# SIMPLE REST CLIENT 1.0
## Guia de Referência

por Carlos Emilio Padilla Severo

### INTRODUÇÃO

Simple Rest Client é um projeto de código aberto que visa a elaboração de uma biblioteca para auxílio a desenvolvedores na construção de aplicações cliente de Web Services. A biblioteca fornece interfaces e classes que buscam simplificar o envio de requisições HTTP à serviços disponibilizados por servidores de Web Services.

### A interface RestClientInterface<T>

A interface genérica RestClientInterface<T> é utilizada para declaração de um objeto de conexão com o Web Services. Vejamos como declarar a variável que servirá de cliente para o envio de métodos de requisição HTTP.

**Exemplo:**

Para exemplificar, criaremos uma referência a uma conexão com um Web Services de notícias acadêmicas. A classe dos objetos correspondentes às notícias acadêmicas é denominada **Noticia**.

*RestClientInterface<Noticia> cliente;*

### A classe RestClient<T>

A classe genérica RestClient<T> irá implementar os comportamentos definidos na interface genérica RestClientInterface<T>.

Inicialmente, invocaremos o método de classe denominado *open()* para obtenção de uma instância de RestClient, a qual permitirá o envio de requisições HTTP à aplicação de Web Services.

**Exemplo:**

```
String uri = "http://localhost:8080/noticiaApp/servicos/noticias";
RequestContentType contentType = RequestContentType.JSON;
RestClientInterface<Noticia> cliente;
cliente = RestClient.open(uri, contentType);
```

A **uri** é um identificador do recurso do servidor de Web Services. A uri é composta de:

endereço do servidor: no exemplo estamos usando localhost para hospedar o servidor de Web Services. Portanto, identificamos o servidor na forma http://localhost:8080/

nome da aplicação: identifica o nome a aplicação de Web Services. No exemplo, o nome é noticiaApp.

Classe de configuração: o nome da classe controladora do Web Services. No exemplo, o nome é servicos.

Nome da classe de Web Services: classe que implementa os recursos solicitados pelos clientes. No exemplo, o nome da classe é noticias.

Já **contentType** é uma constante definida em uma enumeração Java, a qual identifica o tipo de conteúdo solicitado por uma requisição. No exemplo, está sendo enviada uma requisição cujo conteúdo gerado será no formato JSON (*JavaScript Object Notation*).

O método *open()* é um método de classe (estático) definido na classe RestClient, o qual abrirá uma conexão com o servidor de Web Services. A conexão será retornada para o objeto denominado cliente.

A seguir, veremos como enviar métodos de requisições do tipo POST, PUT, DELETE e GET a um Web Services. Note, que após o envio de um método de requisição, devemos fechar a conexão. Isso é realizado através do método *close()*, definido na interface da classe RestClient.

**Exemplo:**

```
String uri = "http://localhost:8080/noticiaApp/servicos/noticias";
RequestContentType contentType = RequestContentType.JSON;
RestClientInterface<Noticia> cliente;
cliente = RestClient.open(uri, contentType);

… //Aqui vai algum método de requisição que veremos a seguir.

cliente.close();
```

### Requisições POST

Para o envio de requisições do tipo POST, devemos utilizar o método específico definido na interface de RestClient.

Após, a obtenção de uma conexão com o método *open()*, utilizamos o método *post(Object o)* para o envio do objeto que será persistido no banco de dados mantido pelo Web Services. Ou seja, utilizamos POST para requisições do tipo INSERT.

**Exemplo:**

```
String uri = "http://localhost:8080/noticiaApp/servicos/noticias";
RequestContentType contentType = RequestContentType.JSON;
RestClientInterface<Noticia> cliente;
Noticia noticia = new Noticia("Lançamento da versão 1.0");
cliente.post(noticia);
```

```
cliente.close();
```

No exemplo anterior, declaramos uma variável do tipo Notícia e instanciamos o objeto, criando uma nova notícia. A notícia será persistida por um serviço disponibilizado no Web Services noticias, o qual é requisitado através do método *post()*.

Após o envio da requisição e seu processamento pelo Web Services, encerramos a comunicação da aplicação cliente com o servidor através do método *close()*.

## Requisições PUT

Um método específico definido na interface da classe RestClient permite o envio de requições do tipo PUT. As requisições PUT são utilizadas para alteração de dados. Ou seja, usaremos requisições do tipo PUT para realizarmos operações UPDATE em tabelas do banco de dados mantido pela aplicação de Web Services.

