getx_pattern

Legenda:

Marcador amarelo = Errado

Marcador verde = Correto

uma proposta pra você que utiliza o GetX

Estamos trabalhando em um novo site para você ! (você!)

Projeto desenvolvido para padronizar os seus projetos com GetX.

Objetivos

O objetivo principal, é tornar sua programação com Flutter + GetX mais agradável e intuitiva do que já é ! (é!)

Quando há um padrão de desenvolvimento a ser seguido, tudo fica mais fácil para quem usa. Mas por quê?

- Porque assim, podemos nos comunicar de forma "universal" com pessoas que também seguem esses mesmos padrões.

- Há infinitas maneiras de resolver o mesmo problema, então os patterns tem um papel muito importante nisso, os mesmos problemas, ainda poderão ser resolvidos de várias maneiras, mas com um determinado fluxo a ser seguido.

- Com isso é muito mais fácil sanar suas dúvidas, quando vocês conversam no mesmo padrão.

- Seu projeto não ficará dependente de quem o construiu, pois(remova a vírgula) outros programadores poderão ler e alterar o código sem dificuldades.

- Você terá um melhor aproveitamento na reutilização do seu código, podendo assim reutilizar diversos widgets de outros projetos por exemplo, que ele se encaixará perfeitamente no seu projeto.

- Que você possa aplicar o uso do GetX em projetos profissionais, tendo uma documentação em mãos, com isso, mais segurança para quem programa, seja você uma empresa ou um estudante em busca de conhecimento.

E além do getx_pattern, também está disponibilizamos uma extensão que irá acelerar muito mais ainda esse processo, o GetX Snippets tem snippets que irão acelerar seu desenvolvimento, podendo criar desde variáveis a classes inteiras e já está disponível para o vscode, Experimente ! (Experimente!)

Agora que você entendeu nossos objetivos, vamos aprender um pouco sobre a estrutura do getx_pattern ! (pattern!)

Se você quiser acelerar seu processo de desenvolvimento, recomendamos que conheça nossa extensão para vscode GetX Snippets

Estrutura

Você pode ficar a (à) vontade para usar a estrutura que desejar em módulos ou em packages, ao passar do tempo estaremos criando exenplos (exemplos) e criando os mesmos das duas maneiras. O exemplo inicial lhe oferecerá as mesmas abas para que você possa acompanhar a construção do projeto de ambas as formas com o mesmo código, então vamos começar ! (começar!)

The Structure for GetX developers xD

Agora que você conhece a proposta e deu uma boa analisada na apresentação da estrutura, vamos parar um pouco para falar rapidamente sobre ela, antes de introduzir você na seção do GetX, para que você possa seguir nossos exemplos sem dificuldade ou dúvidas sobre nossos padrões de nomenclatura ou fluxo, pois bem, vamos falar rapidamente de um por um

agora, não se preocupe se você está se aventurando e não conhece alguns dos conceitos que serão abordados aqui, isso é apenas uma apresentação formal com a estrutura, iremos falar mais detalhadamente sobre cada uma delas, com exemplos, em suas respectivas seções.

Model: É o diretório que irá agrupar todas as nossas classes modelos para nossos objetos.

Sinta-se a (à) vontade para escolher entre Model ou Class, mas tenha em mente que daqui em diante, usarei o termo Model, caso não seja novidade, garanto que logo irá se acostumar com Model.

Providers: É o diretório responsável por agrupar nossos provedores de dados, pode ser tanto um banco de dados ou uma api.

Repository: É um ponto único de acesso aos dados, irá abstrair nossas entidades.

Data: É apenas um diretório responsável por guardar TUDO relacionado aos seus dados, ou seja, seu repository, suas classes e providers.

Controller: Nossos controllers, nada mais são, que os responsáveis pelas nossas regras de negócio, alterações de estado, também é onde criaremos nossos observáveis com seus respectivos estados iniciais e os eventos que serão responsáveis por alterar esses estados.

UI: É tudo que o usuário vê, seus widgets, animações, textos, temas.

Routes: É o diretório responsável por conter o/os nossos arquivos que são resposáveis(responsáveis) por gerênciar(gerenciar) nossas rotas.

Bindings: A classe Binding é uma classe que desacopla a injeção de dependência, enquanto a "ligação" roteia para o gerente de estado e o gerente de dependência. Isso permite conhecer qual tela está sendo exibida quando um controlador específico é usado e saber onde e como descartá-lo. Além disso, a classe Binding permitirá que você tenha o controle de configuração do SmartManager. Você pode configurar as dependências a serem organizadas ao remover uma rota da pilha, ou quando o widget que a utilizou for disposto, ou nenhuma delas.

CASO VOCÊ OPTE PELA OPÇÃO DE ESTRUTURAÇÃO EM MÓDULOS, NOSSA PAGE, CONTROLLER DEVEM SER CONTIDOS DENTRO DE SEUS RESPECTIVOS MÓDULOS.

CASO VOCÊ ESTEJA USANDO BINDINGS PARA INJETAR DEPÊNDENCIAS, OS MESMOS PERTECEM APENAS A SEU MÓDULO.

A ESTRUTURA EM MÓDULOS É RECOMENDADA PARA PROJETOS MUITO GRANDES, DEIXANDO TUDO RELACIONADO AQUELA VIEW, DISPONÍVEL DE FORMA FÁCIL.

