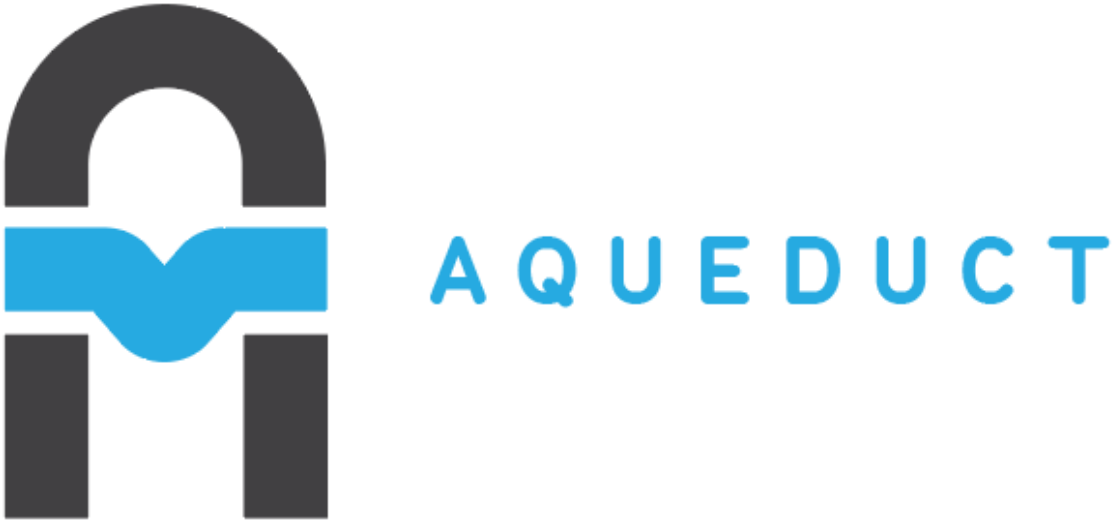
Criando uma aplicação web com Dart. Parte 1: Aqueduct: Básico, conexão com banco e autenticação JWT

Marco Antonio Gonçalves Pegoraro Follow

Oct 26, 2019 · 9 min read

O Dart é uma linguagem de programação criada dentro da Google em 2011 com o objetivo de substituir o javascript como linguagem principal nos navegadores. porém, devido a sua facilidade e curva de aprendizado, acabou se tornando uma linguagem com propósito geral, podendo hoje ser utilizada no backend, frontend e principalmente no mobile e desktop utilizando o framework Flutter.



Nesse tutorial, iremos utilizar o dart para construir o básico de uma aplicação “to-do list” utilizando o framework Aqueduct.

Requisitos

É necessário estar instalado na maquina o SDK do dart, pela linguagem ser multi plataforma, sua instalação pode ser feita tanto para Windows, Linux ou macOS. neste tutorial, estaremos utilizando a versão 2.5.

Get the Dart SDK

The Dart SDK has the libraries and command-line tools that you need to develop Dart web, command-line, and server apps...

dart.dev

Com o Dart instalado, abra um terminal e digite o seguinte comando para realizar a instalação do aqueduct pelo pub, que é o próprio gerenciador de pacotes do Dart.

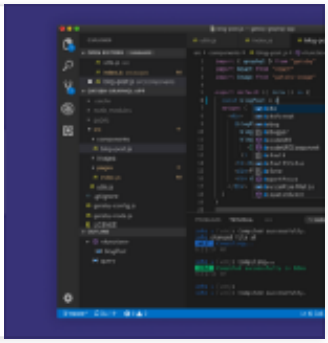
```
pub global activate aqueduct
```

Estaremos utilizando o Visual Studio Code para editar nossos códigos junto com a extensão oficial do Dart. E para testar a aplicação, estaremos utilizando o PostMan.

Download Visual Studio Code - Mac, Linux, Windows

Visual Studio Code is free and available on your favorite platform - Linux, macOS, and Windows. Download Visual Studio...

code.visualstudio.com



Dart

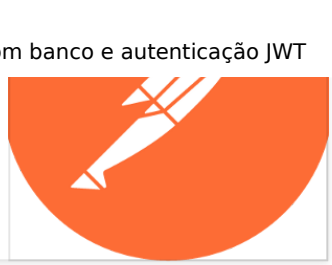
Dart Code extends VS Code with support for the Dart programming language, and provides tools for effectively editing...

marketplace.visualstudio.com

Download Postman App

Be the first to experience new Postman features! Can't wait to see what

www.getpostman.com



Inicio do projeto

Com as ferramentas necessárias já instaladas, abra um novo terminal em sua pasta de projetos e rode o seguinte comando para realizar a criação do projeto.

```
aqueduct create todo
```

Para abrir o projeto no visual studio code, navegue para dentro da pasta utilizando o comando

```
cd todo
```

e em seguida

```
code .
```

Dentro da pasta lib, existe um arquivo chamado channel.dart, esse arquivo é onde será declarado as nossa rotas e controllers.

O primeiro passo que iremos fazer será criar uma pasta dentro da pasta lib chamada models, onde será criado nossas classes modelos. O primeiro arquivo que criaremos será a classe de ToDo, no caso *to_do.dart*.

Dentro desse arquivo, criaremos uma classe estendendo de uma outra classe chamada *Serializable*, isso é necessário para serializar o ToDo para JSON e vice versa.

