tempo que a resposta do **Firestore** ficará cacheada. Por se tratar de um processo assíncrono, precisaremos ter algo que fique ouvindo o retorno, portanto será necessário um listener:

```
remoteConfig.fetch(15).addOnCompleteListener(new OnCompleteListener<Void>() {
    @Override
    public void onComplete(@NonNull Task<Void> task) {
        // tratamento
    }
});
```

Agora precisamos validar se deu tudo certo e caso tenha dado, precisamos deixar os valores que buscamos dessa requisição ativos, para isso temos outro método que ativa esses valores: `activateFetched()`.

```
remoteConfig.fetch(15).addOnCompleteListener(new OnCompleteListener<Void>() {
    @Override
    public void onComplete(@NonNull Task<Void> task) {
```

```

        if (task.isSuccessful()) {
            mFirebaseRemoteConfig.activateFetched();
        }
    }
});

```

Para utilizarmos os valores do Remote Config basta pegarmos a nossa instância e recuperar os valores através de getters. Em nosso caso, temos um boolean portanto usaremos o método `getBoolean()` passando qual é a chave que dará esse valor:

```
boolean valorRemoteConfig = remoteConfig.getBoolean("list_type_single_item");
```

Com base nisso podemos fazermos validações, imprimir Views diferentes, alterar o layout entre diversas outras funcionalidades.

PARA SABER MAIS

Se você quiser saber como integrar o Remote Config ao Analytics pode ver o passo a passo na documentação :

https://firebase.google.com/docs/remote-config/config-analytics?utm_source=studio

Exercícios

1. Para usarmos o Remote Config em nosso projeto precisaremos ter as dependências em nosso projeto, para isso faremos da mesma forma que fizemos com a autenticação. Portanto vá em no assistente do Firebase e escolha a opção Remote Config e configure as dependências no projeto.
2. Agora precisamos configurar a instância do objeto `FirebaseRemoteConfig` dentro da nossa aplicação. Faremos isso no nosso fragment de lista:

```

public class ListaLivrosFragment extends Fragment implements Serializable {

    private FirebaseRemoteConfig mFirebaseRemoteConfig;

}

```

Vamos instanciar nosso objeto no momento que criamos nosso fragment, para isso implemente o método `onCreate` :

```

public class ListaLivrosFragment extends Fragment implements Serializable {

    private FirebaseRemoteConfig mFirebaseRemoteConfig;

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {

```

```

        super.onCreate(savedInstanceState);

        mFirebaseRemoteConfig = FirebaseRemoteConfig.getInstance();

    }
}

```

Agora é necessário passar quais são as chaves e seus valores padrões :

```

public class ListaLivrosFragment extends Fragment implements Serializable {

    private FirebaseRemoteConfig mFirebaseRemoteConfig;

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        mFirebaseRemoteConfig = FirebaseRemoteConfig.getInstance();

        mFirebaseRemoteConfig.setDefaults(R.xml.remote_config_defaults);

    }
}

```

1. Agora vamos criar nosso arquivo local de configurações, para isso com o botão direito na pasta res e escolha a opção de criar um novo Android resource file. Escolha como tipo de arquivo xml e dê o mesmo nome que estamos usando no código: remote_config_defaults, a sua estrutura deve ser similar a esta :

```

<?xml version="1.0" encoding="utf-8"?>
<defaultsMap>
    <entry>
        <key>list_type_single_item</key>
        <value>>false</value>
    </entry>
</defaultsMap>

```

1. Agora precisamos fazer a busca das informações no Firebase, para isso será necessário fazermos uma requisição através do nosso FirebaseRemoteConfig. Como já vimos no curso fazer uma requisição precisa ser algo assíncrono, para isso ele já tem um método que faz isso, precisamos apenas implementar o seu callback.

```

public class ListaLivrosFragment extends Fragment implements Serializable {

    private FirebaseRemoteConfig mFirebaseRemoteConfig;

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}

```

```

mFirebaseRemoteConfig = FirebaseRemoteConfig.getInstance();

mFirebaseRemoteConfig.setDefaults(R.xml.remote_config_defaults);

mFirebaseRemoteConfig.fetch(15).addOnCompleteListener(new OnCompleteListener<Void>() {
    @Override
    public void onComplete(@NonNull Task<Void> task) {
        if (task.isSuccessful()) {

            mFirebaseRemoteConfig.activateFetched();

        }
    }
});
}
}

```

1. Acesse o dashboard do Firebase em nosso projeto. Vá na opção de remote config. Clique no botão para adicionar um novo parâmetro. Na chave você deve colocar o mesmo nome que colocamos em nosso xml: `list_type_single_item`, e como valor padrão deixe o mesmo que usamos em nosso xml também: `false`. Ainda no popup clique em **Adicionar valor para a condição**, que fica na parte superior direita. Isso deve abrir um novo popup que pede um nome, chame de **Teste de Idioma**, você verá ainda nessa tela um dropdown para escolher ao que essa condição estará atrelada, nesse caso você deve escolher idioma do dispositivo, com isso você terá como escolher quais idiomas quer dar suporte, nesse teste escolha inglês - basta digitar inglês e escolher sua opção. Quando fizer isso você verá na tela inicial que haverá mais uma entrada de valor para essa opção que fez agora, dê o valor de `true` para ela.
1. Agora precisamos usar esse valor para poder setar o adapter dependendo da situação, altere a invocação para termos o comportamento desejado:

```

if (mFirebaseRemoteConfig.getBoolean("list_type_single_item")) {
    listaLivros.setAdapter(new ListaLivrosUmItemAdapter(livros));
} else {
    listaLivros.setAdapter(new ListaLivrosAdapter(livros));
}

