

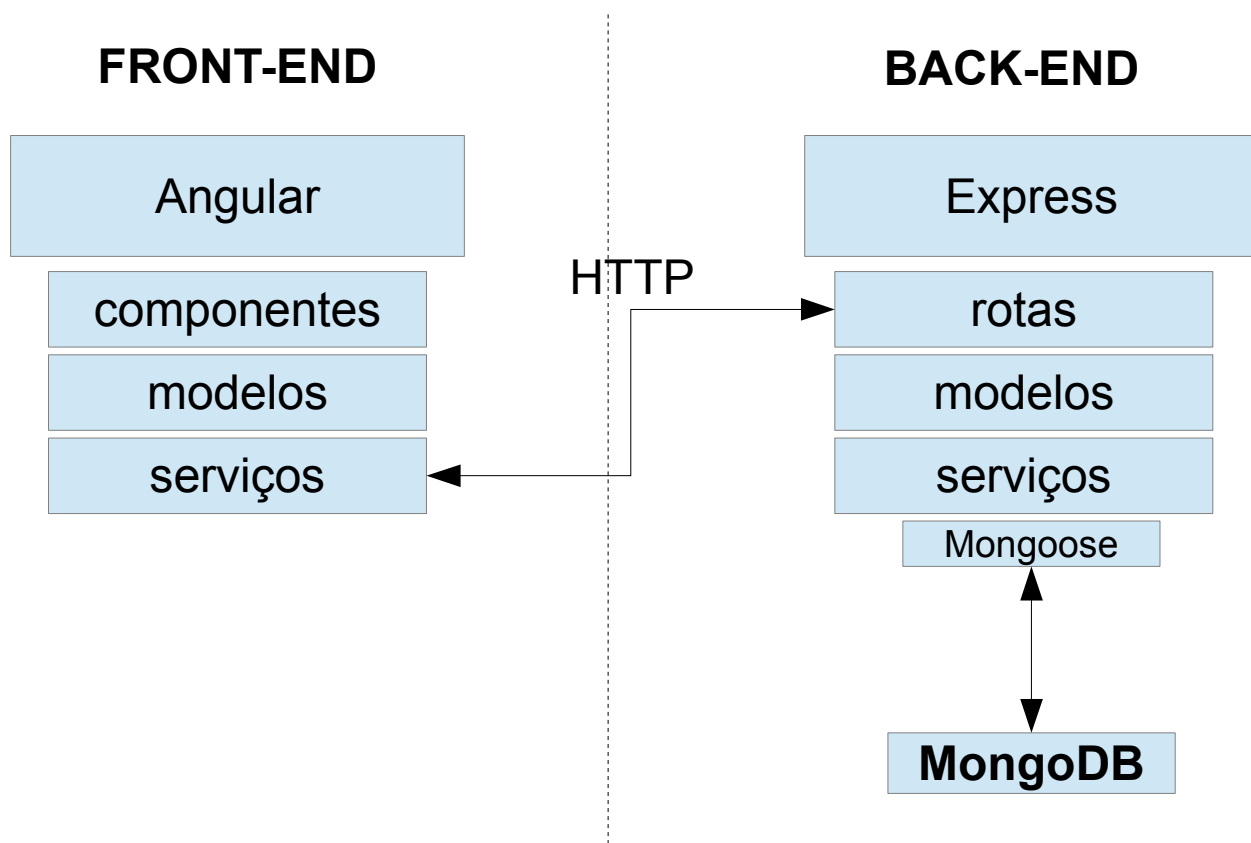
Projeto de Interfaces WEB

CRUD: Mongoose e Express
Aula 12

Introdução

- O objetivo desse mini-projeto é:
 - Criar um servidor simples em Express;
 - Criar um Schema para User, usando o Mongoose
 - Conectar com o MongoDB local (instalar)
 - Operações CRUD em algum cliente REST (ARC?)
 - Conectar a nossa aplicação Angular com o servidor Express.

