Unidade 7

# 

Aula 1 - Networking - Parte 1

Até agora estivemos preso a aplicativos que trabalham offline (sem conexão com a internet), mas sabemos que a maioria dos apps que temos hoje em dia opera de maneira online.

A conectividade é tudo. Nos primórdios dos smartphones, antes da popularização da internet móvel, a quantidade de aplicativos era bem limitada, e entre um modelo de celular e outro sempre víamos mais do mesmo.

Com a popularização das redes móveis e a conectividade dos smartphones, as possibilidades se multiplicaram. Hoje em dia a maioria dos aplicativos está conectada a internet, seja salvando as preferências do usuário, atualizando o conteúdo do app ou até mesmo para jogos online. Definitivamente nossos aplicativos precisam estar conectados à internet.

Para utilizarmos este recurso, você precisará entender alguns conceitos do mundo web, como o protocolo HTTP, Restful e a camada Service do nosso aplicativo. Então vamos lá.

## REST

O primeiro conceito que precisamos entender é a maneira como um aplicativo conversa com o servidor. Para um app manter seus dados salvos para serem recuperados em outro smartphone ou plataforma, é necessário que um servidor opere a seu favor. Este servidor irá guardar os dados, cuidar das regras de negócio, fazer autenticação de usuários, atualizar o conteúdo e muito mais. Em palavras mais breves, o servidor retira toda a complexidade existente do lado do app e a manipula no servidor.

Se referir a esta entidade que “faz tudo” como servidor é um pouco simplista. O nome desta entidade é **web-server.** Arquiteturalmente falando, o web-server está no **backend** e o nosso aplicativo está no **frontend.** Dizemos *frontend* pois não se trata apenas do aplicativo, mas de todos os softwares que consomem os web-servers. Por outro lado, o web-server não é o *backend****,*** mas ele está contido nele juntamente com uma série de outros componentes (banco de dados, servidor de arquivos, outros web-servers, etc.).

Bom, definidos os termos, vamos agora entender como o *frontend* conversa com o *backend.*

Existem dois estilos arquiteturais que definem como o *frontend* deve se comunicar com o *backend:* o REST e o SOAP.

O SOAP (Simple Object Access Protocol, em português Protocolo Simples de Acesso a Objetos) é um protocolo para troca de informações estruturadas em uma plataforma descentralizada e distribuída. Ele se baseia na Linguagem de Marcação Extensível (XML) para seu formato de mensagem.

Já o REST (Representational State Transfer, em português Transferência de Estado Representacional), é uma abstração da arquitetura da World Wide Web, mais precisamente, é um estilo arquitetural que consiste de um conjunto coordenado de restrições arquiteturais aplicadas a componentes, conectores e elementos de dados dentro de um sistema de hipermídia distribuído. REST é o mais utilizado atualmente e é ele que iremos detalhar.

O objetivo do REST é ignorar os detalhes da implementação dos web-services e oferecer uma interface limpa, com alguns métodos, para o usuário (frontend) consumir. Como dissemos acima, o web-service guarda toda a complexidade de uma aplicação, mas deve existir uma forma de acessar os seus métodos. Este meio é o REST.

Ele é frequentemente aplicado à web services fornecendo APIs para acesso a um serviço qualquer na web. Ele usa integralmente as mensagens HTTP para se comunicar através do que já é definido no protocolo sem precisar "inventar" novos protocolos específicos para aquela aplicação.

Você trabalha essencialmente com componentes, conectores e dados. Veja algumas das características do REST:

* Ele usa o protocolo HTTP (verbos, accept headers, códigos de estado HTTP, Content-Type) de forma explícita e representativa para se comunicar. . Os URIs (*Uniform Resource Identifier*, em português Identificador Uniforme de Recursos) são usados para expor a estrutura do serviço. Utilizam uma notação comum para transferência de dados como XML ou JSON. Iremos detalhar mais este trecho adiante.
* Não possui estado entre essas comunicações, ou seja, cada comunicação é independente e uniforme (padronizada) precisando passar toda informação necessária.
* Ele deve facilitar o cache de conteúdo no cliente.
* Deve ter clara definição do que faz parte do cliente e do servidor. O frontend não precisa saber como o backend armazena dados, por exemplo. Assim cada implementação não depende da outra e se torna mais escalável.
* Permite o uso em camadas facilitando a escalabilidade, a confiabilidade e a segurança.
* Frequentemente é criado com alguma forma de extensibilidade.