```

1. Rode o aplicativo com o celular em português e depois altere o idioma e rode novamente para poder ver funcionar.

7.3 FIREBASE NOTIFICATION

Muitas vezes nós temos alguns cenários onde o usuário instala nossa aplicação, mas devido a rotina movimentada acabando não a utilizando com tanta frequência. Para esse tipo de usuário, seria interessante engajá-lo mais dentro de nossa aplicação. Para isso, podíamos enviar emails com promoções, mas assim ele simplesmente poderia cadastrar nossos emails como spam e nossa tentativa de

engajamento falharia. Algo realizado com maior retorno é o envio dessas chamadas para o próprio aparelho de maneira mais invasiva pelo uso de **notificações**, focando em trazer o usuário de volta à aplicação. Podemos facilmente notar esse comportamento em jogos, como **Clash Royale**. Outro cenário são as mensagens que chegam dos mensageiros como o **WhatsApp**, contudo o mais simples e que faz bastante isso é o **Facebook**, que te envia diversas maneiras para retornar, avisando sobre aniversários, eventos que estão ocorrendo ou qualquer interação que tenha ocorrido contigo no sistema.

Para tentarmos fazer este mesmo engajamento em nossa aplicação, precisaremos enviar as notificações para nosso usuário. Para facilitar nossa vida, o próprio **Firebase** possui um módulo específico para envio de notificações, conhecido como **Firebase Notification**.

Para utilizá-lo é necessário, novamente, conectar nossa aplicação ao **Firebase** e também trazer as dependências deste módulo. Teremos novamente o assistente com essa aparência :

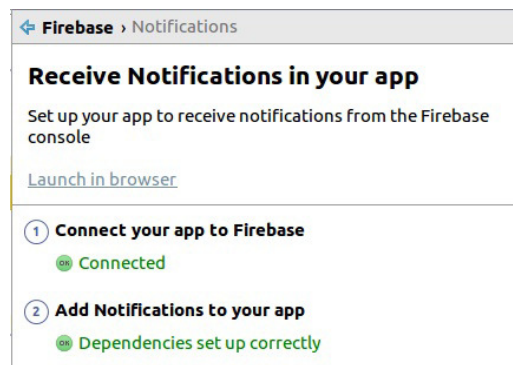


Figura 7.14: Assistente do Firebase

Só de fazermos esta configuração nosso aplicativo já está apto a receber notificações, não sendo necessário ter uma linha de código escrito.

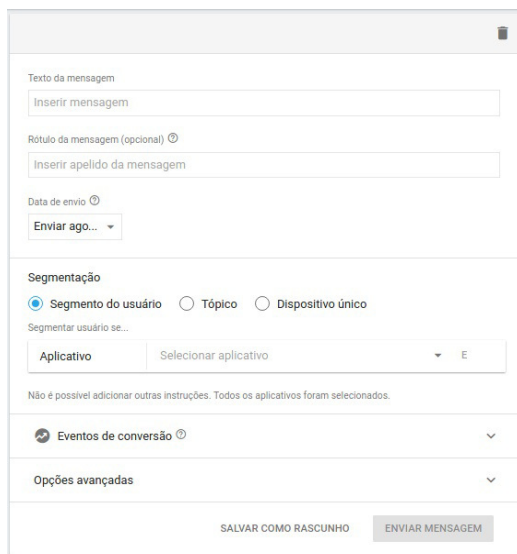
Para que a notificação seja enviada aos usuários, teremos que usar o console do **Firebase**. A primeira tela que teremos é essa:



Figura 7.15: Notificação do Firebase

Ao enviarmos nossa primeira mensagem, teremos um formulário que representa as informações que chegarão ao celular na notificação. Teremos que colocar o corpo da notificação e seu título. As outras informações o próprio **Firebase** se encarregará de preencher, por exemplo, o ícone da notificação que é exibido. Algo bem interessante é que conseguimos segmentar a notificação, enviando apenas para uma

certa parcela de usuários.



The screenshot shows the 'Enviar mensagem' (Send message) interface in the Firebase console. It includes fields for 'Texto da mensagem' (Message text), 'Rótulo da mensagem (opcional)' (Optional message label), and 'Data de envio' (Send date) with a dropdown menu. Below these is the 'Segmentação' (Segmentation) section, which has three radio buttons: 'Segmento do usuário' (selected), 'Tópico', and 'Dispositivo único'. There is a 'Selecionar aplicativo' (Select application) dropdown and a button 'Aplicativo'. A note states: 'Não é possível adicionar outras instruções. Todos os aplicativos foram selecionados.' (Cannot add other instructions. All applications were selected.) At the bottom, there are buttons for 'SALVAR COMO RASCUNHO' (Save as draft) and 'ENVIAR MENSAGEM' (Send message).

Figura 7.16: Enviando notificação pelo console

Ao enviarmos a notificação através do console, os aparelhos a receberão na pilha de notificações do aparelho, com todos os campos já populados e ao fazer o *click* nela seremos redirecionados para a Activity principal da aplicação.