Agora vamos conhecer um pouco mais o Get, o gerenciador de estados mais rápido e leve ! (leve!)

GetX, seu gerenciador de estado reativo.

Primeiramente, é aconselhável que você tenha um conhecimento prévio de flutter e também uma base de conhecimento em gerenciamento de estados, caso não tenha, recomendo uma breve leitura na documentação oficial do flutter sobre este assunto, para acessar click aqui.

O foco do GetX é ter maior desempenho com o mínimo consumo de recursos, veja os benchmarks.

Produtividade, usando uma sintaxe fácil e agradável.

Organização, permitindo a dissociação total da sua camada de apresentação da sua regra de negócio.

O Getx tem o poder de reconstruir apenas o widget quando há uma alteração de estado em uma variável .obs em seu controlador. Isso, porque tudo nele é fluxo, isso nos permite ouvir o evento de cada 'variável'.

Vamos para um primeiro exemplo da aplicação do GetX com o getx_pattern , (pattern,) do bom e velho contador.

Vamos utilizar, neste exemplo, apenas a classe main, para você se familiarizar com os conceitos do GetX.

Apague todo conteúdo da sua main.dart e mantenha, ou copie e cole, apenas este trecho de código:

```
import 'package:flutter/material.dart';

import 'package:get/get.dart';


void main() {
  runApp(MyApp());
}

....
```

view rawmain.dart hosted with ❤ by GitHub

Após isso, vamos criar nossa classe controller, que nos permite criar nossa regra de negócio e controlar nossos estados.

Abaixo do código acima cole o seguinte código:

```
class MyController extends GetxController {
  final _num = 0.obs;
  get num => this._num.value;
  set num(value) => this._num.value = value;

  increment() {
    this.num++;
  }
  decrement() {
    this.num--;
  }
}
```

view rawmy_controller_counter.dart hosted with ❤ by GitHub

O GetxController é o responsável por nos fornecer os métodos necessários para controllar(controlar) nossos observáveis (.obs), permitindo atualizar nossos widgets GetX, respondendo as mudanças de estados da nossa aplicação. No exemplo acima declaramos uma variável e dois métodos, vamos falar sobre eles.

Com GetX, podemos extender a classe GetxController, e com ele podemos criar varáveis observáveis adicionando um simples .obs e seu estado inicial.

final [var] = [initial-value].obs;

Nosso observável só pode ser final, e seu tipo será atribuido(atribuído) a partir do initial-value inserido no momento da declaração, por ser final, o mesmo só pode ser inserido uma vez.

Mas e agora? Se final só pode ser modificada uma vez, como vou alterar o valor dessa variável?

A resposta é tão simples quanto curta, .value. Ao criar sua variável como um observável, ela irá criar, por trás do capô, um RxObject,exemplo, um tipo int Rxint, um tipo List< User>() RxList< User>(), e seu valor e tipo original é guardado no seu atributo value desse RxObject.

No caso das nossas Listas, não é necessário utilizar o .value, então diferente de um RxString onde você acessaria o valor contido com text.value, com as listas você pode trabalhar normalmente, tendo em vista que tanto sua lista, como os objetos dentro dela, são todos observáveis, portanto podemos acessar seus atributos dessa maneira myList.length ou myList[index].name.

Vamos a classe Stateless, responsável pela vizualização(visualização) do nosso contador. Você não leu errado, com GetX você nunca mais irá precisará um stateful quando quiser alterar seus estados, esqueça o setState, nosso widget GetX é reconstruido(reconstruído) sempre que um objeto, nele contido, tenha o estado alterado pelo controller setado no widget.

vamos ao código.

```
class MyApp extends GetWidget {

 final MyController controller = Get.put(MyController());


 @override

 Widget build(BuildContext context) {

  return GetMaterialApp(

   home: Scaffold(

    body: Column(

     mainAxisAlignment: MainAxisAlignment.center,

     crossAxisAlignment: CrossAxisAlignment.center,

     children: <Widget>[

      GetX<MyController>(

       builder: (_) => Text(_.num.toString()),

      ),

      Container(

       height: 300,

       width: 345,

       child: Row(

        mainAxisAlignment: MainAxisAlignment.center,

        children: <Widget>[
```

```
            FloatingActionButton(
              onPressed: () => controller.increment(),
              child: Icon(Icons.add, color: Colors.white),
            ),
            SizedBox(
              width: 30,
            ),
            FloatingActionButton(
              onPressed: () => controller.decrement(),
              child: Icon(Icons.remove, color: Colors.white),
            ),
          ],
        ),
      ),
    ],
  ),
 ),
 );
 }
}
```

view rawMyApp.dart hosted with ❤ by GitHub

Bem, isso é um widget básico, estamos apenas mostrando um text e dois botões responsáveis por acionar os nossos eventos decrement() e increment(), lembra? estão no nosso controller .

Repare um pouco melhor, a primeira coisa que deve ter notado é que estamos utilizando 3 GetX widgets, mas porquê?

Bem, isso deve ser abordado primeiro para que você pode entender como aplicar melhor este widget, e não usá-lo de forma,não(forma, não) errada, mas desnecessária, pois bem, primeiro temos um GetX widget do tipo MyController, onde nossa lógica se encontra.