Vejamos um exemplo de alteração de uma notícia previamente armazenada pelo nosso Web Services de notícias acadêmicas.

**Exemplo:**

```
String uri = "http://localhost:8080/noticiaApp/servicos/noticias";
RequestContentType contentType = RequestContentType.JSON;
RestClientInterface<Noticia> cliente;
noticia.setNoticia("Lançamento da versão 1.0 da biblioteca");
cliente.put(noticia);
cliente.close();
```

Após a alteração da notícia, através do método *setNoticia(String noticia)*. Usamos o método *put(Object obj)* para o envio da notícia ao Web Services de notícias acadêmicas.

## Requisições DELETE

As requisições do tipo DELETE são utilizadas quando desejarmos excluir algum dado. Ou seja, utilizaremos este tipo de requisição quando desejarmos invocar um DELETE em tabelas do banco de dados mantido pelo Web Services.

Vejamos um exemplo de exclusão de uma notícia previamente armazenada pelo nosso Web Services de notícias acadêmicas.

**Exemplo:**

```
String uri = "http://localhost:8080/noticiaApp/servicos/noticias";
RequestContentType contentType = RequestContentType.TEXT;
RestClientInterface<Noticia> cliente;
uri += "deleteNoticia/" + noticia.getId();
cliente = RestClient.open(uri, RequestContentType.JSON);
cliente.delete();
cliente.close();
```

Note que as requisições baseadas no método DELETE recebem um argumento na própria url. Neste exemplo, passamos como parâmetro o **ID** da notícia a ser excluída.

## Requisições GET

O método de requisição GET é utilizado quando desejamos obter uma lista de objetos do Web Services. Note que o GET está relacionado com o comando SELECT, visto que devemos utilizar quando enviamos uma consulta ao banco de dados mantidos pelo Web Services.

Seguindo nosso exemplo, vejamos como carregar as notícias previamente armazenadas no banco de dados mantido pela aplicação no servidor.

**Exemplo:**

```
String uri = "http://localhost:8080/noticiaApp/servicos/noticias";
RequestContentType contentType = RequestContentType.JSON;
RestClientInterface<Noticia> cliente;
List<Noticia> noticias;
cliente = RestClient.open(uri + "todas", RequestContentType.JSON);
cliente.setType(Noticia[].class);
noticias = cliente.get();
cliente.close();
```

Veja que a uri é constituída do caminho onde se encontra o Web Services mais o recurso solicitado pela aplicação cliente. Neste caso, a aplicação cliente enviará uma requisição solicitando "todas" as notícias encontradas no banco de dados do servidor.

O método *setType(Class[] class)*, definido na interface da classe RestClient, deve ser utilizado para definir o tipo de conjunto de objetos retornado em requisições GET. Neste exemplo, será retornado um conjunto de objetos do tipo Noticia.

A lista de notícias retornada na requisição será guardada em noticias, a qual é uma variável do tipo List. Após realizada a requisição, fechamos a transação com o método *close()*.

Também podemos usar o método GET para obter subconjuntos de objetos. Para isso, devemos enviar requisições com parâmetros que identificam valores que devem ser selecionados na cláusula WHERE de consultas SQL. Vejamos um exemplo:

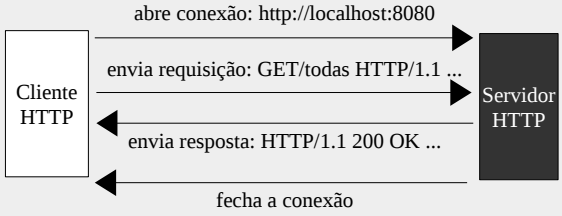
```
String parametro = URLEncoder.encode("23/03/2016, "UTF-8");
String recurso = "pordata/" + parametro;
cliente = RestClient.open(uri + recurso, RequestContentType.JSON);
cliente.setType(Noticia[].class);
noticias = cliente.get();
```

```
cliente.close();
```

Neste exemplo, passamos a data codificada como parâmetro para o recurso “pordata/”. Dessa forma, o servidor irá processar a requisição parametrizada e devolver somente as notícias daquele dia.

## O modelo HTTP de requisição e resposta

Podemos observar que Simple Rest Client apresenta um modelo de comunicação entre aplicações cliente e o servidor, de acordo com o funcionamento de requisições HTTP. Vejamos uma ilustração.




Note que a aplicação cliente abre uma conexão, envia uma requisição através de um método HTTP. Após, o servidor processa a requisição e envia uma resposta. Por fim, a comunicação é encerrada e fechada a conexão.

