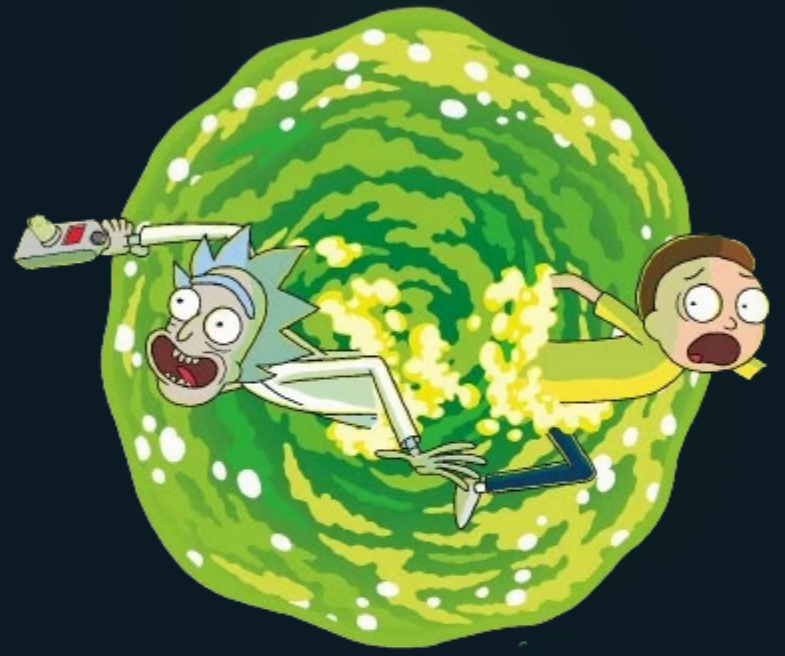
Guia Interdimensional para Construir seu Backend com Node.js e MongoDB

Bem-vindo, recruta da Cidadela! Sua missão é construir o motor por trás do e-commerce mais caótico do multiverso. Este guia é sua Pistola de Portais para navegar pelo desenvolvimento backend usando Node.js, Express e MongoDB.



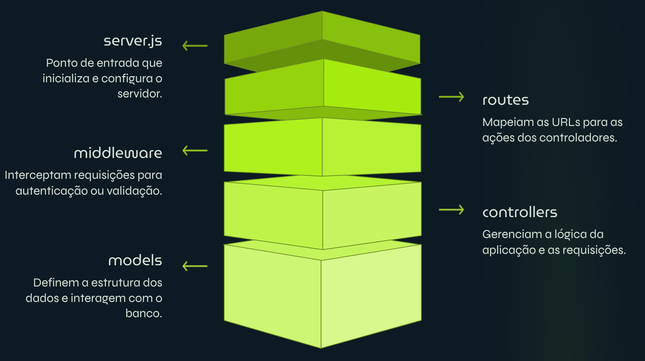
Fase 0: Preparando a Pistola de Portais

Antes de viajar por dimensões, precisamos montar nosso equipamento.

1. Estrutura de Pastas

Uma boa organização evita que seu projeto se transforme em um "Cronenberg". Crie a seguinte estrutura:

```
/
|- controllers/ // A lógica que controla o que acontece em cada rota.
|- models/ // Os "Schemes" ou plantas do nosso banco de dados.
|- routes/ // As URLs da nossa API.
|- middleware/ // Guardas que verificam permissões antes de acessar uma rota.
|- .env // Nossas variáveis secretas.
|- server.js // O ponto de partida do nosso universo.
```



2. Instalação das Dependências

Abra o terminal e instale os pacotes necessários:

```
npm install express mongoose bcryptjs jsonwebtoken cors dotenv joi
```

express
Nosso framework para criar o servidor.

mongoose
Para modelar e conversar com nosso banco de dados MongoDB de forma estruturada.

bcryptjs
Para transformar senhas em códigos indecifráveis (hashing).

jsonwebtoken (JWT)
Para criar passes de acesso interdimensionais (tokens).

cors
Para permitir que nosso front-end converse com o backend.

dotenv
Para carregar nossas variáveis secretas do arquivo .env.

joi
Para validar os dados que chegam, garantindo que ninguém envie um Meeseeks onde deveria haver um e-mail.