Introdução



Introdução

- MongoDB e Node.js
 - MongoDB e Node.js são comumente usadas juntas devido a popularidade da linguagem Javascript e a notação de objeto JSON. O JSON está se tornando rapidamente o padrão de formato de dados transmitidos pela WEB.
- O que é o Mongoose?
 - É uma biblioteca de modelagem de dados (ODM- Object Data Model) que provê um ambiente rigoroso para modelagem dos seus dados, reforçando a sua estrutura mas mantendo a flexibilidade do MongoDB.

Criando o Projeto

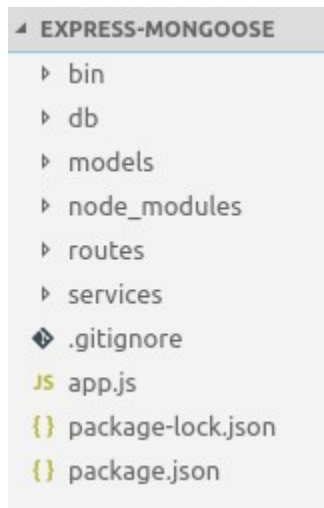
- Para criar o projeto, vamos usar a `express-api`. Caso não tenha esse módulo do Node instalado, faça:
 - **`npm install -g express-generator-api`**
- Depois, pra criar o projeto, simplesmente faça:
 - **`express-api <nome_do_projeto>`**
 - cria a pasta de fato
 - **`cd <nome_do_projeto>`**
 - entra na pasta
 - **`npm install`**
 - instala as dependências na `node_modules`.

Criando o Projeto

- Instalando o Mongoose
 - Entre na pasta do projeto express, criado pela express-api
 - Instale o mongoose:
 - **npm install mongoose --save**
- Crie um projeto, usando a express-api, com o nome **express-mongoose**.

Criando o Projeto

- Diretórios do projeto:



- Crie as pastas **db**, **models** e **services** (sugestão).

Criando o Projeto

- Na pasta **db**:
 - Iremos criar o arquivo js para conexão com o banco de dados MongoDB local.
- Na pasta **models**:
 - Iremos criar o arquivo js responsável pelo Schema de uma “tabela” (collection). Esse arquivo irá nos retornar um Model (classe do Mongoose) que permite as operações de CRUD no esquema criado.
- Na pasta **services**:
 - Iremos criar o arquivo controlador, também um js, o qual usará o arquivo criado na pasta model e fará as operações de CRUD.
- Nas pasta **routes**:
 - Iremos criar um arquivo de rotas em js. Esse arquivo será usado pelo arquivo principal da aplicação (o app.js) para export os endpoints do nosso serviço REST. Sendo assim, possível que clientes façam requisições.

A pasta db

- Crie o arquivo mongo.connection.js

//usando o mongoose (FRONT-END para o mongoDB)

```
var mongoose = require('mongoose');
```

importando o mongoose.

//updating

```
mongoose.set('useFindAndModify', false);
```

evitando depreciações.

//conexão local

```
var mongoDB_URI = 'mongodb://127.0.0.1:27017/piw';
```

```
mongoose.connect(mongoDB_URI, {useNewUrlParser: true});
```

conexão com o banco local. O db deve ser criado antes.

//armazena a conexão em uma variável

```
var db = mongoose.connection;
```

variável com a conexão.

//listeners

```
db.on('connected', ()=>{  
  console.log('Mongoose Connected to '+mongoDB_URI);  
});
```

```
db.on('disconnected', ()=>{  
  console.log('Mongoose Disconnected to '+mongoDB_URI);  
});
```

listeners

```
db.on('error', (err) => {  
  console.log('Mongoose Error: '+err);  
});
```