## Estrutura da aplicação Web Services

Os exemplos apresentados neste guia utilizaram uma aplicação de Web Services que fornece notícias acadêmicas. Usamos esta aplicação para o envio de requisições POST, PUT, DELETE e GET para demonstrar as funcionalidades de Simple Rest Client. Sendo assim, vejamos como é a estrutura do Web Services de notícias acadêmicas.


A aplicação foi organizada em três camadas:

**modelo:** onde está a classe que representa o modelo de uma notícia acadêmica. Uma notícia possui os seguintes atributos.

```
private int id;
private Date data;
private String hora;
private String noticia;
```

**persistencia:** pacote que contém as classes que implementam a lógica de acesso ao banco de dados da aplicação.

A classe ConexaoFabrica fornece um método de classe denominado *getConexao()*, o qual devolve uma instancia de java.sql.Connection para uma classe de acesso a dados (DAO).

Já a classe DAONoticia, fornece métodos para inclusão, alteração, exclusão ou consulta a dados mantidos pela tabela de notícias do banco de dados da aplicação de Web Services.

**services:** classes que implementam os serviços disponibilizados pela aplicação de Web Services RESTful.

A classe ApplicationConfig é a controladora, responsável por direcionar as requisições para os métodos específicos que irão processar a requisição.

A classe Noticia implementa os métodos de tratamento de cada tipo de requisição HTTP enviada pelas aplicações clientes.

O método POST é o responsável pela inserção de uma nova notícia. Vejamos o código do mesmo.

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Path("/postNoticia")
public void postNoticia(String content){
  Noticia noticia;
  noticia = (Noticia) gson.fromJson(content, Noticia.class);
  daoNoticia.incluir(noticia);
}
```

A String *content* é o parâmetro correspondente ao objeto enviado pela aplicação cliente, o qual deve ser persistido pelo método *postNoticia()*. O objeto é enviado no formato JSON, devendo ser convertido para um objeto do tipo Noticia. Após, o método *incluir()* é chamado para que a notícia seja inserida no banco de dados.

O método PUT é o responsável pela alteração de uma notícia. Observe o código a seguir.

```
@PUT
@Consumes(MediaType.APPLICATION_JSON)
@Path("/putNoticia")
public void putNoticia(String content){
  Noticia noticia;
  noticia = (Noticia) gson.fromJson(content, Noticia.class);
  daoNoticia.alterar(noticia);
}
```

O método PUT é muito semelhante a POST. A diferença é que o objeto recebido como argumento deve ser repassado para o método *alterar()*, o qual é responsável pela alteração dos dados na tabela mantida pelo SGBD.

O método DELETE é o responsável pela exclusão de um objeto do banco de dados mantido pela aplicação. Vejamos o código.

```
@DELETE
@Path("/deleteNoticia/{id}")
public void deleteNoticia(@PathParam("id") String id){
   daoNoticia.excluir(Integer.valueOf(id));
}
```

O método recebe como argumento a **id** da notícia a ser excluída. Então, realiza a chamada ao método *excluir()*, o qual é responsável pela exclusão do registro na tabela de notícias do banco de dados.

Na estrutura da classe Noticias de Web Services temos duas formas de retornar um conjunto de notícias. A primeira forma é buscar todas as notícias armazenadas no banco de dados. Já a outra forma é filtrar algumas notícias que serão recuperadas do banco de dados. Usamos o método GET para gerar listas de notícias.

A primeira forma retorna todas as notícias armazenadas no banco de dados, conforme o código a seguir.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/todas")
public String getTodasNoticias(){
   List<Noticia> noticias = daoNoticia.getTodasNoticias();
   return gson.toJson(noticias);
}
```

A segunda forma retorna uma lista de notícias de acordo com o filtro indicado. Neste caso, o filtro indica uma determinada data.

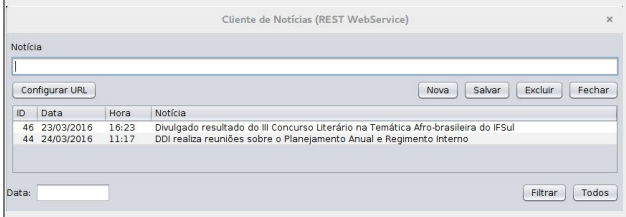
```
@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("/pordata/{data}")
public String getNoticiasPorData(@PathParam("data") String data)
throws ParseException {
   SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");
   Date dataNoticia = df.parse(data);
   List<Noticia> noticias = daoNoticia.getNoticiasDia(dataNoticia);
   return gson.toJson(noticias);
}
```