3. Opções de Banco de Dados: Local vs. Atlas (Nuvem)

Você tem duas opções para seu laboratório de dados.

MongoDB Local

É como construir um laboratório na sua própria garagem. Você instala o MongoDB Community Server diretamente na sua máquina. É rápido para desenvolvimento, mas requer que você gerencie tudo.

MongoDB Atlas (Recomendado)

É como usar um dos laboratórios secretos do Rick espalhados pelo multiverso. O Atlas é a versão em nuvem do MongoDB. É a opção recomendada para este projeto.

Vantagens: Não precisa instalar nada na sua máquina; Possui um nível gratuito perfeito para projetos de desenvolvimento; Seus dados ficam seguros na nuvem, acessíveis de qualquer dimensão (ou computador).

Passos Resumidos para Configurar o Atlas:

1. Crie uma conta gratuita no site do MongoDB Atlas.
2. Crie um novo "Cluster" no plano gratuito (Mo). Escolha um provedor de nuvem e uma região próximos a você.
3. Segurança de Rede: Vá para Network Access e adicione seu endereço IP atual. Para simplificar durante o desenvolvimento, você pode clicar em "Allow Access from Anywhere" (0.0.0.0/0), mas lembre-se que isso não é seguro para produção!
4. Usuário do Banco: Vá para Database Access e crie um novo usuário e senha. Anote-os bem!
5. Obtenha a Connection String: Volte para a visão geral do seu Cluster, clique em "Connect", escolha "Drivers", e copie a string de conexão fornecida. Ela será parecida com mongodb+srv://:@cluster....

4. Conexão com o MongoDB (server.js)

Crie o arquivo server.js e adicione o código para se conectar ao seu banco de dados.

server.js

```
require('dotenv').config();
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');

const app = express();
app.use(cors());
app.use(express.json());

// Conexão com o MongoDB
const MONGO_URI = process.env.MONGO_URI || 'mongodb://localhost:27017/ricknmorty-shop';

mongoose.connect(MONGO_URI)
.then(() => console.log('Conectado ao MongoDB... Wubba Lubba Dub Dub!'))
.catch(err => console.error('Não foi possível conectar ao MongoDB...', err));

// Rotas (vamos adicionar depois)
// app.use('/api/users', require('./routes/userRoutes'));
// app.use('/api/auth', require('./routes/authRoutes'));
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Servidor rodando na dimensão ${PORT}`));
```

Não se esqueça de criar um arquivo .env na raiz e adicionar sua string de conexão que você copiou do MongoDB Atlas. Lembre-se de substituir pela senha que você criou.

```
MONGO_URI=mongodb+srv://seu-usuario:sua-senha@cluster...
```

Controle de Identidades (Usuários e Autenticação)

Vamos começar criando a base para nossos usuários.

1. Modelagem do Usuário (models/User.js)

Aqui definimos como um "Usuário" será salvo no banco. Adicionaremos um campo role para o controle de acesso (Rick vs. Morty) e um campo cart que usaremos mais tarde.

models/User.js

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');

const UserSchema = new mongoose.Schema({
name: { type: String, required: true },
email: { type: String, required: true, unique: true },
password: { type: String, required: true },
role: { type: String, enum: ['user', 'admin'], default: 'user' },
cart: [{
product: { type: mongoose.Schema.Types.ObjectId, ref: 'Product' },
quantity: { type: Number, default: 1 }
}]
});

// Hook para fazer o hash da senha ANTES de salvar no banco
UserSchema.pre('save', async function(next) {
if (!this.isModified('password')) {
return next();
}
const salt = await bcrypt.genSalt(10);
this.password = await bcrypt.hash(this.password, salt);
next();
});

module.exports = mongoose.model('User', UserSchema);
```

2. Controlador de Autenticação (controllers/authController.js)

Aqui fica a lógica para registrar e logar usuários.

controllers/authController.js