Quando a aplicação cumpre todos os itens acima, dizemos que ela é Restful, ou seja, tem capacidade de fazer REST.

Falhando em um dos cinco primeiros itens, a arquitetura não pode ser classificada formalmente como RESTful. Mas nem todo mundo se apega ao formalismo.

## O protocolo HTTP

Já que REST é um conceito arquitetural para comunicação entre plataformas, ele precisa de um protocolo que o normalize.

O REST usa o protocolo HTTP (*Hypertext Transfer Protocol*). Você já deve ter ouvido ou visto esta sigla antes. Ela aparece no seu navegador ao lado da barra de endereço:

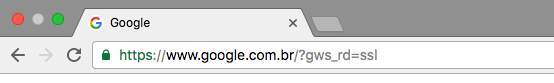


Figura 1 - Exemplo comum de onde utilizamos HTTP e HTTPS

Nesta imagem vemos a sigla https (*Hypertext Transfer Protocol Secure*), que é uma extensão do http.

O *HyperText Transfer Protocol* que em português significa "Protocolo de Transferência de Hipertexto", é um protocolo de comunicação entre sistemas de informação que permite a transferência de dados entre as redes de computadores, principalmente na *World Wide Web (Internet)*.

O HTTP é o protocolo utilizado para transferência de páginas HTML do computador para a Internet, por isso os endereços dos websites (URLs) utilizam no início a expressão "http://", definindo o protocolo usado. Esta informação é necessária para estabelecer a comunicação entre a URL e o servidor Web que armazena os dados, enviando então a página HTML solicitada pelo usuário.

Para que a transferência de dados na Internet seja realizada, o protocolo HTTP necessita estar agregado a outros dois protocolos de rede: TCP (*Transmission Control Protocol*) e IP (*Internet* *Protocol*). Esses dois últimos protocolos formam o modelo TCP/IP, necessários para a conexão entre computadores clientes-servidores.

### Métodos HTTP

Há todo momento quando estamos navegando na web, nosso navegador está enviando requisições para um servidor e o servidor, por sua vez, está nos devolvendo uma resposta em um formato específico ou realizando uma ação de acordo com o que pedirmos para ele fazer.

Nas requisições especificamos o que chamamos de método HTTP ou verbo. Na versão 1.1 do protocolo HTTP (que é a que todos usamos atualmente) temos 9 verbos diferentes.

* **GET**: Requisita uma representação do recurso especificado (o mesmo recurso pode ter várias representações ao exemplo de serviços que retornam XML e JSON). Na prática é um serviço que retorna dados do servidor, em um formato comum entre ele (servidor) e o cliente. Um recurso pode ser entendido como um dado ou informação específica, como uma lista de usuários, e com o GET podemos obter esta lista em diferentes representações (XML, JSON, HTML, etc).
* **POST**: Envia uma entidade e requisita que o servidor aceita-a como subordinada do recurso identificado pelo URI.
* **PUT**: Requisita que um entidade seja armazenada embaixo do URI fornecida. Se a URI se refere a um recurso que já existe, ele é modificado; se o URI não aponta para um recurso existente, então o servidor pode criar o recurso com esse URI.
* **DELETE**: Apaga o recurso especificado.
* **HEAD**: Retorna os cabeçalhos de uma resposta (sem o corpo contendo o recurso).
* **TRACE**: Ecoa de volta a requisição recebida para que o cliente veja se houveram mudanças e adições feitas por servidores intermediários.
* **OPTIONS**: Retorna os métodos HTTP que o servidor suporta para a URL especificada.
* **CONNECT**: Converte a requisição de conexão para um túnel TCP/IP transparente, usualmente para facilitar a comunicação criptografada com SSL (HTTPS) através de um proxy HTTP não criptografado.
* **PATCH**: Usado para aplicar modificações parciais a um recurso.