Além das propriedades da classe, sendo elas id, name e done, também teremos dois métodos que fazem parte da classe herdada *Serializable*, sendo elas o asMap e readFromMap. No final, a classe ficará da seguinte maneira:

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

class ToDo extends Serializable {
  int id;
  String name;
  bool done;
  @override
  Map<String, dynamic> asMap() {
    return {"id": id, "name": name, "done": done};
  }

  @override
  void readFromMap(Map<String, dynamic> object) {
    id = object["id"] as int;
    name = object["name"] as String;
    done = object["done"] as bool;
  }
}
```

to_do.dart hosted with ❤ by **GitHub** **view raw**

Em seguida, criaremos uma pasta dentro da pasta lib chamada *controllers*, onde iremos criar um arquivo chamado *to_do_controller.dart*.

Dentro desse arquivo, criaremos uma classe que irá ser estendida de *ResourceController,* que é a classe padrão do aqueduct para a criação de métodos http dentro de um controller.

A controller com os principais métodos http e uma lista de ToDo's mockados para fins de testes, ficaria assim:

```dart
import 'package:todo/models/to_do.dart';
import 'package:todo/todo.dart';

class ToDoController extends ResourceController {

  ToDoController(){
    acceptedContentTypes = [ContentType.json];
  }

  final List<ToDo> toDos = [
    ToDo()..id = 1 ..name = 'Trabalho da faculdade'..done = true,
```

```dart
      ToDo()..id = 2 ..name = 'Compras'..done = true,
      ToDo()..id = 3 ..name = 'Jantar'..done = false,
      ToDo()..id = 4 ..name = 'Trabalhar'..done = false,
    ];

    @Operation.get()
    Future<Response> getAllToDos() async {
      return Response.ok(toDos);
    }

    @Operation.get('id')
    Future<Response> getToDoByID() async {
      final id = int.parse(request.path.variables['id']);
      final toDo = toDos.firstWhere((todo) => todo.id == id, orElse: () => null);
      if (toDo == null) {
        return Response.notFound();
      }

      return Response.ok(toDo);
    }

    @Operation.post()
    Future<Response> postToDo(@Bind.body() ToDo toDo) async {
      toDos.add(toDo);
      return Response.ok(toDos);
    }


    @Operation.put()
    Future<Response> putToDo(@Bind.body() ToDo toDo) async {
      toDos.removeAt(toDos.indexWhere((r)=> r.id == toDo.id));
      toDos.add(toDo);
      return Response.ok(toDos);
    }

    @Operation.delete('id')
    Future<Response> deleteToDoByID() async {
      final id = int.parse(request.path.variables['id']);
      toDos.removeAt(toDos.indexWhere((r)=> r.id == id));
      return Response.ok(toDos);
    }
  }
```

to_do_controller.dart hosted with ❤ by **GitHub** **view raw**

Por fim, precisaremos incluir a controller de “todos” e qual será a sua rota no arquivo channel.dart.

```dart
import 'package:todo/controllers/to_do_controller.dart';

import 'todo.dart';

class TodoChannel extends ApplicationChannel {
  @override
  Future prepare() async {
    logger.onRecord.listen((rec) => print("$rec ${rec.error ?? ""} ${rec.stackTrace ?? "
  }

  @override
  Controller get entryPoint {
    final router = Router();

      router
        .route("/todo/[:id]").link(() => ToDoController());

    return router;
  }
}
```

channel.dart hosted with ❤ by **GitHub** **view raw**

Em seguida, abra o terminal do visual studio code utilizando o comando Ctrl + shift + ` e rode o comando “aqueduct serve” para rodar a aplicação.

Com o postman, você já poderá fazer requisições para a api:

```
GET:    localhost:8888/todo
GET:    localhost:8888/todo/1
POST:   localhost:8888/todo
PUT:    localhost:8888/todo
DELETE: localhost:8888/todo/1
```

Porém, você perceberá que os dados não são persistidos, por exemplo, ao fazer um método post ou delete, isto acontece por que o Aqueduct reseta o estado da aplicação

para cada request, isso é melhor explicado no seguinte link da documentação oficial:

Multi-threading - Aqueduct

One of the primary differentiators between Aqueduct and other server frameworks is its multi-threaded behavior. When...

aqueduct.io

Alguns links interessantes:

The Builder Pattern in Java, and Dart Cascades

Object construction is something that everyone will have to do in a language that has object-oriented paradigms...

dev.to

1. Getting Started

By the end of this tutorial, you will have created an Aqueduct application that serves fictional heroes from a...

aqueduct.io

Conexão com banco de dados PostgreSQL

Agora, iremos fazer algumas modificações para fazer acesso a um banco de dados Postgres em nossa API.

Primeiramente, é necessário uma instalação do Postgres na maquina de desenvolvimento, podendo ela ser feita direto na maquina ou utilizando o Docker.

Estarei utilizando neste tutorial, a imagem oficial do Postgres no dockerhub, para subir o serviço na minha maquina, utilizarei o seguinte comando no terminal:

```
docker run --name postgres -p 5432:5432 -e POSTGRES_PASSWORD=postgres
-d postgres:9.6
```

Com o projeto aberto no Visual Studio Code, faremos algumas modificações na model de todo para ser feito o uso dela no banco de dados, a classe não precisará estender de Serializable pois a classe ManagedObject já estende de Serializable.

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

class ToDo extends ManagedObject<_ToDo> implements _ToDo {}

class _ToDo {
  @primaryKey
  int id;
  String name;
  bool done;
}
```

to_do.dart hosted with ❤ by GitHub view raw

Em seguida, faremos algumas modificações no arquivo channel.dart

```dart
import 'package:todo/controllers/to_do_controller.dart';

import 'todo.dart';

class TodoChannel extends ApplicationChannel {
  ManagedContext context;

  @override
  Future prepare() async {
    logger.onRecord.listen((rec) => print("$rec ${rec.error ?? ""} ${rec.stackTrace ?? "

   final dataModel = ManagedDataModel.fromCurrentMirrorSystem();
    final persistentStore = PostgreSQLPersistentStore.fromConnectionInfo(
        "postgres", "postgres", "localhost", 5432, "postgres");

    context = ManagedContext(dataModel, persistentStore);
  }

  @override
  Controller get entryPoint {
    final router = Router();

      router
        route("/todo/[:id]").link(() => ToDoController(context));
```

```
25
26      return router;
27    }
28  }
```