```
const User = require('../models/User');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');

// EXEMPLO: (CREATE) POST /users/register
exports.register = async (req, res) => {
    try {
        const { name, email, password } = req.body;
        let user = await User.findOne({ email });
        if (user) {
            return res.status(400).json({ msg: 'Um usuário com este e-mail já existe nesta dimensão.' });
        }
        user = new User({ name, email, password });
        await user.save();
        // 5. Enviar uma resposta de sucesso
        res.status(201).json({ msg: 'Usuário registrado com sucesso!' });
    } catch (err) {
        // 6. Se algo der errado, capturar o erro e enviar uma resposta de erro genérica
        res.status(500).send('Erro no servidor');
    }
};

// SUA VEZ: POST /auth/login
exports.login = async (req, res) => {
    try {
        // DICA: O processo de login é muito parecido com o de registro.
        // 1. Extraia 'email' e 'password' do `req.body`.
        // 2. Encontre o usuário no banco de dados pelo email usando `User.findOne()`.
        // 3. Se o usuário não for encontrado, retorne um erro 400 com uma mensagem genérica de "Credenciais inválidas.".
        // 4. Se o usuário for encontrado, compare a senha enviada com a senha hashada no banco usando `bcrypt.compare()`.
        // 5. Se as senhas não baterem, retorne o mesmo erro 400 de "Credenciais inválidas.". Não seja específico sobre o que está errado (email ou senha) por segurança.
        // 6. Se as senhas baterem, crie o 'payload' para o token JWT. Ele deve conter informações úteis, como o ID e o 'role' do usuário.
        //    Ex: const payload = { user: { id: user.id, role: user.role } };
        // 7. Gere o token usando `jwt.sign()`, passando o payload, sua chave secreta (do .env) e uma data de expiração.
        // 8. Envie o token de volta para o cliente em formato JSON.

    } catch (err) {
        res.status(500).send('Erro no servidor');
    }
};
```

3. Rotas (routes/authRoutes.js)

Crie as rotas para expor os controladores.

routes/authRoutes.js

```
const express = require('express');
const router = express.Router();
const { register, login } = require('../controllers/authController');

router.post('/register', register);
router.post('/login', login);
module.exports = router;
```

O Catálogo do Multiverso (Produtos)

Agora, vamos criar o CRUD para os produtos.

1. Modelagem do Produto (models/Product.js)

models/Product.js

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

const ProductSchema = new mongoose.Schema({
name: { type: String, required: true },
description: { type: String, required: true },
price: { type: Number, required: true },
category: { type: String, required: true },
stock: { type: Number, required: true, default: 0 },
imageUrl: { type: String, required: true }
});

module.exports = mongoose.model('Product', ProductSchema);
```

2. Middlewares de Proteção (middleware/auth.js e middleware/admin.js)

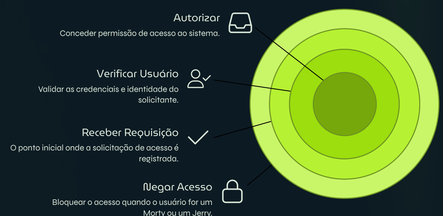
Precisamos de guardas para nossas rotas. Um para verificar se o usuário está logado e outro para verificar se ele é um Rick (admin).

middleware/auth.js

```
const jwt = require('jsonwebtoken');

module.exports = function(req, res, next) {
const token = req.header('x-auth-token'); // ou req.header('Authorization').split(' ')[1];
if (!token) {
return res.status(401).json({ msg: 'Sem token, autorização negada.' });
}
try {
const decoded = jwt.verify(token, process.env.JWT_SECRET);
req.user = decoded.user;
next();
} catch (err) {
res.status(401).json({ msg: 'Token não é válido.' });
}
};
```

middleware/admin.js

```
module.exports = function(req, res, next) {
if (req.user && req.user.role === 'admin') {
next();
} else {
res.status(403).json({ msg: 'Acesso negado. Nível de acesso de Morty detectado.' });
}
};
```



