

# Projeto de Interfaces WEB

## Express-Mongoose-Relacionamentos Aula 13

# Introdução

- O objetivo desse mini-projeto é:
  - Criar um servidor simples em Express;
  - Criar um Schema para User, Product, Cart e Address usando o Mongoose
  - Verificar e testar os relacionamentos:
    - one-to-many
    - many-to-many

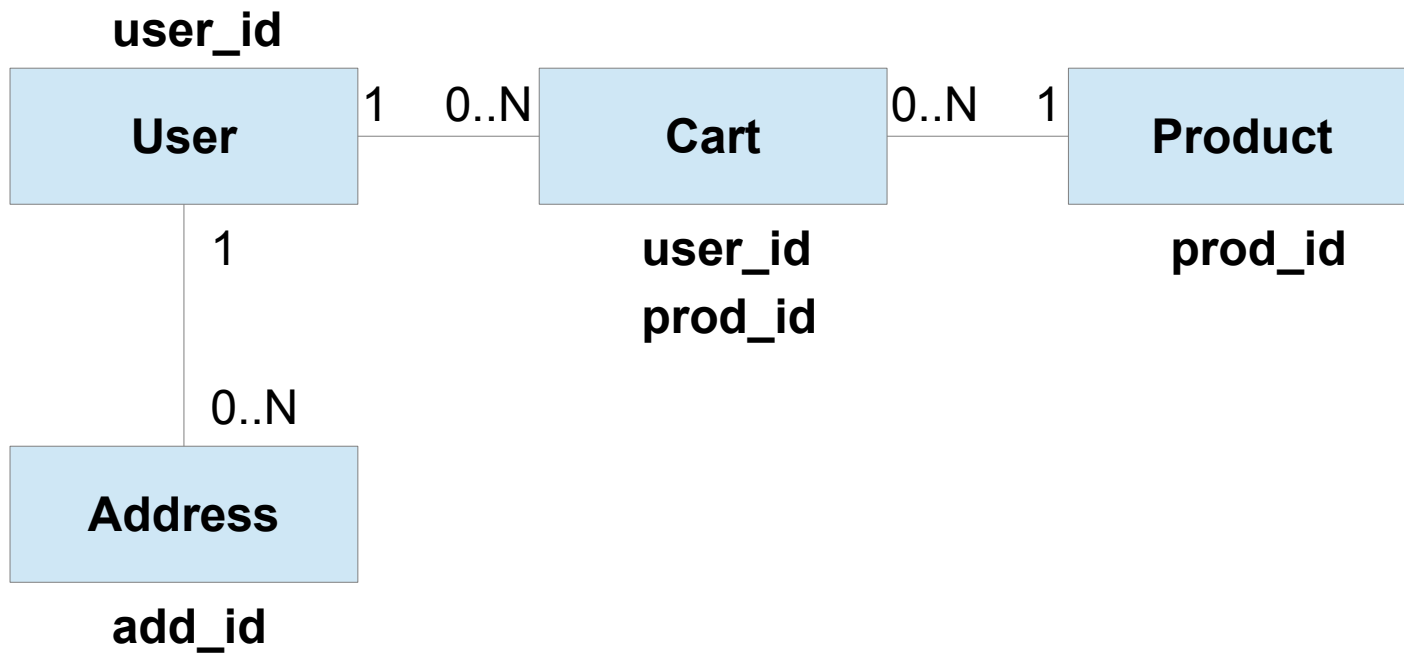
# Criando o Projeto

- 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**.

# Relacionamentos



# One-to-Many

- Relacionamento um-para-muitos (1-n) entre User e Address.
- Um usuário pode ter 0 ou mais endereços.
- No caso do objeto User, ele terá uma lista de Address.