A pasta models

- Crie o arquivo user.model.js

```
var mongoose = require('mongoose');

//criando o schema, o qual servirá para criar o modelo (collections)
var UserSchema = mongoose.Schema(
  {
    firstName: {type:String, required:true, max:100},
    lastName: {type:String, required:true, max:100},
    login: {type:String, required:true, max:100},
    email: {type:String, required:true, max:100},
    zipcode: {type:String, required:true, max:10},
    password: {type:String, required:true, max:20}
  }
);

//criando o modelo a partir do schema acima, o qual servirá para incluir as instâncias
//((documentos))
var UserModel = mongoose.model('users', UserSchema);

//retornando o modelo a ser usado pelo serviço (CRUD).
module.exports = UserModel;
```

Esquema de um user. Responsável em criar a Collection. Não se preocupe com o `_id`.

Model que será usado para criar Documents

Schemas Mongoose

- Os Schemas mongoose oferecem métodos para manipular coleções
 - `.create()` - inserir objeto na coleção
 - `.find(critério)` - busca de documentos com critério
 - `.findById(id)` - busca documento pelo id
 - `.remove(critério)` - remove documento
 - `.findByIdAndRemove(id)` - remove e retorna documento
 - `.findByIdAndUpdate(id, novo_doc)` - atualiza documento
 - ... (dentre outros)

A pasta services

- Crie o arquivo user.service.mongo.js

```
const UserModel = require('../models/user.model');
```

Importa o Model do arquivo anterior.

```
class UserService{
```

```
  //retorna um objeto que representa um User
```

```
  static register(req,res){
```

```
    UserModel.create(req.body).then(
```

```
      (user)=>{
```

```
        res.status(201).json(user);
```

```
      }
```

```
    );
```

```
  }
```

```
  //retorna um vetor de users
```

```
  static list(req,res){
```

```
    UserModel.find().then(
```

```
      (users)=>{
```

```
        res.status(201).json(users);
```

```
      }
```

```
    );
```

```
  }
```

Cria todos os métodos do CRUD. Os métodos create, find, etc já foram implementados pelo Model do mongoose. Todos retornam uma Promise (assíncrona). Temos então que chamar o método “then” e implementar uma função que irá passar os resultados para “res”.

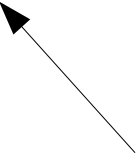
A pasta services (cont.)

```
//retorna um user atualizado
static update(req,res){
  UserModel.findByIdAndUpdate(req.params.id, req.body, {'new':true}).then(
    (user)=>{
      res.status(201).json(user);
    }
  );
}
```

```
//retorna o user deletado
static delete(req,res){
  UserModel.findByIdAndRemove(req.params.id).then(
    (user)=>{
      res.status(201).json(user);
    }
  );
}
```

```
//retorna um user
static retrieve(req,res){
  UserModel.findById(req.params.id).then(
    (user)=>{
      res.status(201).json(user);
    }
  );
}
```

Nesse caso, quero que retorne o objeto modificado.



A pasta services (cont.)

```
//retorn um vetor de user
static retrieveByLogin(req,res){
  UserModel.find({'login':req.params.login}).then(
    (user)=>{
      res.status(201).json(user);
    }
  );
}
```

```
}
```

```
module.exports = UserService;
```

Exporta o serviço para ser usado pelo route.



A pasta routes

- Crie o arquivo users.routes.mongo.js

```
var express = require('express');  
var router = express.Router();  
var userService = require('../services/user.service.mongo');
```

Importando o serviço

```
router.get('/list', function(req, res, next) {  
  userService.list(req, res);  
});
```

```
router.post('/register', function(req, res, next){  
  userService.register(req, res);  
});
```

```
router.put('/update/:id', function(req, res, next){  
  userService.update(req, res);  
});
```

```
router.delete('/delete/:id', function(req, res, next){  
  userService.delete(req, res);  
});
```

```
router.get('/retrieve/:id', function(req, res, next){  
  userService.retrieve(req, res);  
});
```

```
router.get('/retrieve/login/:login', function(req, res, next){  
  userService.retrieveByLogin(req, res);  
});
```

```
module.exports = router;
```

Exporta o router, a ser usado em app.js

Cada router implementa um verbo do HTTP. Post, delete, put, get.