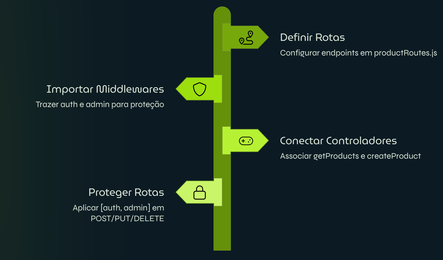
3. Controlador e Rotas de Produtos

A lógica do CRUD de produtos e as rotas que usam os middlewares.

controllers/productController.js

```
const Product = require('../models/Product');

// EXEMPLO: (CREATE) POST /products - Apenas para admins
exports.createProduct = async (req, res) => {
  try {
    // 1. Como está é uma rota de admin, não precisamos verificar quem está criando.
    // O middleware `admin` já fez essa verificação para nós.
    // 2. Crie uma nova instância do modelo Product com os dados do `req.body`.
    const newProduct = new Product(req.body);

    // 3. Salve o novo produto no banco de dados.
    const product = await newProduct.save();

    // 4. Retorne o produto criado com status 201 (Created).
    res.status(201).json(product);
  } catch (err) {
    res.status(500).send('Erro no servidor');
  }
};

// SUA VEZ: (READ) GET /products - Aberto para todos
exports.getProducts = async (req, res) => {
  try {
    // DICA: Esta é uma rota pública para listar produtos.
    // 1. Você pode opcionalmente verificar `req.query` por filtros, como `category`.
    //    Ex: const { category } = req.query;
    // 2. Crie um objeto de filtro. Se houver uma categoria, o filtro será `{ category: category }`. Se não, será um objeto vazio `{}`.
    // 3. Use `Product.find(filterObject)` para buscar os produtos no banco.
    // 4. Retorne a lista de produtos encontrados em formato JSON.
  } catch (err) {
    res.status(500).send('Erro no servidor');
  }
};
// Adicione aqui as outras funções do CRUD para produtos.
// exports.getProductById = async (req, res) => { ... }
// exports.updateProduct = async (req, res) => { ... }
// exports.deleteProduct = async (req, res) => { ... }
```

Rotas de Produtos



routes/productRoutes.js

```
const express = require('express');
const router = express.Router();
const auth = require('../middleware/auth');
const admin = require('../middleware/admin');
const { getProducts, createProduct /*, ...outras*/ } = require('../controllers/productController');

router.get('/', getProducts);
router.post('/', [auth, admin], createProduct);
// ... outras rotas PUT e DELETE protegidas por [auth, admin]
// ... rota GET /:id

module.exports = router;
```

Carrinho Quântico e Pedidos

A lógica do carrinho e dos pedidos é onde a mágica do NoSQL brilha.

1. Controlador do Carrinho (controllers/cartController.js)

Lembre-se que o carrinho é um campo dentro do modelo User.

controllers/cartController.js

```
const User = require('../models/User');

// EXEMPLO: (CREATE/UPDATE) POST /cart/add
exports.addToCart = async (req, res) => {
  // 1. Extraia o ID do produto e a quantidade do corpo da requisição.
  const { productId, quantity } = req.body;
  try {
    // 2. Encontre o usuário logado usando o ID que o middleware `auth` adicionou em `req.user.id`.
    const user = await User.findById(req.user.id);

    // 3. Verifique se o produto já existe no carrinho do usuário.
    const itemIndex = user.cart.findIndex(p => p.product == productId);

    if (itemIndex > -1) {
      // 4a. Se o item já existe, apenas some a nova quantidade.
      user.cart[itemIndex].quantity += quantity;
    } else {
      // 4b. Se não existe, adicione o novo produto e quantidade ao array do carrinho.
      user.cart.push({ product: productId, quantity });
    }

    // 5. Salve as alterações no documento do usuário.
    await user.save();
    res.json(user.cart);
  } catch (err) {
    res.status(500).send('Erro no servidor');
  }
};

// SUA VEZ: (READ) GET /cart
exports.getCart = async (req, res) => {
  try {
    // DICA: Esta rota deve retornar os itens do carrinho do usuário logado com todos os detalhes dos produtos.
    // 1. Encontre o usuário pelo ID vindo de `req.user.id`.
    // 2. Para obter os detalhes completos de cada produto (nome, preço, etc.) em vez de apenas o ID,
    //    use o método `.populate()` do Mongoose na sua busca.
    //    Ex: await User.findById(req.user.id).populate('cart.product');
    // 3. Retorne o `user.cart` populado como resposta.
  } catch (err) {
    res.status(500).send('Erro no servidor');
  }
};

// Adicione aqui as outras funções do CRUD para o carrinho:
// exports.updateCartItem = async (req, res) => { ... }
// exports.removeCartItem = async (req, res) => { ... }
```

