# **O que é a CommonJS?**

Neste capítulo testamos uma maneira de organizar as funcionalidades da nossa aplicação em módulos separados. E utilizamos a convenção CommonJS para isso. Mas afinal o que é a CommonJS? Explique aqui o que você entendeu dela. E qual o objeto do Node que utilizamos para guardar nosso novo módulo que poderá ser invocado futuramente?

## **Resposta do instrutor**

CommonJS é uma convenção para carregamentos de módulos javascript em aplicações server-side. O objeto que o node disponibiliza é o module e a função é passada para o atributo exports desse objeto.

# **Isolando as configurações do express**

Implemente o arquivo express.js isolando as configurações do módulo do express e passe a agora a carregar esse novo arquivo no app.js. Lembre de deixar dentro da função que vai para o module.exports somente o código que realmente precisa ser carregado a cada chamada dessa função. O que não precisa ficar sendo carregado o tempo inteiro deve ficar fora da função.

Lembre de criar esse arquivo numa pasta específica para arquivos de configuração. Assim o código fica bem isolado, inclusive fisicamente. =D

## **Resposta do instrutor**

O arquivo express.js deve ficar algo assim:

var app = require('express')();  
app.set('view engine', 'ejs');  
  
module.exports = function() {  
 return app;  
};

E o app.js passa a ficar assim:

var app = require('./config/express’)();  
  
app.get('/produtos', function(req, res){  
 res.render('produtos/lista');  
});  
  
  
app.listen(3000, function(){  
 console.log("Servidor rodando");  
});

# **O que são as rotas?**

Neste capítulo, separamos as configurações que atendem cada requisição em arquivos específicos e configuramos rotas para atender a url “/produtos” por exemplo. O que são as rotas? Explique o que você entendeu delas.

## **Resposta do instrutor**

Rotas são as configurações que criamos para cada url específica que queremos atender no nosso sistema server-side implementado em javascript.

# **Isolando as rotas**

Implementamos um arquivo específico para as configurações das rotas referentes aos produtos. Fizemos também uma reorganização da estrutura de diretórios do projeto para separar arquivos de código de acordo com sua funcionalidade.

Implemente agora o arquivo produtos.js para configurar as rotas dos produtos, dentro dessa nova estrutura de diretórios, que você precisará criar também. Lembre que o arquivo precisa conhecer a variável que guarda o objeto do express.

Você também precisa ensinar o express onde está a pasta views, já que agora ela ficará em um diretório diferente do default utilizado pelo ejs, essa configuração deve ser feita no arquivo específico que criamos para as configurações do express.

Por fim altere o arquivo app.js para que passe a carregar este novo arquivo criado também como um módulo.

## **Resposta do instrutor**

O arquivo produtos.js deve ficar dentro da pasta casadocodigo/app/routes/ e assim:

module.exports = function(app) {  
 app.get("/produtos",function(req, res) {  
 res.render('produtos/lista');  
 });  
}

O express.js que agora está dentro de casadocodigo/config/ deve ter a nova configuração informando o local das views:

var app = require('express')();  
app.set('view engine', 'ejs');  
app.set('views','./app/views');  
  
module.exports = function() {  
 return app;  
};

E o arquivo app.js fica assim:

var app = require('./config/express')();  
var rotasProdutos = require('./app/routes/produtos')(app);  
  
app.listen(3000, function(){  
 console.log("Servidor rodando");  
});

# **Preparando o Banco de Dados para a aplicação.**

No curso, utilizamos um banco MySQL para armazenar os dados da aplicação. Para que você consiga reproduzir esse funcionamento, instale também o MySQL na sua máquina e já crie o banco de dados e a tabela que iremos utilizar.

Você pode baixar o instalador a partir do próprio site do MySQL e selecionar a versão correspondente ao seu Sistema Operacional: <https://dev.mysql.com/downloads/mysql/>

Ou se preferir utilize algum package manager, de acordo com sua preferência, como o apt-get no Linux, ou o homebrew no Mac.

Após instalado, pode ser que você precise iniciar o banco. Esse passo varia para cada sistema operacional. No Mac, por exemplo, você pode ir em configurações do sistema, seleciona o ícone do MySQL e clicar no botão que inicia o serviço.

Uma vez iniciado, acesse o MySQL no terminal e execute os scripts necessários para a criação do banco e da tabela. Siga os passos a seguir

mysql -u root

Quando ele pedir a senha, simplesmente deixe em branco que é a senha default, ou utilize a sua senha de root caso você tenha configurado alguma diferente.

