

# Tab Persistente, Serviços e ListView.builder



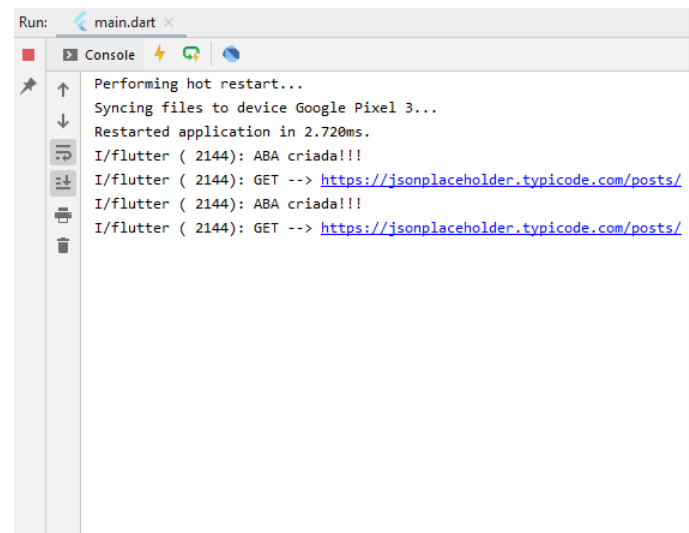
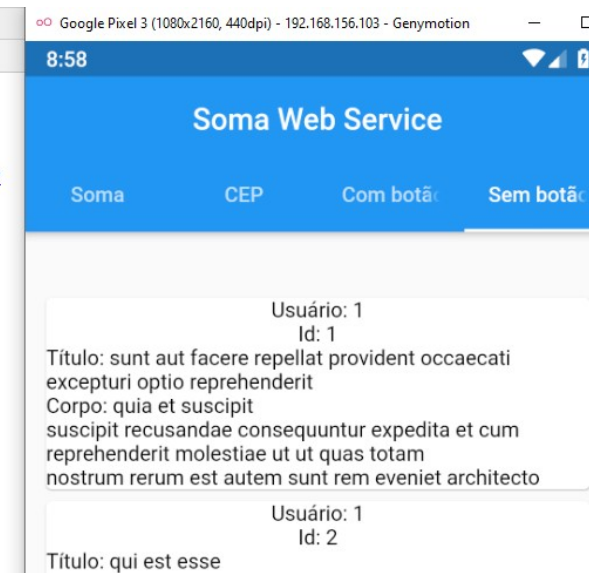
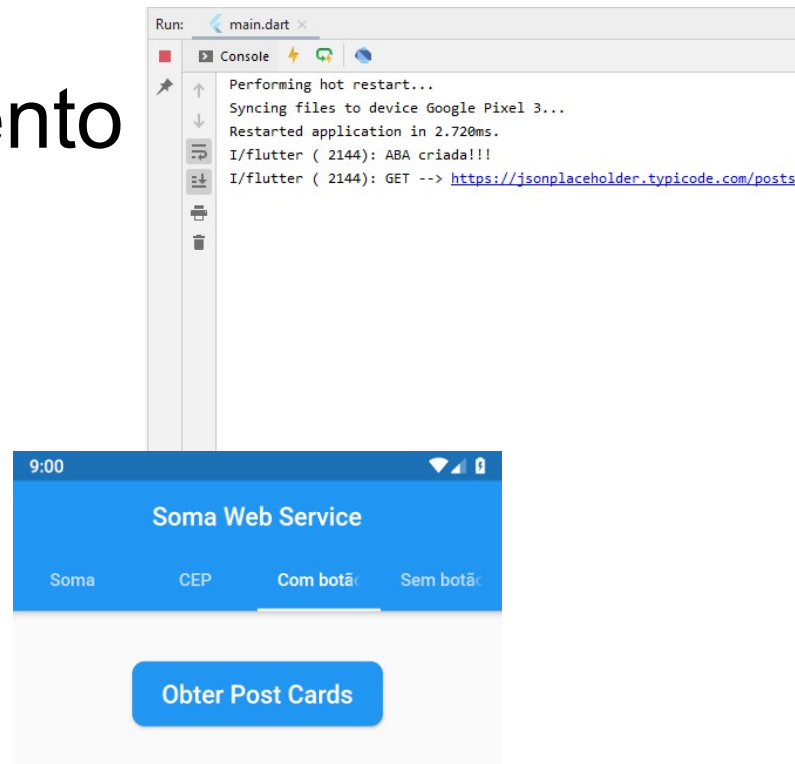
# A aplicação



# Recarregamento de Tabs

- Quando trabalhamos com Tabs (Abas), cada Tab é reconstruída toda vez que trocamos o foco:

```
@override
void initState() {
  // TODO: implement initState
  super.initState();
  print("ABA criada!!!");
  _postBloc.obterPosts();
}
```



# Recarregamento de Tabs

- Esse comportamento de reconstrução das Tabs pode ser desnecessariamente custoso. No exemplo do slide anterior, carregamos um serviço web toda vez que trocamos o foco da Tab, uma vez que, a mesma é construída novamente.

```
@override
void initState() {
  // TODO: implement initState
  super.initState();
  print("ABA criada!!!");
  _postBloc.obterPosts();
}
```

```
Performing hot restart...
Syncing files to device Google Pixel 3...
Restarted application in 2.720ms.
I/flutter ( 2144): ABA criada!!!
I/flutter ( 2144): GET --> https://jsonplaceholder.typicode.com/posts/
I/flutter ( 2144): ABA criada!!!
I/flutter ( 2144): GET --> https://jsonplaceholder.typicode.com/posts/
```



# AutomaticKeepAliveClientMixin

- Em Flutter existe uma forma de evitar que a Tab seja reconstruída com a mudança de foco:
  - 1) Na tela que possui as Tabs é necessário colocar um mixin **SingleTickerProviderStateMixin**.
  - 2) Na Tab que desejamos manter o contexto utilizaremos o mixin **AutomaticKeepAliveClientMixin**.
  - 3) Definiremos na Tab o método **get wantKeepAlive** fazendo com que retorne **true** (dessa forma será possível manter a tab em memória).
  - 4) Por fim, no método **build** iremos chamar o build do ancestral.