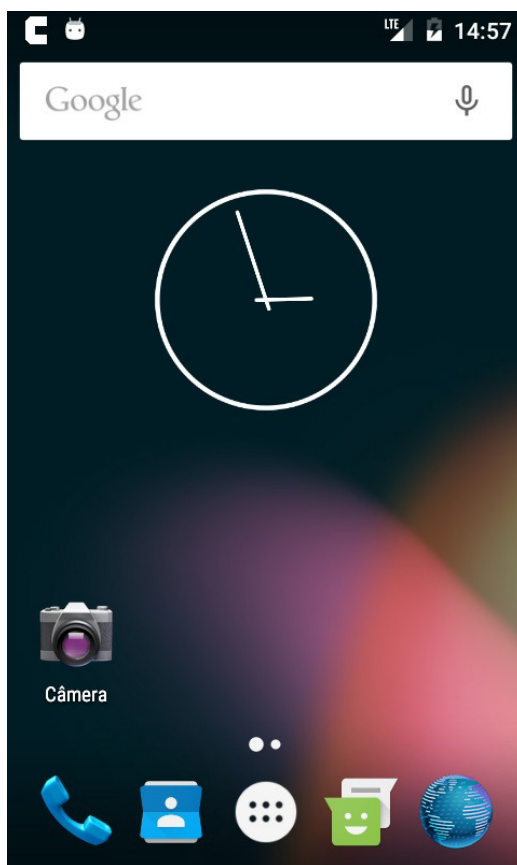


Figura 7.17: Notificação recebida

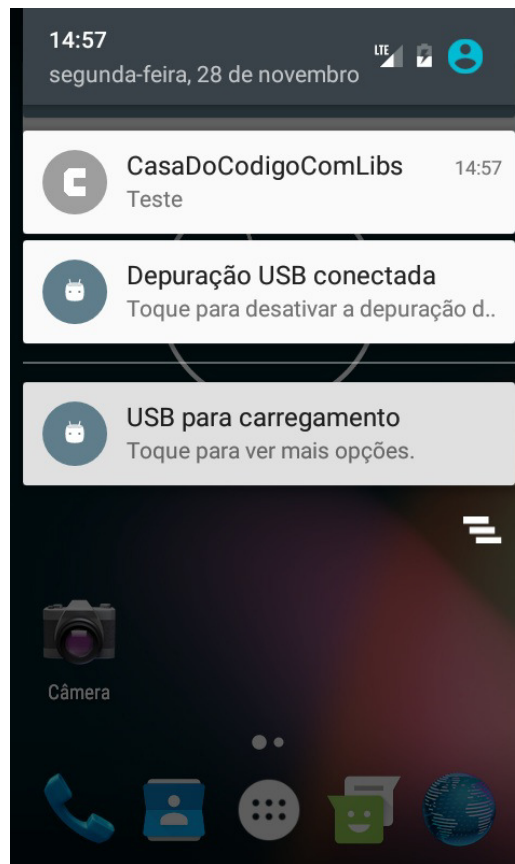


Figura 7.18: Várias notificações

7.4 EXERCÍCIOS

1. Para usarmos o Firebase Notification em nosso projeto precisaremos ter as dependências em nosso projeto, para isso faremos da mesma forma que fizemos com a autenticação. Portanto vá em no assistente do Firebase e escolha a opção Notifications e configure as dependências no projeto.
2. Entre no dashboard do Firebase procure por notificações e mande uma para nosso aplicativo.

7.5 TRATANDO NOTIFICAÇÕES DENTRO DA APLICAÇÃO

Quando estamos utilizando a aplicação e recebemos alguma mensagem não recebemos a notificação. Mas o que ocorre com os dados que podem estar chegando com a notificação? Até esse instante estamos perdendo tudo, pois por padrão a notificação só aparecerá caso a aplicação esteja em segundo plano.

Para podermos recuperar a mensagem recebida enquanto a aplicação está em primeiro é necessário ficarmos ouvindo se recebemos alguma notificação do Firebase. Realizar este trabalho necessitará de alguns passos, sendo que o primeiro é criar a classe especialista que ficará ouvindo esse evento. O próprio Firebase fornece uma classe que ficará ouvindo o envio de notificações, tudo que teremos que fazer é sobrescrever seu comportamento. A classe fornecida atuará como um serviço do sistema

operacional, para podermos alterar seu comportamento teremos que herdar de `FirebaseMessagingService` e sobrescrever o método `onMessageReceived()` que recebe a própria mensagem do Firebase :

```
public class NotificationService extends FirebaseMessagingService {

    @Override
    public void onMessageReceived(RemoteMessage remoteMessage) {
        // Comportamento que daremos
    }
}
```

Agora podemos disparar um evento para nossas `Activities` alertando sobre o recebimento da notificação. Usando a boa prática que vimos no curso basta enviarmos um *post* através do `Eventbus` .

```
public class NotificationService extends FirebaseMessagingService {

    @Override
    public void onMessageReceived(RemoteMessage remoteMessage) {
        EventBus.getDefault().post(new NotificacaoEvent(remoteMessage));
    }
}
```

Agora basta fazermos um `@Subscribe` para esse evento nas `Activities` , para por exemplo exibir um `Toast` apenas falando que recebemos uma notificação do Firebase :

```
@Subscribe()
public void recebeNotificacao(NotificacaoEvent event){

    Toast.makeText(context, "Recebeu uma notificação do servidor", Toast.LENGTH_SHORT).show();

}
```

Agora quando disparamos uma notificação do Firebase com a nossa aplicação em primeiro plano, estamos levando uma `Exception` . Devido ao `FirebaseMessagingService` estar sendo executado em segundo plano, em uma `Thread` secundária. Ao dispararmos o evento através do `Eventbus` , este evento será executado na `Thread` secundária portanto levamos a `Exception` por estarmos tentando manipular a tela que fica na `Thread` principal.

