# Desenvolvimento Backend

## Aula 02 - Gerenciando pacotes e fundamentos do Node.js

Material Didático do Instituto Metrópole Digital - IMD

Termo de uso

# Apresentação

Nesta aula você verá fundamentos importantes do Node.js, entendendo como essa tecnologia funciona por dentro. Além disso, conhecerá a ferramenta NPM, que auxilia na criação de projetos em Node.js.

## Objetivos

- Conhecer o gerenciador de pacotes NPM
- Entender os arquivos package.json e package-lock.json gerados
- Conhecer o *event loop* do Node.js
- Entender a diferença de operações síncronas e assíncronas

# Gerenciando módulos com npm

Link do video da aula: https://youtu.be/ulgDH1_i_Ak

---

O npm é o gerenciador de pacotes oficial para o Node.js. Com ele é possível manter registradas e organizadas todas as dependências que sua aplicação utiliza.

Para iniciar um novo projeto Node.js com o npm, execute:

```
$ npm init
```

Na pasta do projeto, crie um arquivo chamado index.js, com o seguinte conteúdo:

```
console.log('Hello!')
```

Perceba que ao executar o comando `npm init` foi criado um arquivo chamado package.json. Esse arquivo armazena todos os metadados do projeto. Nesse arquivo também é possível configurar os *scripts* de inicialização, conforme exemplo:

```
{
  "name": "aula02",
  "version": "1.0.0",
  "description": "Exemplo simples de uso do npm",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "author": "Gustavo Leitão",
  "license": "ISC"
```

```
}
```

Agora você pode executar o programa:

```
$ npm start
```

# Adicionando uma biblioteca

Para exemplificar como utilizar uma biblioteca, vamos adicionar a dependência "random-words", que permite gerar palavras aleatoriamente. Para isso, execute no *prompt* de comando (na pasta do projeto):

```
$ npm install random-words
```

Altere o arquivo index.js, para utilizar a biblioteca instalada:

```
const randomWords = require('random-words');
console.log(randomWords());
```

Em seguida, execute o programa:

```
$ npm start
```

Perceba que a cada execução uma palavra aleatória é impressa na tela. Essa palavra está sendo gerada pela biblioteca que acabamos de instalar.

# Instalando o Nodemon

O Nodemon é uma biblioteca que facilita o processo de desenvolvimento através do monitoramento dos arquivos do programa. Ao sinal de alguma mudança, o nodemon recarrega o sistema tornando a programação mais produtiva.

Para instalar o nodemon, execute:

```
$ npm install --save-dev nodemon
```

> Perceba que adicionamos a opção "--save-dev". Isso é importante, pois o nodemon será utilizado apenas para fins de desenvolvimento e não deve ser adicionado ao pacote final do software.

Para configurar o nodemon, é necessário alterar o *script* de inicialização no

package.json, conforme exemplo:

```
{
  "name": "aula02",
  "version": "1.0.0",
  "description": "Exemplo simples de uso do npm",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js"
  },
  "author": "Gustavo Leitão",
  "license": "ISC",
  "dependencies": {
    "random-words": "^1.1.1"
  },
  "devDependencies": {
    "nodemon": "^2.0.7"
  }
}
```

Para executar o programa monitorado pelo nodemon, utilize o seguinte comando:

```
$ npm start
```

# Entendendo o package.json e o package-lock.json

Link do video da aula: https://youtu.be/bB-zyTd1t64

---

## Criando múltiplos scripts

O npm permite criar múltiplos scripts. Para este exemplo, vamos criar um novo script chamado "dev" que iniciará o sistema com o nodemon habilitado. Em seguida, podemos alterar o script de start para utilizar o node diretamente, conforme exemplo abaixo:

```
{
  "name": "aula02",
  "version": "1.0.0",
  "description": "Exemplo simples de uso do NPM",
  "main": "index.js",
```

```
  "scripts": {
    "dev": "nodemon index.js",
    "start": "node index.js"
  },
  "author": "Gustavo Leitão",
  "license": "ISC",
  "dependencies": {
    "random-words": "^1.1.1"
  },
  "devDependencies": {
    "nodemon": "^2.0.7"
  }
}
```

Para executar o script de *dev*, utilize o seguinte comando:

```
$ npm run dev
```

# package.json

Como vimos, o package.json armazena todos os metadados do sistema, incluindo as dependências e suas versões. Perceba que as versões possuem três partes, ex: "1.3.4".

