Título

# **Unidade 7**

# Aula 1

# Aula 2

## Padrão de Projeto - Observer

## Biblioteca Ion

Você aprendeu a obter um JSON do diretório do projeto, realizar um parsing utilizando a biblioteca Gson e convertê-lo em um model (no caso um model de E-mail). Agora, aplicando os conceitos de Networking e do padrão Observer vamos utilizar a biblioteca Ion.

Com ela conseguiremos obter um JSON remotamente, ou seja, que não está contido no diretório do projeto. Em outras palavras, vamos obter um JSON que está hospedado em algum lugar da internet.

Para nível de exemplo, o mesmo JSON que utilizamos na aula passada para mostrar a lista de emails foi colocado em uma pasta pública do Dropbox. Através desta pasta publica podemos obter o link do arquivo e consumí-lo.

O link para a pasta o JSON no Dropbox é o seguinte:

<https://dl.dropboxusercontent.com/u/72381225/emails.json>

**DICA**: Se você não sabe o que é o Dropbox, trata-se de um serviço de armazenamento na nuvem. Podemos colocar nossos arquivos lá e ele estará disponível em qualquer lugar com acesso a internet. Você pode fazer esta experiência criando seu próprio arquivo JSON e disponibilizando-o na sua própria conta. Basta colocá-lo na pasta **Public** do seu Dropbox e obter o link de compartilhamento.

Agora vamos ver a real utilização da Ion.

1. Primeiro insira a dependência do Ion no **build.gradle (Module: app)**, como de costume:

compile **'com.koushikdutta.ion:ion:2.+'**

1. Agora vamos preparar a URL. Entre na classe InboxFragment e insira a seguinte constante:

**public static final** String ***BASE\_URL*** = **"**[**https://dl.dropboxusercontent.com/u/72381225/**](https://dl.dropboxusercontent.com/u/72381225/)**"**;

**Entendendo o código:** É uma boa prática definirmos a URL base em uma constante. Se você mudar o usuário ou o diretório onde o arquivo email.json estiver bastará trocar a URL base e todos os lugares que a usarem já estarão atualizados. No nosso exemplo a usaremos uma única vez, então é optativo colocar a URL base em uma constante. Os passos a seguir assumimos que esta URL está em uma constante.

1. Por fim, façamos a chamada da Ion. Substitua o conteúdo do método **initListView()** para corresponder com o seguinte:

**public void** initListView() {

  Ion.*with*(getContext())

.load(***BASE\_URL***+**"emails.json"**)

.asJsonObject()

.setCallback(**new** FutureCallback<JsonObject>() {

      @Override

**public void** onCompleted(Exception e, JsonObject result) {

**if** (result != **null**) {

**if** (result.has(**"emails"**)) {

                  JsonArray list = result.get(**"emails"**).getAsJsonArray();

                  Gson gson = **new** Gson();

                  String json = gson.toJson(list);

**if** (getContext() != **null**) {

**listView**.setAdapter(**new** InboxAdapter(getContext(), Utils.*parseEmails*(json)));

                  }

              }

          }

      }

  });

}

**Entendendo o código**: Vamos analizar por partes. Primeiro a declaração:

Ion.*with*(getContext())

.load(***BASE\_URL***+**"emails.json"**)

.asJsonObject()

.setCallback(**new** FutureCallback<JsonObject>() {

Neste trecho temos a declaração da classe. Semelhante aos EventListeners aprendidos na unidade 3 a biblioteca Ion possui uma interface a ser implementada, chamada FutureCallback, que possui um método que veremos a seguir. O método **load()** é onde especificamos a URL do arquivo que gostaríamos de obter. Neste caso estamos utilizando uma chamada do tipo GET, que relembrando, apenas obtém um arquivo. O método **asJsonObject()** irá converter a resposta da requisição em um objeto JSON. E o método **setCallBack()** irá definir o comportamento a ser realizado quando um objeto vier como resposta da requisição.

@Override

**public void** onCompleted(Exception e, JsonObject result) {

**if** (result != **null**) {

**if** (result.has(**"emails"**)) {

                  JsonArray list = result.get(**"emails"**).getAsJsonArray();

                  Gson gson = **new** Gson();

                  String json = gson.toJson(list);

**if** (getContext() != **null**) {

**listView**.setAdapter(**new** InboxAdapter(getContext(), Utils.*parseEmails*(json)));

                  }

              }

          }

      }

Este método é invocado quando a requisição é completada, ou seja, quando uma resposta é obtida do servidor (no nosso caso o Dropbox é o servidor e a resposta esperada é o arquivo emails.json). Este método possui como parâmetro o objeto **result,** que é onde estará a resposta da requisição, e o objeto **e** que armazena uma Exception, caso algo dê errado. O restante do método trata o arquivo response como um JsonObject, como visto no tópico **Gson.**

1. Execute o código e veja o resultado. Não haverá mudança alguma na aparência do aplicativo, apenas iremos obter o conteúdo da nossa ListView via uma requisição HTTP agora.