Para resolver esse problema precisamos pedir ao `Eventbus` execute a ação na `Thread` principal. Para fazermos isso basta definirmos qual é a `Thread` que deve ser executada, através do parâmetro *threadMode*:

```
@Subscribe(threadMode = ThreadMode.MAIN)
public void recebeNotificacao(NotificacaoEvent event){

    Toast.makeText(context, "Recebeu uma notificação do servidor", Toast.LENGTH_SHORT).show();

}
```

Como estamos definindo um serviço que vai executar fora do padrão, temos que deixá-lo devidamente configurado no manifest :

```
<application>
  <!-- demais configurações -->

  <service
    android:name=".firebase.NotificationService"
    android:enabled="true">
    <intent-filter>
      <action android:name="com.google.firebase.MESSAGING_EVENT" />
    </intent-filter>
  </service>

</application>
```

PARA SABER MAIS

Além de pedirmos para executar na Main podemos pedir para ser executadas de outras formas.

Podemos ver os demais tipos na documentação do EventBus :
<http://greenrobot.org/eventbus/documentation/delivery-threads-threadmode/>

Exercícios

1. Para tratar a notificação caso o usuário esteja dentro da aplicação será necessário termos um serviço ouvindo as notificações. Para isso precisamos criar nossa classe de tratamento. No pacote firebase crie a classe NotificationService:

```
public class NotificationService extends FirebaseMessagingService {
}
```

1. Agora é necessário implementar o método onMessageReceived da classe FirebaseMessagingService :

```
public class NotificationService extends FirebaseMessagingService {

    @Override
    public void onMessageReceived(RemoteMessage remoteMessage) {
        // Comportamento que daremos
    }
}
```

1. Queremos que ao chegar alguma notificação nosso usuário saiba disso, para isso precisamos lançar essa notificação para dentro de nossa aplicação, usaremos para isso o EventBus.

```
public class NotificationService extends FirebaseMessagingService {

    @Override
    public void onMessageReceived(RemoteMessage remoteMessage) {
        EventBus.getDefault().post(remoteMessage);
    }
}
```

1. Para que nosso serviço possa entrar em funcionamento precisamos registra-lo no Manifest :

```
<application>
    <!-- demais configurações -->

    <service
        android:name=".firebase.NotificationService"
        android:enabled="true">
        <intent-filter>
            <action android:name="com.google.firebase.MESSAGING_EVENT" />
        </intent-filter>
    </service>

</application>
```

1. Para tratar o envio do post feito pelo EventBus precisaremos criar um método na nossa MainActivity para receber o evento na thread principal :

```
@Subscribe(threadMode = ThreadMode.MAIN)
public void recebeNotificacao(NotificacaoEvent event){

    Toast.makeText(context, "Recebeu uma notificação do servidor", Toast.LENGTH_SHORT).show();

}
```

1. Execute o aplicativo e mande uma notificação e veja o toast ser exibido.

7.6 RECEBENDO NOTIFICAÇÕES DO NOSSO SERVIDOR

Vimos uma forma de termos esse engajamento do usuário dentro de nossa aplicação, todavia o envio da notificação feita pelo console do Firebase e este método de envio pode ser limitado demais além de não poder ser disparado automaticamente por outro sistema.

Uma forma alternativa de fazer o envio de notificações é criar um servidor que ficará responsável por ter comunicação com o Firebase para solicitar que ele dispare as notificações. Para fazer essa integração do servidor com o Firebase, você pode consultar a documentação :

<https://firebase.google.com/docs/server/setup>

Com o servidor devidamente pronto, podemos fazer com que nossa aplicação comece a receber as notificações diretamente do nosso servidor. Para isso teremos que informar ao nosso servidor qual é a identificação do aparelho, pois quando pedirmos ao Firebase que encaminhe as notificações ele

precisará saber quais aparelhos devem ser notificados.

Para obtermos nossa identificação no `Firebase` é necessário registrar o celular e obter seu *id* que é gerado ao concluir o registro. Já temos uma classe chamada `FirebaseInstanceId` que é especialista em fazer esse registro. Para conseguirmos um objeto desse tipo usamos o método `getInstance()`. Agora basta usarmos essa instância para gerar o id. Para isso, usaremos o método `getToken()` que retornará o registro do `Firebase` com o id:

```
FirebaseInstanceId instance = FirebaseInstanceId.getInstance();  
String token = instance.getToken();
```

Podemos então enviar esse token para o nosso servidor para que ele consiga concentrar a informação de registro de todos usuários. Para isso, podemos usar o `Retrofit` novamente para fazer o envio assíncrono das informações para nosso servidor.

Agora precisamos estar preparados para receber as notificações que virão do nosso servido. Novamente precisaremos ter alguém ouvindo o recebimento das notificações. Podemos então criar um outro serviço da mesma forma fazíamos para tratar a notificação dentro da aplicação.

Para criarmos um serviço temos que ter uma classe que herde de `FirebaseMessagingService` e temos que sobrescrever o método `onMessageReceived()`, que servirá para pegarmos as informações que nosso servidor passou ao `Firebase` para recebermos.

```
@Override  
public void onMessageReceived(RemoteMessage remoteMessage) {  
  
}
```

Usando o parâmetro `RemoteMessage` conseguimos pegar alguns detalhes da mensagem que recebemos. Usaremos o método `getData()` que nos retornará um `Map` onde teremos o conjunto de chaves e valores que chegam dentro do objeto. Para pegarmos a mensagem, usaremos a chave `message`:

```
@Override  
public void onMessageReceived(RemoteMessage remoteMessage) {  
  
    Map<String, String> data = remoteMessage.getData();  
    String message = data.get("message");  
  
}
```