2. Modelagem e Controlador de Pedidos

A melhor prática aqui é embutir (embed) os dados do produto no pedido. Por quê? Se você atualizar o preço de um Plumbus amanhã, o pedido antigo do usuário deve manter o preço da data da compra. Se apenas referenciarmos, o preço mudaria em todo o histórico!

models/Order.js

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

const OrderSchema = new mongoose.Schema({
user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
products: [{
productId: { type: String, required: true },
name: { type: String, required: true },
price: { type: Number, required: true },
quantity: { type: Number, required: true }
}],
total: { type: Number, required: true },
status: { type: String, default: 'Processando' },
createdAt: { type: Date, default: Date.now }
});

module.exports = mongoose.model('Order', OrderSchema);
```

controllers/orderController.js

```
const Order = require('../models/Order');
const User = require('../models/User');
const Product = require('../models/Product');

// EXEMPLO: (CREATE) POST /orders
exports.createOrder = async (req, res) => {
  try {
    // 1. Busque o usuário e popule os produtos do carrinho para ter acesso aos detalhes (preço, estoque, etc.).
    const user = await User.findById(req.user.id).populate('cart.product');
    if (user.cart.length === 0) {
      return res.status(400).json({ msg: 'Seu carrinho está vazio, seu Morty!' });
    }

    let total = 0;
    // 2. Crie um array com os produtos formatados para o pedido, copiando os dados, e calcule o total.
    const orderProducts = user.cart.map(item => {
      // DICA para o aluno: Aqui é um bom lugar para verificar o estoque (`item.product.stock`).
      total += item.quantity * item.product.price;
      return {
        productId: item.product.id,
        name: item.product.name,
        price: item.product.price,
        quantity: item.quantity
      };
    });

    // 3. Crie uma nova instância do modelo Order.
    const order = new Order({
      user: req.user.id,
      products: orderProducts,
      total
    });

    // 4. Salve o novo pedido.
    await order.save();

    // 5. Limpe o carrinho do usuário e salve a alteração.
    user.cart = [];
    await user.save();
    // DICA para o aluno: Aqui você também deve decrementar o estoque dos produtos vendidos.

    res.status(201).json(order);
  } catch (err) {
    res.status(500).send('Erro no servidor');
  }
};

// SUA VEZ: (READ) GET /orders
// Crie a função para buscar todos os pedidos do usuário logado.
// exports.getUserOrders = async (req, res) => { ... }
```

As Regras do Rick (Requisitos Não Funcionais)

Validação de Dados com Joi

Crie um middleware para validar os dados antes que eles cheguem ao controlador.

```
// middleware/validation.js
const Joi = require('joi');

const registerSchema = Joi.object({
name: Joi.string().min(3).required(),
email: Joi.string().email().required(),
password: Joi.string().min(6).required()
});

exports.validateRegister = (req, res, next) => {
const { error } = registerSchema.validate(req.body);
if (error) {
return res.status(400).json({ msg: error.details[0].message });
}
next();
};

// Use na sua rota: router.post('/register', validateRegister, register);
```

Este guia cobre a estrutura e a lógica principal para todos os requisitos.

Agora, é hora de codificar e, lembre-se, não tenha medo de quebrar as coisas. É assim que aprendemos em todas as dimensões! Get Schwifty!