O código completo da aplicação de Web Services notícias acadêmicas pode ser obtido no Github, através do seguinte endereço:

https://github.com/emilio-severo/noticiasAcademicasWebService

## Uma aplicação Desktop cliente de Web Services

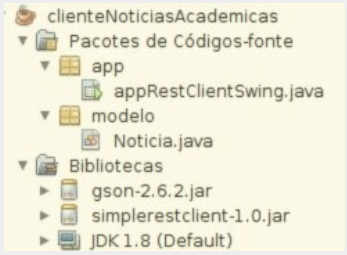
A seguir, veremos como construir uma aplicação cliente do Web Services notícias acadêmicas. A aplicação faz uso de Simple Rest Client para comunicação com os recursos disponibilizados pelo Web Services. A figura abaixo ilustra a interface gráfica da aplicação.




O projeto da aplicação está organizado em dois pacotes.

**app**: onde está a classe principal do aplicativo, apresenta a tela de interação com o usuário.

**modelo**: contém a classe que representa o modelo de objetos do tipo Noticia.




Em bibliotecas devemos adicionar o pacote simplerestclient-1.0.jar, biblioteca que contém os recursos para construção do cliente de Web Services. E, sua dependência, o pacote gson-2.6.2.jar, utilizado para conversão de objetos Java em JSON e vice-versa.

Na classe appRestClientSwing temos os seguintes atributos:

```
RestClientInterface<Noticia> cliente;
List<Noticia> noticias;
Noticia noticia;
URL url;
```

O atributo cliente é uma referência para um objeto do tipo RestClient, o qual será utilizado para envio de requisições HTTP ao Web Services notícias acadêmicas. O atributo notícias é uma lista de objetos do tipo Noticia, a qual é utilizada para armazenar o resultado de requisições do tipo GET. O atributo notícia é utilizado para referenciar um objeto do tipo notícia que será incluído, alterado ou excluído do banco de dados mantido pela aplicação de Web Services notícias

acadêmicas. Já url é um atributo utilizado para identifica o endereço do recurso. A seguir, vejamos o construtor da classe.

```
public appRestClientSwing() {
      initComponents();
      noticia = new Noticia();
      try {
          url = new URL("http://localhost:8080/noticiasAcademicas/" +
                        "servicos/noticias/");
      } catch (MalformedURLException ex) {
         Logger.getLogger(
                        appRestClientSwing.class.getName())
                        .log(Level.SEVERE, null, ex);
      }
      carregarNoticias();
}
```

Após a inicialização dos componentes visuais da interface gráfica da aplicação, um objeto do tipo Noticia é instanciado. Também é instanciado um objeto do tipo URL. Ao final, o método *carregarNoticias()* é chamado para alimentar os dados da Tabela.

```
private void carregarNoticias() {
   cliente = RestClient.open(url + "todas", RequestContentType.JSON);
   cliente.setType(Noticia[].class);
   noticias = cliente.get();
   cliente.close();
   montaTabela();
}
```

O método abre uma conexão com o servidor para enviar uma requisição GET. Assim, o retorno da requisição é armazenado em uma lista de objetos do tipo Noticias. Após, o método *montaTabela()* é chamado para atualizar a interface gráfica da aplicação.

```
private void montaTabela() {
   SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");
   DefaultTableModel dtm;
   dtm = (DefaultTableModel) tbNoticias.getModel();
   dtm.setNumRows(0);
   for (Noticia noticia : noticias) {
        dtm.addRow(new Object[]{noticia.getId(),
                        df.format(noticia.getData()), noticia.getHora(),
                        noticia.getNoticia()});
     }
   }
```

Ao clicar no botão salvar será chamado o método responsável para inserir ou alterar uma notícia. Vejamos o código do evento clique do botão salvar.

```
private void btSalvarActionPerformed(java.awt.event.ActionEvent evt)
{
  if (noticia.getId() == 0) {
     noticia = new Noticia(edNoticia.getText());
     cliente = RestClient.open(url + "postNoticia",
                                RequestContentType.JSON);
     cliente.post(noticia);
  }
  else{
     noticia.setNoticia(edNoticia.getText());
     cliente = RestClient.open(url + "putNoticia",
                                RequestContentType.JSON);
     cliente.put(noticia);
  }
  cliente.close();
  carregarNoticias();
  novaNoticia();
}
```