# address.model.js

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

```
//criando o schema, o qual servirá para criar o modelo (collections)
```

```
var AdressSchema = mongoose.Schema(  
  {  
    street: {type:String, required:true, max:20},  
    number: {type:Number, required:true},  
  }  
);
```

O nome 'ADDRESS' servirá para referenciar o modelo em outro Schema.



```
//criando o modelo a partir do schema acima, o qual servirá para incluir as instâncias (documentos)
```

```
var AdressModel = mongoose.model('ADDRESS', AdressSchema);
```

```
//retornando o modelo a ser usado pelo serviço (CRUD).
```

```
module.exports = AdressModel;
```

# 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},  
    login: {type:String, required:true, max:100},  
    password: {type:String, required:true, max:20},  
    addresses: [{type:mongoose.Schema.Types.ObjectId,ref:'ADDRESS'}]  
  }  
);
```

Referenciando o 'ADDRESS'.

UserSchema terá uma propriedade chamada 'addresses' que é uma lista cujos elementos são ObjectId. Cada ObjectId referencia um objeto 'ADDRESS' que, como vimos no slide anterior, é na verdade um Schema para Address.

```
//criando o modelo a partir do schema acima, o qual servirá para incluir as instâncias (documentos)
```

```
var UserModel = mongoose.model('USER', UserSchema);
```

```
//retornando o modelo a ser usado pelo serviço (CRUD).
```

```
module.exports = UserModel;
```



# user.service.mongo.js

```
class UserService{  
  //...SUPRIMIDO  
  
  //retorna um vetor de users  
  static list(req,res){  
    UserModel.find()  
    .populate('addresses')  
    .then(  
      (users)=>{  
        res.status(201).json(users);  
      }  
    ).catch(  
      (error)=>{  
        res.status(500).json(error);  
      }  
    );  
  }  
  //...SUPRIMIDO  
}
```

O **.populate** é opcional. Caso o desenvolvedor não o coloque, o método find irá retornar os usuários com uma lista de endereços, onde cada endereço é na verdade um ObjectId (usuário leve).

Caso o desenvolvedor use o **.populate**, ele deverá passar o nome da propriedade a qual será “populada”. No caso, ‘addresses’. Ao ver isso, o Mongoose irá então substituir cada ObjectId por um objeto endereço (usuário pesado).

# Many-to-Many

- Relacionamento de muitos para muitos (N-N).
- Um usuário pode ter vários produtos. E um mesmo produto (não o produto físico individual) pode pertencer a vários usuários.
- Quando temos esse tipo de relacionamento, podemos criar uma tabela intermediária. No nosso caso, iremos chamá-la de “cart”, ou carrinho, o qual simula um carrinho de compras.
- **OBS.:** Uma outra abordagem, é não criar a tabela intermediária mas criar um vetor de cada tipo em cada schema, assim como no relacionamento 1-n.

# product.model.js

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

```
//criando o schema, o qual servirá para criar o modelo (collections)
```

```
var ProductSchema = mongoose.Schema(  
  {  
    name: {type:String, required:true, max:20},  
    price: {type:Number, required:true},  
    qty: {type:Number, required:true}  
  }  
);
```

```
//criando o modelo a partir do schema acima, o qual servirá para incluir as instâncias (documentos)
```

```
var ProductModel = mongoose.model('PRODUCT', ProductSchema);
```

```
//retornando o modelo a ser usado pelo serviço (CRUD).
```

```
module.exports = ProductModel;
```

```
const ProductModel =
require('../models/product.model');

class ProductService{

  //retorna um objeto que representa um Product
  static register(req,res){
    ProductModel.create(req.body).then(
      (prd)=>{
        res.status(201).json(prd);
      }
    ).catch(
      (error)=>{
        res.status(500).json(error);
      }
    );
  }
}
```

```
//retorna um vetor de Products
static list(req,res){
  ProductModel.find().then(
    (products)=>{
      res.status(201).json(products);
    }
  ).catch(
    (error)=>{
      res.status(500).json(error);
    }
  );
}

module.exports = ProductService;
```

# cart.model.js

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

```
//criando o schema, o qual servirá para criar o modelo (collections)
```

```
var CartSchema = mongoose.Schema({
```

```
  {
    user:{
      type: mongoose.Schema.ObjectId,
      ref: 'USER',
      required: true
```

Perceba que tanto “user” como “product” são do tipo ObjectId e ambos referenciam ‘USER’ (schema User) e ‘PRODUCT’ (schema Product).