Agora você vai entender o que estamos tentando fazer este tempo todo. Vamos ao exemplo:

Imagine como um exemplo o nosso TDP, que sua finalidade é consultar as informações sobre filmes. Você cadastraria todos os filmes existentes? Ou cadastraria apenas os filmes dos últimos dez anos? Se você respondeu sim, então boa sorte, pois isto vai tomar uma boa parte do seu tempo. Caso respondeu não, como 99% das pessoas respondem a esta pergunta, você precisa começar a pensar em onde conseguir tantas informações sobre os filmes.

A boa notícia é que existem web-servers que disponibilizam essas informações gratuitamente. Vamos tentar pegar os dados do popular filme **Clube da luta.** Abra no seu navegador o exemplo abaixo:

<https://api.themoviedb.org/3/movie/550?api_key=d272326e467344029e68e3c4ff0b4059>

Implicitamente você está fazendo uma requisição **GET** usando o **HTTPS** e como resultado você receberá um arquivo JSON todo bagunçado. Para melhor visualização pegamos este **JSON** e deixamos de uma forma mais legível para você:

{

"adult": false,

"backdrop\_path": "/8uO0gUM8aNqYLs1OsTBQiXu0fEv.jpg",

"belongs\_to\_collection": null,

"budget": 63000000,

"genres": [

{

"id": 18,

"name": "Drama"

}

],

"homepage": "http://www.foxmovies.com/movies/fight-club",

"id": 550,

"imdb\_id": "tt0137523",

"original\_language": "en",

"original\_title": "Fight Club",

"overview": "A ticking-time-bomb insomniac and a slippery soap salesman channel primal male aggression into a shocking new form of therapy. Their concept catches on, with underground \"fight clubs\" forming in every town, until an eccentric gets in the way and ignites an out-of-control spiral toward oblivion.",

"popularity": 5.56269,

"poster\_path": "/811DjJTon9gD6hZ8nCjSitaIXFQ.jpg",

"production\_companies": [

{

"name": "Regency Enterprises",

"id": 508

},

{

"name": "Fox 2000 Pictures",

"id": 711

},

{

"name": "Taurus Film",

"id": 20555

},

{

"name": "Linson Films",

"id": 54050

},

{

"name": "Atman Entertainment",

"id": 54051

},

{

"name": "Knickerbocker Films",

"id": 54052

}

],

"production\_countries": [

{

"iso\_3166\_1": "DE",

"name": "Germany"

},

{

"iso\_3166\_1": "US",

"name": "United States of America"

}

],

"release\_date": "1999-10-14",

"revenue": 100853753,

"runtime": 139,

"spoken\_languages": [

{

"iso\_639\_1": "en",

"name": "English"

}

],

"status": "Released",

"tagline": "How much can you know about yourself if you've never been in a fight?",

"title": "Fight Club",

"video": false,

"vote\_average": 8.1,

"vote\_count": 5357

}

O resultado desta requisição é um JSON veja que ele realmente possui algumas informações sobre o filme (em inglês). Mas o que é JSON?

## JSON

JSON (JavaScript Object Notation) é uma notação que descreve um objeto da linguagem de programação JavaScript. Ela é amplamente utilizada na web e seus objetos são definidos através da notação JSON.

Esta notação é muito usada nas aplicações mobile por ser leve (arquivos geralmente pequenos), diminuindo o tempo de resposta das requisições (pois são menos dados trafegando na rede) e é de fácil compreensão.

JSON usa o padrão “chave-valor” onde para cada chave há um valor. Vejamos um trecho do exemplo acima:

"tagline": "How much can you know about yourself if you've never been in a fight?"

"title": "Fight Club"

"tagline"é a chave para o valor "How much can you..." e "title" é a chave para o valor "Fight Club".

No começo pode parecer estranho, mas logo você pega o jeito. Como o foco do nosso curso não é explicar conceitos de conectividade, passamos por este assunto rapidamente para que possamos fazer uso na aula seguinte, mas aconselhamos que você procure por leituras complementares, em especial sobre o HTTP e o JSON.