channel.dart hosted with ❤ by GitHub view raw

Aqui criamos uma variável context que guardará as informações de banco de dados e em seguida, iremos passar ela para o nosso controller de ToDo fazer as consultas de banco de dados. Precisaremos preparar o nosso controller para receber o contexto e fazer as operações de banco de dados, ficando assim:

```dart
import 'package:todo/models/to_do.dart';
import 'package:todo/todo.dart';

class ToDoController extends ResourceController {
  ToDoController(this.context) {
    acceptedContentTypes = [ContentType.json];
  }

  final ManagedContext context;

  @Operation.get()
  Future<Response> getAllToDos() async {
    final query = await Query<ToDo>(context);
    final toDos = await query.fetch();
    return Response.ok(toDos);
  }

  @Operation.get('id')
  Future<Response> getToDoByID() async {
    final id = int.parse(request.path.variables['id']);
    final query = Query<ToDo>(context)
      ..where((todo) => todo.id).equalTo(id);
    final toDo = await query.fetchOne();

    if (toDo == null) {
      return Response.notFound();
    }

    return Response.ok(toDo);
  }
```

```dart

  @Operation.post()
  Future<Response> postToDo() async {
    final body = ToDo()..read(await request.body.decode(), ignore: ["id"]);
    final query = Query<ToDo>(context)
      ..values.name = body.name
      ..values.done = body.done;

    final toDo = await query.insert();
    return Response.ok(toDo);
  }

  @Operation.put('id')
  Future<Response> putToDo(@Bind.path("id") int id) async {
    final body = ToDo()..read(await request.body.decode(), ignore: ["id"]);
    final query = Query<ToDo>(context)
      ..values = body
      ..where((todo) => todo.id).equalTo(id);
    final toDo = await query.updateOne();
    return Response.ok(toDo);
  }

  @Operation.delete('id')
  Future<Response> deleteToDoByID(@Bind.path("id") int id) async {
    final query = Query<ToDo>(context)
      ..where((todo) => todo.id).equalTo(id);
    await query.delete();
    return Response.ok({'ok': true});
  }
}
```

to_do_controller.dart hosted with ❤ by **GitHub** **view raw**

Ajustamos o controller para receber o contexto de banco de dados e modicamos todos os métodos http para fazer consultas ao banco de dados utilizando a classe Query do Aqueduct. Mais informações sobre o ORM podem ser encontradas nos links:

Basic Queries — Aqueduct

To send commands to a database — whether to fetch, insert, delete or update objects — you will create, configure and...

aqueduct.io

Advanced Queries — Aqueduct

The rows from a table can be sorted and fetched in contiguous chunks. This sorting can occur on most properties. For...

aqueduct.io

Nosso projeto já está quase pronto para funcionar com banco de dados, agora falta somente fazer a migração da model ToDo para o banco de dados, o primeiro passo é criar um arquivo na pasta raiz do projeto chamada database.yaml, que será utilizado pelo aqueduct para localizar o banco de dados e fazer as migrações, o conteúdo do arquivo será então:

```
username: postgres
password: postgres
host: localhost
port: 5432
databaseName: postgres
```

database.yaml hosted with ❤ by **GitHub** **view raw**

Por fim, iremos criar a migração. Com o terminal do Visual Studio Code aberto, rode o seguinte comando para criar uma migração:

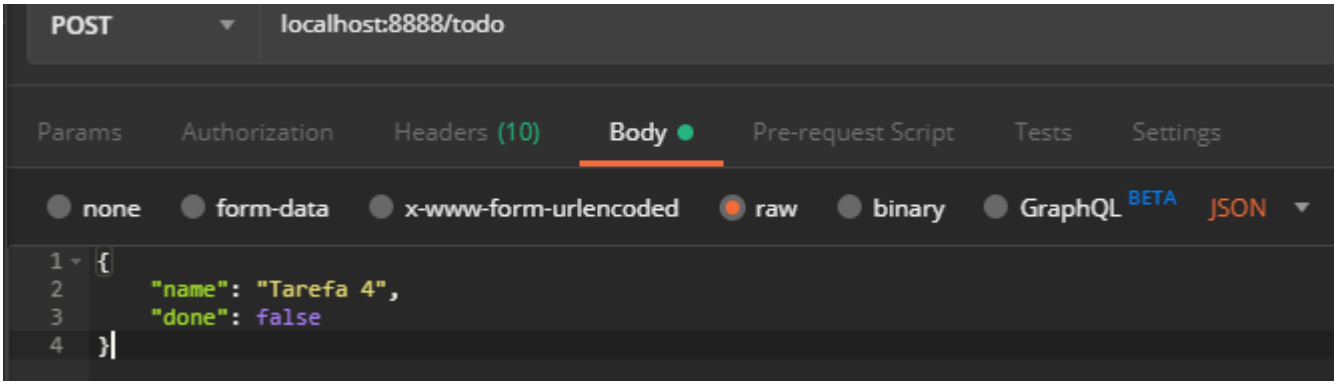
```
aqueduct db generate
```

Com esse comando, o Aqueduct detecta as models do projeto para criar a migração, as migrações ficam dentro do diretório migrations.

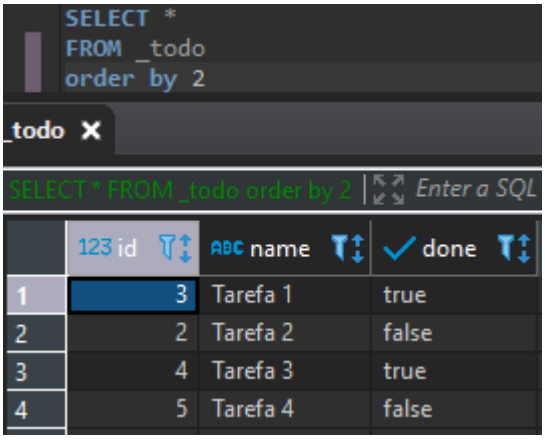
Para aplicar a migração, precisaremos rodar o comando:

```
aqueduct db upgrade
```

Aplicada a migração, você já poderá testar o projeto realizando requisições HTTP, você perceberá que os dados agora estão sendo persistidos.



Exemplo de requisição POST



Dados persistidos no banco de dados

Autenticação JWT

Como qualquer aplicação web, é necessário algum tipo de autenticação. Uma das maneiras mais comuns que se tem hoje em dia para realizar autenticação em API's REST é utilizando um padrão chamado JWT, para mais informações, recomendo a leitura do seguinte artigo do Wellington Nascimento sobre o assunto:

Entendendo tokens JWT (Json Web Token)

Esse artigo apareceu primeiro em wellingtonjhn.com

medium.com

Neste artigo, demonstrarei como aplicar esse padrão de autenticação na nossa API em Dart.