A seguir os scripts de criação do banco e da tabela passo a passo:

create database casadocodigo\_nodejs;  
  
use casadocodigo\_nodejs;  
  
CREATE TABLE livros (  
 id int(11) NOT NULL AUTO\_INCREMENT PRIMARY KEY,  
 titulo varchar(255) DEFAULT NULL,  
 descricao text,  
 preco decimal(10,2) DEFAULT NULL);

Aproveite para já inserir alguns registros também:

insert into livros(titulo, descricao, preco)  
values ('Comecando com nodejs', 'livro introdutório sobre nodejs', 39.90);  
  
insert into livros(titulo, descricao, preco)  
values ('Comecando com javascript', 'livro introdutório sobre javascript', 39.90);  
  
insert into livros(titulo, descricao, preco)  
values ('Comecando com express', 'livro introdutório sobre express', 39.90);

# **Instalando o driver do MySQL e implementando a consulta no Node.**

Agora estamos utilizando um banco de verdade para buscar os dados da aplicação da Casa do Código. Você já tem o banco instalado, configurado e populado por ter completado o exercício anterior. Agora instale o driver do mysql utilizando o npm, lembrando de passar o argumento que salva essa informação no arquivo de metadados da app, e altere a implementação do tratamento da rota “/produtos” dentro do produtos.js para que o código passe a conectar com o banco, faça um select na tabela de livros e envie de volta para a tela o json retornado do banco.

## **Resposta do instrutor**

Para instalar o driver do MySQL deve ter sido rodado o seguinte comando:

npm install mysql --save

Quando é feita a instalação de um módulo via npm, uma nova pasta para esse módulo é criada dentro da pasta node\_modules, portanto seu projeto agora deve ter uma pasta chamada mysql.

Como utilizamos o argumento --save essa informação deve ter ficado registrada no arquivo package.json como uma das dependências do projeto.

O arquivo produtos.js com a nova implementação deve ter ficado assim:

module.exports = function(app) {  
 app.get("/produtos",function(req, res) {  
 var mysql = require('mysql');  
 var connection = mysql.createConnection({  
 host: "localhost",  
 user: "root",  
 password : "",  
 database: "casadocodigo\_nodejs",  
 });  
  
 connection.query('select \* from livros', function(err, results){  
 res.send(results);  
 });  
  
 connection.end();  
  
 });  
}

# **Exibindo na tela os dados do Banco.**

Aprendemos neste capítulo como consultar os dados no banco e exibi-los na tela de maneira dinâmica.

Implemente agora esta funcionalidade adicionando o retorno do select no comando que renderiza a página no arquivo produtos.js.

E depois altere o lista.ejs para que a tabela seja escrita dinamicamente utilizando javascript junto ao html da maneira que o ejs suporta.

## **Resposta do instrutor**

O produtos.js agora deve passar como parâmetro na chamada da função res.render() além do caminho da view a ser exibida, o resultado da consulta em formato json. Atribuímos nesse json o nome “lista” para o resultado a ser enviado, indicado que essa é a lista de livros a ser exibida na página. O código deve ficar mais ou menos assim:

module.exports = function(app) {  
 app.get("/produtos",function(req, res) {  
 var mysql = require('mysql');  
 var connection = mysql.createConnection({  
 host: "localhost",  
 user: "root",  
 password : "",  
 database: "casadocodigo\_nodejs",  
 });  
  
 connection.query('select \* from livros', function(err, results){  
 res.render('produtos/lista', {lista: results});  
 });  
  
 connection.end();  
  
 });  
}

Agora que alteramos também o lista.ejs para exibir a lista dinamicamente inserindo código javascript direto no html, ele deve ficar assim:

<html>  
 <body>  
 <table>  
 <tr>  
 <td>id</td>  
 <td>titulo</td>  
 <td>descricao</td>  
 <td>preco</td>  
 </tr>   
 <% for(var i=0; i<lista.length; i++) {%>  
 <tr>  
 <td><%= lista[i].id %></td>  
 <td><%= lista[i].titulo %></td>  
 <td><%= lista[i].preco %></td>  
 <td><%= lista[i].descricao%></td>  
 </tr>  
 <% } %>  
 </table>  
 </body>  
</html>

# **Isolando a criação da conexão com a ConnectionFactory**

Vimos nesse capítulo que é interessante separar os códigos de infra-estrutura da aplicação dos códigos de negócio. Para isso, criamos um diretório chamado “infra” e lá dentro criamos um arquivo chamado connectionFactory para isolar a criação da conexão com o banco. Faça o mesmo no seu projeto criando a sua própria connectionFactory e alterando o código do produtos.js para que passe a carregar esse novo módulo ao invés de sempre criar uma conexão em cada rota configurada nele.

Compartilhe seu código com a gente. O que você achou dessa forma de criar a conexão? Porque a chamamos de Factory?