## A biblioteca Gson

Na aula 1 da unidade 6 criamos a interface da tela **Entrada (Inbox)**, para preencher as células da ListView, criamos um array de objetos **Email**, pré-definidos. Na vida real não será assim, pois neste caso os e-mails deveriam vir de um backend em formato JSON. Pensando nisto vamos aprender agora como manipular um objeto JSON usando a biblioteca Gson que foi escrita na linguagem Java.

A Gson é uma biblioteca que pode ser utilizada para converter objetos em JSON (chamamos este processo de serialização). É possível também fazer o processo inverso (chamado deserialização). A Gson pode trabalhar com objetos arbitrários incluindo objetos pré-existentes que não temos acesso ao código (código fechado).

Existem algumas outras bibliotecas que realizam esta função, porém a maioria requer que seja inserida uma série de anotações Java nas nossas classes e isto é uma coisa que não conseguiríamos fazer se não tivéssemos acesso ao código-fonte destas classes. A maioria delas também não dá suporte aos Generics do Java, mas o Gson considera estes dois empasses.

### Objetivos do Gson

* Providenciar simples métodos **toJson()** e **fromJson()** para converter objetos em JSON e vice-versa.
* Permite realizar conversões em classes com código fechado.
* Suporte aos Generics Java.
* Permite uma representação personalizada para os objetos.
* Suporta objetos complexos arbitrariamente.

Vamos mudar nosso exemplo e passar a usar o JSON como fonte de dados da nossa lista de e-mail. Com isso estaremos a um passo de obter os e-mails pela internet, via HTTP.

1. Preparamos um projeto como ponto de partida para você. Descompacte o arquivo **Unidade\_7\_-\_Aula\_1\_-\_Exemplo\_1.zip** e abra o projeto no seu Android Studio.
2. Adicionamos um arquivo JSON ao projeto contendo os e-mails que serão listados. Este arquivo se encontra na pasta assets (**.../app/src/main/assets**), mesma pasta onde colocamos as fontes na unidade passada. Dê uma olhada nele e entenda como é feito um arquivo JSON.
3. Como sempre, adicione uma nova dependência no seu arquivo **build.gradle (Module: app)**:

compile **'com.google.code.gson:gson:2.6.2'**

1. Para transformar este JSON em e-mails, fazemos o procedimento chamado Deserialização, ou também conhecido como Parsing. Primeiro devemos ler o arquivo transformando-o em String, depois o convertemos para um tipo de array especial, o JsonArray, e por último, através deste array criamos um ArrayList de objetos Email. Entre na classe **Utils.java** e insira o método responsável por ler um arquivo do diretório do projeto (chamado bundle) e devolve em formato String:

**public static** String loadJSONFromAsset(String fileName) {

String json = **null**;

**try** {

InputStream inputStream = MailListApplication.*getInstance*().getApplicationContext().getAssets().open(fileName);

**int** size = inputStream.available();

**byte**[] buffer = **new byte**[size];

inputStream.read(buffer);

inputStream.close();

json = **new** String(buffer, **"UTF-8"**);

} **catch** (IOException ex) {

ex.printStackTrace();

**return null**;

}

**return** json;

}

1. Tendo nosso JSON em formato de String o Gson entra em ação. Ainda na classe Utils, adicione o seguinte método:

**public static** ArrayList<Email> parseEmails (String jsonString){

*// Novo parser (serializa/deserializa)*

JsonParser parser = **new** JsonParser();

*// Obtém elemento JSON através parser da jsonString*

JsonElement element = parser.parse(jsonString);

*// Inicializa um ArrayList para guardar os objetos Email*

ArrayList<Email> emailsList = **new** ArrayList<>();

*// Verifica se o elemento JSON contém apenas um JsonObject ou um conjunto deles*

**if** (element.isJsonArray()) {

JsonArray emails = element.getAsJsonArray();

**for** (**int** i = 0; i < emails.size(); i++) {

*// Obtém o JsonObject correspondente do index i do array email*

JsonObject emailJson = emails.get(i).getAsJsonObject();

*// Obtém os valores de cada chave do JsonObject*

String name = emailJson.get(**"from"**).getAsString();