O primeiro parâmetro é a string que ficará no final da URL (por exemplo, “/list” ou “/update/:id”).

O segundo parâmetro é uma função que irá chamar o serviço. Todas as funções retornam um Observable.

Em app.js

```
var express = require('express');  
var cookieParser = require('cookie-parser');  
var bodyParser = require('body-parser');
```

```
//mongo conn  
require('./db/mongo.connection');
```

**Cria a conexão com o Mongo.
Deve ser feito primeiro!**

```
//router  
var users = require('./routes/users.routes.mongo');
```

Importa as rotas de User, arquivo anterior.

```
//main  
var app = express();
```

```
//configuração  
app.use(bodyParser.json());  
app.use(bodyParser.urlencoded({ extended: false }));  
app.use(cookieParser())  
app.use(function(req, res, next) {  
  res.header("Access-Control-Allow-Origin", "*");  
  res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");  
  res.header("Access-Control-Allow-Methods", "GET, POST, OPTIONS, PUT, DELETE");  
  next();  
});
```

```
//endpoint para users  
app.use('/users', users);  
  
module.exports = app;
```

IMPORTANTE: Aqui é criado a primeira string da URL que será concatenada com cada string do arquivo de rotas. Por exemplo **/user/list** irá chamar o **"/list"** do arquivo de rotas.

Usando um cliente REST

- Podemos testar nossa API com um cliente REST qualquer.
- Vamos usar o ARC, aplicativo do Chrome, mas pode ser qualquer um que você ache melhor.
- Primeiro, vamos iniciar o nossos serviços
 - `./mongod --dbpath ../datadir` (inicia o Mongo)
 - `npm start` (inicia o servidor Express)

ARC list

ARC

HTTP request

Socket

History

Today

PUT

http://localhost:3000/users/update/5cdd...

GET

http://localhost:3000/users/list

DELETE

http://localhost:3000/users/delete/5...

DELETE

http://localhost:3000/users/delete/5...

Saved

PUT

UPDATE BY ID

GET

RETRIEVE BY LOGIN

GET

RETRIEVE BY ID

POST

REGISTER - Express

GET

LIST - Express

Projects

Request

MethodGETRequest URLhttp://localhost:3000/users/listSEND

Parameters

Headers

Variables

Toggle source mode

Insert headers set

Header namecontent-typeHeader valueapplication/json

X

ADD HEADER

Headers are valid

Headers size: 30 bytes

ARC retrieve by id

Request

Method

GET

Request URL

http://localhost:3000/users/retrieve/5cdd96ecfbce0c0f001207bf

SEND

Parameters

Headers

Variables

ARC register

Request



Method

POST

Request URL

http://localhost:3000/users/register



SEND

Parameters ^

Headers

Body

Variables

Body content type

application/json



Editor view

Raw input



FORMAT JSON

MINIFY JSON

```
{
  "firstName": "Jefferson",
  "lastName": "de Carvalho Silva",
  "login": "jeff",
  "email": "jeff@gmail.com",
  "zipcode": "60411205",
  "password": "456"
}
```

ARC retrieve by login

Request



Method

Request URL

GET



http://localhost:3000/users/retrieve/login/tom



SEND



Parameters ^

ARC update by id

Method

PUT

Request URL

http://localhost:3000/users/update/5cdd72edd80f6331df02850a

SEND

Parameters ^

Headers

Body

Variables

Body content type

application/json

Editor view

Raw input

FORMAT JSON MINIFY JSON

```
{
  "firstName": "Jefferson",
  "lastName": "SILVA",
  "login": "jeff",
  "email": "jeffERSON@gmail.com",
  "zipcode": "60411205",
  "password": "123456"
}
```

ARC delete

Method

DELETE



Request URL

http://localhost:3000/users/delete/5cdd96ecfbce0c0f001207bf



SEND

Parameters ^