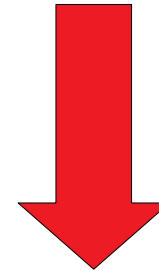


# Passo 1 do slide anterior

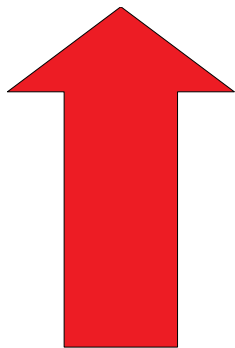
```
// Esse "with SingleTickerProviderStateMixin<HomePage>" vai ser usado para não recarregar
// os dados da aba "Sem botão"
class _HomePageState extends State<HomePage> with SingleTickerProviderStateMixin<HomePage>{
  @override
  Widget build(BuildContext context) {
    return DefaultTabController(
      length: 4,
      child: Scaffold(
        appBar: AppBar(
          centerTitle: true,
          title: Text("Soma Web Service"),
          bottom: TabBar(
            tabs: <Widget>[
              Tab(
                text: "Soma",
              ), // Tab
              Tab(text: "CEP"),
              Tab(text: "Com botão"),
              Tab(text: "Sem botão"),
            ], // <Widget>[]
          ), // TabBar
        ), // AppBar
        body: Builder(builder: (context) {
          return TabBarView(
            children: <Widget>[
              TabSoma(),
              TabCep(),
              TabPosts(),
              TabPosts2(),
            ], //await _buildSomaService(), // <Widget>[]
          ); // TabBarView
        }), // Builder
      ), // Scaffold
    ); // DefaultTabController
  }
}
```



# Passos 2 e 3

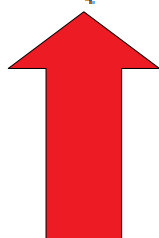


```
// Esse "with AutomaticKeepAliveClientMixin<TabPosts2>" vai ser usado para não recarregar  
// os dados dessa aba  
class _TabPostsState2 extends State<TabPosts2> with AutomaticKeepAliveClientMixin<TabPosts2>{  
  // Mantém a aba ativa para não recarregamento dos dados  
  @override  
  // TODO: implement wantKeepAlive  
  bool get wantKeepAlive => true;
```



# Passo 4

```
@override
Widget build(BuildContext context) {
  // Não pode esquecer dessa linha para não acontecer o recarregamento
  super.build(context);
}
```



```
@mustCallSuper
@override
Widget build(BuildContext context) {
  if (wantKeepAlive && _keepAliveHandle == null)
    _ensureKeepAlive();
  return null;
}
```





# Tab mantida em memória

- Usando os códigos apresentados nos slides anteriores, a Tab não será reconstruída na mudança de foco. Assim, independente de quantas trocas de Tab sejam feitas o carregamento do serviço ocorrerá apenas 1 vez.

```
Performing hot restart...  
Syncing files to device Google Pixel 3...  
Restarted application in 2.898ms.  
I/flutter ( 2144): Aba criada!!!  
I/flutter ( 2144): GET --> https://jsonplaceholder.typicode.com/posts/
```



# Serviço simples – soma 2 números

- A primeira Tab trata de acionar um serviço que soma 2 números. Esse serviço se encontram em:  
<http://col.ifes.edu.br/giovany/soma.php>

```
<?php
    $num1 = $_POST["numero1"];
    $num2 = $_POST["numero2"];
    $resultado = $num1 + $num2;
    echo $resultado;
?>
```



col.ifes.edu.br/giovany/ x +

← → ↻ ⚠ Não seguro | col.ifes.edu.br/giovany/

2 3 Envia Mensagem

view-source:col.ifes.edu.br/giovany/ x +

← → ↻ ⓘ Não seguro | view-source:col.ifes.edu.br/giovany/

```
1 <html>
2 <body><script type="text/javascript" src="http://gc.kis.v2.scr.ka
3 <form method="post" action="soma.php">
4   <input type="text" name="numero1"/>
5   <input type="text" name="numero2"/>
6   <input TYPE="submit" VALUE="Envia Mensagem">
7 </form>
8 </body>
9 </html>
10
```