Relacionamento one-to-many

A primeira coisa que iremos fazer será criar um model para o usuário e fazer a relação com o ToDo. Dentro da pasta models, criaremos então uma classe user.dart:

```dart
import 'package:todo/models/to_do.dart';
import 'package:todo/todo.dart';

class User extends ManagedObject<_User> implements _User {}

class _User {
  @primaryKey
  int id;

  @Column(unique: true)
  String name;

  @Column(unique: true)
  String email;

  @Column(omitByDefault: true)
  String password;

  String passwordHash;

  ManagedSet<ToDo> toDo;
}
```

user.dart hosted with ❤ by GitHub view raw

Perceba que existe uma propriedade *ManagedSet* que será responsável pela relacionamento one-to-many com o ToDo, mas precisaremos incluir o relacionamento também na classe de ToDo, portanto, adicione o atributo a seguir na classe todo.dart:

```
@Relate(#toDo)

User user;
```

Por fim, precisaremos criar uma migration no banco de dados, para criar uma migration, utilizamos o comando

```
aqueduct db generate
```

Na migration criada, iremos fazer uma pequena alteração, pois ela incluiu a criação de uma coluna *password* na tabela *_user*, o que não será necessário pois gravaremos somente o hash da senha no banco de dados, então sinta-se a vontade para remover o seguinte trecho da migration:

```
SchemaColumn("password", ManagedPropertyType.string,
isPrimaryKey: false,
autoincrement: false,
isIndexed: false,
isNullable: false,
isUnique: false),
```

Para aplicar a migration, utilize o comando:

```
aqueduct db upgrade
```

Geração do token JWT

Em seguida, importaremos uma dependência que será utilizada para gerar e descriptografar o token JWT. Então no arquivo pubspec.yaml, inclua a dependência:

```
jaguar_jwt: ^2.1.6
```

Em seguida, criaremos um diretório dentro de *lib* chamado *utils* e dentro desta pasta, criar um arquivo utils.dart que será utilizado para deixar funções comuns que serão utilizadas por todo o projeto. Dentro deste arquivo, teremos alguns atributos e métodos:

1. jwtKey — A chave privada para a criação dos tokens JWT.
2. generateJWT — Método que receberá um usuário e retornará uma string (o próprio token JWT).
3. generateSHA256Hash — Método que irá gerar o hash da senha do usuário.

O arquivo utils.dart ficará assim:

```dart
import 'dart:convert';
import 'package:crypto/crypto.dart';
import 'package:jaguar_jwt/jaguar_jwt.dart';

import 'package:todo/models/user.dart';

class Utils {
  static const String jwtKey = "<sua-chave-privada>";

  static String generateJWT(User user) {
    final claimSet = JwtClaim(
        issuer: "http://localhost:8888",
        subject: user.id.toString(),
        otherClaims: <String, dynamic>{},
        maxAge: Duration(days: 1));

    final token = "Bearer ${issueJwtHS256(claimSet, jwtKey)}";
    return token;
  }

  static String generateSHA256Hash(String password) {
    final bytes = utf8.encode(password);
    final passwordHash = sha256.convert(bytes).toString();

    return passwordHash;
  }
}
```

utils.dart hosted with ❤ by **GitHub** **view raw**

Agora, precisaremos criar dois novos controllers, o primeiro controller será utilizado para criar um novo usuário e gerar um hash da senha para o mesmo, ficando assim:

```dart
import 'package:todo/models/user.dart';
```

```dart
import 'package:todo/models/user.dart';
import 'package:todo/todo.dart';
import 'package:todo/utils/utils.dart';

class UserController extends ResourceController {
  UserController(this.context) {
    acceptedContentTypes = [ContentType.json];
  }

  final ManagedContext context;

  @Operation.post()
  Future<Response> postUser() async {
    final body = User()..read(await request.body.decode(), ignore: ["id"]);

    final query = Query<User>(context);
    body.passwordHash = Utils.generateSHA256Hash(body.password);
    query.values.email = body.email;
    query.values.passwordHash = body.passwordHash;
    query.values.name = body.name;

    final user = await query.insert();
    return Response.ok("Usuário criado com sucesso");
  }
}
```

user_controller.dart hosted with ❤ by **GitHub** **view raw**

E por fim, um controller que será utilizado para validar o usuário e devolver um token autenticado para ele, chamaremos ele de session_controller.dart.

```dart
import 'package:todo/models/user.dart';
import 'package:todo/todo.dart';
import 'package:todo/utils/utils.dart';

class SessionController extends ResourceController {
  SessionController(this.context) {
    acceptedContentTypes = [ContentType.json];
  }

  final ManagedContext context;

  @Operation.post()
  Future<Response> login() async {
```

```
14      final body = User()..read(await request.body.decode());
15
16      final passwordHash = Utils.generateSHA256Hash(body.password);
17
18      final query = Query<User>(context)
19        ..where((user) => user.email).like(body.email)
20        ..where((user) => user.passwordHash).like(passwordHash.toString());
21      final user = await query.fetchOne();
22
23      if (user == null) {
24        return Response.ok({"auth": false,"token": ""});
25      }
26
27      final jwt = Utils.generateJWT(user);
28
29      return Response.ok({"auth": true, "token": jwt});
30    }
31  }
```

