

Boas-vindas!

Esteja confortável, pegue uma água e se acomode em um local tranquilo que já começamos.

Como você **chega?**

1

2



3

Esta aula será

- gravada

Resumo

da última aula

- ✓ Usar programação síncrona e assíncrona e aplicá-la no uso de arquivos
- ✓ Conhecer o módulo nativo do Nodejs para interagir com arquivos
- ✓ Conhecer o uso de arquivos com callbacks e promessas
- ✓ Conhecer as vantagens e desvantagens do FileSystem, bem como exemplos práticos.

Perguntas?

AULA 5 – BACKEND

Gerenciador de pacotes NPM

Objetivos da aula

- Revisar o que é o Node.js e seu uso no back-end
Entenda a diferença entre um módulo nativo e um módulo de terceiros
- Conhecer a função do NPM e o processo de instalação da dependência
- Conheça o processo de atualização de dependências.

Vamos revisar os **principais**
pontos da aula anterior?

Glossário - Aula 4

Arquivo: Sequência de informações que podem ser armazenadas em um disco, resolvendo a persistência na memória

Persistência de memória: modelo de armazenamento que só persiste ao longo do programa, se o programa terminar ou for reiniciado, a informação é perdida. Normalmente são arrays e objetos

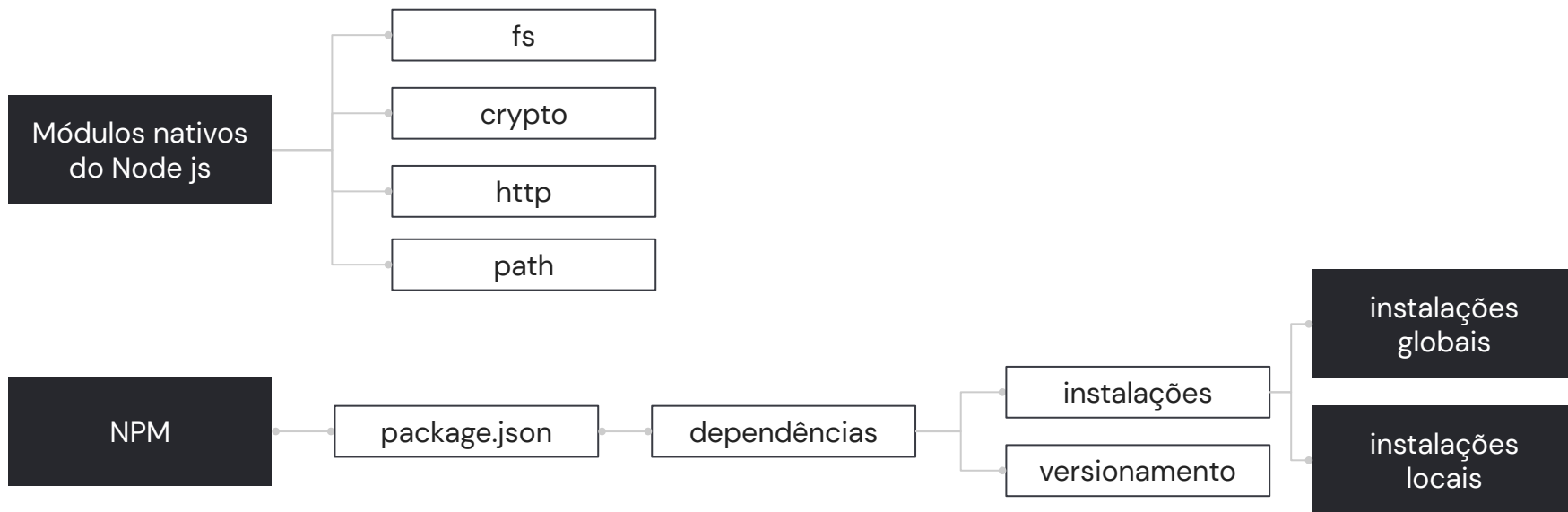
Persistência de arquivos: Modelo de armazenamento utilizando arquivos de um FileSystem, evitando a perda de informações ao reiniciar um programa

fs: Abreviação de “fileSystem”, que é o módulo interno do Node js para trabalhar com arquivos.

