# **O que são testes automatizados?**

[Sugira uma melhoria](https://cursos.alura.com.br/course/node-js/section/7/exercise/1#)

Neste capítulo, vimos o conceito de teste automatizados e implementamos na aplicação.

Para você o que são testes automatizados? Para que eles são importantes? Tente descrever com suas próprias palavras. Se preciso, pesquise mais sobre o assunto ao final do curso. É um conteudo muito interessante e útil!

Testes automatizados são uma forma de escrever código para testar os possíveis cenários que as regras de negócio da aplicação devem cumprir.

Eles são muito importantes porque ajudam a garantir o funcionamento do código e facilitam muito as futuras manutenções, além de orientar o próprio desenvolvimento e acabam servido até como documentação. Para muitos, essa é a melhor forma de documentar o código, na verdade.

## **Resposta do instrutor**

Testes automatizados são uma forma de escrever código para testar os possíveis cenários que as regras de negócio da aplicação devem cumprir.

Eles são muito importantes porque ajudam a garantir o funcionamento do código e facilitam muito as futuras manutenções, além de orientar o próprio desenvolvimento e acabam servido até como documentação. Para muitos, essa é a melhor forma de documentar o código, na verdade.

# **Quando utilizar testes automatizados.**

[Sugira uma melhoria](https://cursos.alura.com.br/course/node-js/section/7/exercise/2#)

Utilizamos os testes automatizados na aplicação para verificar o correto funcionamento do Controller das rotas relacionadas à produtos.

Em que outras situações você imagina que seria útil utilizá-las?

Tente pensar no maior número de caso possíveis faça as devidas implementações! Sua aplicação ficará muito bem documentada e com uma facilidade grande para manutenções!

## **Resposta do instrutor**

Todos os casos de uso e regras de negócio da aplicação merecem testes automatizados. Sejam eles testes unitários (aqueles que verificam cada função do código) ou de integração (aqueles que verificam funcionalidades num fluxo por inteiro).

Pesquise mais sobre esse assunto e implemente em todas as regras de negócio para todos os possíveis cenários que você conseguir imaginar.

# **Instalação do Mocha e infra para testes**

[Sugira uma melhoria](https://cursos.alura.com.br/course/node-js/section/7/exercise/3#)

Para evitar ter que escrever a infra do teste todo na mão usamos o mocha, uma lib para escrever testes automatizados em JavaScript.

Comece então instalando o mocha com ajuda do npm, como já foi feito várias vezes ao longo do curso, lembrando que dessa vez é interessante passar o parâmetro informando que ele deve ser instalado somente para o ambiente de desenvolvimento.

O mocha usa um um padrão de nome de diretório para saber onde encontrar os casos de teste. Crie esse diretório seguindo o padrão, implemente uma primeira função básica somente para testar se tudo está funcionando e rode o teste com Mocha, conforme foi mostrado na aula.

## **Resposta do instrutor**

Para instalar o Mocha com o parâmetro de instalação somente para ambiente de desenvolvimento:

sudo npm install mocha --save-dev

A pasta onde ficarão seus casos de testes deve se chamar test. Você pode criá-la na raiz do projeto. Dentro dela você pode criar um arquivo chamado produtos.js, por exemplo.

Para executar o teste, você deve fazê-informando o caminho completo onde foi instalado o Mocha e executando um Script que também se chama Mocha:

node\_module/mocha/bin/mocha

Se você tiver qualquer dificuldade, abra uma dúvida no fórum!

# **Caso de teste para a listagem de produtos.**

[Sugira uma melhoria](https://cursos.alura.com.br/course/node-js/section/7/exercise/4#)

Implementamos o caso de teste “listagem json” para verificar se a busca de produtos realmente retorna um json e aproveitamos para verificar também o Status Code do retorno.

Implemente esse mesmo teste no seu código, utilizando o Mocha.

Dentro do arquivo que você criou na pasta test, escreva o código com a function describe que recebe como parâmetro a descrição do que o arquivo testa e a função que de fato implementa o teste.

Lembre de utilizar o “#” na frente do nome da descrição para seguir a convenção.

Dentro da function crie um caso de testa que verifica se o Status Code retornado foi realmente 200 e se o Header Content-Type realmente é Json.

Lembre de utilizar também a lib assert para facilitar a implementação dos testes e função de finalização para indicar ao Mocha quando ele deve realmente finalizar a execução dos testes e exibir o relatório com os resultados.

Após a implementação rode o teste e verifique se ele funcionou e também se a aplicação realmente está se comportando conforme esperado.

