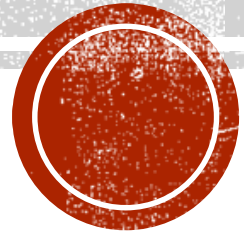
EXPRESS



GERENCIANDO MÓDULOS COM NPM

- Assim como o Gems do Ruby, ou o Maven do Java, o Node.js também possui o seu próprio gerenciador de pacotes: ele se chama NPM (Node Package Manager)

COMANDOS

- npm install nome_do_módulo : instala um módulo no projeto;
- npm install -g nome_do_módulo : instala um módulo global;
- npm install nome_do_módulo --save : instala o módulo no projeto, atualizando o package.json na lista de dependências;
- npm list : lista todos os módulos do projeto;
- npm list -g : lista todos os módulos globais;
- npm remove nome_do_módulo : desinstala um módulo do projeto;
- npm remove -g nome_do_módulo : desinstala um módulo global;
- npm update nome_do_módulo : atualiza a versão do módulo;
- npm update -g nome_do_módulo : atualiza a versão do módulo global;
- npm -v : exibe a versão atual do NPM;
- npm adduser nome_do_usuário : cria uma conta no NPM, através do site https://npmjs.org.
- npm whoami : exibe detalhes do seu perfil público NPM (é necessário criar uma conta antes);
- npm publish : publica um módulo no site do NPM (é necessário ter uma conta antes).

ENTENDENDO O PACKAGE.JSON

- Todo projeto Node.js é chamado de módulo.
- O termo módulo surgiu do conceito de que a arquitetura do Node.js é modular. E todo módulo é acompanhado de um arquivo descritor, conhecido pelo nome de package.json
- Este arquivo é essencial para um projeto Node.js. Um package.json mal escrito pode causar bugs ou impedir o funcionamento correto do seu módulo, pois ele possui alguns atributos chaves que são compreendidos pelo Node.js e NPM.

```
{
  "name": "express",
  "version": "1.0.0",
  "description": "projeto ",
  "main": "index.js",
  "dependencies": {},
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [ "node"],
  "author": "jose alves",
  "license": "ISC"
}
```


- Com esses atributos, você já descreve o mínimo possível o que será sua aplicação. O atributo name é o principal. Com ele, você descreve o nome do projeto, nome pelo qual seu módulo será chamado via função require(‘teste') .

- Em description , descrevemos o que será este módulo.

ESCOPOS DE VARIÁVEIS GLOBAIS

- Assim como no browser, utilizamos o mesmo JavaScript no Node.js. Ele também usa escopos locais e globais de variáveis.
  única diferença é na forma como são implementados esses escopos.
  No client-side, as variáveis globais são criadas da seguinte maneira:

```
window.hoje = new Date();
alert(window.hoje);
```

- Em qualquer browser, a palavra-chave window permite criar variáveis globais que são acessadas em qualquer lugar. Já no Node.js, usamos uma outra keyword para aplicar essa mesma técnica:

```
global.hoje = new Date();
console.log(global.hoje);
```

- Ao utilizar global , mantemos uma variável global acessível em qualquer parte do projeto, sem a necessidade de chamá-la via require ou passá-la por parâmetro em uma função.

COMMONJS, COMO ELE FUNCIONA?

- O Node.js utiliza nativamente o padrão CommonJS para organização e carregamento de módulos. Na prática, diversas funções deste padrão serão usadas com frequência em um projeto Node.js.
- A função require('nome-do-modulo') é um exemplo disso, ela carrega um módulo.
- E para criar um código JavaScript que seja modular e carregável pelo require , utilizam-se as variáveis globais: exports ou module.exports .
-

A seguir, apresento-lhe dois exemplos de códigos que utilizam-se padrão do CommonJS. Primeiro, crie o código hello.js :

```
module.exports = function (msg) {
  console.log(msg);
};
```

Depois, crie o código human.js com o seguinte código:

```
exports.hello = function (msg) {
  console.log(msg);
};
```

A diferença entre o hello.js e o human.js está na maneira como eles serão carregados. Em hello.js , carregamos uma única função modular e, em human.js , é carregado um objeto com funções modulares.
Essa é a grande diferença entre eles. Para entender melhor na prática,
crie o código app.js para carregar esses módulos:

```
var hello = require('./hello');
var human = require('./human');
hello('Olá pessoal!');
human.hello('Olá galera!');
```

Tenha certeza de que os códigos hello.js , human.js e app.js estejam na mesma pasta e rode no console o comando: node app.js .
E então, o que aconteceu? O resultado foi praticamente o mesmo: o app.js carregou os módulos hello.js e human.js via require() , em seguida foi executada a função hello() que imprimiu a mensagem Olá pessoal! e, por último, o objeto human , que executou sua função human.hello('Olá galera!') .