Sufixo Sync: Usado na frente das operações fs para indicar que o processo será síncrono e, portanto, bloqueante.

fs.promises: Usado para indicar que as operações de arquivo devem ser executadas como uma promessa e, portanto, sem bloqueio. Eles trabalham com async await ou com .then e .catch

MAPA DE CONCEITOS



Revendo nossa ferramenta de trabalho: Node Js

Sobre o coração dos nossos projetos

O **Node js** não é apenas mais um módulo ou uma biblioteca para trabalhar. Em vez disso, é um ambiente de desenvolvimento completo no qual nossos programas javascript vivem e são executados.

Ele surgiu da necessidade de executar javascript fora do navegador e se tornou um dos pilares do desenvolvimento web

[Documentação de NodeJs](#)



Node js

Ele surgiu da necessidade de executar javascript fora do navegador e se tornou um dos pilares do desenvolvimento web.

Possui o mesmo mecanismo V8 do Google Chrome, que **permite converter código javascript em código de máquina para que seja processado corretamente.**

Além disso, possui muitas funcionalidades internas da mesma linguagem javascript graças aos seus ajustes com ECMAScript.

O poder do Node js no back-end

O Node js foi planejado principalmente para o back-end, o que significa que **seu desenvolvimento foi baseado no back-end em primeiro lugar**. Seu sistema de desenvolvimento baseado em eventos permite ao desenvolvedor construir aplicações leves, rápidas e até mesmo em tempo real.

Isto sem esquecer que o suporte desta maravilha tecnológica se baseia na utilização e processamento de Javascript, que possui uma infinidade de funções e estruturas que permitem resolver diferentes problemas todos os dias.

**Até este ponto, quão bem você domina
a linguagem Javascript?**

Projeto em node

Duração: 15 minutos

Atividade de revisão para fortalecer os conceitos de node e javascript

- ✓ Crie um projeto em nodejs que gere 10.000 números aleatórios no intervalo de 1 a 20.
- ✓ Crie um objeto cujas chaves são os números gerados e o valor associado a cada chave será o número de vezes que esse número foi gerado. Exiba os resultados no console.

Módulos nativos do Node js

Módulos nativos do Node js

Como precisamos de programas mais complexos, precisamos de operações mais complexas, e como precisamos de operações mais complexas, precisamos de ferramentas mais úteis.

É por isso que, desde que instalamos o Nodejs em nosso computador, já temos uma série de módulos nativos (ou seja, eles já moram dentro dele), para poder resolver esse tipo de tarefa com eficiência e sem ter que reprogramar tudo (lembre-se não reinventar a roda).

[Módulos ativos de NodeJs](#)

Módulos nativos em Nodejs

fs

Módulo utilizado para gerenciar arquivos

Ele é usado para lidar com outro modelo de persistência.

crypto

Permite operações de criptografia e criptografia para informações confidenciais

Serve para melhorar a segurança dos dados

http

Permite criar um servidor básico sob o protocolo http

Ele é usado para criar nosso primeiro servidor de solicitação/resposta

path

Permite o correto tratamento de rotas

É usado para evitar ambiguidade ao trabalhar com rotas

Importante

Lembre-se de que usamos soluções de terceiros para melhorar nosso trabalho. Usar um módulo que nos permite resolver um problema anterior nos permite focar no problema atual.

Hands on Lab

Prática dos módulos nativos: fs + criptografia

Duração: 15 minutos

Nesta instância da aula, vamos nos aprofundar na criação de promessas e no uso de `async await` com um exercício prático.

De que maneira?

- ✓ O professor demonstrará como fazer e você poderá replicar em seu computador. Se surgirem dúvidas, você pode compartilhá-las para resolvê-las junto com a ajuda dos tutores.

Hands on Lab

Prática dos módulos nativos: fs + criptografia

Duração: 15 minutos

Como fazemos?

- ✓ Será criada uma classe “UserManager” que permitirá que os usuários sejam salvos em um arquivo. O usuário será recebido com uma senha de string simples e a senha com hash deve ser salva com criptografia. Use os módulos nativos fs e crypto. O UserManager deve ter os seguintes métodos:
 - O método "Criar usuário" deve receber um objeto com os campos:
 - Nome
 - Sobrenome
 - Idade
 - Curso

Hands on Lab

Prática dos módulos nativos: fs + criptografia

Duração: 15 minutos

- O método deve salvar um usuário em um arquivo "Usuarios.json", lembrando que a senha deve ser hash por segurança
- O método "Validar Usuário" receberá o nome de usuário para validar, seguido da senha, ele deve ser capaz de ler o json gerado anteriormente com o array de usuários e fazer a comparação de senha. Se o nome de usuário e a senha forem iguais, retorne uma mensagem "Logado", caso contrário, indicar erro se o usuário não existir ou se a senha não corresponder.