Agora precisamos criar a notificação para o usuário. Para isso, usaremos a classe `NotificationCompat.Builder` que criará uma notificação que funcionará em todos os celulares. Precisaremos passar alguns parâmetros para que tudo funcione corretamente, como título da notificação e o ícone que deverá ser exibido. Além disso, podemos falar que é uma notificação que poderá ser removida automaticamente da bandeja. Como a notificação ficará sendo exibida fora do aplicativo

precisaremos passar **contexto** para ela no momento de criação do objeto:

```
@Override
public void onMessageReceived(RemoteMessage remoteMessage) {

    Map<String, String> data = remoteMessage.getData();
    String message = data.get("message");

    Notification notification = new NotificationCompat.Builder(getBaseContext())
        .setSmallIcon(R.drawable.casadocodigo)
        .setContentTitle(message)
        .setAutoCancel(true)
        .build();

}
```

Geralmente quando clicamos em uma notificação ela nos redireciona para o aplicativo, portanto temos que criar nossa `Intent` que irá para a tela desejada :

```
@Override
public void onMessageReceived(RemoteMessage remoteMessage) {

    Map<String, String> data = remoteMessage.getData();
    String message = data.get("message");

    Intent intent = new Intent(getApplicationContext(), CarrinhoActivity.class);

    Notification notification = new NotificationCompat.Builder(getBaseContext())
        .setSmallIcon(R.drawable.casadocodigo)
        .setContentTitle(message)
        .setAutoCancel(true)
        .build();

}
```

Agora precisamos vincular nossa `Intent` à nossa `Notification`. Contudo a nossa `Intent` não será executada pela nossa aplicação, portanto precisamos pedir para outra aplicação executá-lo por nós, parecido com uma procuração no mundo real. Vamos então criar o objeto que represente nossa procuração. Esse objeto é a `PendingIntent`. Para criá-la usaremos o método `getActivity()` que necessita de alguns parâmetros, sendo um deles a própria `Intent`. Como ele será executado por outra aplicação, temos apenas que falar quem somos. Por isso, temos que passar um contexto à ela. Ainda temos que passar um código que identifique aquela nossa `PendingIntent` que será representado como um inteiro e também temos que passar uma *flag* que representa o comportamento que a notificação deverá ter. Esse comportamento é pertinente a cada aplicação. O **WhatsApp**, por exemplo, reaproveita a notificação que estava na bandeja e faz as mudanças em cima daquela notificação. Outro caso é o que o **Facebook** faz, que é criar uma nova notificação para cada evento que ocorre. Para nossa aplicação vamos fazer um comportamento diferente de ambas, que é ter sempre uma única notificação relacionada a nossa aplicação na bandeja, ou seja, se já tivermos uma aquela será cancelada e a nova será a vigente. Para este comportamento usaremos a constante `PendingIntent.FLAG_CANCEL_CURRENT` :


```

@Override
public void onMessageReceived(RemoteMessage remoteMessage) {

    Map<String, String> data = remoteMessage.getData();
    String message = data.get("message");

    Intent intent = new Intent(getApplicationContext(), CarrinhoActivity.class);

    PendingIntent pendingIntent =
        PendingIntent.getActivity(getBaseContext(), 123,
            intent,
            PendingIntent.FLAG_CANCEL_CURRENT);

    Notification notification = new NotificationCompat.Builder(getBaseContext())
        .setSmallIcon(R.drawable.casadocodigo)
        .setContentTitle(message)
        .setAutoCancel(true)
        .setContentIntent(pendingIntent)
        .build();

}

```

A única coisa que ainda não fizemos é pegar essa notificação e a adicionar na bandeja de notificações do sistema, para este trabalho teremos que pegar o especialista de notificações do Android, a classe `NotificationManager` :

```
NotificationManager manager = (NotificationManager) context.getSystemService(NOTIFICATION_SERVICE);
```

Agora só precisamos disparar essa nossa notificação, para isso usaremos nosso gerenciador. Basta invocarmos o método `notify()` , onde passaremos a notificação e além disso um identificador para aquela notificação.

```

@Override
public void onMessageReceived(RemoteMessage remoteMessage) {

    Map<String, String> data = remoteMessage.getData();
    String message = data.get("message");

    PendingIntent pendingIntent =
        PendingIntent.getActivity(getBaseContext(), 123,
            new Intent(getApplicationContext(), CarrinhoActivity.class),
            PendingIntent.FLAG_CANCEL_CURRENT);

    Notification notification = new NotificationCompat.Builder(getBaseContext())
        .setSmallIcon(R.drawable.casadocodigo)
        .setContentTitle(message)
        .setColor(Color.WHITE)
        .setAutoCancel(true)
        .setContentIntent(pendingIntent)
        .build();

    Context context = getBaseContext();
    NotificationManager manager = (NotificationManager) context.getSystemService(NOTIFICATION_SERVICE);

    manager.notify(123, notification);

}

```

Para que nosso serviço funcione, precisamos deixá-lo cadastrado no manifest:

```
<service
    android:name=".fcm.FCMListener"
    android:enabled="true">
    <intent-filter>
        <action android:name="com.google.firebase.MESSAGING_EVENT" />
    </intent-filter>
</service>
```

EXERCICIOS

1. Vá na pasta do curso, FJ-59 e pegue o zip CasaDoCodigoServer.zip e o descompacte na area de trabalho.
2. No dashboard do Firebase, clique em configurações -> usuários e permissões. Isso deve fazer uma nova página abrir, onde você escolherá a opção conta de serviço. Nessa página, você irá criar uma nova nova conta de serviço. Dê o nome de servidor e peça para uma chave privada no formatado json. Pegue o conteúdo do json gerado pelo firebase, vá no projeto CasaDoCodigoServer e procure na pasta resources um json e altere o **conteúdo** com o json que o firebase gerou.
3. Para ouvirmos o envio de notificações do nosso servidor teremos que criar uma classe para ficar ouvindo o evento e assim que chegar gerar uma notificação.

```
public class MeuListener extends FirebaseMessagingService{

    @Override
    public void onMessageReceived(RemoteMessage remoteMessage) {

        Map<String, String> data = remoteMessage.getData();
        String message = data.get("message");

        PendingIntent pendingIntent =
            PendingIntent.getActivity(getBaseContext(), 123,
                new Intent(getApplicationContext(), CarrinhoActivity.class),
                PendingIntent.FLAG_CANCEL_CURRENT);

        Notification notification = new NotificationCompat.Builder(getBaseContext())
            .setSmallIcon(R.drawable.casadocodigo)
            .setContentTitle(message)
            .setColor(Color.WHITE)
            .setAutoCancel(true)
            .setContentIntent(pendingIntent)
            .build();

        Context context = getBaseContext();
        NotificationManager manager = (NotificationManager) context.getSystemService(NOTIFICATION_SERVICE);
    }
}
```

```

        manager.notify(123, notification);
    }
}