Mas o que ele tem diferente dos outros ? (outros?) Bem, a propriedade init, ela é responsável por criar nossa instancia do objeto controller e dispinibiliza-lo(disponibilizá-lo) na função builder desde widget, você já deve ter percebido, mas é isso mesmo, nosso controllador(controlador) corresponde ao nosso _ (underline), com isso você pode acessar seus observáveis apenas usando a instância do seu controlador, dessa forma _.num.

Agora que temos nosso primeiro widget GetX com controlador inicializado, não precisamos inicializa-lo novamente, apenas setar o tipo no dele,caso(dele, caso) estejam na mesma classe/page, dessa forma Getx<[type]>, e ele se encarregará do resto para você, fácil não? E você pode ficar tranquilo, sem boilerplate, o GetX é sensacional, e ainda sabe a hora certa de remover esse controlador da memória. Você também pode controllar(controlar) isso se quiser, com GetX é assim, ele deixa você livre para focar na qualidade do seu código, sem se preocupar com questões repetitivas e fáceis de lidar como essa, ele cuida de tudo pra você.

Mas e o .value?

Bem, é ai que entra uma das propostas do getx_pattern, nós achamos melhor manter tudo que for controlado na interface dentro do controlador, isso inclui os estados de suas variáveis e os eventos que levarão a isso, assim como nosso exemplo temos a variável num, que é declarado no controller e a mesma, é usada no Getx widget, e nossas funções/eventos, responsáveis por alterar o estado dessa nossa variável. Portanto, carregar _.num.value vai contra as boas práticas, preferimos que o controllador(controlador) seja responsável até por isso, criando um get e um set para que possamos abstrair o .value do código da sua vizualização(visualização), deixando seu código mais legível e sem boilerplate.

Além do widget observável GetX< Controller >(), o GetX Também possui o Obx(() => Widget()), sendo este, responsável por reconstruir nossos widgets considerados pesados.

E o que o GetX entende por widget leve e pesado ? (pesado?)

Vamos falar um pouco sobre isso.

O widget Text(_.count) do nosso exemplo é um widget 'leve', pois estamos o alterando DIRETAMENTE Isso quer dizer:

GetX< MyController>(builder: (_) => Text(_.var))

Uma observação importante neste ponto é que Containers não são considerados filhos diretos.

Dessa forma o exemplo a seguir também é valido(válido) para o uso do GetX

GetX< MyController>(builder: (_) => Container(child: Text(_.var)))

Widgets são considerados pesados quando possui uma grande hierarquia de filhos, como uma lista ou um card complexo. Nesses casos, ao tentar usar o GetX widget para reconstruir esse list ou card, você receberá um erro de uso indevido.

Para esses casos, você pode fazer o uso do Obx(() => WidgetPesado()).

Onde WidgetPesado se subentende uma Lista, que pode possuir Widgets Getx para a alteração de cada elemento e o Obx para atualização da lista completa.

Seu uso se mostra muito eficiente ao ser combinado com GetView, estendendo sua Page, juntamente com bindings para ela, você verá isso no exemplo mais sólido desta documentação seções adiante.

Mas basicamente você ira(irá) ter uma insttancia(instância) do seu controller ao invés do _ .

Isso foi uma breve introdução sobre o uso doGetX(do GetX), vamos aplicar isso também em outros instantes, se você não entendeu tudo até aqui, não se preocupe.

Se quer saber mais sobre o GetX, visite seu package no pubdev, ou seu repositório no GitHub aqui.

Data

Aqui não teremos muitos o que discutir, é apenas um repositório onde você irá abstrair/empacotar TUDO relacionado aos seus dados, seus modelos, seus repositories e provedores de dados. Caso você opte por usar a versão em módulos , Data terá o mesmo papel, deixando seus dados disponivéis(disponíveis) para todos os seus módulos, deixando só o que é vital para seu módulo nele!!

Isso foi pensado para que você possa manter sua estrutura de diretórios o menor possível ao trabalhar com flutter, e ao mesmo tempo, ser algo intuitivo e didático para acelerar sua curva de aprendizado.

Vamos começar o uso do pattern aqui, ao avançar das seções você vamos utilizar o JSONPlaceHolder para exemplificar o consumo de uma API com essa estrutura.

Vamos lá!!!

Crie os seguintes repositórios, dentro de lib.

```
|-- lib
 |-- app
  |-- data
 main.dart
```

Provider

Em algumas outras estruturas, o termo 'provider', pode ser abordado de várias formas, mas aqui, ele existe única e exclusivamente, para fazer requisições http ou persistência num banco de dados. Caso você use os dois, crie os respectivos diretórios e/ou arquivos dentro deste.

Caso haja muitas requisições, em um único arquivo, você pode optar por separar por entidades, ou manter no mesmo arquivo, isso é uma escolha pessoal e vai de cada programador.

Vamos começar a construir nosso exemplo por aqui.

Comece criando um diretório chamado provider, dentro de Data que irá conter todos os arquivos que fornecem dados para nossa aplicação, juntamente com a classe responsável por capturar/enviar nossos dados neste exemplo. Se você quiser criar essa classe de forma mais rápida, use nossa extensão para vscode. GetX Snippets.

