

Requisições HTTP

Usabilidade, Desenv. Web, Mobile e Jogos

Prof. Gustavo Custodio
gustavo.custodio@anhembi.br

Requisições HTTP

- A maioria dos aplicativos móveis dependem de dados que vêm de uma fonte externa.
 - Aplicativos de notícias.
 - Aplicativos par enviar e-mails.
- Chamamos estes serviços de *web services*.
 - Frequentemente fornecidos por meio de APIs REST.

Requisições HTTP

- Quando um aplicativo conecta com um *web service*, ele faz uma requisição HTTP.
- O *backend* recebe a solicitação.
 - Responde enviando dados de volta para o aplicativo.
 - Normalmente em json ou xml.



Requisitos HTTP

Desenvolvendo a Aplicação

Requisições HTTP

- Vamos desenvolver uma aplicação que faz requisições HTTP.
- Dentro do arquivo `pubspec.yaml`:

```
1  dependencies:  
2    flutter:  
3      sdk: flutter  
4    http: ^0.13.0
```

MockLab

- Vamos simular um web service (neste caso, uma API REST) utilizando o Mock Lab.
 - <https://app.mocklab.io/>
 - Faça o cadastro.
 - Clique em *new* e dê o nome Pizza para esse *web service*.
 - Escolha GET como verbo e digite /pizzalist.

MockLab

GET a JSON resource

GET /json/1

200

```
{ "id": 1, "value": "things" }
```

Match a JSON POST on the request body

POST /json

201

```
{ "message": "Success" }
```

Only match if Accept header is for application/json

PUT /json/2

200

```
{ "message": "Success" }
```

Only match if query parameter is present and in the correct format

GET /search

200

Name

Pizzas

Add description

GET /pizzalist

Advanced ?

Scenario ?

Response

Direct Fault Proxy

Status

200

Enable templating ?


Header

Test Requester

MockLab

- Coloque no body o conteúdo do arquivo `pizzas.json`.
- Verifique qual é o domínio do *web service*.

WireMock - My Mock APIs

Name	Domains		
Example Mock API	3d40g.mocklab.io	Owner	

- Exemplo: <https://3d40g.mocklab.io/pizzalist>

Modelo

- Queremos transformar essa lista de elementos em formato JSON para uma lista de objetos.
- Crie uma pasta chamada `model` dentro do projeto.
 - E dentro dessa pasta, adicione um arquivo `pizza.json`.

Modelo

- Dentro desse arquivo criaremos um objeto para representar um json que é extraído do web service.

```
1  class Pizza {
2      int id;
3      String nomePizza;
4      String descricao;
5      double preco;
6      String urlImagem;
7
8      Pizza(this.id, this.nomePizza, this.descricao,
9            this.preco, this.urlImagem);
10 }
```

Modelo

- Agora adicione um construtor especial para ler o conteúdo do json:

```
1    Pizza.fromJson(Map<String, dynamic> json)
2      : id = json["id"],
3        nomePizza = json["nomePizza"],
4        descricao = json["descricao"],
5        preco = json["preco"],
6        urlImagem = json["urlImagem"];
```

Consumindo o Web Service

- Agora vamos adicionar um arquivo `httphelper.dart`.
 - Dentro dele adicione uma classe chamada `HttpHelper`;

```
1  import 'dart:io';
2  import 'package:http/http.dart' as http;
3  import 'dart:convert';
4  import 'model/pizza.dart';
5
6  class HttpHelper {
7      final String dominio = "3d40g.mocklab.io";
8      final String caminho = "pizzalist";
9  }
```

Consumindo o Web Service

- O objetivo dessa classe é conter todas as funcionalidades das requisições HTTP.
- Agora criamos dentro dela um método para acessar o *endpoint* criado.
 - O acesso deve ser realizado forma assíncrona.
 - Criamos uma função assíncrona chamada `getListaPizzas()`.

Consumindo o Web Service

```
1 Future<List<Pizza>> getListaPizzas() async {
2     // Lista de pizzas
3     final List<Pizza> pizzas = [];
4
5     Uri url = Uri.https(dominio, caminho);
6     http.Response result = await http.get(url);
7
8     // Se o acesso ao web service for bem-sucedido.
9     if (result.statusCode == HttpStatus.ok) {
10         final jsonResponse = json.decode(utf8.decode(result.bodyBytes));
11         // Adicionar cada pizza retirada do arquivo json
12         for (var response in jsonResponse) {
13             var pizza = Pizza.fromJson(response);
14             pizzas.add(pizza);
15         }
16     }
17     return pizzas;
18 }
```

Consumindo o Web Service

- Agora adicionamos uma classe para mostrar o resultado da consulta à API.
- Crie uma classe que herda de StatefulWidget chamada PainelPizzaria.