O clique no botão excluir dispara o evento responsável pela realização de uma requisição do tipo DELETE. Vejamos o código do método a seguir.

```
private void btExcluirActionPerformed(java.awt.event.ActionEvent ev)
{
   cliente = RestClient.open(url +
                        "deleteNoticia/" +
                         noticia.getId(), RequestContentType.JSON);
   cliente.delete();
   cliente.close();
   carregarNoticias();
   novaNoticia();
}
```

O código completo da aplicação cliente para Desktop pode ser obtida no Github. Através do seguinte endereço:

https://github.com/emilio-severo/clienteNoticiasAcademicasDesktop

**Uma aplicação Android cliente de Web Services**

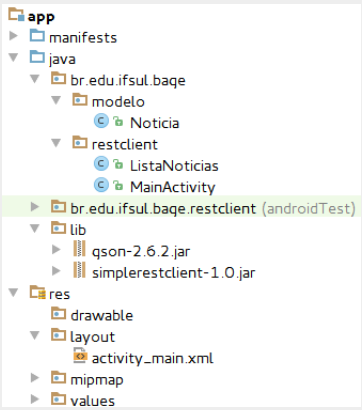
Agora veremos como utilizar Simple Rest Client em uma aplicação cliente do Web Services de notícias acadêmicas escrita para Android.

# SIMPLE REST CLIENT 1.0
## Guia de Referência

por Carlos Emilio Padilla Severo

O projeto é constituído de duas activities. A primeira activity corresponde a tela de entrada do aplicativo, a qual apresenta um cadastro de notícias onde o usuário poderá incluir, alterar ou excluir notícias. Já, a segunda activity é a lista de notícias previamente cadastradas no banco de dados do Web Services de notícias acadêmicas.

Quando o usuário seleciona uma notícia, será redirecionado para a tela de cadastro onde poderá alterar ou excluir a notícia selecionada. A figura a seguir ilustra a estrutura do projeto.




A seguir, vejamos o código da activity de cadastro de notícias. Ou seja, a activity principal da aplicação. Iniciamos pela descrição dos atributos da classe Java correspondente à atividade. Temos seis atributos: alguns atributos que referenciam componentes da interface gráfica (caixa de texto e botões), um atributo que referencia um objeto do tipo Noticia e um atributo referente ao cliente de Simple Rest Client.

```
private EditText texto;
private Button botaosalvar;
private Button botaolistar;
private Button botaoexcluir;
private Noticia noticia;
private RestClientInterface<Noticia> cliente;
```

O método herdado o*nCreate()*, após a inicialização dos elementos de interface gráfica, deverá verificar se algum parâmetro foi enviado pela atividade listagem de notícias. Visto que a atividade de cadastro pode ser chamada pela atividade de listagem, após o usuário selecionar uma determinada notícia a ser editada. Vejamos o código.

```
Intent intent = getIntent();
noticia = (Noticia) intent.getSerializableExtra("noticia");
if(noticia != null)
   texto.setText(noticia.getNoticia());
```

A seguir, são implementados os manipuladores de eventos para cada botão da interface gráfica da atividade de cadastro.

**Clique no botão salvar**

```
botaosalvar.setOnClickListener(new View.OnClickListener() {
  @Override
  public void onClick(View v) {
      if(noticia == null) {
        noticia = new Noticia(texto.getText().toString());
        noticia.setHora(
          new   SimpleDateFormat("hh:mm").format(noticia.getData())
        );
        new Task().execute("POST");
       }
       else{
            noticia.setNoticia(texto.getText().toString());
            new Task().execute("PUT");
       }
       novaNoticia();
       Toast.makeText(getApplicationContext(), "Notícia salva.",
                           Toast.LENGTH_SHORT).show();
   }
});
```

Quando o usuário clicar no botão salvar, primeiramente, verifica-se se há uma instância de notícia sendo editada no momento. Caso o objeto notícia for nulo, então é enviada uma requisição POST para o Web Services para que o objeto seja inserido como novo no banco de dados. Caso contrário, após a edição da notícia já existente, deve-se enviar uma requisição PUT para que o objeto de notícia seja atualizado no banco de dados da aplicação de Web Services.