```
  },
  product:{
    type: mongoose.Schema.ObjectId,
    ref: 'PRODUCT',
    required: true
```

```
  }
});
```

```
//criando o modelo a partir do schema acima, o qual servirá para incluir as instâncias (documentos)
```

```
var CartModel = mongoose.model('CART', CartSchema);
```

```
//retornando o modelo a ser usado pelo serviço (CRUD).
```

```
module.exports = CartModel;
```

# cart.service.mongo.js

```
const CartModel = require('../models/cart.model');
```

```
class CartService{
```

```
  //retorna um objeto que representa um Cart
```

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

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

```
      (cart)=>{
```

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

```
      }
```

```
    ).catch(
```

```
      (error)=>{
```

```
        res.status(500).json(error);
```

```
      }
```

```
    );
```

```
  }
```

# cart.service.mongo.js

```
//retorna um vetor de Carts
static list(req,res){
  CartModel.find()
    .populate('user') //nome da propriedade em cart
    .populate('product') //nome da propriedade em cart
    .then(
      (carts)=>{
        res.status(201).json(carts);
      }
    ).catch(
      (error)=>{
        res.status(500).json(error);
      }
    );
}
```

```
module.exports = CartService
```

Mesma ideia dos addresses. No entanto, eu escolho popular tanto user, quando product. Assim, um objeto do tipo cart será listado com esses dois objetos dentro.

# cart.routes.mongo.js

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

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

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

module.exports = router;
```



# Abordagem sem cart

*user.model.js*

```
var UserSchema = mongoose.Schema(  
  {  
    firstName: {type:String, required:true, max:100},  
    login: {type:String, required:true, max:100},  
    password: {type:String, required:true, max:20},  
    addresses: [{type:mongoose.Schema.Types.ObjectId,ref:'ADDRESS'}],  
    products: [{type:mongoose.Schema.Types.ObjectId,ref:'PRODUCT'}]  
  }  
);
```

*product.model.js*

```
var ProductSchema = mongoose.Schema(  
  {  
    name: {type:String, required:true, max:20},  
    price: {type:Number, required:true},  
    qty: {type:Number, required:true},  
    users: [{type:mongoose.Schema.Types.ObjectId,ref:'USER'}]  
  }  
);
```

# Abordagem sem cart

*user.service.mongo.js*

```
static list(req,res){
  UserModel.find()
    .populate('products')
    .then(
      (users)=>{
        res.status(201).json(users);
      }
    ).catch(
      (error)=>{
        res.status(500).json(error);
      }
    );
}
```

*product.service.mongo.js*

```
static list(req,res){
  ProductModel.find()
    .populate('users')
    .then(
      (products)=>{
        res.status(201).json(products);
      }
    ).catch(
      (error)=>{
        res.status(500).json(error);
      }
    );
}
```

# Testando com o ARC (1-n)

- Adicione dois endereços (address):

|   |  |
|---|--|
| Method  | Request URL                              |
| POST ▼  | http://localhost:3000/addresses/register |
| Parameters ^  |  |
| Headers   |  |
| Body  |  |
| Body content type   | Editor view                              |
| application/json ▼  | Raw input                                |
| FORMAT JSON MINIFY JSON   |  |
| <pre>{   "street": "Rua Orlando Pinho",   "number": 677 }</pre> |  |

|  |  |
|--|--|
| Method   | Request URL                              |
| POST ▼   | http://localhost:3000/addresses/register |
| Parameters ^   |  |
| Headers  |  |
| Body   |  |
| Body content type  | Editor view                              |
| application/json ▼   | Raw input                                |
| FORMAT JSON MINIFY JSON  |  |
| <pre>{   "street": "Rua Mario Mamede",   "number": 333 }</pre> |  |

# Testando com o ARC (1-n)

- Adicione um usuário (user)

Method POST Request URL http://localhost:3000/users/register

Parameters ^

Headers Body

Body content type application/json Editor view Raw input

FORMAT JSON MINIFY JSON

```
{
  "firstName": "Fábio",
  "login": "fabio",
  "password": "123456Ty"
}
```

201 Created 8.40 ms



```
{
  "addresses": [Array[0]],
  "_id": "5cfa4cbeb8c26a19969a9466",
  "firstName": "Fábio",
  "login": "fabio",
  "password": "123456Ty",
  "__v": 0
}
```

Note que o usuário foi criado com uma lista vazia de addresses.