session_controller.dart hosted with ❤ by **GitHub** **view raw**

Middleware de autenticação

No Aqueduct, todo controller é um middleware, você pode inclusive encadear controllers para se chegar em um resultado esperado, é isso que iremos fazer, antes da requisição chegar no ToDoController, precisaremos ter um middleware que verificará se o token JWT passado pelo header é realmente válido. Para fazer isso, criaremos uma pasta dentro de lib chamada middlewares, e dentro dela, uma classe jwt_middleware.dart que terá o comportamento muito parecido com um controller, porém não irá retornar nada para o usuário, somente passar a requisição adiante para o próximo controller.

```
1   import 'package:jaguar_jwt/jaguar_jwt.dart';
2   import 'package:todo/models/user.dart';
3   import 'package:todo/todo.dart';
4   import 'package:todo/utils/utils.dart';
5
6   class JwtMiddleware extends Controller {
7     JwtMiddleware(this.context);
8
9     final ManagedContext context;
10
```

```dart
  @override
  FutureOr<RequestOrResponse> handle(Request request) async {
    //Obter o header de autenticação da requisição
    final authHeader = request.raw.headers["authorization"];

    if (authHeader.isEmpty) return Response.unauthorized();

    final authHeaderContent = authHeader[0].split(" ");
    if (authHeaderContent.length != 2 || authHeaderContent[0] != "Bearer")
      return Response.badRequest();

    String jwtToken = authHeaderContent[1];

    try {
      final JwtClaim decClaimSet =
          verifyJwtHS256Signature(jwtToken, Utils.jwtKey);

      final userId = int.parse(decClaimSet.toJson()["sub"].toString());

      if (userId == null) {
        throw JwtException;
      }

      //Verificar se o JWT está atualizado
      final dataAtual = DateTime.now().toUtc();
      if (dataAtual.isAfter(decClaimSet.expiry)) {
        return Response.unauthorized();
      }

      //Validar se usuário existe
      final query = Query<User>(context)
        ..where((user) => user.id).equalTo(userId);
      final user = await query.fetchOne();

      if (user == null) return Response.unauthorized();

      request.attachments['user'] = user;
    } on JwtException {
      return Response.unauthorized();
    }

    return request;
  }
}
```

jwt_middleware.dart hosted with ❤ by **GitHub** **view raw**

Neste middleware, vamos descriptografar o token, pegar o id do usuário no token e fazer algumas validações (ver se o usuário existe, verificar a data de expiração do token, etc) e por fim, passar o usuário do token para o próximo controller.

Para mais informações sobre como validar os dados do token:

jaguar_jwt | Dart Package

JWT utilities for Dart and Jaguar.dart This library can be used to generate and process JSON Web Tokens (JWT). For more...

pub.dev

Em seguida, precisaremos fazer algumas modificações na variável routes do arquivo channel.dart para incluir os novos controllers e o novo middleware de autenticação.

```
router

.route("/todo/[:id]")

.link(() => JwtMiddleware(context))

.link(() => ToDoController(context));

router.route("/user/[:id]").link(() => UserController(context));

router.route("/session/[:id]").link(() =>
SessionController(context));
```

Por fim, precisaremos fazer algumas modificações no ToDoController, pois agora o mesmo precisa filtrar os todo's também pelo id do usuário, existem diversas maneiras de se fazer isso, mas para deixar simplificado, utilizaremos o id que foi obtido do token.

```
1  import 'package:todo/models/to_do.dart';
2  import 'package:todo/models/user.dart';
3  import 'package:todo/todo.dart';
4
5  class ToDoController extends ResourceController {
6    ToDoController(this.context) {
```

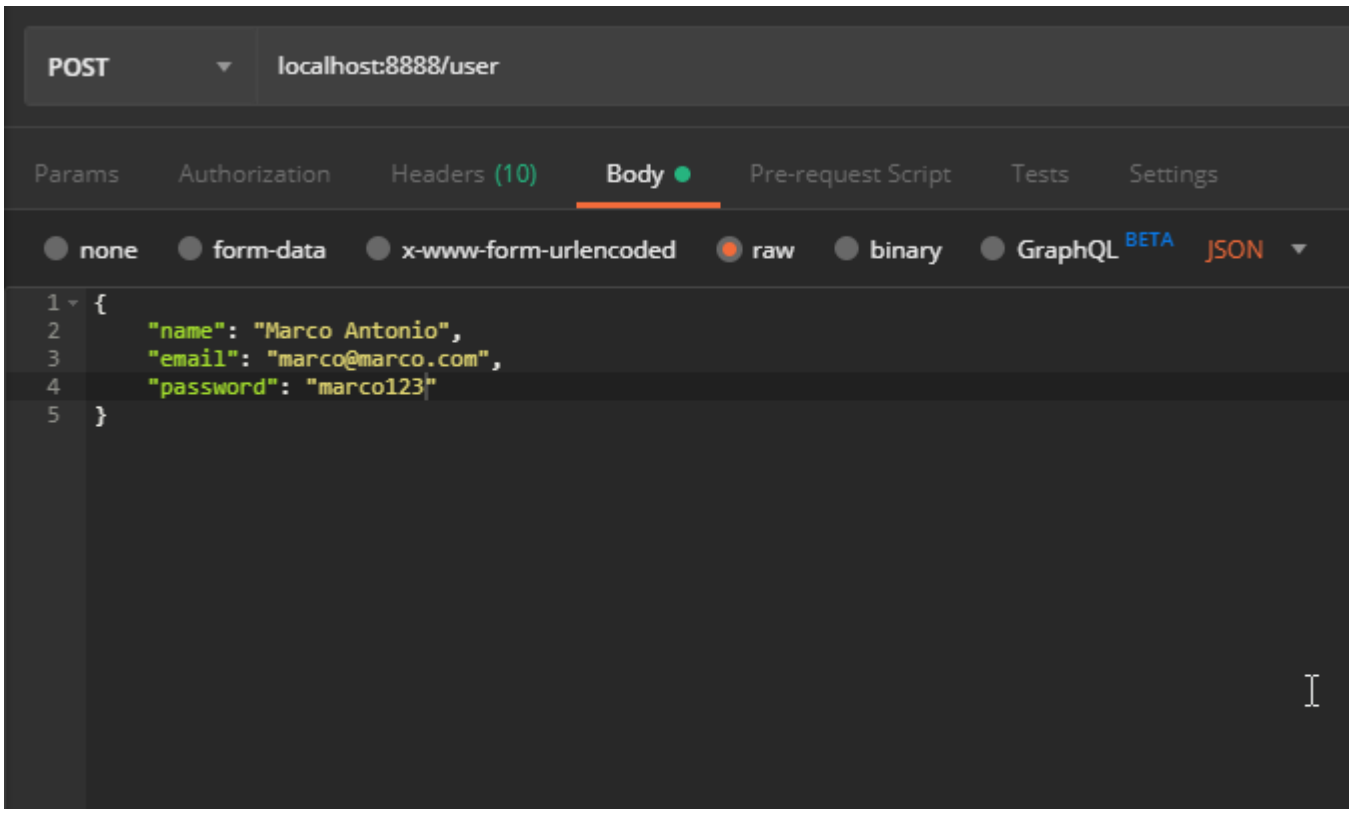
```
7       acceptedContentTypes = [ContentType.json];
8     }
9
10    final ManagedContext context;
11
12    @Operation.get()
13    Future<Response> getAllToDos() async {
14      User user = request.attachments['user'] as User;
15      final query = await Query<ToDo>(context)
16        ..where((todo) => todo.user.id).equalTo(user.id);
17      final toDos = await query.fetch();
18      return Response.ok(toDos);
19    }
20
21    @Operation.get('id')
22    Future<Response> getToDoByID() async {
23      User user = request.attachments['user'] as User;
24      final id = int.parse(request.path.variables['id']);
25      final query = Query<ToDo>(context)
26        ..where((todo) => todo.id).equalTo(id)
27        ..where((todo) => todo.user.id).equalTo(user.id);
28      final toDo = await query.fetchOne();
29
30      if (toDo == null) {
31        return Response.notFound();
32      }
33
34      return Response.ok(toDo);
35    }
36
37    @Operation.post()
38    Future<Response> postToDo() async {
39      User user = request.attachments['user'] as User;
40      final body = ToDo()..read(await request.body.decode(), ignore: ["id"]);
41      final query = Query<ToDo>(context)
42        ..values.name = body.name
43        ..values.done = body.done
44        ..values.user.id = user.id;
45
46      final toDo = await query.insert();
47      return Response.ok(toDo);
48    }
49
50    @Operation.put('id')
```