**Clique no botão excluir**

```
botaoexcluir.setOnClickListener(new View.OnClickListener() {
  @Override
   public void onClick(View v) {
       if(noticia != null){
          new Task().execute("DELETE");
          novaNoticia();
          Toast.makeText(getApplicationContext(), "Notícia excluída.",
                              Toast.LENGTH_SHORT).show();
       }
       else
          Toast.makeText(getApplicationContext(),
          "Selecione uma notícia.",  Toast.LENGTH_SHORT).show();
   }
});
```

Quando o usuário clicar no botão excluir, devemos verificar se há uma notícia selecionada atualmente. Caso afirmativo, uma requisição DELETE é enviada para o Web Services de notícias acadêmicas, passando-se como parâmetro o ID da notícia a ser excluída. Caso contrário, uma mensagem é enviada ao usuário informando que uma notícia deve ser selecionada na lista para que a mesma possa ser excluída.

**Clique no botão listar**

```
botaolistar.setOnClickListener(new View.OnClickListener() {
   @Override
    public void onClick(View v) {
       Intent intent = new Intent(getBaseContext(), ListaNoticias.class);
       startActivity(intent);
   }
});
```

O clique no botão listar remete o usuário para a tela da atividade de listagem de notícias. Nesta tela o usuário poderá visualizar as notícias já cadastradas e selecionar alguma para edição.

**O método novaNoticia()**

```
private void novaNoticia(){
    texto.setText("");
    noticia = null;
}
```

Após a execução de salvar ou excluir sesrá acionando o método novaNoticia(). Isso faz com que a aplicação apague o conteúdo da caixa de texto que contém a descrição da notícia e torne nulo o objeto de referência a uma notícia.

**AsyncTask**

O uso da classe AsyncTask permite a construção de atividades que rodam tarefas em segundo plano, cujos resultados são publicados em componentes da interface gráfica da própria atividade. Esta é uma funcionalidade útil da tecnologia, pois desonera a activity de tarefas que demandam alguns segundos durante sua execução. Vejamos o código da classe Task, a qual estende as funcionalidades de AsyncTask a fim de especificar o comportamento das requisições HTTP necessárias pela atividade de cadastro.

```
private class Task extends AsyncTask<String, Void, Void>{
    @Override
     protected Void doInBackground(String... params) {
     String url;
     url="http://10.8.1.10:8080/noticiasAcademicas/servicos/noticias/";
     if(params[0].equals("POST")) {
        url += "postNoticia";
        cliente = RestClient.open(url, RequestContentType.JSON);
        cliente.post(noticia);
     }
     else if(params[0].equals("PUT")) {
        url += "putNoticia";
        cliente = RestClient.open(url, RequestContentType.JSON);
        cliente.put(noticia);
     }
     else if(params[0].equals("DELETE")) {
        url += "deleteNoticia/" + noticia.getId();
        cliente = RestClient.open(url, RequestContentType.TEXT);
        cliente.delete();
        }
        cliente.close();
        return null;
     }

  }
```

A activity que apresenta a lista de notícias previamente cadastradas ao usuário possui os seguintes atributos:

```
private List<Noticia> noticias;
private ArrayAdapter<Noticia> arrayAdapterNoticias;
RestClientInterface<Noticia> cliente;
```

Uma lista de objetos do tipo notícias que será usada para apresentar as notícias ao usuário na interface gráfica da aplicação. Um objeto do tipo *ArrayAdapter* que irá fazer a vinculação da lista de objetos com o componente de interface gráfica. E, um objeto cliente do servidor de notícias acadêmicas.

O método carregarNoticias() faz a instanciação da AsyncTask responsável pela requisição GET que irá retornar uma lista de notícias do servidor.

```
private void carregarNoticias(){
    Task task = new Task();
     try {
         noticias = task.execute().get();
     } catch (InterruptedException e) {
          e.printStackTrace();
     } catch (ExecutionException e) {
          e.printStackTrace();
     }
     Toast.makeText(getApplicationContext(), "Notícias carregadas.",
                              Toast.LENGTH_SHORT).show();
}
```

Por fim, a estrutura da classe Task que estende as funcionalidades de AsyncTask, implementando a lógica de tratamento da requisição GET, a qual irá retornar uma lista de notícias para a aplicação.

```
private class Task extends AsyncTask<Void, Void, List<Noticia>> {
   @Override
    protected List<Noticia> doInBackground(Void... params) {
      List<Noticia> n;
      cliente = RestClient.open("http://10.8.1.10:8080/"+
                                 "noticiasAcademicas/servicos/noticias/todas",
                                  RequestContentType.JSON);
      cliente.setType(Noticia[].class);
      n = cliente.get();
      cliente.close();
      return n;
   }
}
```

O código completo da aplicação cliente para Android pode ser obtida no Github. Através do seguinte endereço:

https://github.com/emilio-severo/clienteNoticiasAcademicasAndroid