Streams

Definição

- Stream é uma sequência de eventos assíncronos. É definido um fluxo e quando adicionamos dados a esse fluxo ele será capaz de atualizar uma interface com o usuário.
- O uso de Streams nos direciona a uma programação mais reativa, ou seja, avisamos da mudança de dados e o Flutter promove a atualização dos widgets.
- Como primeiro exemplo veremos a classe **BotaoAzulServicoWeb** utilizado na aula anterior.

BotaoAzulServicoWeb

- Na definição do widget não foi implementado nada acerca do uso de Streams.
- O mais importante desse slide é a importação de **'dart:async'** para que possamos usar Streams.
- O resto do código basicamente define os atributos que serão customizados pelo programador.

```
import 'dart:async';
import 'package:flutter/material.dart';
import 'package:somawebservice/services/api_response.dart';
import 'package:somawebservice/widget/botao_azul.dart';

class BotaoAzulServicoWeb extends StatefulWidget {
  String texto;
  double tamanho_fonte;
  Color cor_fonte;
  Function pre_servico;
  Function acionar_servico;
  Function(ApiResponse response) pos_servico;
  FocusNode marcador_foco;

  BotaoAzulServicoWeb(
      {this.texto,
      this.tamanho_fonte,
      this.cor_fonte,
      this.pre_servico = null,
      this.acionar_servico,
      this.pos_servico = null,
      this.marcador_foco});

  @override
  _BotaoAzulServicoWebState createState() => _BotaoAzulServicoWebState();
}
```


BotaoAzulServicoWeb

```dart
class _BotaoAzulServicoWebState extends State<BotaoAzulServicoWeb> {
  final _streamController = StreamController<bool>();

  @override
  void initState() {
    // TODO: implement initState
    super.initState();
  }

  @override
  void dispose() {
    // TODO: implement dispose
    super.dispose();
    _streamController.close();
  }
```

- Aqui já temos algo muito importante; a definição do **StreamController**. Esse objeto ficará responsável por controlar o fluxo da Stream. É ele que será avisado da mudança de um dado.
- Notar que o **StreamController** recebe o tipo de dado que ele irá controlar, nesse caso um **bool**.
- Uma outra questão importante é a chamada do método **close()** quando o fluxo da Stream não for mais necessário.