Comece criando um arquivo chamado api.dart, ou o nome que desejar.

```
|-- lib
 |-- app
  |-- data
   |-- provider
    api.dart
 main.dart
```

É dentro deste aquivo(arquivo) que iremos escrever nossa classe, copie o exemplo a seguir, ou se estiver usando nossa extensão basta começar escrevendo getprovider e pronto, ela lhe trará uma classe como a do exemplo!!

```
import 'dart:convert';

import 'package:getx_pattern/app/data/model/model.dart';

import 'package:http/http.dart' as http;

import 'package:meta/meta.dart';


//nossa url base

const baseUrl = 'https://jsonplaceholder.typicode.com/posts/';


//nossa classe responsável por encapsular os métodos http

class MyApiClient {

//seu client http, pode ser http, http.Client, dio, apenas traga seus métodos para cá e funcionarão normalmente :D

final http.Client httpClient;

MyApiClient({@required this.httpClient});
```

```
  //um exemplo rápido, aqui estamos recuperando todos os posts disponibilizados pela api(100)
  getAll() async {
    try {
      var response = await httpClient.get(baseUrl);
      if(response.statusCode == 200){
        Iterable jsonResponse = json.decode(response.body);
          List<MyModel> listMyModel = jsonResponse.map((model) => MyModel.fromJson(model)).toList();
        return listMyModel;
      }else print ('erro');
    } catch(_){ }
  }
}
```

view rawapi.dart hosted with ❤ by GitHub

Não se preocupe com os erros até o momento, ao decorrer das seções posteriores, vamos criar todas as depêndencias(dependências).

Como podem ver, temos uma classe MyApiClient com um parâmetro requerido, nosso objeto http.Client, que será usado neste exemplo para fazer todas as requisições à api.

Por fim, temos nosso método getAll que tem como papel, recuperar todos os posts da api.

Iremos adicionar mais métodos futuramente.

Model

Na nossa classe modelo teremos nossos atributos e dois métodos, toJson e fromJson responsáveis por converter nosso objeto em json ou um json em nosso objeto.

Geralmente quando trabalhamos com API's, usamos esses métodos para criar objetos a partir da resposta da nossa api, ou criar um json para enviar para nossa api.

Comece criando o diretório model e um arquivo model.dart.

```
|-- lib
  |-- app
    |-- data
      |-- provider
        api.dart
      |-- model
        model.dart
  main.dart
```

Agora vamos ao código, copie e cole o seguinte código, ou se estiver usando nossa extensão comece escrevendo getmodel

```
class MyModel {


  int id;

  String title;

  String body;


  MyModel({ id, title, body});


  MyModel.fromJson(Map<String, dynamic> json){

     this.id = json['id'];

     this.title = json['title'];

     this.body = json['body'];

  }


  Map<String, dynamic> toJson(){

    final Map<String, dynamic> data = new Map<String, dynamic>();

    data['name'] = this.title;

    data['body'] = this.body;

    return data;

  }

}
```

view rawmodel.dart hosted with ❤ by GitHub

Repository

É o responsável por seperar(separar) suas entidades , (entidades,) basicamente, entidades são todas as "tabelas" da sua base de dados que irão interagir com seu ou seus providers.

O repository tem o objetivo de abstrair e separar, sua fonte de dados do seus controladores, tendo assim, um ponto único de falha, ou seja, se um dia você vier a trocar a api ou base de dados do seu projeto, basta alterar seus arquivos providers, não sendo necessário nenhuma alteração no repository, pois ele só é responsável por chamar a função do provider, não havendo lógica nenhuma ali.

ex Imagine que você possua um aplicativo de venda de produtos, que possua apenas os clientes e os produtos de um estabelecimento.

Podemos identificar facilmente nossas entidades nos fazendo uma pergunta.

Irei receber e/ou enviar dados dessa entidade? Se a resposta for sim, então ela precisa de um repository.

No nosso exemplo teríamos três repositories, UserRepository, ProdutoRepository, EstabelecimentoRepository.

Algumas vezes, podemos tirar essas entidades nos baseando em nossas classes, mas muitas vezes existem classes auxiliares que não estão necessariamente na sua base de dados ou api, portanto preferimos nos basear no que realmente interage com sua base de dados.

Dessa forma, tiramos muita responsabilidade do nosso controller, ele não precisa saber de onde vem os dados, apenas consumi-los, outra vantagem é que o repository faz a nossa ligação Controller <-> Dados.

Tendo assim, uma melhor organização, deixando o código compreensivo e é extremamente fácil a manutenção, de forma intuitiva.

No nosso exemplo, nosso repository será responsável pelo meio de campo entre a api e o controller, sendo o responsável pela chamada da api no controller.

Vamos então criar o nosso diretório repository e um arquivo posts_repository.dart

```
|-- lib
 |-- app
  |-- data
   |-- provider
    api.dart
   |-- model
    model.dart
   |-- repository
    posts_repository.dart
 main.dart
```

Vejamos o exemplo, copie e cole o código a seguir, ou se possuir nossa extensão basta apenas começar a escrever getrepository

```
import 'package:getx_pattern/app/data/provider/api.dart';

import 'package:meta/meta.dart';

//Repositório responsável pela entidade post !!

//lembrando que cada entidade que persiste de alguma forma com o banco de dados, seja ela uma api ou sqlite

//deve possuir um repositório excluisivo

class MyRepository {

 final MyApiClient apiClient;


 MyRepository({@required this.apiClient}) : assert(apiClient != null);


 getAll() {

  return apiClient.getAll();

 }

}
```

view rawposts_repository.dart hosted with ❤ by GitHub

Com isso, finalizamos a parte do nosso projeto que cuida dos dados da nossa aplicação, vamos consumi-los agora ! (agora!)