Para brincar, testar e ver os benefícios você pode criar um arquivo **emails.json** na sua própria conta do Dropbox e disponibilizá-lo como público.

Vamos voltar ao padrão Observer e relacioná-lo com  este exemplo.

Quando utilizamos o método **setCallBack()** na verdade estamos instanciando um Observer. O Ion manda uma requisição ao servidor dizendo “preciso do arquivo emails.json” e o Observer fica esperando a resposta do servidor. Enquanto isso os códigos seguintes são executados e você consegue continuar a interagir com o app normalmente. Quando o servidor dá o arquivo emails.json ao Ion, o Observer logo intercepta isto e faz o devido tratamento (como visto no método **onCompleted()**). Na prática podemos ver que isto acontece de forma assíncrona, ou seja, enquanto o Observer está esperando a resposta do servidor, o aplicativo continua funcionando e quando a resposta é obtida, as tarefas que já estavam em execução no app não são interrompidas para que o Observer faça o tratamento da resposta, pelo contrário, as duas coisas acontecem ao mesmo tempo.

Então para fechar a questão do padrão Observer, tenha em mente que quando precisamos realizar uma tarefa demorada mas não podemos bloquear o uso do aplicativo enquanto esta tarefa não é terminada, utilize o padrão Observer e crie tarefas assíncronas.

Agora iremos aprender a obter as imagens do servidor.

## Biblioteca Glide

No arquivo emails.json contido no nosso Dropbox possui um campo chamado **image** para alguns objetos do JSON. Este campo possui uma URL que direciona para uma imagem de perfil de alguma pessoa (estas imagens são públicas e disponibilizadas pelo serviço <https://randomuser.me>).

O que iremos fazer é obter estas imagens através destas URLs e inserir ao lado esquerdo de cada célula de e-mail no lugar da inicial do remetente.

A biblioteca perfeita para tal feito é a Glide.

Com a Glide podemos fazer requisições assíncronas de imagens e ela ainda possui uma técnica de cache que salva a imagem baixada temporariamente no pacote do seu app para economizar seu plano de dados. Então ela não fará download das imagens sempre que for requisitado, apenas se a imagem não foi baixada ainda.

Vamos testá-la.

1. Insira a seguinte dependência no arquivo build.gradle (Module: app):

compile **'com.github.bumptech.glide:glide:3.7.0'**

1. Agora visite a classe InboxAdapter e altere o método getView() e insira o seguinte trecho de código antes do retorno da função:

**if** (**data**.get(position).getImageUrl() != **null**) {

  Glide.*with*(**context**).load(**data**.get(position)

          .getImageUrl())

          .asBitmap()

          .centerCrop()

          .into(**new** BitmapImageViewTarget(picture) {

      @Override

**protected void** setResource(Bitmap resource) {

          RoundedBitmapDrawable circularBitmapDrawable =

                  RoundedBitmapDrawableFactory.*create*(**context**.getResources(), resource);

          circularBitmapDrawable.setCircular(**true**);

          picture.setImageDrawable(circularBitmapDrawable);

      }

  });

}

1. Seu método completo ficará da seguinte forma:

@Override

**public** View getView(**int** position, View view, ViewGroup parent) {

*// Se a view for nula, infla do layout*

**if** (view == **null**)

      view = *inflater*.inflate(R.layout.***row***, **null**);

*// A ButterKnife pode ser um pouco problemática em um Adapter, então colete as Views de row do modo tradicional*

TextView brief = (TextView) view.findViewById(R.id.***row\_message***);

  TextView from = (TextView) view.findViewById(R.id.***row\_contact***);

  TextView subject = (TextView) view.findViewById(R.id.***row\_subject***);

**final** ImageView picture = (ImageView) view.findViewById(R.id.***contact\_picture***);

*// Define os respectivos textos dado uma position*

brief.setText(**data**.get(position).getMessage());

  from.setText(**data**.get(position).getFrom());

  subject.setText(**data**.get(position).getSubject());

  ColorGenerator generator = ColorGenerator.*MATERIAL*;

**int** color = generator.getColor(**data**.get(position).getFirstLetter());

  TextDrawable drawable = TextDrawable.*builder*()

          .buildRoundRect(**data**.get(position).getFirstLetter(), color, 46);

  picture.setImageDrawable(drawable);

**if** (**data**.get(position).getImageUrl() != **null**) {

      Glide.*with*(**context**).load(**data**.get(position)

              .getImageUrl())

              .asBitmap()

              .centerCrop()

              .into(**new** BitmapImageViewTarget(picture) {

          @Override

**protected void** setResource(Bitmap resource) {

              RoundedBitmapDrawable circularBitmapDrawable =

                      RoundedBitmapDrawableFactory.*create*(**context**.getResources(), resource);

              circularBitmapDrawable.setCircular(**true**);

              picture.setImageDrawable(circularBitmapDrawable);

          }

      });

  }

**return** view;

}

**Entendendo o código:** Vamos entender algumas partes do novo código:

**if** (**data**.get(position).getImageUrl() != **null**)

Este trecho apenas compara se o campo imageUrl no objeto de E-mail realmente possui uma URL. Se sim, iniciamos a requisição da imagem:

Glide.*with*(**context**).load(**data**.get(position)

              .getImageUrl())

              .asBitmap()

              .centerCrop()

              .into(**new** BitmapImageViewTarget(picture) {

Inicializamos a biblioteca Glide com algumas configurações. O método load() especifica a URL a da imagem. O método getImageUrl() é o comando para obter a imagem em si. Especificamos o tipo da nossa imagem com o método asBitmap(), e neste caso a imagem será um bitmap. Entre outras configurações que você pode encontrar na documentação da biblioteca (<https://github.com/bumptech/glide>).