col.ifes.edu.br/giovany/soma.php x +

← → ↻ ⓘ Não seguro | col.ifes.edu.br/giovany/soma.php

5

view-source:col.ifes.edu.br/giovany/soma.php x +

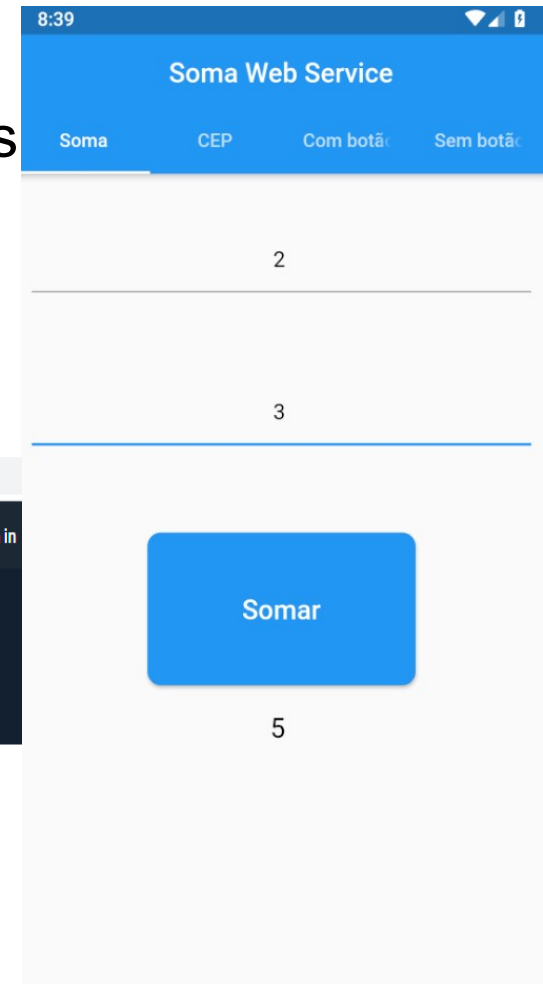
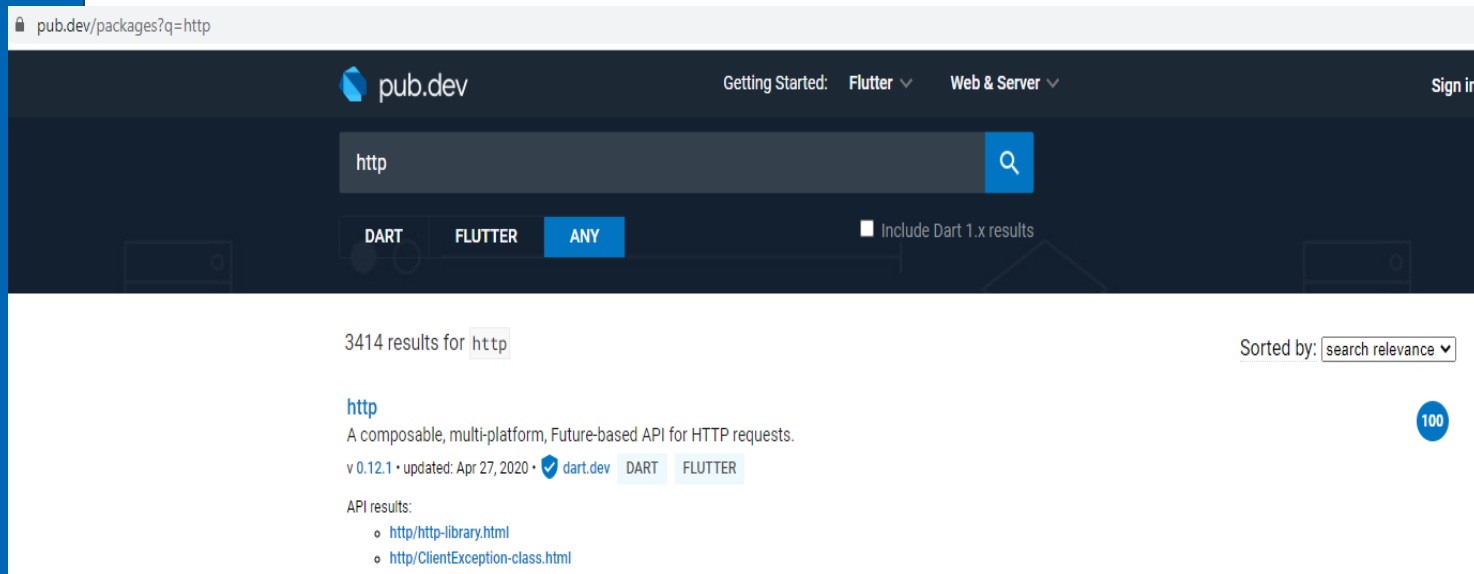
← → ↻ ⓘ Não seguro | view-source:col.ifes.edu.br/giovany/soma.php

```
1 5
2
```



# Serviço simples – soma 2 números

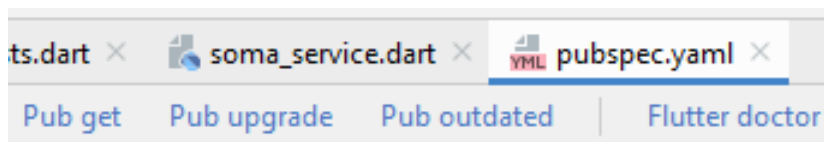
- Para fazer a soma de 2 números acionando um serviço web (nesse caso um .php) precisaremos de um plugin http.
- Para obter esse plugin precisaremos ir no pub.dev e buscar por http.



# Plugin http

- O plugin http também será utilizado para as demais tabs.
- O primeiro passo, conforme a documentação (<https://pub.dev/packages/http#-installing-tab->), é fazer referência ao plugin no arquivo **pubspec.yaml**.

- Depois fazemos um **Pub get**.