Controller

Os controladores são a parte vital da sua aplicação, neles você irá criar suas variáveis .obs que guardará valores que podem ser alterados durante a aplicação.

Seu controlador é responsável também por consumir seus dados>, através dos seus repositórios, que por sua vez, só realizam as chamadas dos dados dos seus providers

Regra: Todo controller deve possuir um, e apenas um, repository, sendo esse, um atributo requerido para inicializar seu controller no seu GetX widget.

Caso você venha a precisar de dados de dois repositories diferentes em uma mesma page, você deve usar dois GetX widgets. Recomendamos que haja, no mínimo, um controller para cada page.

Só existe uma exceção para que você possa usar o mesmo controllador(controlador) para varias(várias) pages, e é bem simples:

IMPORTANTE

Você poderá usar um controlador em várias páginas, única e exclusivamente, se os dados de todas as pages, consumirem um único repositório.

O objetivo disso é fazer com que você use o GetX e aproveite todo seu poder, assim sempre que precisar manipular duas entidades, você precisará de dois controlladores(controladores) diferentes e uma view.

Por que? Imagine que você possua um controller com dois repositórios, e esse controller está sendo usado com um GetX widget em uma page, utilizando dados recuperados pelo controller dos dois repositórios.

Sempre que uma entidade for modificada, o controllador(controlador) irá atualizar seus widgets responsáveis pelas duas variáveis, sendo que uma delas não necessitava alteração. Portanto separar um repositório por controlador, pode ser uma boa prática ao se trabalhar com o GetX widget, tendo um controllador(controlador) responsável para cada widget, que de alguma forma, mostra essas informações apartir(a partir) deles, renderizando apenas o widget que teve sua variável .obs alterada.

Então vamos criar nosso diretório controller e um arquivo controller.dart

```
|-- lib
 |-- app
  |-- data
   |-- provider
    api.dart
   |-- model
    model.dart
   |-- repository
    posts_repository.dart
  |-- controller
   |-- home_controller
    home_controller.dart
main.dart
```

Vamos ao código do nosso exemplo, copie e cole o código abaixo, ou se estiver usando nossa extensão basta começar escrevendo getcontroller e pronto ! (pronto!)

```
import 'package:get/get.dart';

import 'package:getx_pattern/app/data/model/model.dart';

import 'package:getx_pattern/app/data/repository/posts_repository.dart';

import 'package:meta/meta.dart';


class MyController extends GetxController {

  //repository required

  final MyRepository repository;

  MyController({@required this.repository}) : assert(repository != null);


  //use o snippet getfinal para criar está variável

  final _postsList = List<MyModel>().obs;

  get postList => this._postsList.value;

  set postList(value) => this._postsList.value = value;

  ///função para recuperar todos os posts

  getAll(){

    repository.getAll().then( (data){ this.postList = data; } );

  }

}
```

view rawcontroller.dart hosted with ❤ by GitHub

Vamos falar um pouco sobre o que temos no controller até o momento.

Primeiramente podemos ver nosso repositório como um @required, isso pois vamos utilizar nosso repository para recuperar nossos dados, reparem que não preciso saber da onde vem esses dados, eu apenas preciso me preocupar em consumir o meu repository.

Logo em seguida, vemos nossa variável postsList que é um .obs, responsável pela nossa lista de posts, a mesma que iremos exibir na nossa page na próxima seção.

Reparem que ela foi criada como privado, por isso criamos nossos get and set. Como foi abordado antes, gostamos de deixar toda responsabilidade de alteração de estados dentro do controller, portanto ao atribuir algum valor a essa variável, isto será feito dentro do controller, sem necessidade de usar o .value, o mesmo já está presente no seu controller.

Por último temos a funçao(função) getAll do nosso controllador(controlador), reparem também, que usamos o mesmo nome para a função independente da classe que ela se encontrava, isso facilita na localização independente do arquivo, ao

ver essa função no controller você poderia ir direto para seu provider, sem precisar procurar no seu repository, caso você usasse um nome diferente.

Você deve ter percebido até aqui que não usamos o Future, ao invés disso, usamos o .then

Essa função aguarda o retorno de uma função e atribui a data, com isso você não precisa mais usar FutureBuilders nunca mais !! (mais!!)

Basta chamar a função com funcao().then atribuir o retorno ao seu .obs e consumir com umGetX widget.

Com isso podemos aguardar na nossa page até que a lista seja preenchida com um loading por exemplo, você verá isso melhor na próxima seção.

Agora vamos criar um controller simples, apenas recebendo o post selecionado na lista home.

Pra isso vamos usar um recurso muito simples e interessante do Getx, arguments você pode passar ele assim que você chama uma rota, e pode receber o valor numa váriavel(variável) quando o controllador(controlador) for inicializado.

```
import 'package:get/get.dart';

import 'package:getx_pattern/app/data/model/model.dart';

import 'package:getx_pattern/app/data/repository/posts_repository.dart';

import 'package:meta/meta.dart';


class DetailsController extends GetxController {


  final MyRepository repository;

  DetailsController({@required this.repository}) : assert(repository != null);


 final _post = MyModel().obs;

 get post => this._post.value;

 set post(value) => this._post.value = value;


  //pratique

editar(post){

  print('editar');

}

  //pratique

delete(id){

  print('deletar');

}

}
```

view rawdetails_controller.dart hosted with ❤ by GitHub

Bindings

Ideal para seu gerenciamento de dependências, os bindings podem inicializar seus controladores e repositórios, apis e o que você precisar, sem a necessidade de chama-los diretamente na View ! (View!)