Gerenciando módulos de terceiros: NPM

O que é NPM?


NPM significa “**Node Package Manager**”, que se refere a um aplicativo Node. Isso permite que a comunidade de desenvolvedores crie seus próprios módulos , para poder carregá-los na nuvem para que outros desenvolvedores possam usá-los.

Para o trabalho do pacote, teremos um arquivo em nosso projeto chamado package.json





[Documentação de NPM](#)

Pode consultar a página oficial

 Nicely Pruned Marigolds

ProTeamsPricingDocumentation



Search

You don't have two-factor authentication (2FA) enabled on your account. [Configure 2FA](#) or [visit our docs](#) to learn more.

Popular libraries

lodash

react

axios

tslib

chalk

react-dom

commander

express

vue

moment

Discover packages

Front-end

Back-end

CLI

Documentation

CSS

Testing

IoT

Coverage

Mobile

Frameworks

Robotics

Math

By the numbers

Packages

2.302.390

Downloads · Last Week

50.219.028.642

Downloads · Last Month

222.350.658.434

package.json

O que é package.json?

package.json é um arquivo que geramos dentro de nossos projetos, que conterá diferentes especificações dele, coisas como:

- O nome do seu projeto
- A versão do seu projeto
- Alguns scripts para rodar o projeto
- Do que depende o projeto?

[Leitura complementar package.json](#)

```
{ } package.json X
{ } package.json > ...
1  {
2    "name": "projeto",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
   ▶ Depurar
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "author": "",
10   "license": "ISC"
11  }
12
```

Dependências

Quando nosso projeto precisa usar dependências de terceiros do npm, um novo campo é adicionado ao nosso package.json chamado "dependencies" que conterá os módulos que instalamos naquele projeto e, portanto, indica que o projeto precisa de dependências de terceiros. Essas dependências são instaladas para poder rodar corretamente.

```
9    "author": "",  
10   "license": "ISC",  
11   "dependencies": {  
12     "cowsay": "^1.5.0"  
13   }
```

Instalando nossa primeira dependência

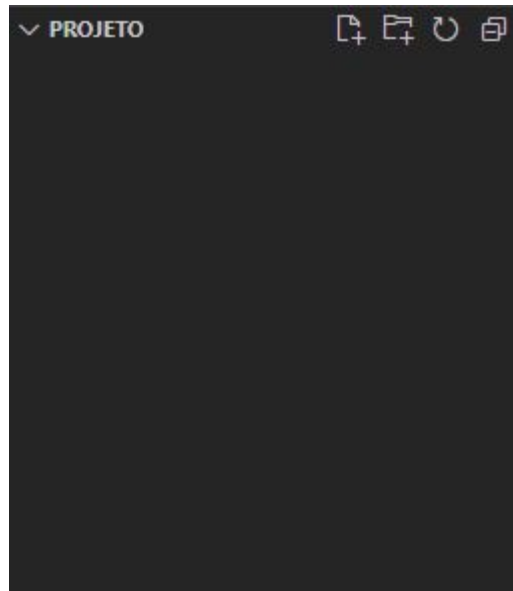


Para trabalhar com dependências precisamos:

- **Passo 1:** Tenha um projeto vazio
- **Passo 2:** Execute o comando `npm init`
- **Passo 3:** Execute o comando `npm install`
- **Passo 4:** Usando o módulo

1 Tenha um projeto vazio

Nossos projetos devem ser **ambientes isolados** que **não obstruam com outros projetos**. Então precisaremos ter uma pasta vazia.