EXPRESS

- O Express estende as capacidades do servidor padrão do Node.js adicionando **middlewares** e outras capacidades como views e rotas.
- Middlewares são funções que lidam com requisições. Uma pilha de middlewares pode ser aplicada em uma mesma requisição para se atingir diversas finalidades (segurança, logging, auditoria etc.). Cada middleware passará o controle para o próximo até que todos sejam aplicados.

ESTRUTURA DO PROJETO E PACKAGE.JSON

- app

  controllers -> controladores chamados pelas rotas da aplicação
  models -> models que representam o domínio do problema
  routes -> rotas da aplicação
  views -> views do template engine
  config -> configuração do express, banco de dados etc.
  public -> todos os arquivos acessíveis diretamente pelo navegador

- O próximo passo será criar o arquivo package.json na raiz do projeto.

  Use o comando npm init

- Adicionar Express

  npm i express –s

  O parâmetro --save grava em package.json a dependência e sua versão.

CRIANDO O MÓDULO DE CONFIGURAÇÃO DO EXPRESS

Classe express dentro do pacote config

```
//config/express.js
var express = require('express'),
    load = require('express-load');

module.exports = function () {
    var app = express();
    app.set('port', 3000);

    app.use(express.static('./public‘));
    return app;
}
```

CLASSE SERVER

```
var http = require('http');
var app = require('./config/express')();

http.createServer(app).listen(app.get('port'),function() {
    console.log('Express Serever rodando '+app.get('port'));
});
```

CARREGANDO DEPENDÊNCIAS COM EXPRESSLOAD

- Com Express, podemos evitar chamadas à função require dentro de nossos controllers e routes através do módulo **express-load**
- O primeiro passo é instalar o módulo dentro da pasta raiz de nosso projeto através do gerenciador de pacotes do Node.js:
  npm install express-load –save
- Em seguida, precisamos importar o módulo dentro do nosso arquivo de configuração do Express:

```
// config/express.js
var load = require('express-load');
 load('models')
 .then('controllers')
 .then('routes')
 .into(app);
```

- A função load carregará todos os scripts dentro das pastas app/models, app/controllers e app/routes. No final, a função into adiciona dinamicamente na instância do Express propriedades que apontam para esses módulos.

ORDEM DE CARREGAMENTO

- Um ponto importante é que precisamos carregar as pastas seguindo a ordem *models, controllers e routes*, caso contrário não conseguiremos, por exemplo, ter acesso aos nossos controllers em nossas rotas caso os módulos com nossos controllers tenham sido carregados por último.

LISTANDO DADOS

- nosso servidor precisa retornar uma lista de contatos que mais tarde será consumida pelo nosso front-end usando AngularJs. Disponibilizaremos este recurso através de um identificador. No protocolo HTTP, usamos URLs para isso:
  http://localhost:3000/contatos
- Lembre-se que no Express esta URL é chamada de **rota** (route) e precisa ser configurada. Vamos criar o arquivo app/routes/contato.js:

```
// app/routes/contato.js
module.exports = function(app) {
        var controller = app.controllers.contato;
        app.get('/contatos', controller.listaContatos);
};
```

Precisamos criar o controller, através do arquivo app/controllers/contato.js:

```
// app/controllers/contato.js
var contatos = [
    {       _id: 1, nome: 'Contato Exemplo 1',
        email: 'cont1@empresa.com.br'    },
    {       _id: 2, nome: 'Contato Exemplo 2',
        email: 'cont2@empresa.com.br'    },
    {       _id: 3, nome: 'Contato Exemplo 3',
        email: 'cont3@empresa.com.br'    }];
module.exports = function () {
    var controller = {};
    controller.listaContatos = function (req, res) {
        res.json(contatos);
    };
    controller.listaContato = function (req, res) {
    return controller;
}
```

RETORNANDO CONTATO DA LISTA

Adicione a linha no arquivo app/controllers/contato.js

```
app.get('/contatos/:id', controller.obtemContato);
```

Adicione no controller

```
controller.obtemContato = function (req, res) {
    var idContato = req.params.id;
    var contato = contatos.filter(function (contato) {
        return contato._id == idContato;
    })[0];
    contato ?
        res.json(contato) :
        res.status(404).send('Contato não encontrado');
};
```

Nesse código, quando um contato não é encontrado, alteramos o status da resposta para 404 (não encontrado) através da função res.status e enviamos como resposta uma mensagem com uma pista do problema ocorrido.
Já podemos testar:
http://localhost:3000/contatos/2

EXERCÍCIO

- Criar uma aplicação de series com pelo menos 5 series a serie deve ter nome e id, ela deve ter 3 rotas
  - /series onde deve listar todas as series
  - /series/:id busca a serie por id
  - /series/:nome busca a serie por nome