No nosso exemplo iremos inicializar dois controladores, juntamente com seus respectivos repositórios, criando um binding inicial no seu arquivo main para nossa home e outro através da rota em que iremos acessar nossa DetailsPage.

Vamos criar os bindings e depois falar um pouco sobre eles.

```
|-- lib
 |-- app
  |-- bindings
  my_binding.dart
  |-- data
   |-- provider
    api.dart
   |-- model
    model.dart
   |-- repository
    posts_repository.dart
  |-- controller
   |-- home_controller
    home_controller.dart
 main.dart
```

Com nosso diretório bindings criado, ou em seus respectivos módulos, crie dois arquivos, home_bindings.dart e details_binding.dart.

Repare que você pode fazer isso para todas as suas views, desde que siga a orientação de usar sempre um controllador (controlador) para cada page.

Vamos aos arquivos.

```
import 'package:getx_pattern/app/controller/home/home_controller.dart';

import 'package:getx_pattern/app/data/provider/api.dart';

import 'package:getx_pattern/app/data/repository/posts_repository.dart';

import 'package:http/http.dart' as http;

import 'package:get/get.dart';


class HomeBinding implements Bindings {
  @override
  void dependencies() {
    Get.lazyPut<HomeController>(() {
      return HomeController(
```

```
        repository:
            MyRepository(apiClient: MyApiClient(httpClient: http.Client())));
    });
  }
}
```

view rawhome_binding.dart hosted with ❤ by GitHub

```
import 'package:get/get.dart';
import 'package:getx_pattern/app/controller/details/details_controller.dart';
import 'package:getx_pattern/app/data/provider/api.dart';
import 'package:getx_pattern/app/data/repository/posts_repository.dart';
import 'package:http/http.dart' as http;

class DetailsBinding implements Bindings{
  @override
  void dependencies() {
    Get.lazyPut<DetailsController>(() {
      return DetailsController(
        repository:
            MyRepository(apiClient: MyApiClient(httpClient: http.Client())));
    });
  }

}
```

view rawdatails_binding.dart hosted with ❤ by GitHub

Pronto ! (Pronto!) Agora podemos ir para a próxima seção fazer o uso dos nossos bindings ! (bindings!)

UI

Aqui vamos agrupar tudo relacionado a nossa interface do usuário.

Também temos uma proposta de como você deve agrupar esses elementos internamente.

Uma das maiores vantagens do flutter é o modo como fica fácil "componentizar" sua UI, portanto neste diretório você deve criar um diretório para cada page, neste diretório, além da sua page, podemos adicionar um outro diretório, de widgets para cada page, dessa forma sua page principal pode ser lida facilmente, pois é montada com vários widgets.

E se você já adivinhou, podemos fazer isso também no nivel(nível) da UI, criando um diretório de widgets globais, que virão a ser utilizados em mais de uma page na sua aplicação.

Caso você não precise usar widgets em determinada page, basta ignorar essa orientação, mas sempre que for criar uma page, preferimos que você crie um diretório para ela, pois futuramente pode vir a ter widgets externos à ela, dessa maneira você não irá perder tempo resolvendo erros de importação.

Vamos entender melhor adicionando seu diretório a estrutura.

```
|-- lib
 |-- app
  |-- data
   |-- provider
    api.dart
   |-- model
    model.dart
   |-- repository
    posts_repository.dart
  |-- controller
   |-- home_controller
    home_controller.dart
  |-- ui
   |-- android
    |-- widgets
     my_reusable_global_widget.dart
    |-- home
     |-- widgets
      my_reusable_home_widget.dart
     home_page.dart
   |-- ios
   |-- theme
    my_theme.dart
 main.dart
```

Agora vamos ao nosso exemplo, copie e cole o código abaixo, ou se estiver usando nossa extensão basta começar escrevendo getpage e pronto ! (pronto!)

```
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:getx_pattern/app/controller/home/controller.dart';
import 'package:getx_pattern/app/data/provider/api.dart';
import 'package:getx_pattern/app/data/repository/posts_repository.dart';
import 'package:http/http.dart' as http;

class HomePage extends GetView<HomeController> {
//repository and controller  injection bindings
```