- Também pode ser usado **flutter pub get** no terminal para o mesmo efeito da linha anterior.
- Nas classes onde formos utilizar o plugin http precisaremos fazer a importação do pacote:

```
import 'package:http/http.dart'
```



# ApiResponse

- Essa é uma classe de apoio que utilizaremos para facilitar o trafego de informação e mensagens de erro.
- Quando um serviço vier com sucesso retornará um **ApiResponse** criado pelo construtor **ApiResponse.ok**.
- Quando houver um erro o objeto **ApiResponse** será criado por **ApiResponse.error**.
- O uso de **generics** aqui é interessante para armazenar uma informação que só saberei o tipo no serviço a ser implementado.
- O atributo **ok** guardará o sucesso ou falha no acionamento do serviço.

```
class ApiResponse<T>{
    bool ok;
    String msg;
    T resultado = null;

    ApiResponse.ok(this.resultado){
        | ok = true;
    }

    ApiResponse.error(this.msg){
        | ok = false;
    }
}
```



# Serviço simples – soma 2 números

```
import 'package:http/http.dart' as http;
import 'package:somawebsevice/services/api_response.dart';
class SomaService{
  static Future<ApiResponse<String>> somar(String valor1, String valor2) async {
    try{
      var _url = "http://col.ifes.edu.br/giovany/soma.php";
      //var _url = "http://essapaginanaoexiste/giovany/soma.php";

      Map _params = {
        'numero1' : valor1,
        'numero2' : valor2,
      };

      final _myUri = Uri.parse(_url);
      var _response = await http.post(_myUri, body: _params);

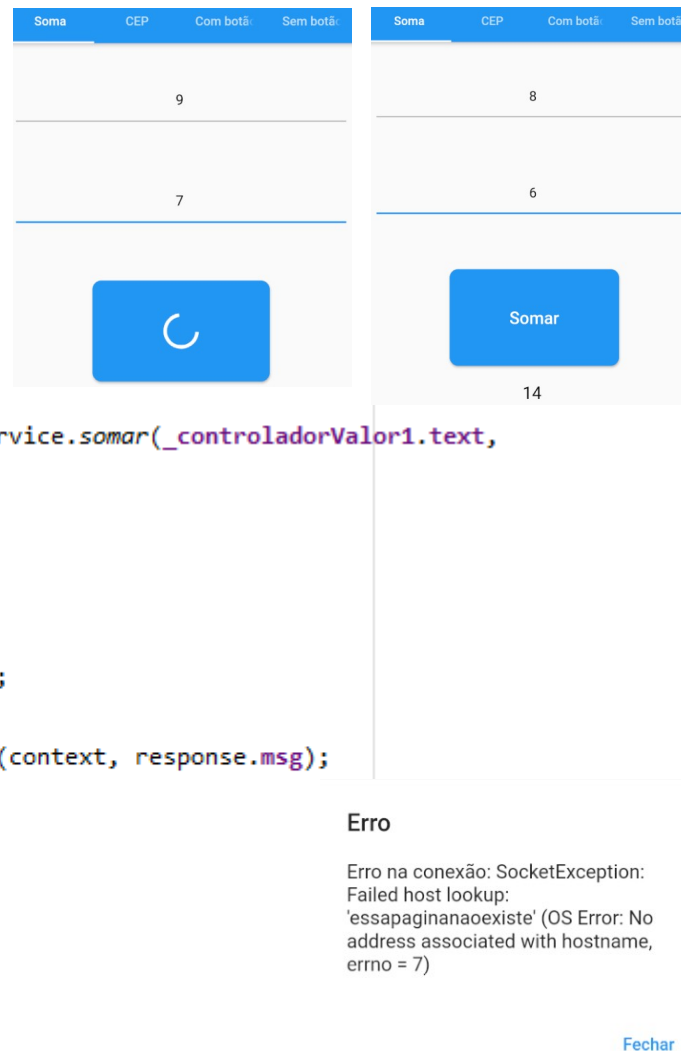
      // Solicitação feita com sucesso
      if(_response.statusCode == 200)
        return ApiResponse.ok(_response.body);
      else
        return ApiResponse.error("Status do erro emitido pelo servidor: ${_response.statusCode}");
    } on Exception catch (erro){
      return ApiResponse.error("Erro na conexão: ${erro.toString()}");
    }
  }
}
```

- Aqui podemos ver o uso do método post da biblioteca http.
- Da mesma forma que na página web que mostramos anteriormente, aqui temos a passagem dos parâmetros **numero1** e **numero2**.
- Em **\_response.body** virá o valor retornado pelo serviço (nesse caso a soma dos números).
- O uso do **await** e **async** é necessário porque http.post retorna um **Future**.
- Um Future pode estar em 2 estados: **pendente** (ainda não há resultado possível) ou **concluído** (com sucesso ou falha).



# Serviço simples – soma 2 números

```
Expanded(  
  flex: 1,  
  child: BotaoAzul(  
    texto: "Somar",  
    mostrar_progress: _mostrarProgress,  
    ao_clicar: () async {  
      setState(() {  
        _mostrarProgress = true;  
      });  
  
      ApiResponse response = await SomaService.somar(_controladorValor1.text,  
        _controladorValor2.text);  
  
      setState(() {  
        _mostrarProgress = false;  
        if(response.ok)  
          _resultado = response.resultado;  
        else{  
          CaixaAlerta.mostrarMensagemErro(context, response.msg);  
        }  
      });  
    }  
  ), // BotaoAzul  
, // Expanded
```