# **Caso de teste para a listagem de produtos.**

[Sugira uma melhoria](https://cursos.alura.com.br/course/node-js/section/7/exercise/4#)

Implementamos o caso de teste “listagem json” para verificar se a busca de produtos realmente retorna um json e aproveitamos para verificar também o Status Code do retorno.

Implemente esse mesmo teste no seu código, utilizando o Mocha.

Dentro do arquivo que você criou na pasta test, escreva o código com a function describe que recebe como parâmetro a descrição do que o arquivo testa e a função que de fato implementa o teste.

Lembre de utilizar o “#” na frente do nome da descrição para seguir a convenção.

Dentro da function crie um caso de testa que verifica se o Status Code retornado foi realmente 200 e se o Header Content-Type realmente é Json.

Lembre de utilizar também a lib assert para facilitar a implementação dos testes e função de finalização para indicar ao Mocha quando ele deve realmente finalizar a execução dos testes e exibir o relatório com os resultados.

Após a implementação rode o teste e verifique se ele funcionou e também se a aplicação realmente está se comportando conforme esperado.

## **Resposta do instrutor**

Dentro da pasta test, criamos o arquivo produtos.js, que por enquanto tem o seguinte código:

var http = require('http');  
var assert = require('assert');  
  
  
describe('#ProdutosController', function() {  
 it('#listagem de produtos json', function (done) {  
 var configuracoes = {  
 hostname: 'localhost',  
 port: 3000,  
 path: '/produtos',  
 headers: {  
 'Accept':'application/json'  
 }  
 };  
 http.get(configuracoes, function(res){  
 assert.equal(res.statusCode, 200);  
 assert.equal(res.headers['content-type'], 'application/json; charset=utf-8’);  
 done();  
 });  
 });  
});

A função de finalização é necessária por causa do funcionamento assíncrono do Node.js. Caso ela não seja utilizada é possível que o Mocha tente rodar o relatório final antes de realmente ter terminado a execução das funcionalidades testadas.

Para executar o teste, basta rodar o seguinte comando, contando que você esteja na pasta raiz:

node\_modules/mocha/bin/mocha

Como o Mocha sabe que deve procurar os arquivos dentro da pasta “test” nem é necessário informar quais arquivos devem ser executados.

# **Executando o teste sem precisar subir o servidor**

[Sugira uma melhoria](https://cursos.alura.com.br/course/node-js/section/7/exercise/5#)

Para facilitar a escrita dos testes, passamos a utilizar mais um módulo javascript: o **Supertest**, que integra com o Mocha e ajuda a implementar o código do teste, deixando-o mais limpo mais claro.

Além da facilidade na escrita do teste, ganhamos também a possibilidade de executar o teste sem precisar que a aplicação a ser testada esteja rodando, pois o Supertest consegue se integrar facilmente com o Express e assim é possível testar diretamente a rota.

Primeiro instale o módulo do Supertes utilizando o npm e passando a informação de que ele deve ser mantido apenas em ambiente de desenvolvimento.

Altere o seu arquivo de teste para que ele passe a usar também o Supertest. Nesta nova forma de implementação, você deve carregar arquivo de configurações do Express para dentro do arquivo de testes, já invocando a função que ele guarda.

Além disso, você deve agora escrever o caso de teste utilizando o objeto do Supertest, que você também vai carregar, passando o objeto do express como parâmetro. Depois disso é só escrever o código que faz as mesmas validações do Content-Type e Status Code, passando como parâmetro o Accept ‘application/json’ para a rota ‘/produtos’.

Implemente esse código e execute-o para ver se realmente está funcionando!

## **Resposta do instrutor**

O código inteiro do arquivo de testes agora fica assim:

var express = require('../../config/express')()  
var request = require('supertest')(express);  
  
describe('#ProdutosController', function() {  
 it('#listagem de produtos json', function (done) {  
 request.get('/produtos')  
 .set('Accept', 'application/json')  
 .expect('Content-Type', /json/)  
 .expect(200,done)  
  
 });  
});

Para executá-lo é o mesmo comando utilizado no exercício anterior

node\_modules/mocha/bin/mocha

O código fica bem mais enxuto e você nem precisa mais subir o server para executar o teste.

# **Implementando o teste para o cadastro**

[Sugira uma melhoria](https://cursos.alura.com.br/course/node-js/section/7/exercise/6#)

Precisamos também validar a funcionalidade de cadastro de produtos. Para isso escrevemos dois testes: um para testar o cadastro com dados inválidos e outro com dados válidos.

Implemente esses dois novos casos de teste no seu arquivo de testes também. Lembre que agora você continua testando a rota “/produtos”, porém passando o verbo “post”. E para enviar o corpo da requisição, ou seja, o json com os dados do produto, você deve usar a função send() do objeto do supertest.