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('HomePage')),
      body: Container(
        child: GetX<MyController>(
            initState: (state) { Get.find<MyController>().getAll() ;},
            builder: (_) {
              return
              _.postList.length < 1
              ? Center(child: CircularProgressIndicator(),)
              :
              ListView.builder(
                itemBuilder: (context, index) {
                  return ListTile(
                    title: Text(_.postList[index].title),
                    subtitle: Text(_.postList[index].body),
                  );
                },
                itemCount: _.postList.length,
              );
            }),
      ),
    );
  }
}
```

view rawhome_page.dart hosted with ❤ by GitHub

Vamos falar um pouco sobre o que está acontecendo em nossa page.

Repare que não estamos declarando nosso controller em lugar algum, estamos estendendo uma GetView< HomeController >, em nosso arquivo main. nossa classe HomeBinding() está sendo injeta na home, sendo ela a página inicial recebendo nosso HomeBinding.

Você pode ter acesso ao seu controllador(controlador) usando controller.att, normalmente.

Isso deixa nosso código muito mais limpo e legível.

Em nossa função build temos um Scaffold simples, com um container e um GetX widget.

Nele, iniciamos nosso controlador, e no initState do GetX buscamos o controlador e chamamos a função getAll, responsável por atribuir a nossa lista de posts _postsList no nosso controlador

Então verificamos o tamanho da lista e assim que começarmos a reber(receber) a lista de posts criamos nossa ListViewBuilder contendo um ListTile exibindo o title e o body do nosso objeto em um determinado index, lembrando que o length, é o tamanho da nossa lista, mais um motivo para o GetX widget estar envolvendo nossa ListViewBuilder.

Lembrando que, como nossa função getAll é async, isso quer dizer que não vai interromper o funcionamento da aplicação, por isso temos um CircularProgressIndicator para exibir enquanto não recebemos nossa postsList no nosso controller.

Estamos quase prontos, só vamos falar sobre algumas coisa que podem agradar você, antes de irmos para a seção de rotas e modificar nossa main.dart para você executar o app :)

Vamos falar sobre nosso theme, nele você pode definir temas para as cores do seu app, ou modificar o tema de algun(algum) botões de maniera(maneira) fácil, portanto pensamos em demonstrar o uso para que você possa tirar maior proveito do flutter !

Veja um exemplo, usaremos appTheme em nossa main.dart mais adiante.

```
import 'package:flutter/material.dart';


final ThemeData appThemeData = ThemeData(

  primaryColor: Colors.blueAccent,

  accentColor: Colors.blue,




);
```

view rawapp_theme.dart hosted with ❤ by GitHub

Agora vamos as rotas, e logo depois usaremos nosso TEMA

Rotas

A ideia é que você possa gerenciar suas rotas de forma fácil, limpa e segura.

Mudamos nossa maneira de manipular suas rotas, a proposta é que você separe suas rotas de sua navegação de pages.

Porque?

Para que suas rotas sejam sempre CONSTANTES, dessa maneira você pode navegar dessa maneira com GetX Get.toNamed(Routes.ROTA_CONSTANTE).

Tanto Routes quanto AppPAges, serão classes abstratas, para que você não cometa nenhum engano instânciado-as(instanciando-as) por engano, viu nós cuidamos de você.

Então agora chega de procurar as rotas nas pages, e chega de alterar uma por uma quando necessário alteração, basta alterar em um único lugar e seu app continuará funcionando normalmente !! (normalmente!!)

Crie os diretórios dessa forma Crie o diretório routes e o arquivo my_routes.dart que conterá nossas rotas.

```
        |-- lib
```

```
|-- app
 |-- data
  |-- provider
   api.dart
  |-- model
   model.dart
  |-- repository
   posts_repository.dart
 |-- controller
  |-- home_controller
   home_controller.dart
 |-- ui
  |-- android
   |-- widgets
    my_reusable_global_widget.dart
   |-- home
    |-- widgets
     my_reusable_home_widget.dart
    home_page.dart
  |-- ios
  |-- theme
   app_theme.dart
 |-- routes
  app_routes.dart
  app_pages.dart
main.dart
```

Vamos a um exemplo, onde definimos apenas nossa rota principal e rota de detalhes.

Copie e cole o código abaixo, ou se estiver usando nossa extensão comece escrevendo getroutes, Isso(isso) lhe trará uma classe completa com algumas constantes como exemplo de rotas.

```
part of './app_pages.dart';


abstract class Routes{


  static const INITIAL = '/';

  static const DETAILS = '/details';
```

```
}
```

view rawapp_routes.dart hosted with ❤ by GitHub

Procure saber tudo que você pode fazer com as rotas, como middleware, parâmetros e mais na documentação oficial do GetX.

Agora nosso arquivo de navegação, Apppages, essa classe conterá um array do tipo GetPage, e dentro dela, iremos declarar todas as nossas navegações, usando nossa classe Routes para recuperar os nomes das nossas rotas.

```
import 'package:get/get.dart';

import 'package:getx_pattern/app/ui/android/details/details_page.dart';

import 'package:getx_pattern/app/ui/android/home/home_page.dart';

part './app_routes.dart';


class AppPages {


  static final routes = [

    GetPage(name: Routes.INITIAL, page:()=> HomePage(),),

    GetPage(name: Routes.DETAILS, page:()=> DetailsPage(), binding: DetailsBinding()), //dependencias de details via rota


  ];

}
```

view rawapp_pages.dart hosted with ❤ by GitHub

Internacionalização

Se você deseja fazer aplicativos que irão romper nossas fronteiras, isso não é um problema quando se trabalha com GetX!

Você pode criar chaves para palavras, frases e até mesmo parâmetros dinâmicos são permitidos ! (permitidos!)

E para usar é fávil(fácil) e rápido, basta adicionar .tr em seus textos ! (textos!)

```
Text('bem vindo'.tr),
```

Vamos ver como ela se encaixa em nossas estruturas.

Crie o diretório translations e o arquivo app_translations.dart que conterá todas as traduções de maneira organizada, permitindo uma ou mais linguas(línguas). Crie Também os respectivos diretórios para cada lingua(língua), iremos falar sobre eles depois.