## **Resposta do instrutor**

Factory é um padrão de desenvolvimento muito utilizado no mundo Orientado a Objetos quando queremos isolar a criação de um objeto complexo para que os outros objetos que o utilizam passem apenas a pedir a sua criação sem necessariamente saber como ele está sendo construído. Por isso, a classe responsável por fazer essa criação fica conhecida como Fábrica, ou Factory, pois ela tem a função de fabricar o objeto desejado.

O código da nossa connectionFactory fica assim:

var mysql = require('mysql');  
  
module.exports = function() {  
 return mysql.createConnection({  
 host: 'localhost',  
 user: 'root',  
 password: '',  
 database: 'casadocodigo\_nodejs'  
 });  
}

Veja como fica então o produtos.js invocando agora função createConnection() a partir da connectionFactory:

var connectionFactory = require('../infra/connectionFactory');  
  
module.exports = function(app) {  
 app.get("/produtos",function(req, res) {  
  
 var connection = connectionFactory();  
  
 connection.query('select \* from livros', function(err, results){  
 res.render('produtos/lista', {lista: results});  
 });  
  
 connection.end();  
  
 });  
}

# **Configurando o carregamento automático dos módulos.**

Para melhorar o design do código e evitar ter vários ‘requires’ espalhados pela aplicação, utilizamos uma estratégia de fazer carregamento automático das módulos através da lib express-load.

Instale o express-load usando o npm e configure-o para gerenciar os carregamentos dos módulos da sua app também. Faça isso no arquivo específico para configurações do express, informando quais diretórios devem ser carregados na app.

Faça com que o express-load carregue também os módulos criados por você mesmo para suportar a infra da aplicação. Lembre de fazer o wrapper das funções para evitar o carregamento automático. Para verificar que essa estratégia funcionou, crie logs nas funções para ver em que momento elas acontecem de fato.

Após isso, você pode remover os “requires” dos seus arquivos de código, exceto é claro, do express.js.

## **Resposta do instrutor**

Para instalar o express-load:

npm install express-load --save

Agora o express.js deve fazer load de mais um módulo, que é o express-load, justamente para que ele faça o carregamento automático dos outros módulos que vamos adicionando ao projeto. O código deve ficar mais ou menos assim:

var express = require('express');  
var load = require('express-load');  
  
module.exports = function() {  
  
 var app = express();  
 app.set('view engine', 'ejs');  
 app.set('views','./app/views');  
  
 load('routes',{cwd: 'app'})  
 .then('infra')  
 .into(app);  
  
 return app;  
  
}

Lembre que a ordem em que você carrega os módulos importa, pois as rotas precisam dos módulos da infra para funcionar. Além disso, um outro ponto importante é a presença do segundo parâmetro da função load(), onde passamos um json com um atributo cwd. Essa atributo indica que o express-load deve procurar arquivos para receberem os módulos carregados dentro da pasta app e não da raiz do projeto.

Alteramos também o arquivo produtos.js, visto que agora ele não precisa mais carregar o connectionFactory, mas sim passa a receber a connection a partir da variável que foi criada pelo express-load e que tem o nome do diretório raiz que informamos ao express-load. Fica assim:

module.exports = function(app) {  
 app.get("/produtos",function(req, res) {  
  
 var connection = app.infra.connectionFactory();  
  
 connection.query('select \* from livros', function(err, results){  
 res.render('produtos/lista', {lista: results});  
 });  
  
 connection.end();  
  
 });  
}

Agora o app.js não precisa mais fazer require do produtos.js:

var app = require('./config/express')();  
// removida a linha que servia somente para carregar o arquivo de rotas.   
// var rotasProdutos = require('./app/routes/produtos')(app);  
  
  
app.listen(3000, function(){  
 console.log("Servidor rodando");  
});

Por fim, criamos o wrapper na connectionFactory para evitar que o load invoque a função automaticamente sem que a gente chame de fato:

var mysql = require('mysql');  
  
function createDBConnection(){  
 return mysql.createConnection({  
 host: 'localhost',  
 user: 'root',  
 password: '',  
 database: 'casadocodigo\_nodejs'  
 });  
}  
  
module.exports = function() {  
 return createDBConnection;  
}

# **Criando um arquivo para isolar acesso a produtos no banco.**

Neste capítulo criamos um novo arquivo na pasta infra para isolar todas as funcionalidades de acesso ao banco de dados referentes aos produtos. Chamamos esse arquivo de produtosBanco.js e movemos o select para lá.

Crie esse arquivo no seu código também com uma função para fazer o select. Atribua a implementação dessa função a uma variável lista Lembre que a função criada no produtosBanco precisa receber a conexão e a função de callback a ser executada no retorno.

this.lista = function(connection, callback){  
 //código da função  
}

Por fim crie uma outra função que engloba esse trecho de código e que deve ser a função atribuida aomodule.exports.