2 Execute o comando npm init

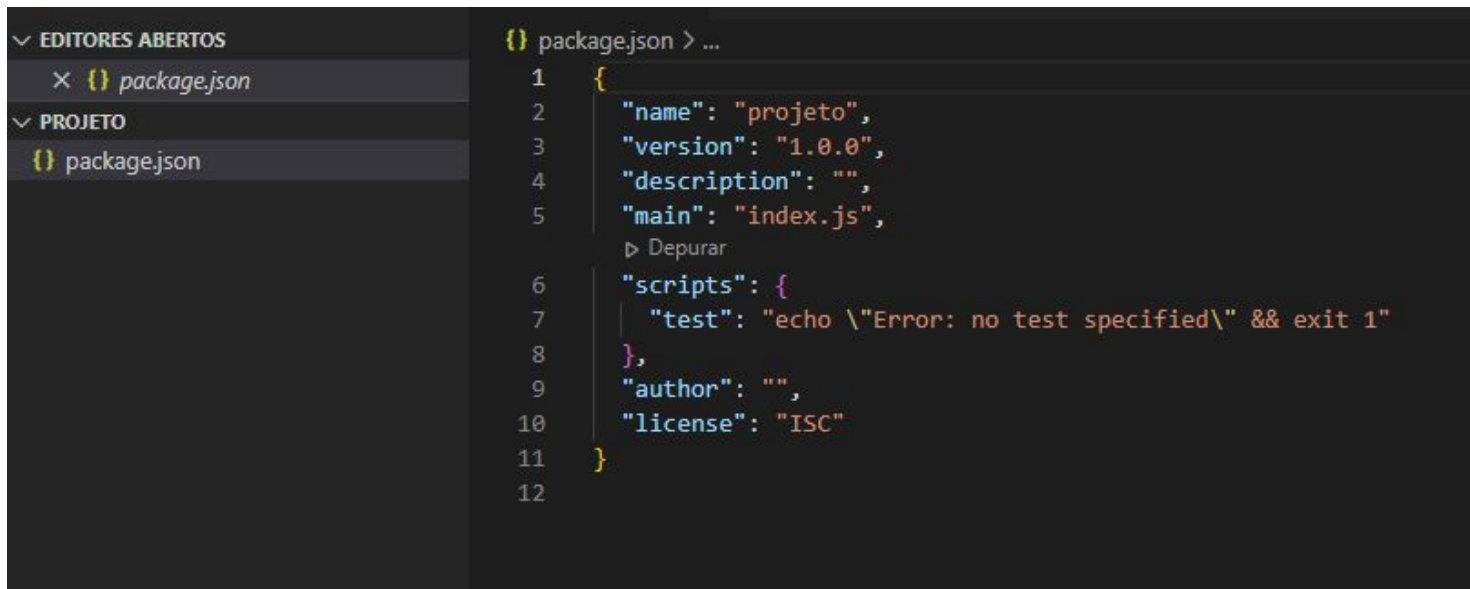
O comando `npm init` permitirá que você crie um `package.json` dentro da pasta onde você está. Verifique se você está dentro da pasta em que estará trabalhando .

```
CursoBackend\Aula 5\projeto> npm init
```

```
package name: (projeto)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
```


2 Execute o comando npm init

Por fim, será gerado um package.json com as especificações definidas



The screenshot shows a code editor interface. On the left, a sidebar displays the file explorer with 'EDITORES ABERTOS' (Open Editors) and 'PROJETO' (Project) sections. Under 'EDITORES ABERTOS', there is a file named 'package.json'. Under 'PROJETO', there is also a file named 'package.json'. The main editor area shows the content of the selected 'package.json' file. The content is a JSON object with the following properties: 'name' (projeto), 'version' (1.0.0), 'description' (empty string), 'main' (index.js), 'scripts' (an object with a 'test' property), 'author' (empty string), and 'license' (ISC). The 'test' script is defined as 'echo \"Error: no test specified\" && exit 1'. The editor has a dark theme and line numbers are visible on the left side of the code area.

```
{ } package.json > ...
1  {
2    "name": "projeto",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "author": "",
10   "license": "ISC"
11 }
12
```