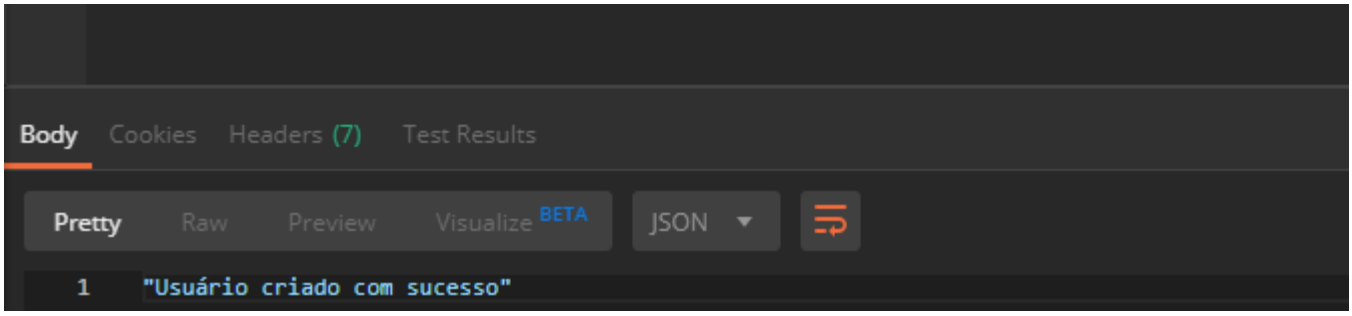
```dart
  Future<Response> putToDo(@Bind.path("id") int id) async {
    User user = request.attachments['user'] as User;
    final body = ToDo()..read(await request.body.decode(), ignore: ["id"]);
    final query = Query<ToDo>(context)
      ..values = body
      ..where((todo) => todo.id).equalTo(id)
      ..where((todo) => todo.user.id).equalTo(user.id);
    final toDo = await query.updateOne();
    return Response.ok(toDo);
  }

  @Operation.delete('id')
  Future<Response> deleteToDoByID(@Bind.path("id") int id) async {
    User user = request.attachments['user'] as User;
    final query = Query<ToDo>(context)
      ..where((todo) => todo.id).equalTo(id)
      ..where((todo) => todo.user.id).equalTo(user.id);
    await query.delete();
    return Response.ok({'ok': true});
  }
}
```

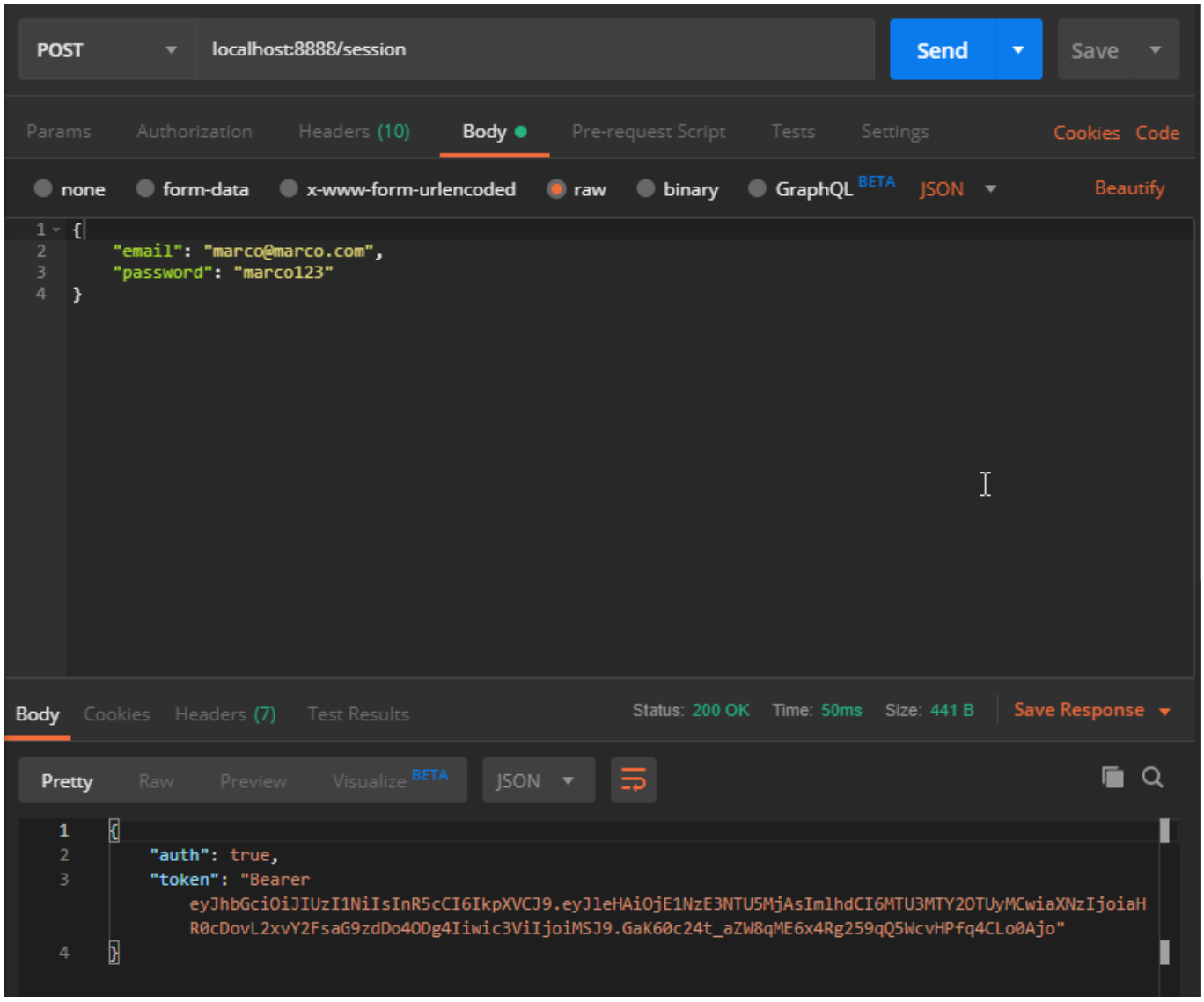
to_do_controller.dart hosted with ❤ by GitHub view raw