# Testando com o ARC (1-n)

- Atualize o usuário anterior para “linkar” os dois endereços a ele:

The screenshot displays a REST client interface with the following details:

- Method:** PUT
- Request URL:** `http://localhost:3000/users/update/5cfa4cbeb8c26a19969a9466`. An arrow points to the ID `5cfa4cbeb8c26a19969a9466` with the label "id do usuário inserido anteriormente".
- Parameters:** None.
- Body:** The body content type is `application/json` and the editor view is `Raw input`. The JSON body is:

```
{
  "addresses" : ["5cfa4c92b8c26a19969a9464", "5cfa4caab8c26a19969a9465"],
  "firstName" : "Fábio",
  "login" : "fabio",
  "password" : "123456Ty"
}
```

Arrows point to the two address IDs in the array with the label "ids do endereços inserido anteriormente".
- Response:** The status is `201 Created` with a response time of `19.00 ms`. The response body is:

```
{
  "-addresses": [Array[2]
    0: "5cfa4c92b8c26a19969a9464",
    1: "5cfa4caab8c26a19969a9465"
  ],
  "_id": "5cfa4cbeb8c26a19969a9466",
  "firstName": "Fábio",
  "login": "fabio",
  "password": "123456Ty",
  "__v": 0
}
```

# Testando com o ARC (n-n)

- Adicione dois produtos:

Method POST Request URL `http://localhost:3000/products/register`

Parameters ^

Headers Body

Body content type `application/json` Editor view `Raw input`

FORMAT JSON MINIFY JSON

```
{
  "name": "Shoes One",
  "price": 300,
  "qty": 5
}
```

Method POST Request URL `http://localhost:3000/products/register`

Parameters ^

Headers Body

Body content type `application/json` Editor view `Raw input`

FORMAT JSON MINIFY JSON

```
{
  "name": "Shoes Two",
  "price": 200.89,
  "qty": 5
}
```

# Testando com o ARC (n-n)

- Adicione uma entrada para cada produto e o mesmo usuário em cart.

Method POST Request URL http://localhost:3000/carts/register

Parameters ^

Headers Body

Body content type application/json Editor view Raw input

FORMAT JSON MINIFY JSON

```
{
  "user": "5cfa4cbeb8c26a19969a9466",
  "product": "5cf958ba7c2cf03b3a5c4f76"
}
```

Method POST Request URL http://localhost:3000/carts/register

Parameters ^

Headers Body

Body content type application/json Editor view Raw input

FORMAT JSON MINIFY JSON

```
{
  "user": "5cfa4cbeb8c26a19969a9466",
  "product": "5cf958ba7c2cf03b3a5c4f75"
}
```

id do usuário

ids de produtos diferentes

# Testando com o ARC (n-n)

- List o conteúdo do carrinho:
  - GET : `http://localhost:3000/carts/list`

```
-0: {
  "_id": "5cf959027c2cf03b3a5c4f77",
  "user": {
    "addresses": [Array[2]]
    0: "5cf954272672353a5f086e9c",
    1: "5cf9543c2672353a5f086e9d"
  ],
  "_id": "5cf953f12672353a5f086e9b",
  "firstName": "Fábio",
  "login": "fabio",
  "password": "123456Ty",
  "__v": 0
},
-"product": {
  "_id": "5cf958ba7c2cf03b3a5c4f75",
  "name": "Shoes One",
  "price": 200.89,
  "qty": 5,
  "__v": 0
},
  "__v": 0
},
```

Note que o Mongoose “populou” user e product, ou seja, o Mongoose substitui os ids pelos objetos reais. Isso é possível graças ao **.populate**.