String subject = emailJson.get(**"subject"**).getAsString();

String message = emailJson.get(**"message"**).getAsString();

*// Instancia e adiciona um novo objeto Email na lista de e-mails*

emailsList.add(**new** Email(subject, name, message));

}

}

*// Retorna a lista de e-mails*

**return** emailsList;

}

**Entendendo o código:** A primeira coisa a ser feita é a instanciação de um parser de JSON (JsonParser) e através dele deserializamos o JSON em formato de String usando o JsonElement. JsonElements podem conter um JsonArray ou um JsonObject. Um JsonArray é composto por JsonObjects, e no nosso caso, um JsonObject representa um único e-mail isolado. Como temos uma lista de e-mails, utilizamos o JsonArray. Tendo identificado que nosso JsonElement é um JsonArray, iteramos por este array e instanciamos novos objetos Email baseado no conteúdo de cada JsonObject contido neste JsonArray.

1. Feito isto basta usarmos na nossa classe **InboxFragment**. Primeiro apague o array de e-mails
2. Agora substitua o conteúdo do método **initListView()** pelo seguinte:

**public void** initListView() {

String json = Utils.*loadJSONFromAsset*(**"emails.json"**);

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

}

1. Rode a aplicação e veja o resultado.

Você não verá diferenças na interface do aplicativo, pois apenas mudamos o jeito de popular nossa ListView. Apesar do uso da Gson neste momento parecer não ter necessidade, ele acabou limpando a classe InboxFragment. Se tivéssemos 20 mil itens para serem apresentados na lista, a classe ficaria muito grande e difícil de gerenciá-la. Separar os dados da lógica é uma boa prática.

Você verá a real utilidade da Gson na aula seguinte, quando puxarmos os dados (e-mails) da internet.

## Resumo

Esta aula foi um tanto quanto teórica, mas você pode conhecer o coração dos aplicativos. Hoje em dia são poucos os aplicativos sem um back-end por trás. Então você aprendeu o conceito básico de Cliente-Servidor, Back-end e Front-End, comunicação pela rede através do protocolo HTTP, e por fim o formato de representação de objetos JSON. Estes conceitos serão usados por toda sua carreira como desenvolvedor de aplicativos e no curso de iOS isto será um pouco mais aprofundado.

Aula 2 - Networking - Parte 2

# 

## Padrão de Projeto - Observer

Agora iremos falar do último padrão: o Observer. Ele servirá para entendermos melhor como são feitas as chamadas HTTP no app.

### Objetivo

* Definir uma dependência um-para-muitos entre objetos de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.
* Encapsular os componentes núcleo em uma abstração Subject, e componentes variáveis em uma hierarquia Observer.
* 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.

### Problema

Precisamos interceptar ou receber notificações quando um objeto muda de estado de forma que o fluxo principal do programa não precise esperar por estas notificações, ou seja, de forma assíncrona.

### Discussão

O padrão Observer define um objeto que é o "guardião" do modelo de dados e/ou lógica de negócios (o Subject). Também delega todas as funcionalidades de “View” para desacoplar e distinguir objetos Observers, ou seja, um Observer só estará vinculado a apenas uma View. Observers registram-se juntamente com os Subjects ao qual foram cridos. Sempre que o Subject muda, um broadcast é disparado para todos os observers registados.

O protocolo descrito acima especifica um modelo de interação "puxar". Ao invés do Subject “empurrar” o que mudou aos Observers, cada Observer se encarrega de “puxar”.

O padrão Observer capta a maior parte da arquitetura MVC que tem sido uma parte da comunidade Smalltalk por anos.

### Estrutura

O Subject representa a abstração central. O Observer representa a abstração variável do contexto. O Subject solicita aos objetos Observers que realizem e tomem uma ação quando um evento acontece. Cada observador pode chamar de volta o Subject, conforme necessário.