No final dessa função, deve ser retornado o this para que o objeto inteiro fique visível para quem carregar uma instância dele.

No arquivo produtos.js, para listar os livros, passe a carregar o novo arquivo produtosBanco de maneira semelhante como você já estava carregando a connectionFactory, armazene o retorno dessa chamada numa variável e a partir dessa variável, invoque a função que você criou, passando a connection e a função de callback.

Lembre que agora você tem o objeto em mãos e para invocar uma função dele use o nome do atributo ao qual ela foi atribuida. Nesse caso, lista.

Após as mudanças suba novamente a app e teste se tudo continua funcionando.

## **Resposta do instrutor**

O arquivo produtosBanco.js fica assim:

module.exports = function(){  
 this.lista = function(connection, callback){  
 connection.query('select \* from livros',callback);  
 }  
 return this;  
}

A implementação no produtos.js passa a ficar assim:

module.exports = function(app) {  
 app.get("/produtos",function(req, res) {  
  
 var connection = app.infra.connectionFactory;  
 var produtosBanco = app.infra.produtosBanco;  
  
 produtosBanco.lista(connection, function(err, results){  
 res.render('produtos/lista', {lista: results});  
 });  
  
 connection.end();  
  
 });  
}

# **Passando a connection como argumento para a Classe produtosNoBanco.**

Para evitar passar a connection em todas as funções declaradas dentro de produtosBanco, mudamos a implementação para passar a connection direto na invocação da classe.

Altere a sua classe produtosBanco para que ela passe a funcionar assim também.

Note que agora você precisa criar uma nova função para envelopar a função lista() para que o express-load consiga carregar tudo normalmente.

Essa nova função é quem passará a receber a connection e a função que de fato implementa o select passa a receber somente a função de callback.

No arquivo que invoca essa classe, passe a fazer isso utilizando o new para evitar ter o thisdescontextualizado.

## **Resposta do instrutor**

A classe produtosBanco passa a ficar assim:

module.exports = function(){  
 return function(connection){  
 this.lista = function(callback){  
 connection.query('select \* from livros',callback);  
 }  
 return this;   
 };  
}

E a produtos.js agora deve instanciar cada objeto da classe produtosBanco, utilizando o new, passando a connection no construtor e a função de callback na chamada do lista:

module.exports = function(app) {  
 app.get("/produtos",function(req, res) {  
  
 var connection = app.infra.connectionFactory();  
 var produtosBanco = new app.infra.produtosBanco(connection);  
  
 produtosBanco.lista(function(err, results){  
 res.render('produtos/lista', {lista: results});  
 });  
  
 connection.end();  
  
 });  
}

# **Função produtora e o prototype**

Criamos a função produtora para servir como o construtor da classe e passamos a atribuir as novas funções ao objeto prototype do javascript. Além disso, passamos a guardar a connection recebida no construtor em uma variável \_connection privada, guardando assim o estado e fazendo nosso arquivo ficar mais parecido ainda com uma classe.

Na classe produtosBanco, implemente essa função produtora que recebe a conexão e a guarda numa variável privada. Chame a função produtora de ProdutosBanco(connection) e siga a convenção para nomenclatura de variável privada do javascript para criar o atributo que vai armazenar a connection.

Passe agora a retornar dentro da function passada para o module.exports essa nova função produtora que você acabou de criar.

Por fim, mude a função lista para que ela passe a ser atribuida ao atributo prototype da ProdutosBanco e não mais ao this:

ProdutosBanco.prototype.lista = function(callback){  
 // implementacao da funcao …  
}

Essa estratégia ficou muito parecida com o Pattern DAO, muito conhecido no mundo Orientado a Objetos, portanto mude o nome do arquivo produtosBanco para ProdutosDAO fazendo com que assim ele seja facilmente reconhecido como uma classe que implementa este Pattern.

Já que renomeamos a classe, vamos também refatorar o nome da função produtora para ProdutosDAO e dos arquivos que instanciam essas classe. Faça isso e poste o código final aqui pra gente.

## **Resposta do instrutor**

A classe ProdutosDAO fica assim:

function ProdutosDAO(connection) {  
 this.\_connection = connection;  
}  
  
ProdutosDAO.prototype.lista = function(callback) {  
 this.\_connection.query('select \* from livros',callback);  
}  
  
module.exports = function(){  
 return ProdutosDAO;  
};

E o seu uso na produtos.js agora fica assim:

var produtosDAO = new app.infra.ProdutosDAO(connection);  
  
produtosDAO.lista(function(err, results){  
 res.render('produtos/lista', {lista: results});  
 });