```
            |-- lib
              |-- app
                |-- data
                  |-- provider
                    api.dart
                  |-- model
```

```
        model.dart
      |-- repository
        posts_repository.dart
    |-- controller
      |-- home_controller
        home_controller.dart
    |-- ui
      |-- android
        |-- widgets
          my_reusable_global_widget.dart
        |-- home
          |-- widgets
            my_reusable_home_widget.dart
          home_page.dart
      |-- ios
      |-- theme
        app_theme.dart
    |-- routes
      app_routes.dart
    |-- translations
      app_translations.dart
      |-- en_US
        en_us_translation.dart
      |-- pt_BR
        pt_br_translation.dart
      |-- es_MX
        es_mx_translation.dart
  main.dart
```

Vamos ver o que temos no nosso arquivo app_translations.

Copie e cole o código abaixo e depois iremos falar sobre.

```
import 'package:getx_pattern/app/translations/en_US/en_us_translations.dart';
import 'package:getx_pattern/app/translations/es_MX/es_mx_translations.dart';
import 'package:getx_pattern/app/translations/pt_BR/pt_br_translations.dart';
```

```
abstract class AppTranslation {

  static Map<String, Map<String, String>>

  translations =

{

    'pt_BR' : ptBR,

    'en_US' : enUs,

    'es_mx' : esMx

};


}
```

view rawapp_translations.dart hosted with ❤ by GitHub

Nossa classe é responsavel(responsável) por manter todas as linguas(línguas) que serão disponibilizadas.

Repare no nosso tipo, estamos declarando esta primeira parte do Map< String, o 'próximo map' será recebido de acordo com nossas tradução(nossas traduções) específicas que passarmos, as mesmas serão criadas separadamente agora, buscando sempre manter uma arquitetura clean e organizada.

```
final Map<String, String> enUs = {

  'oi' : 'Hello'

};
```

view rawen_us_translation.dart hosted with ❤ by GitHub

```
final Map<String, String> esMx = {

  'oi':'Holla'

};
```

view rawes_mx_translation.dart hosted with ❤ by GitHub

Criamos a pt_br apenas para mostrar que podemos usar mais de uma lingua(língua).

Agora sempre que o aplicativo reconhecer o idioma do celular do usuário, ou que ele seja trocado dinâmicamente(dinamicamente), através de um botão por exemplo, isso pode ser feito com GetX em apenas uma linha

```
Get.updateLocale(Locale('en', 'US'));
```

E para usar basta concatenar sua chave com .tr, você pode também passar parâmetros para serem traduzidos.

Procure saber mais sobre internacionalização na documentação oficial do GetX.

A classe main

```
import 'package:flutter/material.dart';

import 'package:get/get.dart';

import 'package:getx_pattern/app/routes/app_pages.dart';

import 'package:getx_pattern/app/translations/app_translations.dart';
```

```
import 'app/ui/android/home/home_page.dart';
import 'app/ui/theme/app_theme.dart';

void main() {
  runApp(
    GetMaterialApp(
      debugShowCheckedModeBanner: false,
      initialRoute: Routes.INITIAL, //Rota inicial
      initialBinding: HomeBinding(), // dependencias iniciais
      theme: appThemeData, //Tema personalizado app
      defaultTransition: Transition.fade, // Transição de telas padrão
      getPages: AppPages.pages, // Seu array de navegação contendo as rotas e suas pages
      home: HomePage(), // Page inicial
      locale: Locale('pt', 'BR'), // Língua padrão
      translationsKeys: AppTranslation.translations, // Suas chaves contendo as traduções<map>
    )
  );
}
```

view rawmain.dart hosted with ❤ by GitHub

Agora você está pronto para testar !!!!, dê um flutter run ! (run!)

IMPORTANTE

Esta publicação tem como intuito ver a aceitação da comunidade com esse modelo proposto.

Não está finalizado ainda, e disponibilizaremos exemplos mais exemplos em um futuro próximo.

Qualquer dúvida criei um grupo no telegram que estará na seção de informações.

Isso é uma proposta para auxiliar no processo de desenvolvimento com GetX, não é obrigatório o uso desse pattern caso você não precise de organização, ou queira fazer por aprendizado.

Isso acelerou meu processo de desenvolvimento padronizando meu fluxo de programação.

Deixe uma estrela no repositório e aproveite ao máximo, tudo isso foi feito totalmente para auxiliar você que(você que) desenvolve com GetX.

Tutotiais(Tutoriais) e Exemplos

Estamos trabalhando nisso

By Paulo Silva : Getting Location with GetX and getx_pattern - GitHub Repository

By William Silva : getx_pattern_modulable, a version of the getx pattern divided into modules GitHub Repository

Posts

Dando continuidade ao nosso primeiro exemplo, onde ensinamos você sobre a nossa arquitetura proposta, agora em duas versões, orientado a packages e orientada a módulos , (módulos.)

Vamos agora realizar o restante das requisições http para que você fixe ainda mais nossa proposta e possa aplicar em seus projetos.

Então vamos lá.

Estamos trabalhando nisso, acompanhe aqui

GetX Pattern Web/App

Um dos melhores exemplos é nosso próprio futuro site ! (site!)

Acompanhe e colabore na construção do getx pattern ! (getx_pattern!)

Acompanhe o projeto clicando aqui

Weather

Estamos trabalhando nisso

Login Firebase

Estamos trabalhando nisso

Dúvidas frequentes

Estamos trabalhando nisso

Informações

Package GetX - Clique aqui

Repositório GetX - Clique aqui

Extension GetX Snippets - Clique aqui

Repositório GetX Snippets - Clique aqui

Repositório getx_patterm - Clique aqui

Developer getx_pattern telegram group