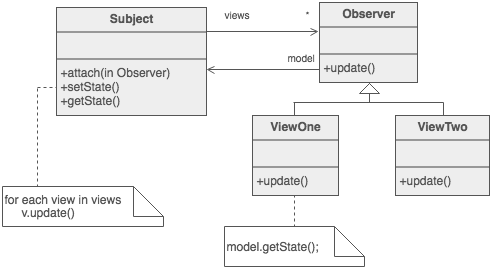


Figura 2 - Diagrama de classes que representa o padrão Observer

## 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 e-mails foi colocado em uma pasta pública do Dropbox. Através desta pasta pública podemos obter o link do arquivo e consumi-lo.

O link para a pasta 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 da 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 analisar 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. A 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 a 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. Precisamos atualizar nosso método **parseEmail()** para que ele guarde a URL das imagens que aparecerão na nossa lista. Para isto, entre na classe **Utils**, e adicione a seguinte linha de código (veja com atenção o local onde ela está posicionada):

public staticArrayList<Email> parseEmails (String jsonString){  
  
 // Novo parser (serializa/deserializa)  
 JsonParser parser = new JsonParser();  
  
 // Obtém elemento JSON através parser da jsonString  
 JsonElement element = parser.parse(jsonString);  
  
 // Inicializa um ArrayList para guardar os objetos Email  
 ArrayList<Email> emailsList = new ArrayList<>();  
  
 // Verifica se o elemento JSON contém apenas um JsonObject ou um conjunto deles  
 if (element.isJsonArray()) {  
  
 JsonArray emails = element.getAsJsonArray();  
 for (int i = 0; i < emails.size(); i++) {  
  
 // Obtém o JsonObject correspondente do index i do array email  
 JsonObject emailJson = emails.get(i).getAsJsonObject();  
  
 // Obtém os valores de cada chave do JsonObject  
 String name = emailJson.get("from").getAsString();  
 String subject = emailJson.get("subject").getAsString();  
 String message = emailJson.get("message").getAsString();

String image = **null**;  
 **if** (emailJson.get(**"image"**) != **null**) {  
 image = emailJson.get(**"image"**).getAsString();  
 }

// Instancia e adiciona um novo objeto Email na lista de e-mails  
 emailsList.add(new Email(subject, name, message));  
 }

}  
  
// Retorna a lista de e-mails  
return emailsList;

}

1. Ainda é preciso inserir esta **image** no construtor do e-mail (na próxima instrução):

public staticArrayList<Email> parseEmails (String jsonString){  
  
 // Novo parser (serializa/deserializa)  
 JsonParser parser = new JsonParser();  
  
 // Obtém elemento JSON através parser da jsonString  
 JsonElement element = parser.parse(jsonString);  
  
 // Inicializa um ArrayList para guardar os objetos Email  
 ArrayList<Email> emailsList = new ArrayList<>();  
  
 // Verifica se o elemento JSON contém apenas um JsonObject ou um conjunto deles  
 if (element.isJsonArray()) {  
  
 JsonArray emails = element.getAsJsonArray();  
 for (int i = 0; i < emails.size(); i++) {  
  
 // Obtém o JsonObject correspondente do index i do array email  
 JsonObject emailJson = emails.get(i).getAsJsonObject();  
  
 // Obtém os valores de cada chave do JsonObject  
 String name = emailJson.get("from").getAsString();  
 String subject = emailJson.get("subject").getAsString();  
 String message = emailJson.get("message").getAsString();

String image = null;  
 if (emailJson.get("image") != null) {  
 image = emailJson.get("image").getAsString();  
 }

// Instancia e adiciona um novo objeto Email na lista de e-mails  
 **emailsList**.**add(new Email(subject, name, message, image));**  
 }  
 }  
  
 // Retorna a lista de e-mails  
 return emailsList;  
  
}

1. Agora visite a classe **InboxAdapter** e altere o método **getView()** inserindo 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);

      }

  });

}

**Entendendo o código:** Estes métodos irão preencher o local onde é destinado a inicial do nome do contato do e-mail (remetente) com uma imagem, se ela existir. Adicionalmente a Glide arredonda a imagem para corresponder com o nosso design

1. Para manipular a variável **picture** dentro da implementação do método **setResource()**, é preciso que ela seja modificada para **final**. Com isto você não poderá atribuir mais nada à ela, pois ela se tornará uma constante e apenas seus atributos poderão ser manipulados. Coloque a palavra chave **final** na declaração de **picture**.
2. 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 Email 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);

No trecho acima atribuímos a imagem obtida e tratada ao nosso ImageView.