Consumindo o Web Service

```
1  import 'package:flutter/material.dart';
2  import 'httphelper.dart';
3  import 'model/pizza.dart';
4
5  class PainelPizzaria extends StatefulWidget {
6    const PainelPizzaria({super.key});
7
8    @override
9    State<PainelPizzaria> createState() => _PainelPizzariaState();
10 }
11
12 class _PainelPizzariaState extends State<PainelPizzaria> {
13   final HttpHelper helper = HttpHelper();
14
15   @override
16   Widget build(BuildContext context) {}
17 }
```


Consumindo o Web Service

- O método build conterá o seguinte código:

```
1  @override
2  Widget build(BuildContext context) {
3      return Scaffold(
4          appBar: AppBar(
5              title: Text("Pizzaria"),
6          ),
7          body: _mostrarListaPizzas(context),
8      );
9  }
```

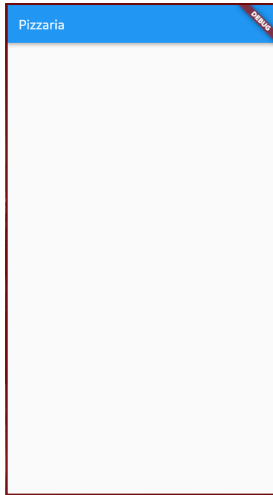
Consumindo o Web Service

```
1 ListView _mostrarListaPizzas(BuildContext context) {
2   List<Widget> listTiles = [];
3
4   helper.getListaPizzas().then((listaPizzas) {
5     for (Pizza pizza in listaPizzas) {
6       listTiles.add(
7         ListTile(
8           title: Text(pizza.nomePizza),
9           // toStringAsFixed(2) determina duas casas apos a virgula.
10          subtitle: Text(
11            "${pizza.descricao} - R\$ ${pizza.preco.toStringAsFixed(2)}"),
12          ),
13        );
14      }
15    });
16    return ListView(
17      children: listTiles,
18    );
19  }
```

Consumindo o Web Service

- **Problema:** A requisição para o *Web Service* é lenta, então a `ListView` será criada antes da requisição ser completada.
 - Podemos resolver isso utilizando o `FutureBuilder`.

Consumindo o Web Service



FutureBuilder

- O padrão de receber dados de forma assíncrona e atualizar uma interface com base no resultado é bem comum.
- Por isso, o Flutter fornece um widget para essa tarefa.
 - FutureBuilder.

FutureBuilder

- Esse widget tem duas propriedades obrigatórias:
 - future: contendo o objeto Future.
 - builder: utilizado para construir a interface com o resultado do Future.
- Vamos adicionar um FutureBuilder na nossa aplicação.

FutureBuilder

```
1  body: FutureBuilder(  
2    future: helper.getListasPizzas(),  
3    builder: ((context, snapshot) {  
4      // Verifica se os dados já foram recuperados  
5      // Caso sim, crie o widget  
6      if (snapshot.hasData) {  
7        return _mostrarListaPizzas(context, snapshot.data!);  
8        // Caso não, mostre o Widget de espera  
9      } else {  
10         return Center(  
11           child: CircularProgressIndicator(),  
12         );  
13       }  
14     })),  
15  ),
```

FutureBuilder

- Neste exemplo, o snapshot corresponde aos dados que o Future retorna.
- O CircularProgressIndicator indica que o resultado de algo está sendo aguardado.
- Dessa forma, se a operação assíncrona do Future estiver completa, o ListView é construído na tela.
 - Caso contrário, mostre o CircularProgressIndicator.

FutureBuilder

- Vamos alterar o método `_mostrarListaPizzas`:

FutureBuilder

```
1  ListView _mostrarListaPizzas(BuildContext context, List<Pizza> listaPizzas) {
2    List<Widget> listTiles = [];
3
4    for (Pizza pizza in listaPizzas) {
5      listTiles.add(
6        ListTile(
7          title: Text(pizza.nomePizza),
8          subtitle: Text(
9            "${pizza.descricao} - R\$ ${pizza.preco.toStringAsFixed(2)}"),
10        ),
11      );
12    }
13    return ListView(
14      children: listTiles,
15    );
16  }
```

Resultado

Pizzaria
Mussarela Queijo mussarela, tomate - R\$ 45.00
Calabresa Calabresa fatiada e cebola - R\$ 46.00
Alho Mussarela, alho e parmesão - R\$ 52.00
Atum Atum ralado e cebola - R\$ 54.00

Exercício 1

- Modifique o aplicativo criado:
 - Adicione uma tela que é mostrada quando clicamos em uma das pizzas no `ListView`.
 - Essa tela deve mostrar uma imagem da pizza clicada, sua descrição e seu preço.

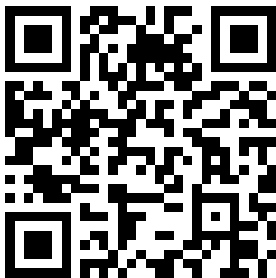
Referências

 Simone Alessandria and Brian Kayfitz.

Flutter Cookbook: Over 100 proven techniques and solutions for app development with Flutter 2.2 and Dart.

Packt Publishing Ltd, 2021.

Conteúdo



<https://gustavotcustodio.github.io/usabilidade.html>

Obrigado

gustavo.custodio@anhembi.br