Após ter implementando, execute o teste e veja se está ok.

## **Resposta do instrutor**

Os dois novos casos de teste são implementados também no arquivo produtos.js (da pasta “test”), também dentro da função describe() e seu código é o seguinte:

it('#cadastro de um novo produto com dados invalidos', function (done) {  
 request.post('/produtos')  
 .send({titulo:"",descricao:"livro de teste"})  
 .expect(400,done)  
  
 });  
  
 it('#cadastro de um novo produto com tudo preenchido', function (done) {  
 request.post('/produtos')  
 .send({titulo:"novo livro",preco:20.50,descricao:"livro de teste"})  
 .expect(302, done)  
 });

Para executar o teste é o mesmo comando já conhecido.

# **Separando o banco de teste e o de desenvolvimento**

[Sugira uma melhoria](https://cursos.alura.com.br/course/node-js/section/7/exercise/7#)

Criamos um novo banco de dados na aplicação para conseguir separar execuções de ambientes de teste de desenvolvimento.

Para que isso realmente funcione, é preciso criar o ambiente de teste e informar à aplicação quando ela deve utilizá-lo.

Crie um novo banco de dados no MySQL somente para ser executado em ambiente de testes. Dê a ele um nome que evidencie isso facilmente.

Altere o connectionFactory para que ele passe a verificar a variável de ambiente antes de criar a conexão com o banco.

No Controller da Rota que faz a listagem de produtos, mude a função que define a Rota para que ela passe a receber como parâmetro também a variável next, que armazena a próxima função a ser executada.

Na função de callback da Rota passe a verificar a variável erros (que é passada como parâmetro para função que lista os produtos) e caso haja algum erro, retorne invocando a função next e passando como parâmetro a própria variável erros.

Por fim, execute o teste passando a variável de ambiente com o valor que indica que este é um teste.

## **Resposta do instrutor**

Para o criar o novo banco, basta logar no MySQL e executar a função que cria um banco. Veja o exemplo:

create database casadocodigo\_nodejs\_teste;

A ConnectionFactory verificando o ambiente passa a ficar assim:

var mysql = require('mysql');  
  
function createDBConnection(){  
 if (!process.env.NODE\_ENV || process.env.node === 'dev') {  
 return mysql.createConnection({  
 host: 'localhost',  
 user: 'root',  
 password: '',  
 database: 'casadocodigo\_nodejs'  
 });  
 }  
  
 if (process.env.NODE\_ENV == 'test') {  
 return mysql.createConnection({  
 host: 'localhost',  
 user: 'root',  
 password: '',  
 database: 'casadocodigo\_nodejs\_teste'  
 });  
 }  
}  
  
module.exports = function() {  
 return createDBConnection;  
}

A Rota da listagem de produtos no Produtos.js agora fica assim:

app.get("/produtos",function(req, res, next) {  
  
 var connection = app.infra.connectionFactory;  
 var produtosBanco = new app.infra.produtosBanco(connection);  
  
 produtosBanco.lista(function(err, results){  
  
 if(err){  
 return next(err);  
 }  
  
 res.format({  
 html: function(){  
 res.render("produtos/lista",{lista:results});  
 },  
 json: function(){  
 res.json(results);  
 }  
 });  
  
 });  
  
 connection.end();  
  
 });

Para executar o teste, passando a variável de ambiente fica algo assim:

NODE\_ENV=test node\_modules/mocha/bin/mocha

# **Limpando o banco para cada caso de teste**

[Sugira uma melhoria](https://cursos.alura.com.br/course/node-js/section/7/exercise/8#)

Para garantir a consistência dos testes, optamos por sempre limpar o banco antes de executá-los. Como não queremos ficar repetindo esse código em cada chamada de caso de teste, decidimos utilizar o recurso beforeEach do Mocha, que permite executar um código sempre antes da execução de cada caso de teste.

Implemente essa função no seu arquivo de teste, fazendo com que ela dispare uma query no banco que deleta todos os dados da tabela de produtos.

Depois disso, rode os testes novamente e veja se realmente o banco está sendo sempre limpo.

## **Resposta do instrutor**

No arquivo de testes produtos.js, basta implementar a chamada função beforeEach da seguinte maneira:

beforeEach(function(done) {  
 var connection = express.infra.connectionFactory();   
 connection.query("delete from livros", function(ex,result){  
 if(!ex){  
 done();  
 }  
 });  
 });

Pesquise como fazer a mesma query de uma maneira mais rápida utilizando a lib **node-database-cleaner**