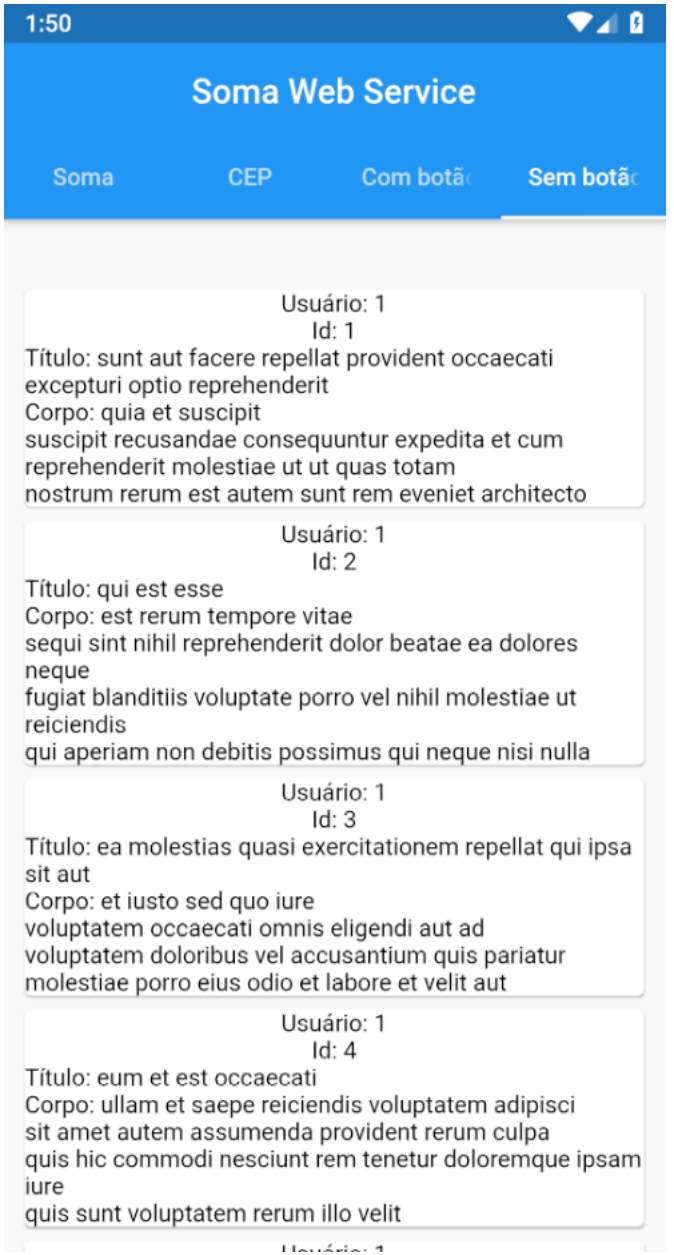
build

- O widget **StreamBuilder** recebe o tipo de dado que ele irá tratar (deve ser o mesmo que o **StreamController**).
- Em **stream** setamos a *stream* do StreamController definido no widget (*_streamController*).
- **initialData** recebe o valor inicial que será passado em **snapshot.data**.
- Quando **_streamController.add(true)** é acionado o **StreamBuilder** reconstrói o BotaoAzul e **snapshot.data** terá o valor **true** (fazendo o progress aparecer).
- Quando **_streamController.add(false)** é acionado o **StreamBuilder** reconstrói o BotaoAzul e **snapshot.data** terá o valor **false** (fazendo o progress desaparecer).

```dart
@override
Widget build(BuildContext context) {
  return StreamBuilder<bool>(
    stream: _streamController.stream,
    initialData: false,
    builder: (context, snapshot){
      return BotaoAzul(
        texto: widget.texto,
        mostrar_progress: snapshot.data,
        ao_clicar: () async {
          if (widget.pre_servico != null){
            if (!widget.pre_servico()){
              return;
            }
          }
          // Vai colocar o progress visível
          _streamController.add(true);

          // Aciona o serviço
          ApiResponse response = await widget.acionar_servico();

          // Vai voltar com o texto para o botão
          _streamController.add(false);

          // Vai atualizar a tela com os resultados obtidos no serviço
          widget.pos_servico(response);
        },
      ); // BotaoAzul
    },
  ); // StreamBuilder
  }
}
```

Segundo exemplo

- Também utilizamos Streams no carregamento dos dados da Tab_Posts2 (Sem botão).
- O ListView utilizado (ListViewPostCard) é o mesmo da última aula mas para obter a listagem para sua exibição utilizamos Streams e o padrão BLoC (Business Logic of Component).
- Com o padrão BLoC é possível separar lógica de negócio da interface. Basicamente o BloC encapsulará o StreamController, controlará eventos de entrada e colocará a saída desses eventos na stream.
- O link: https://www.flutterparainiciantes.com.br/gerenciamento-de-estado/bloc possui maiores orientações acerca do padrão BLoC.



SimpleBloc

- A ideia da classe SimpleBloc é simplificar a criação de novas classes que usam o padrão BLoC.
- Na prática SimpleBloc encapsula o StreamController buscando simplificar seu acesso.

```dart
import 'dart:async';

class SimpleBloc<T> {
  final _controller = StreamController<T>();

  Stream<T> get stream => _controller.stream;

  void add(T object){
    _controller.add(object);
  }

  void dispose() {
    _controller.close();
  }
}
```


PostBloc

```dart
import 'package:somawebservice/domain/post.dart';
import 'package:somawebservice/services/api_response.dart';
import 'package:somawebservice/services/posts_service.dart';
import 'package:somawebservice/util/simple_bloc.dart';

class PostBloc extends SimpleBloc<ApiResponse>{
  Future<ApiResponse<List<Post>>> obterPosts() async{
    // Obtendo os posts
    ApiResponse<List<Post>> response = await PostsService.obterPosts();
    this.add(response);
    return response;
  }
}
```

- **PostBloc** herda de **SimpleBloc** informando que o tipo do dado que será controlado pelo fluxo de dados é um **ApiResponse**.
- No método **obterPosts** temos a chamada ao serviço de obtenção de Posts e atualização do fluxo através da chamada ao método **add**.
- Na prática essa chamada ao método **add** irá acionar o StreamBuilder que estiver ligado ao StreamController do nosso SimpleBloc. E isso irá atualizar a interface gráfica (o widget que se encontra dentro de StreamController, no nosso caso o ListViewPostCard).

TabPosts2

```dart
import 'package:flutter/material.dart';
import 'package:somawebservice/domain/post.dart';
import 'package:somawebservice/domain/post_bloc.dart';
import 'package:somawebservice/services/api_response.dart';
import 'package:somawebservice/tabs/localwidget/listview_post_card.dart';

class TabPosts2 extends StatefulWidget {
  @override
  _TabPostsState2 createState() => _TabPostsState2();
}

// 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;
```

- Essa parte inicial do código não trata do uso de Streams.
- Tem-se aqui a questão dessa ser uma Tab Persistente, ou seja, que é necessário fazer o uso do mixin **AutomaticKeepAliveClientMixim** e a definição de **wantKeepAlive** para **true**.
- Notar que não há a necessidade de importar a biblioteca `import 'dart:async';` pois ela já está definida no SimpleBloc, do qual PostBloc, que será mostrado nos próximos slides, herda.