O primeiro número (da esquerda para direita) é chamado de *major version*. Deve ser incrementado quando uma mudança maior é adicionada à biblioteca e que potencialmente causa incompatibilidade com as versões anteriores (ou seja, você precisará fazer alguma adaptação no seu programa).

O segundo número é chamado de *minor version* e é incrementado sempre que uma nova funcionalidade é disponibilizada na biblioteca sem causar incompatibilidade com as versões anteriores.

Já o último número é chamado de *patch version* e deve ser incrementado sempre que um ajuste é feito na biblioteca, mas que não incremente nenhuma funcionalidade. Geralmente correções de *bugs*, melhoria de desempenho, são exemplos de mudanças que alteram apenas o *patch version*.

Perceba que há um símbolo antes do número da versão no arquivo package.json. Esse símbolo é chamado de modificador e pode assumir alguns valores. Veja dois dos principais:

- **"^"** - instala qualquer versão compatível. Permite instalar versões mais recentes com mudança em *minor* e em *patch version*.
- **"~"** - Significa versão equivalente. Permite instalar versões mais recentes com

mudança apenas de *patch version*.

Há outros modificadores de acesso. A lista completa pode ser acessada em https://nodejs.dev/learn/semantic-versioning-using-npm.

## package-lock.json

O package-lock é gerado automaticamente e tem o objetivo de registrar a versão exata que foi instalada. Com isso, caso seja executado o comando de instalação em outro momento ou até mesmo em outro ambiente, a versão registrada no package-lock.json será instalada.

Esse comportamento é importante para manter a consistência de versões em uma equipe de desenvolvimento e também para garantir que a versão utilizada no ambiente de desenvolvimento seja exatamente igual à que será utilizada em produção.