```

1. Agora precisamos falar para o android que temos uma classe especialista em ouvir requisições vindas do firebase, para isso teremos que remover o serviço que tínhamos feito no último exercício e adicionar esse novo :

```

<service
    android:name=".fcm.FCMListener"
    android:enabled="true">
    <intent-filter>
        <action android:name="com.google.firebase.MESSAGING_EVENT" />
    </intent-filter>
</service>

```

1. Para que possamos de fato receber a notificação nosso servidor precisará saber qual é o identificador do nosso aparelho, para isso crie um serviço que fará um requisição `@Post` para o seguinte endpoint : `/device/register/{email}/{id}` , não esqueça de criar um método no `WebClient` para podermos usar esse serviço.
1. Agora precisamos gerar o token de identificação do aparelho. Para isso usaremos a classe `FirebaseInstanceId` no momento em que estivermos nos logando na aplicação:

```

FirebaseInstanceId instance = FirebaseInstanceId.getInstance();

String token = instance.getToken();

client.cadastraCelular(user.getEmail(), token);

```

1. No navegador faça uma requisição para `/device/lancaFirebase` e veja a notificação chegar no aplicativo

UTILIZANDO BANCO DE DADOS

8.1 COLOCANDO FORMA DE PAGAMENTO

Nosso sistema está bem completo até o momento, o único problema é que ainda não conseguimos realizar a venda, para isso vamos precisar ter uma forma de pagamento que nosso sistema.

A melhor forma seria cadastrar o cartão de crédito de nosso usuário dentro da nossa aplicação, para isso precisaremos criar uma nova tela onde vamos deixar todo comportamento para nossos cartões.

Vamos começar criando uma nova `Activity` e vamos chama-la de `CartaoActivity` :

```
public class CartaoActivity extends AppCompatActivity {

}
```

Agora precisamos definir o layout e o referenciar em nossa `Activity` , como já vimos anteriormente, vamos deixar nossa exibição de tela em `Fragments` , então nosso layout apenas com um `FrameLayout` .

```
public class CartaoActivity extends AppCompatActivity {

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_cartao);
        ButterKnife.bind(this);
    }

}
```

Agora precisamos definir nossos `Fragments` , que a priori serão um para exibir a lista e uma tela de cadastro.

A tela de listagem terá um `RecyclerView` e um botão para podermos fazer a troca de fragment :

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout xmlns:android="http://schemas.android.com/apk/res/an
droid"
    xmlns:app="http://schemas.android.com/apk/res-auto"
```

```

android:layout_width="match_parent"
android:layout_height="match_parent">

<android.support.v7.widget.RecyclerView
    android:id="@+id/lista_cartoes"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="2dp" />

<android.support.design.widget.FloatingActionButton
    android:id="@+id/btn_add_cartao"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom|right"
    android:layout_margin="10dp"
    android:src="@drawable/card_add"
    app:backgroundTint="#98c5f2"
    app:fabSize="normal" />

</android.support.design.widget.CoordinatorLayout>

```

Com esse layout teremos um resultado similar a isso :

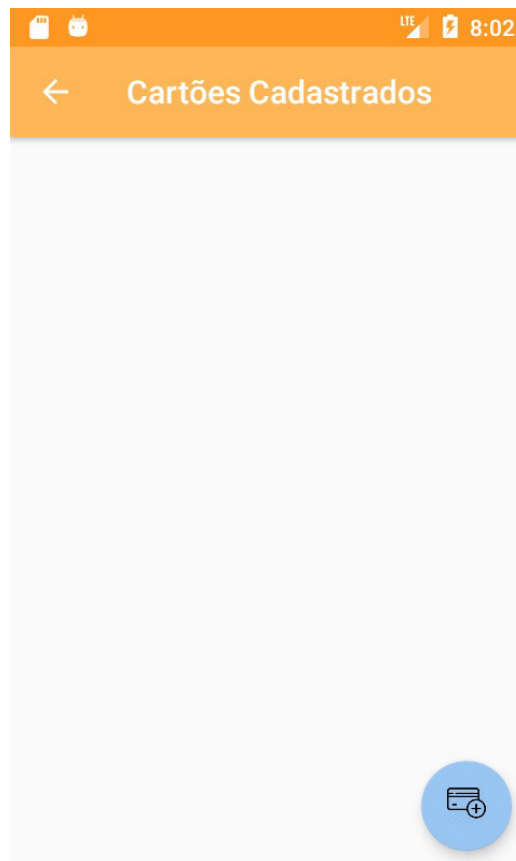


Figura 8.1: Listagem de Cartao

Ainda fica faltando fazermos a tela de cadastro, onde colocaremos os campos pertinentes ao cartão, como na imagem :

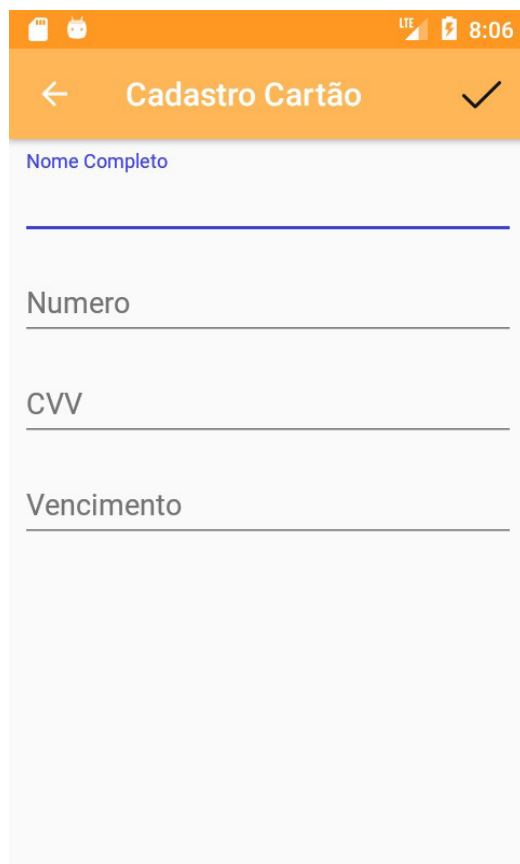


Figura 8.2: Cadastro de Cartao

Para chegarmos nesse layout, precisamos definir o nosso xml da seguinte maneira :

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_margin="3dp"
        android:orientation="vertical">

        <android.support.design.widget.TextInputLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_margin="3dp">

            <EditText
                android:id="@+id/formulario_nome"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:hint="Nome Completo" />

        </android.support.design.widget.TextInputLayout>

        <!-- Demais campos -->
    </LinearLayout>
</ScrollView>
```

```
</LinearLayout>
```

```
</ScrollView>
```