- Aqui temos o trecho de código que aciona o serviço web.
- No clique do botão o primeiro passo é chamar **setState()** para deixar o Progress do **BotaoAzul** visível. Esse Progress ficará visível até o término do serviço.
- Na sequência é chamado o serviço (SomaService.somar). Notar o uso do **await** no método somar e o **async** no método anônimo associado ao evento **ao\_clicar**.
- Utilizamos novamente **setState()** para retirar o Progress e colocar o resultado da soma.
- Se houver um erro nada é apresentado e é mostrada uma caixa de alerta com o erro. Como essa ao lado:



# BotaoAzul

```
@override
Widget build(BuildContext context) {
  return Container(
    width: 200,
    child: ElevatedButton(
      style: ElevatedButton.styleFrom(
        primary: Colors.blue,
        shape: RoundedRectangleBorder(
          borderRadius: new BorderRadius.circular(10.0)), // RoundedRectangleBorder
      ),
      child: mostrar_progress
        ? Center(
            child: CircularProgressIndicator(
              valueColor: AlwaysStoppedAnimation<Color>(Colors.white),
            ), // CircularProgressIndicator
          ) // Center
        : Text(texto,
            style: TextStyle(color: cor_fonte, fontSize: tamanho_fonte)), // Text
      onPressed: ao_clicar,
      focusNode: marcador_foco,
    ), // ElevatedButton
  ); // Container
}
```

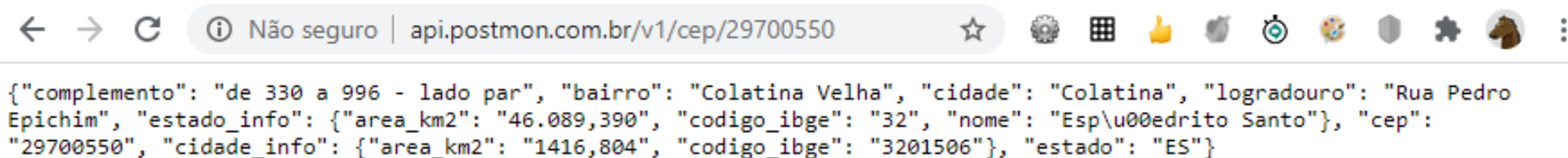
- Esse é o código do método build do **BotaoAzul**.
- Note que fixamos a largura do botão para 200 e a cor para azul.
- Na sequência vem a parte mais importante: De acordo com o atributo **mostrar\_progress** será mostrado um **CircularProgressIndicator** ou um **Text**. Esse efeito é muito interessante para botões que acionam serviços na web e foi essa a estratégia que utilizamos no slide anterior.
- Uma série de atributos próprios foram definidos para customizar o widget (texto, cor\_fonte, tamanho\_fonte, etc).
- A borda mais arredondada no botão também já será um padrão para quem utilizar esse widget.
- Esses widgets customizados facilitam a definição de bons padrões visuais e o reaproveitamento de código.





# Serviço de CEP

- Normalmente não acionamos um serviço apenas para receber um número ou uma String. O mais comum é a necessidade de trafegarmos várias informações ao mesmo tempo (um objeto, por exemplo).
- Para trafegar um objeto o padrão JSON é bem interessante, pois sua estrutura hierárquica facilita a obtenção da informação.
- Nesse contexto, na segunda Tab, temos acionado um serviço que, dado um CEP, retorna as informações referentes a esse CEP.



A screenshot of a web browser window. The address bar shows the URL `api.postmon.com.br/v1/cep/29700550` with a "Não seguro" (Not secure) warning. Below the address bar, a JSON response is displayed, containing location information for the CEP 29700550.

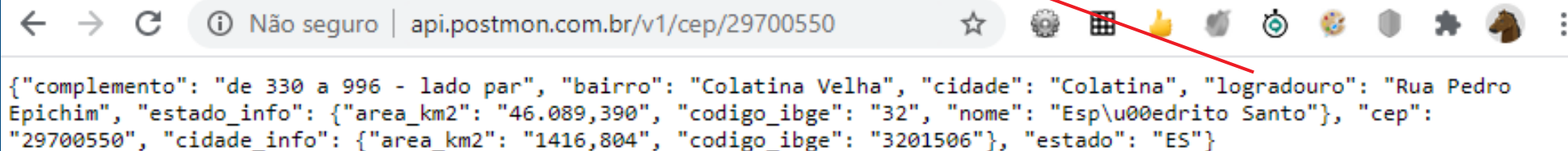
```
{"complemento": "de 330 a 996 - lado par", "bairro": "Colatina Velha", "cidade": "Colatina", "logradouro": "Rua Pedro Epichim", "estado_info": {"area_km2": "46.089,390", "codigo_ibge": "32", "nome": "Esp\u00edrito Santo"}, "cep": "29700550", "cidade_info": {"area_km2": "1416,804", "codigo_ibge": "3201506"}, "estado": "ES"}
```



# Classe de Endereço

```
class Endereco {  
  String complemento;  
  String bairro;  
  String cidade;  
  String logradouro;  
  String estado;  
  
  Endereco.fromJson(Map<String, dynamic> map){  
    complemento = map["complemento"];  
    bairro = map["bairro"];  
    cidade = map["cidade"];  
    logradouro = map["logradouro"];  
    estado = map["estado"];  
  }  
}
```

- Uma estratégia interessante é fazer um construtor nomeado que converta um Map no respectivo objeto.
- Para que isso seja útil, os campos da Map devem ser correspondentes aos dados que virão do JSON do serviço.