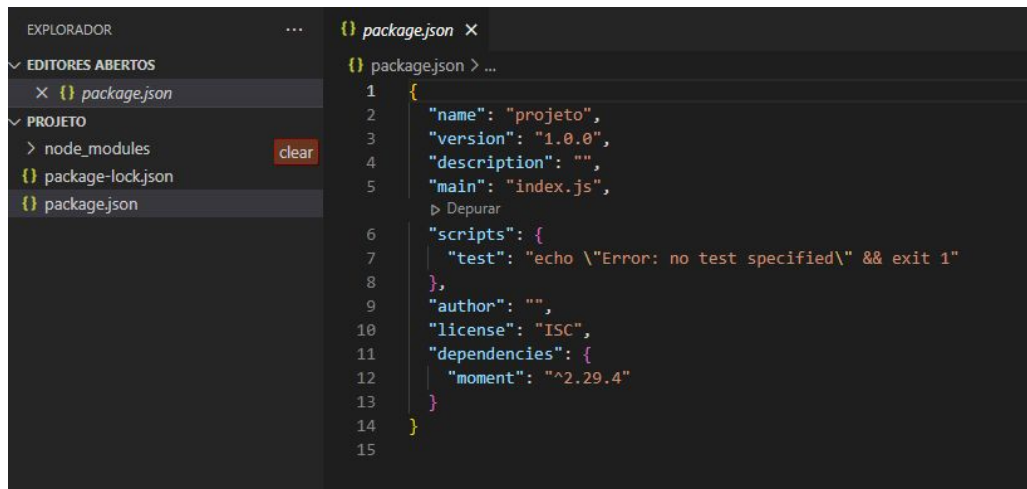
com `npm init -y` você pode pular as etapas de seleção, indicando os cuidados a serem tomados ao usar `npm init -y`

3 Execute o comando npm install

Uma vez gerado nosso package.json, com o comando `npm install nome_do_modulo` podemos **instalar o módulo que precisamos**. Neste exemplo, instalamos o momento que é usado para gerenciar datas e horas com eficiência.

Uma vez instalado, **uma pasta node_modules será gerada** (onde ficará o momento), e a **dependência do momento será adicionada ao meu package.json**

```
\CursoBackend\Aula 5\projeto> npm install moment
```

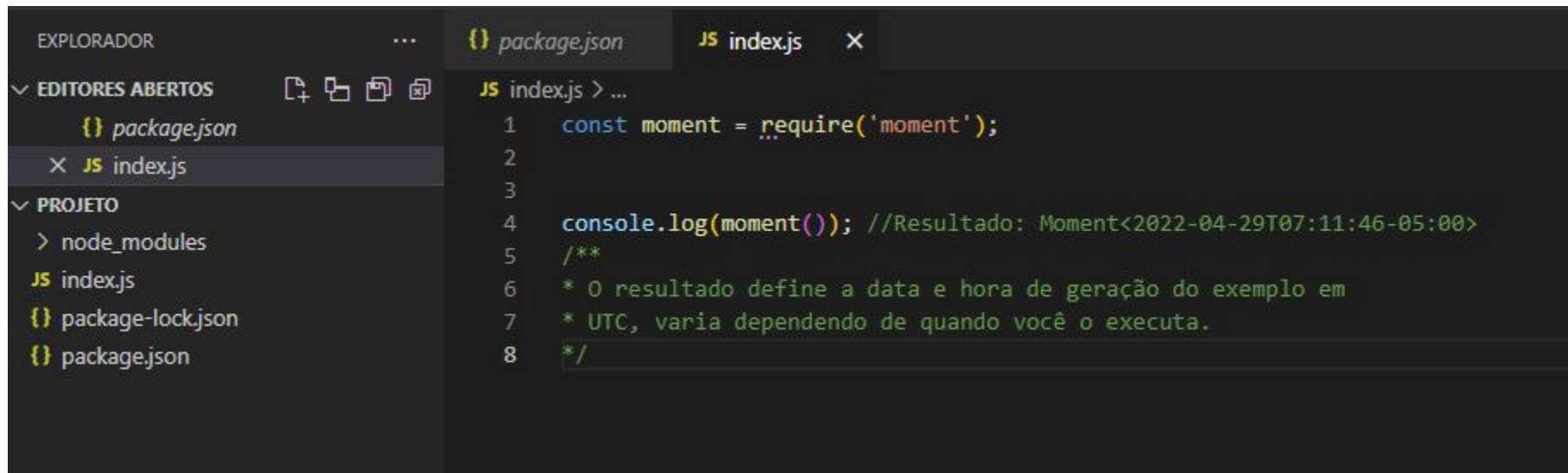


The screenshot shows the VS Code interface. On the left, the Explorer sidebar shows the project structure: 'EDITORES ABERTOS' with 'package.json' and 'PROJETO' with 'node_modules', 'package-lock.json', and 'package.json'. A 'clear' button is next to 'node_modules'. The main editor shows the 'package.json' file with the following content:

```
1 {
2   "name": "projeto",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1"
8   },
9   "author": "",
10  "license": "ISC",
11  "dependencies": {
12    "moment": "^2.29.4"
13  }
14 }
```

4 Usando o módulo

Depois de tudo instalado, podemos importar o módulo da mesma forma que fazemos com nossos módulos nativos.