Agora é necessário apenas criar os `Fragment`s que estarão responsáveis de gerenciar esses layouts :

```
public class FormularioCartaoFragment extends Fragment {

    @BindView(R.id.formulario_nome)
    EditText nome;

    @BindView(R.id.formulario_vencimento)
    EditText vencimento;

    @BindView(R.id.formulario_numero)
    EditText numero;

    @BindView(R.id.formulario_cvv)
    EditText cvv;

    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.formulario_cartao_fragment, container, false);
        ButterKnife.bind(this, view);

        return view;
    }
}
```

E também nossa listagem :

```
public class ListaCartoesFragment extends Fragment {

    @BindView(R.id.lista_cartoes)
    RecyclerView lista;

    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.lista_cartoes_fragment, container, false);
        ButterKnife.bind(this, view);

        return view;
    }
}
```

Nesse ponto estamos trabalhando com para ter um cartão, contudo ainda não mapeamos ele ainda em nossa aplicação, então para isso vamos criar uma classe que possa representar nosso `Cartao` :

```

public class Cartao implements Serializable {

    private String nomeCompleto;
    private Long numero;
    private Integer cvv;
    private String validade;

    // getters e setters

}

```

Agora podemos trabalhar melhor em nossos Fragments :

```

public class ListaCartoesFragment extends Fragment {

    @BindView(R.id.lista_cartoes)
    RecyclerView lista;

    private List<Cartao> cartoes;

    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.lista_cartoes_fragment, container, false);
        ButterKnife.bind(this, view);

        return view;
    }
}

```

Pensando em nosso Fragment de lista, teremos que criar um novo Adapter para que possamos exibir nossa lista de cartões :

```

public class CartaoAdapter extends RecyclerView.Adapter {
    private List<Cartao> cartoes;

    public CartaoAdapter(List<Cartao> cartoes) {
        this.cartoes = cartoes;
    }

    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
        View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_cartao, parent, false);

        return new ViewHolder(view);
    }

    @Override
    public void onBindViewHolder(RecyclerView.ViewHolder viewHolder, int position) {
        Cartao cartao = cartoes.get(position);
        ViewHolder holder = (ViewHolder) viewHolder;
        holder.nome.setText(cartao.getNomeCompleto());
        holder.numero.setText(cartao.getNumero().toString());
        holder.validade.setText(cartao.getValidade());
    }
}

```



```

        holder.cvv.setText(cartao.getCvv().toString());
    }

    @Override
    public int getItemCount() {
        return cartoes.size();
    }

    class ViewHolder extends RecyclerView.ViewHolder {

        @BindView(R.id.titular_cartao_item)
        TextView nome;

        @BindView(R.id.numero_cartao_item)
        TextView numero;

        @BindView(R.id.validade_cartao_item)
        TextView validade;

        @BindView(R.id.cvv_cartao_item)
        TextView cvv;

        public ViewHolder(View view) {
            super(view);
            ButterKnife.bind(this, view);
        }
    }
}

```

Só fica faltando fazermos o layout para cada item de nossa listagem, vamos tentar deixar bem próximo de um cartão de verdade :

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/cartao_item"
    android:layout_width="match_parent"
    android:layout_height="130dp"
    android:layout_margin="10dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">

        <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="30dp"
            android:background="@drawable/degrade">

            <TextView
                android:id="@+id/titular_cartao_item"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_margin="2dp"
                android:textSize="20dp"
                android:textStyle="bold|italic" />

        </RelativeLayout>
    </LinearLayout>

```

```

<TextView
    android:id="@+id/numero_cartao_item"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom"
    android:layout_margin="2dp"
    android:gravity="center"
    android:paddingTop="20dp"
    android:textSize="17dp"
    android:textStyle="bold|italic" />

<LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:orientation="vertical">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="2dp"
            android:text="Validade do cartão : "
            android:textSize="15dp" />