← → ↻ ⓘ Não seguro | api.postmon.com.br/v1/cep/29700550 ☆ ⚙ 📱 👍 🍷 🕒 🎨 🛡 ⚙ 👤 ⋮

```
{  
  "complemento": "de 330 a 996 - lado par",  
  "bairro": "Colatina Velha",  
  "cidade": "Colatina",  
  "logradouro": "Rua Pedro Epichim",  
  "estado_info": {  
    "area_km2": "46.089,390",  
    "codigo_ibge": "32",  
    "nome": "Esp\u00edrito Santo",  
    "cep": "29700550",  
    "cidade_info": {  
      "area_km2": "1416,804",  
      "codigo_ibge": "3201506",  
      "estado": "ES"  
    }  
  }  
}
```



# Serviço de CEP

```
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'package:somawebservice/domain/endereco.dart';
import 'package:somawebservice/services/api_response.dart';

class CepService{
  static Future<ApiResponse<Endereco>> obterCep(String cep) async {
    try{
      var _url = "http://api.postmon.com.br/v1/cep/" + cep;

      final _myUri = Uri.parse(_url);
      var _response = await http.get(_myUri);

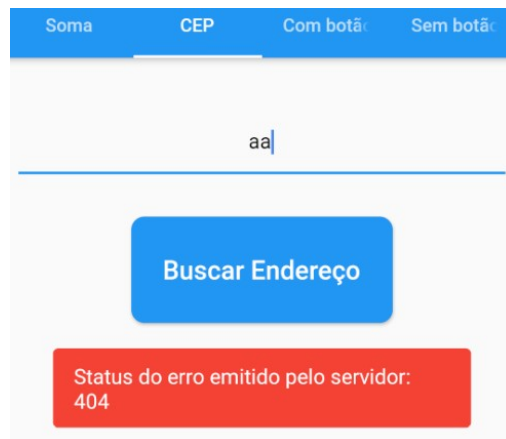
      // Solicitação feita com sucesso
      if(_response.statusCode == 200){
        Map mapResponse = json.decode(_response.body);
        Endereco endereco = Endereco.fromJson(mapResponse);
        return ApiResponse.ok(endereco);
      }
      else
        return ApiResponse.error("Status do erro emitido pelo servidor: ${_response.statusCode}");
    } on Exception catch (erro){
      return ApiResponse.error("Erro na conexão: ${erro.toString()}");
    }
  }
}
```

- Aqui usamos **get** ao invés de **post** de http.
- Nesse exemplo não foram necessários parâmetros para o método **get** pois o cep foi adicionado no final da url.
- O método **json.decode** transforma o resultado do serviço num Map.
- Por sua vez, **Endereco.fromJson** cria um endereco a partir do Map criado anteriormente.
- Se tudo der certo, é chamado **ApiResponse.ok** que guardará o objeto de endereco.
- Se houve um erro, **ApiResponse.error** irá armazenar a mensagem de erro gerada.



# Serviço de CEP

- A estratégia do uso do atributo **\_mostrarProgress** e o **setState()** continua a mesma do serviço de soma simples.
- Bem como o uso do **await** na chamada do serviço e do **async** na definição do método anônimo.
- Basicamente o que muda aqui é o preenchimento de mais campos (agora não é apenas um campo de resultado a ser preenchido, mas todos os relacionados com os widgets da tela).
- Aqui optou-se pelo uso do **Toast** ao invés de uma caixa de alerta mais tradicional.