The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar shows the project structure with files like `package.json`, `index.js`, `package-lock.json`, and `node_modules`. The main editor area displays the `index.js` file with the following code:

```
1  const moment = require('moment');
2
3
4  console.log(moment()); //Resultado: Moment<2022-04-29T07:11:46-05:00>
5  /**
6   * O resultado define a data e hora de geração do exemplo em
7   * UTC, varia dependendo de quando você o executa.
8   */
```



Break

5 minutos e voltamos!





Break

10 minutos e voltamos!



Para pensar

Se os módulos que instalo forem de terceiros, como sei o que posso fazer com eles e o que não?

Devo fazer a instalação do módulo em todos os projetos que vou fazer?

Deixe sua opinião no chat

Instalações globais e instalações locais

Global ou local?

Instalar uma dependência localmente significa que **aquele módulo instalado pertencerá e será usado somente dentro daquele projeto**. Isso implica que se você quisesse usar a mesma dependência em outro projeto, teria que fazer a instalação novamente, pois eles não são compartilhados .

```
projeto> npm install modulo_a_instalar
```

Por outro lado, **instalar uma dependência globalmente implica instalar o módulo para todos os projetos**, evitando a necessidade de instalá-lo toda vez que criamos um novo projeto. Para instalar globalmente, colocamos apenas o sinalizador `-g`.

```
projeto> npm install -g modulo_a_instalar
```


O mal das instalações globais



Desvantagens de uma instalação global

Se instalarmos uma **dependência global**, estaremos **limitando todos os nossos projetos para usá-la com a versão instalada**. Isso significa que se eu instalei minha dependência global na versão 1.0.0, todos os projetos usarão a versão 1.0.0.

E se a dependência for posteriormente atualizada para a versão 2.0.0 e eu quiser usar seus recursos mais recentes, o que acontecerá com todos os projetos que estavam executando apenas a versão 1.0.0? **Eles se tornam obsoletos e você tem que atualizar o código de todos eles**, o que é um problema muito complicado.

Controle de versão de dependência

Controle de versão de dependência

Como qualquer programa, este não costuma ficar estático em uma única versão. **Sempre que fazemos uma alteração em nosso código, estamos alterando a versão dele.**

A mesma coisa acontece com o que instalamos, todas as dependências de terceiros são regidas por versões .

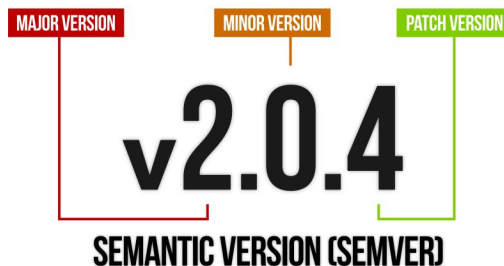
Uma versão define pontos específicos nos quais um código tem certas características, certa sintaxe, certa funcionalidade e até mesmo certos bugs.

```
"dependencies": {  
  "aws-sdk": "^2.1122.0",  
  "bcrypt": "^5.0.1",  
  "cookie-parser": "^1.4.6",  
  "cors": "^2.8.5",  
  "dotenv": "^16.0.0",  
  "express": "^4.18.0",  
  "jsonwebtoken": "^8.5.1",  
  "mongoose": "^6.3.1",  
  "multer": "^1.4.4",  
  "multer-s3": "^2.10.0",  
  "passport": "^0.5.2",  
  "passport-jwt": "^4.0.0",  
  "passport-local": "^1.0.0",  
  "socket.io": "^4.5.0"  
}
```

Gerenciamento de versões no NPM

As versões são baseadas em 3 elementos básicos:

- **Versões principais** (primeiro dígito): Refere-se a grandes mudanças, tanto que não são mais compatíveis com as versões anteriores .
- **Versões menores** (segundo dígito): Refere-se a mudanças em determinados recursos e funcionalidades que não afetam as versões anteriores , ou seja, podemos atualizá-lo sem afetar a estrutura do projeto.
- **Patches** (último dígito): Refere-se a correções de bugs ou gerenciamento de defeitos do código atual. Nada está sendo mudado estruturalmente falando , estamos apenas consertando as coisas.