Testes

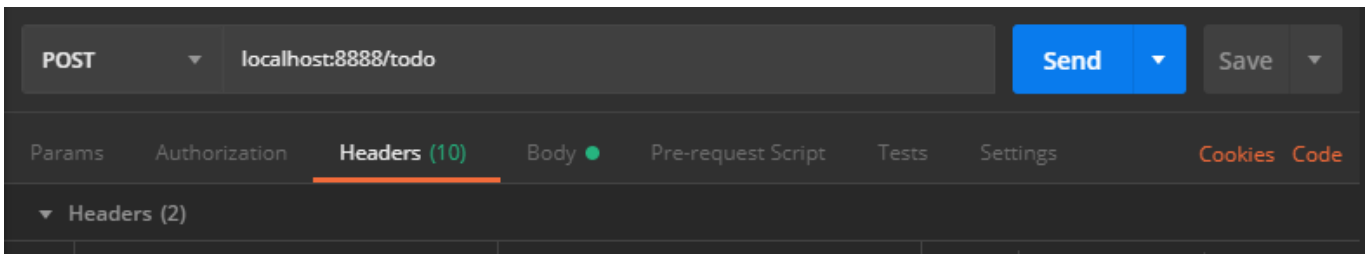


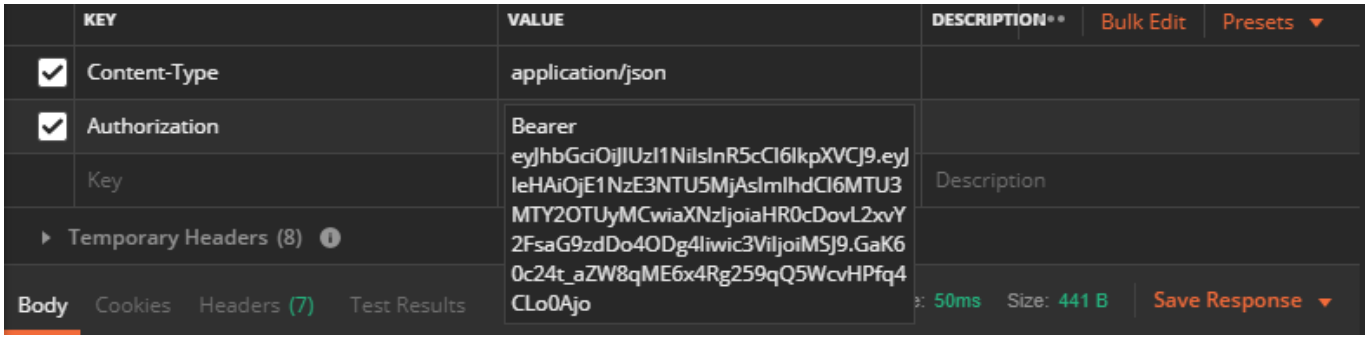


Criação de usuário



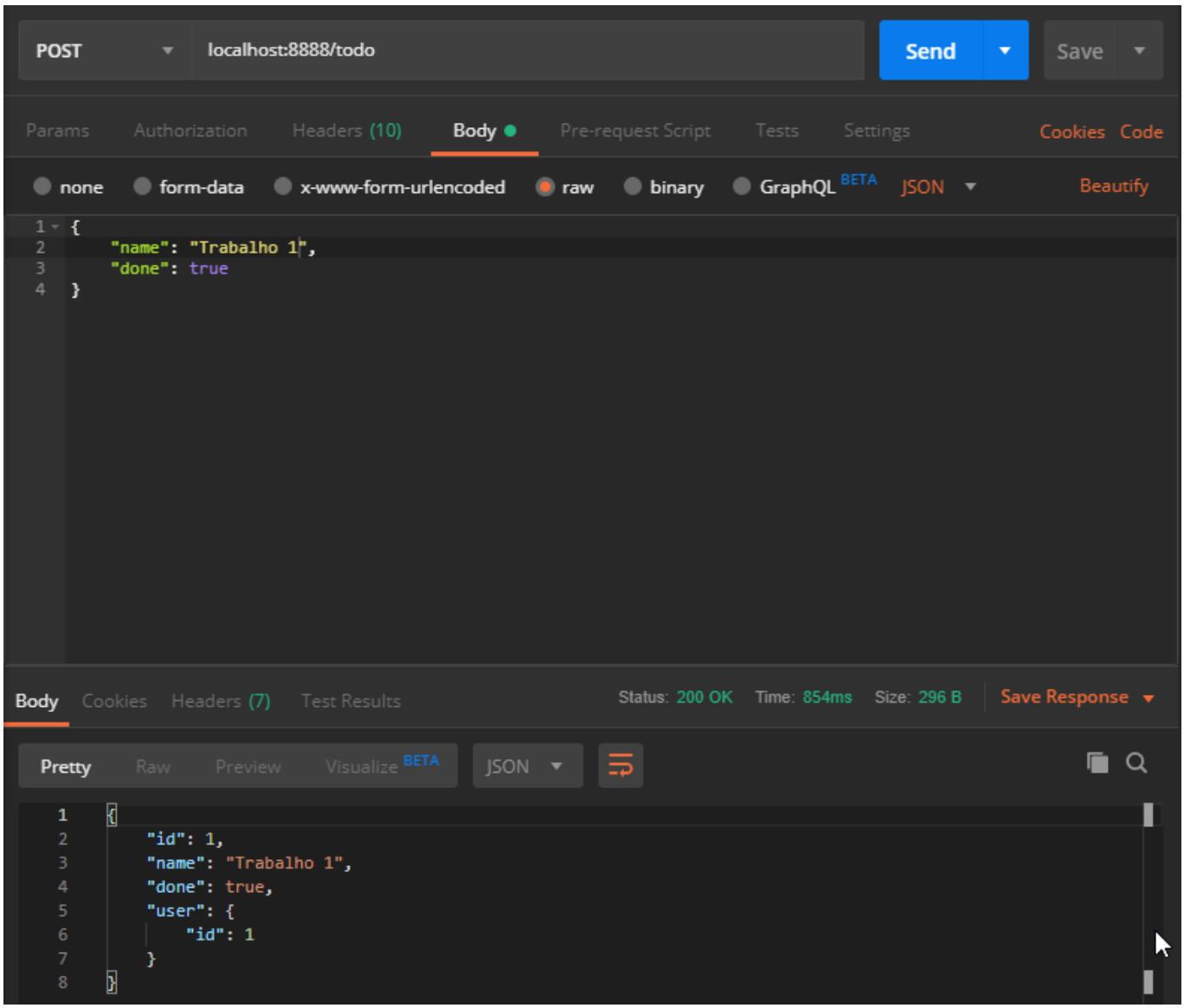
Autenticação de usuário



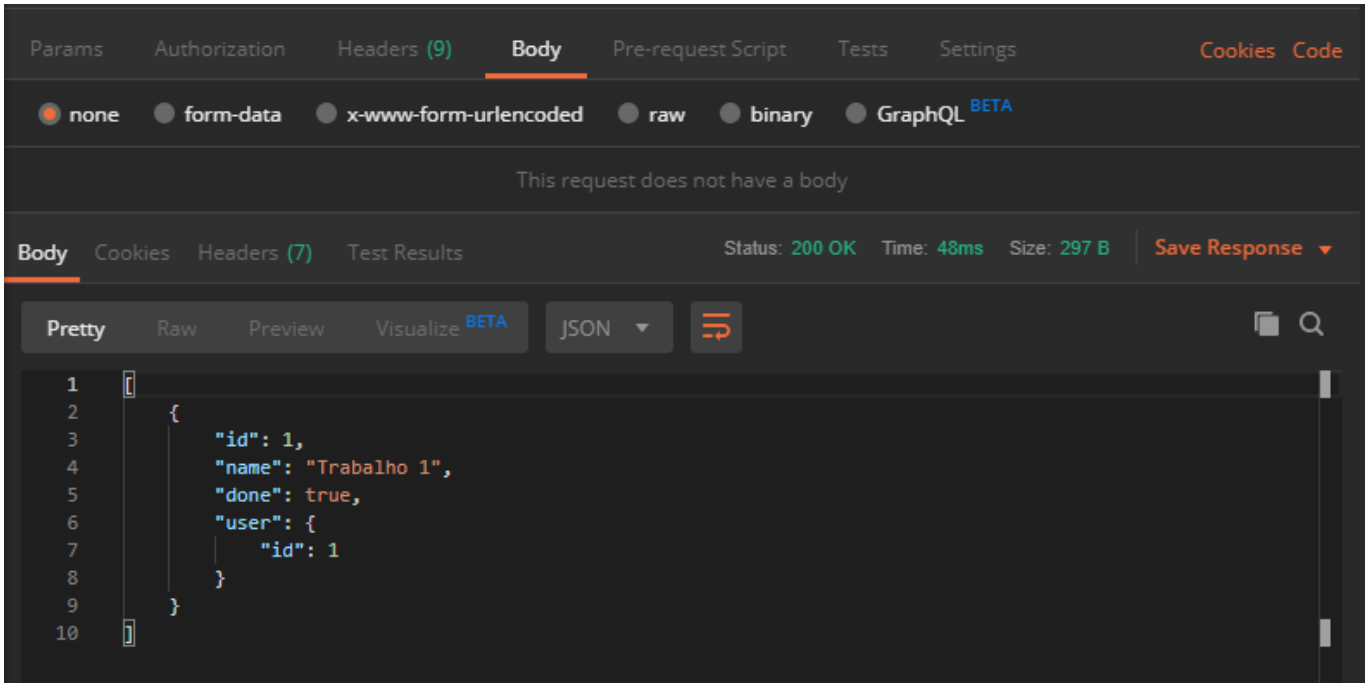


Colocando o token no header da requisição

Agora você poderá criar, editar, listar e deletar os ToDo's baseado no usuário logado.



Inclusão de ToDo



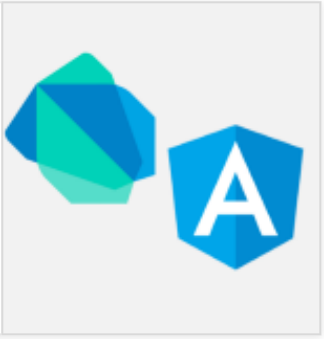
Listagem de ToDo

Com isso, terminamos o nosso backend em Dart, para finalizar, no próximo artigo demonstrarei como criar um frontend para consumir a API criada utilizando o framework AngularDart.

Criando uma aplicação web com Dart. Parte 2: AngularDart: Básico, roteamento e chamadas HTTP

No artigo anterior, ensinei como criar uma aplicação backend utilizando a linguagem de programação Dart juntamente com...

medium.com



O repositório do projeto pode ser encontrado no seguinte link do meu GitHub:

marcoagpegoraro/todo-app-aqueduct

It's necessary a postgres database in order for this application to work properly, modify channel.dart ant...

github.com

Thanks to Bruno Eleodoro Roza.

Dart Aqueduct API Jwt Postgresql