TabPosts2

```dart
// Usando o padrão Bloc da Google para trabalhar com Streams
final _postBloc = PostBloc();

@override
void initState() {
  // TODO: implement initState
  super.initState();
  _postBloc.obterPosts();
}

@override
void dispose() {
  // TODO: implement dispose
  super.dispose();
  _postBloc.dispose();
}
```

- Aqui temos a definição do **PostBloc** (que irá encapsular o **StreamController**).
- No **initState()** é feita a chamada para a obtenção dos posts. Como essa chamada é assíncrona, o método **build** (que será apresentado nos próximos slides) será acionado antes mesmo de seu término. Entretanto, quando o serviço terminar o StreamController (dentro do _postBloc) irá acionar a reconstrução da tab.
- Já no **dispose()** temos a liberação do fluxo de dados do **StreamController** que se encontra dentro do _postBloc.

build

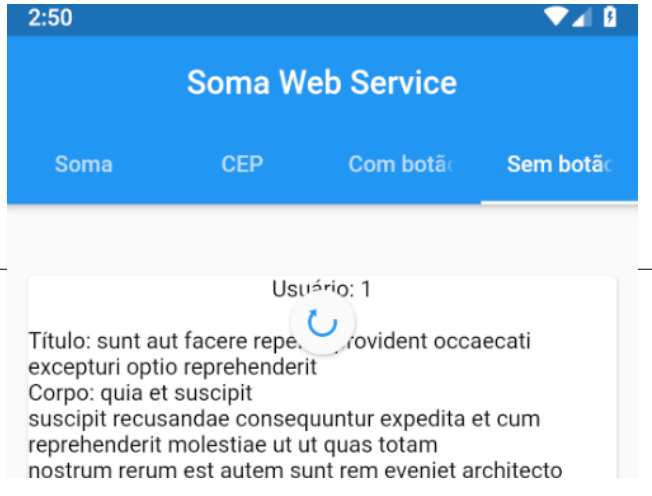
```dart
@override
Widget build(BuildContext context) {
  // Não pode esquecer dessa linha para não acontecer o recarregamento
  super.build(context);

  return StreamBuilder<ApiResponse>(
    stream: _postBloc.stream,
    builder: (context, snapshot) {
      // Vai colocar um progress enquanto os posts não forem carregados
      if (!snapshot.hasData) {
        return Center(
          child: CircularProgressIndicator(),
        ); // Center
      }

      ApiResponse<List<Post>> apiResponse = snapshot.data;

      // Se houve algum erro
      if (!apiResponse.ok)
        return Center(
          child: Text(
            apiResponse.msg,
            style: TextStyle(
              color: Colors.red,
              fontSize: 22,
            ), // TextStyle
          ), // Text
        ); // Center
```

- Aqui temos a primeira parte do método **build**.
- No parâmetro **stream**, da definição do **StreamBuilder**, é setado o **stream** do **_postBloc** (que é o stream do StreamController dentro desse Bloc).
- No método definido para o parâmetro **builder** temos o uso de **snapshot.hasData**. Isso é utilizado para sabermos se já temos dados para serem apresentados, se não tivermos é exibido um Progress.
- Se já temos dados, não cairemos no **if(! snapshot.hasData)** e obteremos esses dados, que podem ser uma falha (o serviço recebeu uma mensagem de erro, por exemplo) ou a lista de Posts. Ao lado temos o código que trata o caso de uma falha.

RefreshIndicator

```
        return RefreshIndicator(
          // Ele obriga um método que retorna um Future para garantir
          // que está sendo chamado um método assíncrono para atualizar
          // o Stream, mas ele não usa esse retorno
          onRefresh: _postBloc.obterPosts,
          // Aqui temos a lista de posts sendo informada para o ListView
          child: ListViewPostCard(apiResponse.resultado)
        ); // RefreshIndicator
      },
    ); // StreamBuilder
  }
}
```

- Aqui poderíamos ter apenas o **return ListViewPostCard(apiResponse.resultado)**. Mas optamos por usar um RefreshIndicator para atualizar a listagem caso o usuário “puxe” a listagem para baixo.
- **onRefresh** recebe um método que retorna um **Future** e quando esse Future estiver completo, o **RefreshIndicator** retira o progress e atualiza seu **child**.
- A listagem então vem pronta em **apiResponse.resultado** (lembrar que **apiReponse** veio de **snapshot.data**).



Dúvidas?