[Leitura complementar sobre versionamento](#)

Política de atualização de dependência

Operadores para atualizar versões

Em nosso package.json podemos colocar operadores que nos permitem ter um melhor controle das versões:

- O operador `^` é usado para instalar a versão secundária mais alta , isso significa que ele não atualizará para uma versão principal, protegendo assim nosso código de incompatibilidades
- O operador `~` é usado para instalar apenas patches , o que significa que ele não altera versões secundárias, apenas pequenas correções de bugs de código.
- Se não colocarmos nenhum operador, será instalada a versão exata que colocamos.

```
"dependencies": {  
  "aws-sdk": "^2.1122.0",  
  "bcrypt": "~5.0.1",  
  "cookie-parser": "1.4.6"  
}
```

Comandos para atualizar no NPM

`npm outdated` é um comando que lerá as dependências instaladas em nosso `package.json` e, dependendo do operador que tivermos colocado, indicará o que é "conveniente" para nós. Também nos informa qual é a versão mais recente encontrada na internet, caso estejamos interessados.

```
ecommerce_backend> npm outdated
```

<u>Package</u>	<u>Current</u>	<u>Wanted</u>	<u>Latest</u>	<u>Location</u>	<u>Depended by</u>
aws-sdk	2.1122.0	2.1128.0	2.1128.0	node_modules/aws-sdk	ecommerce_backend
express	4.18.0	4.18.1	4.18.1	node_modules/express	ecommerce_backend
mongoose	6.3.1	6.3.2	6.3.2	node_modules/mongoose	ecommerce_backend

Para realizar a atualização, usaremos o `npm update` comando que se encarregará de fazer as alterações que indicarmos.

Calculadora de idade

Duração: XX minutos

Fazendo um programa que depende de módulos externos

Faça um programa que use a dependência momentjs (deve ser instalado por npm install).

- ✓ Você deve ter uma variável que armazene a data atual (use moment())
- ✓ Você deve ter uma variável que armazene apenas a data do seu nascimento (utilizar moment).
- ✓ Valide com um se a variável contém uma data válida (use o método isValid());
- ✓ Por fim, mostre ao console quantos dias se passaram desde que você nasceu até hoje. (use o método diff())
- ✓ Extra: Altere seu moment para a versão 1.6.0, pois não é a mesma versão principal, observe a alteração ao executar o programa.

Perguntas?

Como foi a aula?

1

Que bom

O que foi super legal na aula e podemos sempre trazer para as próximas?

2

Que pena

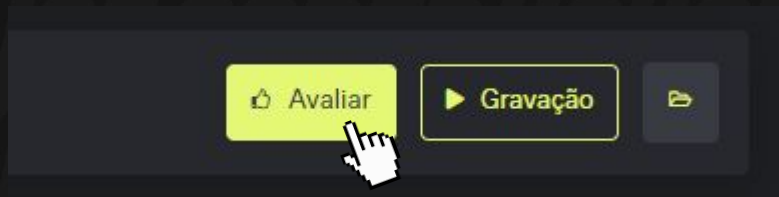
O que você acha que não funcionou bem e precisamos melhorar?

3

Que tal

Qual sugestão deveríamos tentar em próximas aulas?

O que você achou da aula?



Seu feedback vale pontos para o Top 10!! 😎



Deixe sua opinião!

1. Acesse a plataforma
2. Vá na aula do dia
3. Clique em **Avaliar**

Resumo

da aula de hoje:

- ✓ Revisamos o que é o Node.js e seu uso no back-end
- ✓ Entendemos a diferença entre um módulo nativo e um módulo de terceiros
- ✓ Conhecemos a função do NPM e o processo de instalação da dependência
- ✓ Conhecemos o processo de atualização de dependências.



**Obrigado por estudar
conosco!**