  @Override

**protected void** setResource(Bitmap resource) {

/ ...

  }

Similar ao Ion este é o método invocado pelo Observer quando uma resposta é obtida. Neste caso a resposta estará no parâmetro resource.

RoundedBitmapDrawable circularBitmapDrawable =

                      RoundedBitmapDrawableFactory.*create*(**context**.getResources(), resource);

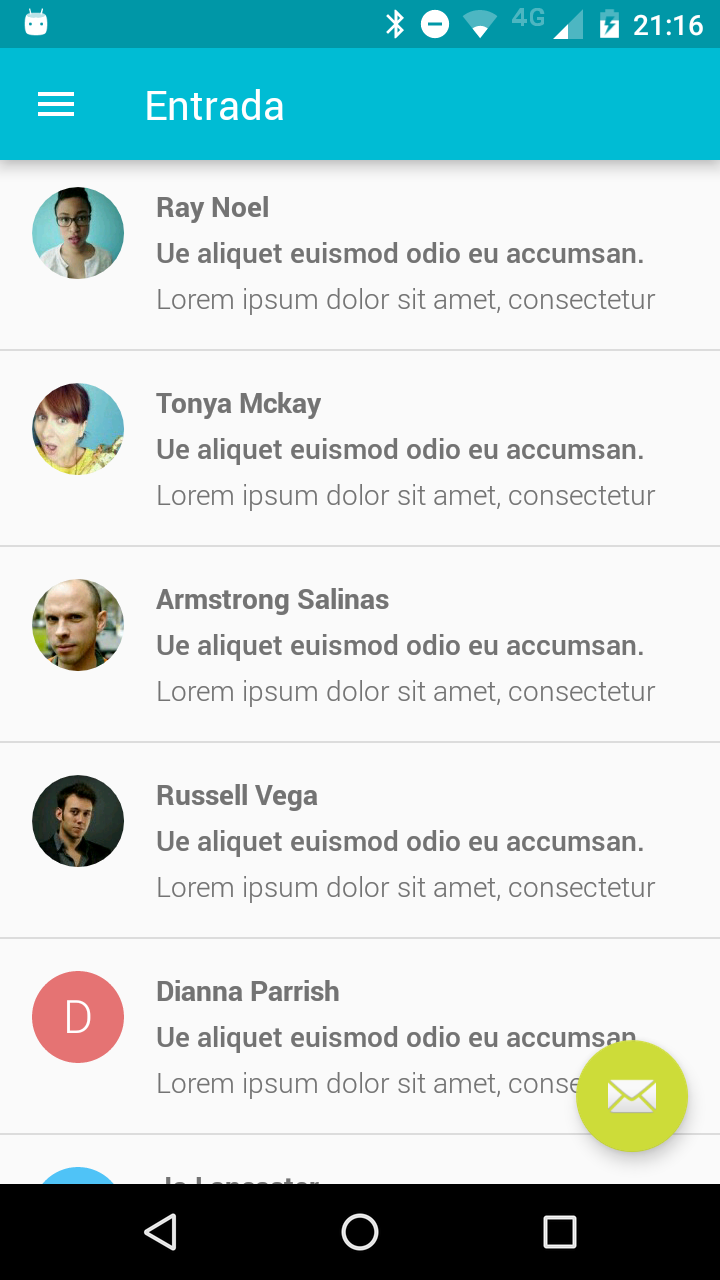
circularBitmapDrawable.setCircular(**true**);

Para adequar ao nosso design, este trecho de código irá deixar nossas imagens circulares.

picture.setImageDrawable(circularBitmapDrawable);

Neste trecho atribuímos a imagem obtida e tratada ao nosso ImageView.

Agora podemos executar o código e ver o resultado final. Você obterá a seguinte tela:



## Considerações finais

Caro aluno, ou melhor, desenvolvedor Android. Existe ainda um mundo de funcionalidades que o Android disponibiliza para nós, afinal este é um sistema operacional que está superando até mesmo os desktops. Anualmente novas funcionalidades e atualizações são lançadas, então não podemos parar nunca. Esteja sempre em contato, praticando, lendo e melhorando. É assim que nos tornamos um bom profissional. Se houver uma dificuldade não desista, afinal o Android é tão amplo que seu limite é a sua imaginação. Nos vemos no curso de iOS.