        <TextView
            android:id="@+id/validade_cartao_item"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="2dp"
            android:textSize="15dp" />
    </LinearLayout>

    <LinearLayout
        android:layout_width="0dp"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:orientation="vertical">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="2dp"
            android:text="CVV"
            android:textSize="15dp" />

        <TextView
            android:id="@+id/cvv_cartao_item"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_margin="2dp"
            android:textSize="15dp" />
    </LinearLayout>
</LinearLayout>

```

```
</LinearLayout>
</android.support.v7.widget.CardView>
```

Com esse layout teremos um resultado parecido com isto :

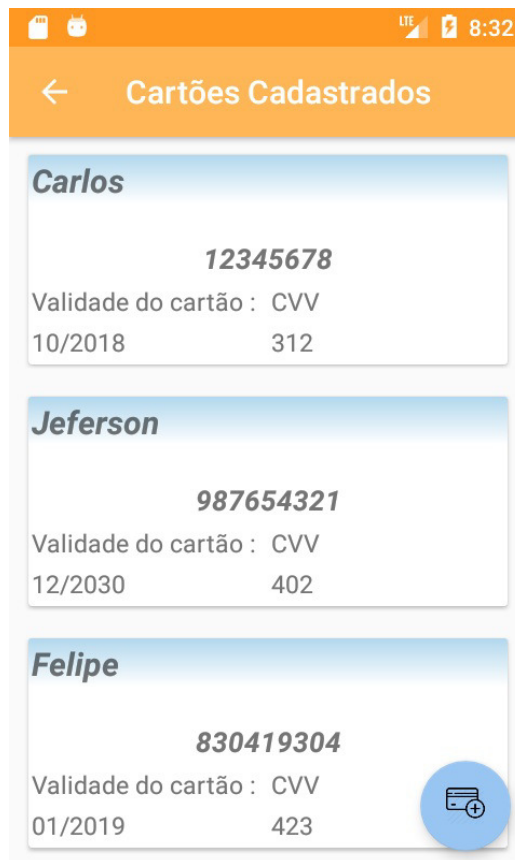


Figura 8.3: Exemplo dos Cartões

Para termos isso tudo funcional precisaremos adicionar nossos dados num banco de dados, estamos acostumados a ter que fazer tudo na mão, ou seja, criar nossa classe que irá estender de `SQLiteOpenHelper` e implementar nossos métodos, o grande problema disso é que com o passar do tempo nossos modelos vão ser atualizados e quando isso ocorre, temos que modificar nosso DAO .

Para quem está habituado a trabalhar com sistemas web em java, ao ver esta situação, acredita que estamos trabalhando de maneira arcaica, dado que já existe uma especificação que cuida disso, a JPA (Java Persistence API).

Se tivéssemos algo similar a JPA no Android nosso modelo ficaria da seguinte forma :

```
@Entity
public class Cartao implements Serializable {

    @Id
    private Long id;
```

```

@NotNull
private String nomeCompleto;
@NotNull
private Long numero;
private Integer cvv;
private String validade;
}

```

Com nosso objeto mapeado, bastaria nesse ponto o enviar para o banco de dados para ser persistido, usando JPA temos uma classe que encapsula todas as chamadas para o banco de dados, criando os comandos com base no objeto mapeado, essa classe se chama `EntityManager`, ela só consegue fazer isso tudo em tempo de execução pois usa um recurso muito poderoso da linguagem chamado `Reflection`, que infelizmente não é recomendado para utilizar no Android.

Contudo o uso de JPA é bem interessante para evitar todo o trabalho manual de criação do banco e suas queries, aumentando a produtividade do desenvolvedor, dado essa circunstância gostaríamos de ter algo similar no Android, pensando nisso foi desenvolvida uma biblioteca para este propósito, conhecida como `GreenDao`, ela por trás dos panos gerará em tempo de compilação toda parte de código para podermos trabalhar com requisições para o `SQLite`, sem uso de `Reflection`.

Para a utilizarmos precisamos deixar uma instância da sessão do banco de dados disponível para ser utilizada:

```

DaoMaster.DevOpenHelper helper = new DaoMaster.DevOpenHelper(context, "cdc-bd");
Database db = helper.getWritableDatabase();
DaoSession session = new DaoMaster(db).newSession();

```

Vamos entender inicialmente cada uma dessas linhas, o `GreenDao`, já possui uma classe que vai herdar de `SQLiteOpenHelper`, essa classe é `DevOpenHelper`, que tivemos que passar para ela tanto o nome do banco e o contexto, e o restante ele fará por nós.

Em sequência nós precisamos pegar abrir uma sessão no banco de dados, fazemos isso através da classe `DaoMaster`, que precisa de uma instância do banco de dados, para isso usamos nosso helper para nos fornecer um instância que possamos escrever.

A classe `DaoSession` que representa nossa sessão, é gerada em tempo de compilação, nesse procedimento ela busca todas nossas classes que estão mapeadas com `@Entity` e gera para cada uma delas um `Dao` que já provê todos os métodos necessários.

Para recuperarmos nosso `Dao` para nossa entidade `Cartão` basta fazermos:

```

CartaoDao dao = session.getCartaoDao();

```

PARA SABER MAIS

Você pode ler mais sobre o GreenDao na documentação oficial :

<http://greenrobot.org/greendao/>

8.2 EXERCÍCIOS

1. Adicionando as dependências e configurando o gradle
2. Criando a Activity
3. Criando os Fragments
4. Criando o Modelo
5. Mapeando o modelo
6. Colocando o Dao para ser injetado pelo dagger
7. Usando o dao nos fragments
8. Usando o cartão na finalização da compra