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

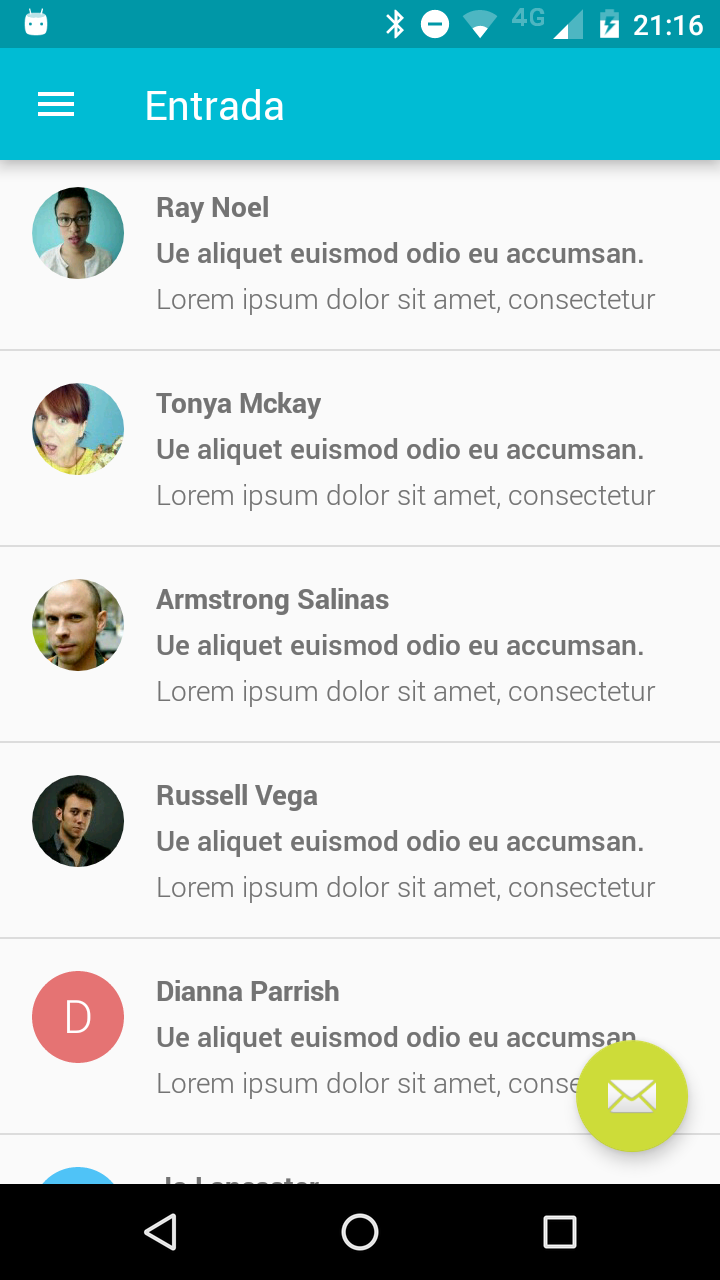


Figura 3 - Exemplo final da nossa tela Inbox (Entrada)

## Resumo

Após aprendermos os conceitos sobre Networking na aula passada, chegou a hora de aplica-los. Nesta aula você aprendeu a obter objetos pela rede, e como exemplo hospedamos um objeto json para que ele seja obtidos. Você viu também que podemos pegar qualquer imagem da internet utilizando a biblioteca Glide, basta termos a respectiva URL. Com estes conceitos fechamos os conteúdo do curso de Android.

Neste curso foi apresentado tudo que você precisa saber para criar seu próprio aplicativo sozinho. Você agora pode pesquisar e se aventurar mais a fundo nas bibliotecas de terceiros e na documentação do Android, já que você sabe como funciona uma aplicativo como um todo. Bons estudos!

## 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 bons profissionais. Se houver uma dificuldade não desista, afinal o Android é tão amplo que seu limite é a sua imaginação.