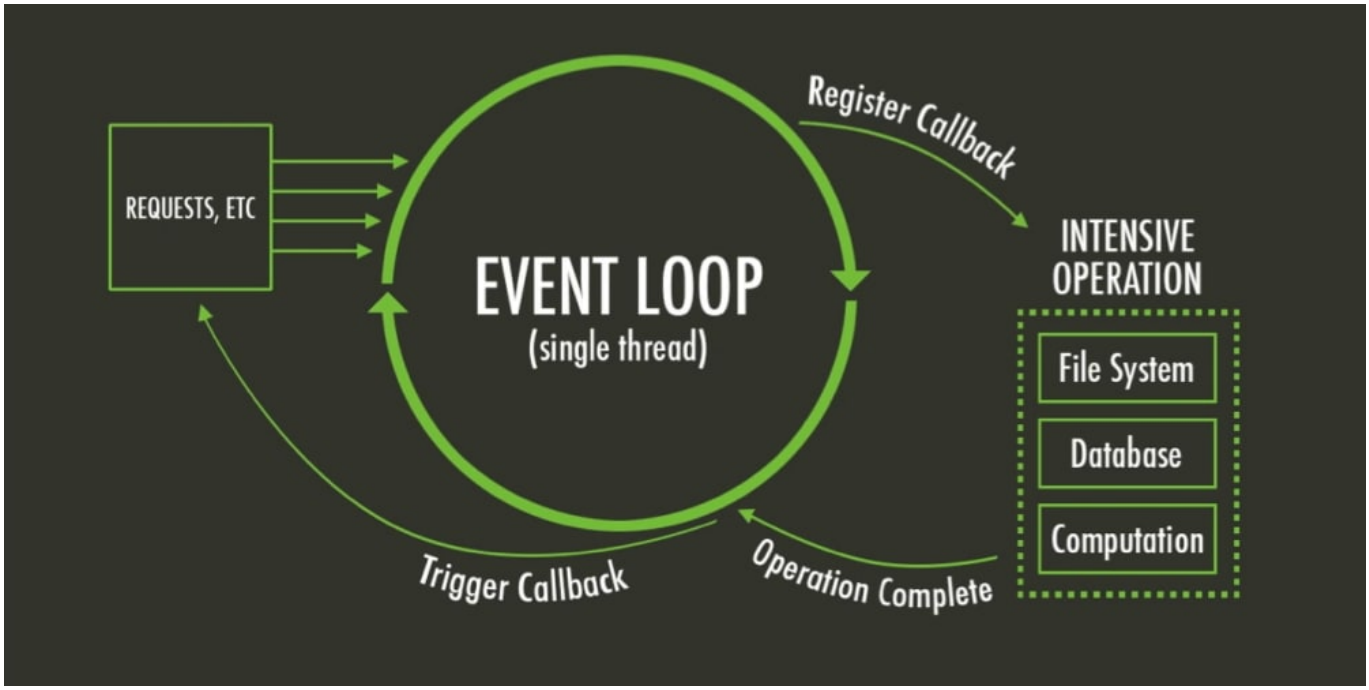
# *Event loop*

Link do video da aula: https://youtu.be/3grOVRie66w

---

Vamos agora ver como funciona o Node.js por dentro. Vamos estudar o *event loop* uma tecnologia que permite ao Node.js atender múltiplos usuários simultaneamente de forma simples e descomplicada. Como o Node.js faz isso?

**Figura 1** - Event loop




Veja a ilustração do *event loop* que mostra essa tecnologia. Perceba na imagem

que existe um grande bloco chamado *event loop* que executa todas as operações em sequência. Ou seja, as instruções são executadas uma após a outra, mesmo que oriundas de múltiplos usuários.

No entanto, isso pode gerar um problema quando o tratamento de uma requisição for custoso, pois os outros usuários seriam impactados. Como o Node.js resolve isso?

Para resolver isso, o Node.js trata operações de entrada e saída (E/S) de maneira diferente das operações comuns. As operações comuns são executadas no próprio *event loop*, mas as operações de E/S não. As operações de E/S (que normalmente são mais custosas) são executadas à parte em *threads* gerenciadas pelo Node.js. Quando as operações de E/S são encerradas, o Node.js notifica através de um *callback* para que a resposta possa ser tratada no próprio *event loop*. Dessa forma, o *event loop* fica livre processando outras requisições enquanto as operações de E/S são realizadas.

Assim, do ponto de vista do programador, há uma única *thread* (fluxo de execução) o que também simplifica muito a programação, evitando a necessidade de lidar com concorrência e sincronia.

Essa técnica também traz vantagens do ponto de vista de recurso computacional, uma vez que criar uma *thread* por usuário é custoso do ponto de vista de recursos computacionais (uso de memória, principalmente).

Assim, com o *event loop*, o Node.js consegue tratar múltiplos usuários de forma prática e eficiente.

Para mais detalhes, assista a videoaula e acesse https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

# Execução síncrona e assíncrona

Link do video da aula: https://youtu.be/P2CryAQv7Lg

---

Vamos agora mostrar exemplos de operações síncronas e assíncronas para reforçar o entendimento do *event loop*.

Para isso, faremos um exemplo com leitura de arquivo. O Node.js permite realizar esta operação de forma assíncrona (desejável) e também de forma síncrona. Acompanhe a diferença dessas abordagens nas seções seguintes.

## Leitura síncrona

Na leitura síncrona o *event loop* fica "preso" aguardando a leitura do arquivo

terminar, para seguir adiante. Apesar da simplicidade, essa forma bloqueia o tratamento de outras requisições enquanto o Node.js estiver realizando a operação de leitura no arquivo.

```
const fs = require('fs')
const data = fs.readFileSync('./tmp.txt', {encoding: 'utf-8', flag:
'r'})
console.log(data)
console.log('Executou aqui!')
```

## Leitura assíncrona

Já na forma assíncrona o *event loop* fica liberado para seguir adiante. Para tratar a resposta é necessário definir uma função (chamada de *callback*) que será chamada quando a operação de leitura finalizar.

Com isso, perceba que a impressão da mensagem "Executou aqui!" acontece provavelmente antes mesmo da leitura terminar - isso pode variar dependendo do tempo de leitura do arquivo.

```
const fs = require('fs')
fs.readFile('./tmp.txt', {encoding: 'utf-8', flag: 'r'}, function
(err, data){
    if (!err){
        console.log(data)
    }
})
console.log('Executou aqui!')
```

Apesar de em primeira análise parecer mais complexo, esse tipo de uso é o que faz o Node.js ser bastante eficiente computacionalmente com pouco recurso computacional. Sendo assim, evite utilizar operações de E/S síncronas para extrair o máximo da tecnologia.

# Resumo

Nesta aula você viu como o gerenciador de pacotes npm é importante para facilitar a criação de um projeto e gerenciamento de suas dependências. Viu também como o *event loop* funciona e sua importância no tratamento de operações simultâneas. Por fim, viu exemplos de operações síncronas e assíncronas e o impacto delas no *event loop*.