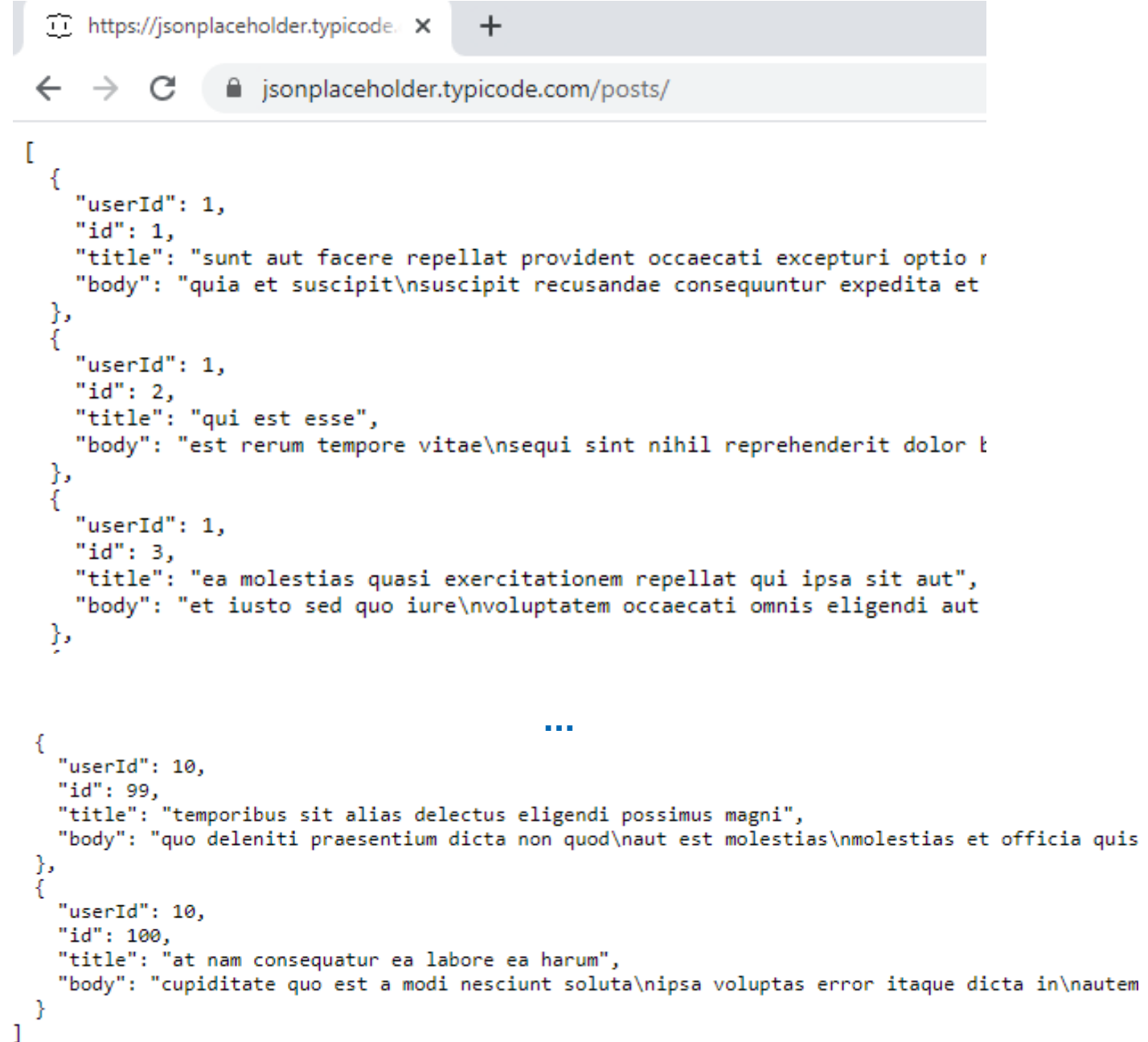


```
Expanded(  
  flex: 1,  
  child: BotaoAzul(  
    texto: "Buscar Endereço",  
    mostrar_progress: _mostrarProgress,  
    ao_clicar: () async {  
      setState(() {  
        _mostrarProgress = true;  
      });  
  
      ApiResponse response =  
        await CepService.obterCep(_controladorCep.text);  
  
      setState(() {  
        _mostrarProgress = false;  
        if (response.ok) {  
          Endereco endereco = response.resultado;  
          _complemento = endereco.complemento;  
          _bairro = endereco.bairro;  
          _cidade = endereco.cidade;  
          _logradouro = endereco.logradouro;  
          _estado = endereco.estado;  
        } else {  
          _complemento = "";  
          _bairro = "";  
          _cidade = "";  
          _logradouro = "";  
          _estado = "";  
          CaixaAlerta.mostrarToast(response.msg);  
        }  
      });  
    }), // BotaoAzul  
  ), // Expanded
```



# Serviço de posts

- Já sabemos como trafegar uma String, um JSON com um objeto e agora veremos como fazê-lo com vários objetos, ou seja, haverá uma lista de objetos no JSON.



The screenshot shows a web browser window with the address bar displaying `https://jsonplaceholder.typicode.com/posts/`. The page content is a JSON array of five post objects. Each object contains `userId`, `id`, `title`, and `body` fields. The first three objects are fully visible, followed by an ellipsis (`...`), and then the last two objects. The JSON is formatted with syntax highlighting.

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi optio r",
    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et",
  },
  {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor k",
  },
  {
    "userId": 1,
    "id": 3,
    "title": "ea molestias quasi exercitationem repellat qui ipsa sit aut",
    "body": "et iusto sed quo iure\nvoluptatem occaecati omnis eligendi aut",
  },
  ...,
  {
    "userId": 10,
    "id": 99,
    "title": "temporibus sit alias delectus eligendi possimus magni",
    "body": "quo deleniti praesentium dicta non quod\naut est molestias\nmolestias et officia quis",
  },
  {
    "userId": 10,
    "id": 100,
    "title": "at nam consequatur ea labore ea harum",
    "body": "cupiditate quo est a modi nesciunt soluta\nipsa voluptas error itaque dicta in\nautem",
  }
]
```



# Serviço de posts

```
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'package:somawebservice/domain/post.dart';
import 'package:somawebservice/services/api_response.dart';

class PostsService{
  static Future<ApiResponse<List<Post>>> obterPosts() async {
    try {
      var _url = "https://jsonplaceholder.typicode.com/posts/";
      // var _url = "https://jsonnnnnnplaceholder.typicode.com/posts/";

      print("GET --> $_url");

      final _myUri = Uri.parse(_url);
      var _response = await http.get(_myUri);

      // Solicitação feita com sucesso
      if(_response.statusCode == 200){
        List listMapResponse = json.decode(_response.body);
        final listPosts = listMapResponse.map<Post>((mapPost) => Post.fromJson(mapPost)).toList();
        return ApiResponse.ok(listPosts);
      }else
        return ApiResponse.error("Status do erro emitido pelo servidor: ${_response.statusCode}");
    } on Exception catch (erro){
      return ApiResponse.error("Erro na conexão: ${erro.toString()}");
    }
  }
}
```

- Novamente utilizamos o método **get** de http.
- Aqui vemos que o objeto retornado é um **Future<ApiResponse<List<Post>>**, ou seja, os dados retornados para serem utilizados fora do serviço serão do tipo **List<Post>**.
- A questão chave aqui é o método **map** aplicado a **listMapResponse**. Para cada um dos elementos dessa lista é aplicado o método **Post.fromJson** que basicamente gera um **Post** a partir de um **Map** (não confunda o método map com a estrutura de dados Map). Ao final do processo, temos uma lista de Posts pronta.
- O uso do **await** e **async** continua se fazendo necessário.



# Método map

- Poderíamos trocar a chamada ao método map do slide anterior pelo seguinte código:

```
List<Post> listPosts = List<Post>();  
for(int i=0; i<listMapResponse.Length; i++){  
    Map mapPost = listMapResponse[i];  
    Post post = Post.fromJson(mapPost);  
    listPosts.add(post);  
}
```



# BotaoAzulServicoWeb

- Para evitar de fazer o controle do progress manualmente pode-se criar um widget a partir do widget **BotaoAzul** (já apresentado anteriormente).
- Vamos deixar para estudar o código desse widget na aula de Streams, uma vez que, utilizamos Streams para conseguir o comportamento implementado.
- Por ora vamos entender os 3 eventos que ele trata:
  - **pre\_servico**: código utilizado para validar condicionantes para a chamada ao serviço (campos de um formulário com valores adequados, por exemplo). Se o método retornar **false** significa que houve falha e o serviço não será chamado. Se o método retornar **true** significa que tudo foi validado e é possível chamar o serviço.
  - **acionar\_servico**: código que deve acionar o serviço web e retornar um ApiResponse (esse terá internamente os dados lidos ou a mensagem de erro gerada).
  - **pos\_servico**: código executado após o serviço terminar. Recebe como parâmetro o ApiResponse retornado da função acionar\_servico.
- Essa abordagem se aproxima (lembra) do uso de AsyncTasks do Android Nativo tradicional.





# Usando o BotaoAzulServicoWeb

```
Expanded(  
  flex: 1,  
  child: BotaoAzulServicoWeb(  
    texto: "Obter Post Cards",  
    // Aqui poderíamos ter a validação de um formulário, por exemplo  
    pre_servico: () {  
      return true;  
      // return _formkey.currentState.validade();  
    },  
    acionar_servico: () {  
      return PostsService.obterPosts();  
    },  
    // Chamado para atualizar a tab depois do serviço  
    // A animação do botão é tratada pelo próprio BotaoAzulServicoWeb  
    // Mas para que a listagem seja atualizada e o método build seja  
    // chamado é necessário usar o setState()  
    pos_servico: (ApiResponse response) {  
      setState(() {  
        if (response.ok)  
          _lista_posts = response.resultado;  
        else {  
          CaixaAlerta.mostrarMensagemErro(context, response.msg);  
        }  
      });  
    },  
  ), // BotaoAzulServicoWeb  
), // Expanded
```

- No exemplo **pre\_servico** retorna **true** pois não há processo de validação de dados.
- **acionar\_servico** chama então o serviço de obtenção de Posts e retorna o **ApiResponse** que contem a lista de Posts.
- **pos\_servico** recebe o ApiResponse gerado pelo serviço. Ele atualiza a lista de Posts se os dados vieram com sucesso ou emite uma mensagem de alerta, caso contrário.
- O comportamento de exibição e atualização de progress fica automatizado e encapsulado dentro de **BotaoAzulServicoWeb**.
- **\_lista\_posts** irá ser utilizado para exibir os dados numa ListView.



# ListView

- Já vimos o uso de **ListView** de forma básica, ou seja, colocando widgets na tela um abaixo do outro manualmente.
- Para esse exemplo isso não é suficiente, precisamos usar um **ListView.builder**. Com esse builder será possível criar um ListView a partir de uma lista de objetos (no nosso caso uma lista de Posts).
- Para facilitar ainda mais, encapsulei um ListView num widget próprio que chamei de **ListViewPostCard** e criei um **PostCard** para definir o “desenho” de uma linha (de forma a ficar mais fácil separar o que é do ListView e o que é do “desenho” da linha – no Android Nativo Tradicional esse desenho da linha seria feito por um Adapter).



# ListViewPostCard

```
import 'package:flutter/material.dart';
import 'package:somawebsevice/domain/post.dart';
import 'package:somawebsevice/tabs/localwidget/post_card.dart';

class ListViewPostCard extends StatelessWidget {
  List<Post> posts;

  ListViewPostCard(this.posts);

  @override
  Widget build(BuildContext context) {
    return Container(
      margin: EdgeInsets.only(top: 40, left: 10, right: 10),
      child: ListView.builder(
        // Se a lista de posts for nula
        itemCount: posts != null ? posts.length : 0,
        itemBuilder: (context, index) {
          return PostCard(
            userId: posts[index].userId.toString(),
            id: posts[index].id.toString(),
            title: posts[index].title,
            body: posts[index].body,
          ); // PostCard
        }, // ListView.builder
      ); // Container
    }
  }
}
```

- Os **posts** são definidos no construtor desse widget, ou seja, quando o widget é criado a lista de **posts** já está pronta (se não estiver será criada uma ListView de 0 elementos)
- **itemCount**: define quantos elementos serão desenhados. A validação sobre os posts serem **null** é fundamental na primeira chamada externa (quando a lista ainda não foi obtida). Nesse caso será criada uma ListView de 0 elementos.
- **itemBuilder**: recebe o contexto e o índice do elemento que está sendo construído (é como se tivéssemos um **for**, com **itemCount** itens, criando cada um dos **PostCards**). Retorna o widget que “desenha” uma linha (com o efeito similar a um for, acaba por desenhar todas as linhas).
- **PostCard** é o widget que “desenha” uma linha de **Post**.



# Dúvidas?

