



INSTITUTO DE GESTÃO E TECNOLOGIA  
DA INFORMAÇÃO

---

# **Desenvolvimento Back End com JavaScript**

---

Guilherme Henrique de Assis

**2022**

## **Desenvolvimento Back End com JavaScript**

Guilherme Henrique de Assis

© Copyright do Instituto de Gestão e Tecnologia da Informação.

Todos os direitos reservados.

## Sumário\_Toc95839390

---

Capítulo 1.	Introdução.....	5
	Backend vs Frontend.....	5
	APIs.....	7
	Node.js .....	9
	Node.js Event Loop .....	11
	Módulos do Node.js.....	15
	NPM .....	19
	IDE de desenvolvimento .....	22
	Ferramentas para consumo de endpoints.....	24
Capítulo 2.	Express .....	26
	Instalação .....	26
	Hello World .....	32
	Configurações iniciais .....	33
	Rotas .....	37
	Middlewares .....	41
	Tratamento de erros .....	44
	Gravação de logs.....	46
	Servindo arquivos estáticos .....	48
Capítulo 3.	Organização de projetos Node.js.....	50
	Route.....	51

Controller .....	52
Service .....	52
Repository .....	53
Capítulo 4. GraphQL .....	55
Referências.....	56

## Capítulo 1. Introdução

---

Esta disciplina tem por objetivo discutir a respeito de desenvolvimento de APIs. Para isso, utilizaremos como tecnologias o JavaScript, o Node.js e o framework Express. Este capítulo fará uma breve introdução de conceitos importantes para o entendimento da disciplina. Iniciaremos contextualizando o desenvolvimento de APIs dentro do desenvolvimento backend, inclusive diferenciando este do desenvolvimento frontend. Depois falaremos sobre APIs e webservices, faremos uma introdução ao Node.js e seu gerenciador de pacotes, o NPM e, por fim, será discutido a respeito de IDEs de desenvolvimento e ferramentas para consumo de endpoints.

### Backend vs Frontend

---

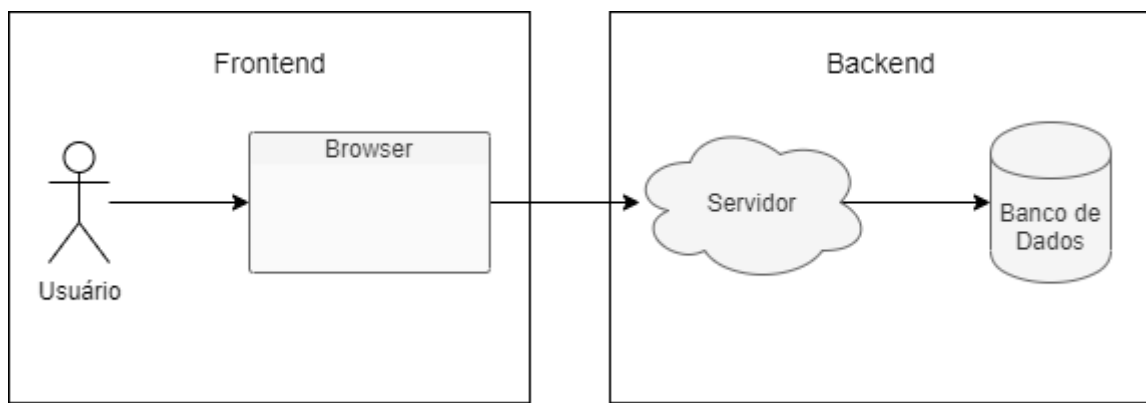
Um sistema geralmente pode ser dividido, a grosso modo, em frontend e backend. O backend se refere a parte que fica hospedada no servidor, focando principalmente em como a aplicação funciona, sendo também responsável por interagir com o banco de dados, gravando e buscando registros.

Existem várias linguagens de programação que podem ser utilizadas no desenvolvimento backend, como por exemplo Java, C#, PHP, entre outras. Esta disciplina irá focar na linguagem JavaScript, que pode ser utilizada no backend juntamente com o Node.js.

Enquanto o backend foca mais na parte da aplicação que fica no servidor, se preocupando por exemplo com as regras de negócio da aplicação e com a comunicação com banco de dados, o frontend é parte da aplicação com a qual o usuário interage. Se tratando de web, o frontend basicamente utiliza HTML, CSS e JavaScript. O HTML é o

responsável por estruturar a página, o CSS por estilizar esta estrutura e o JavaScript por fazer com que a interface seja dinâmica. As páginas podem ser montadas no servidor e devolvidas ao usuário já prontas, utilizando tecnologias como PHP ou que podem ser montadas no próprio browser do usuário, utilizando para isso frameworks e bibliotecas como Angular e React. A imagem abaixo ilustra a separação entre frontend e backend.

**Figura 1 – Frontend vs Backend.**



Quando um usuário acessa um e-commerce através da web, a interface que ele está navegando, buscando produtos, adicionando produtos ao carrinho e fazendo compras representa o frontend da aplicação. Neste exemplo, o backend é quem envia para o frontend a lista dos produtos, finaliza a compra, retira a quantidade comprada do estoque, entre outras funções. Todas as vezes que a interface do frontend precisa buscar ou enviar dados, ela tem que se comunicar com o backend. Uma maneira muito comum de se fazer essa comunicação é através da utilização de APIs, que é o principal assunto desta disciplina.

API é a sigla para *Application Programming Interface* e é basicamente um conjunto de serviços que são expostos de forma a permitir a comunicação entre sistemas. Através de uma API, uma aplicação pode acessar recursos de outra sem precisar saber como eles foram implementados. A aplicação que está expondo seus recursos pode controlar quais serão publicados, o que será realizado a partir de seu consumo, quem poderá acessar suas funcionalidades e quais poderão ser acessadas. Uma API pode ser vista como um contrato, definindo uma espécie de acordo entre as partes interessadas, sendo este acordo representado pela documentação da API. Ao enviar uma requisição estruturada em um formato específico, a aplicação irá entender o que está sendo solicitado, irá processar a informação e devolver uma resposta ao solicitante.

Os serviços que fazem parte de uma API também são chamados de webservices. Eles são utilizados para transferir dados através de protocolos de comunicação para diferentes plataformas, independentemente do seu sistema operacional e linguagem de programação.

Um webservice somente transmite as informações, não sendo por si só uma aplicação possível de ser acessada pela web. Eles permitem reutilizar funcionalidades de sistemas já existentes, sem que seja preciso implementá-las do zero. Por isto ele é muito utilizado para fazer integração de sistemas, pois uma aplicação pode expor suas funcionalidades internas a outras aplicações, de forma que as outras a utilizem apenas sabendo o contrato que deve ser seguido na chamada do método. Dessa forma a aplicação que expõe os serviços pode controlar o que está sendo fornecido e o que está sendo alterado internamente, sem que os consumidores saibam o que está ocorrendo. Ultimamente muitas empresas estão criando APIs bem documentadas de suas aplicações, de forma a possibilitar a fácil integração de outras aplicações.

Existem várias vantagens na utilização de webservices. Uma delas é esta facilidade de integração de sistemas, pois o funcionamento de um webservice depende apenas do protocolo HTTP e um formato como XML ou JSON, sendo assim possível trocar informações entre dois sistemas de modo que um possa não saber nada do outro.

Outra vantagem é a reutilização de código provida pelo uso de webservices, uma vez que uma funcionalidade desenvolvida em determinada linguagem e aplicação pode ser reaproveitada por qualquer outra aplicação, independentemente da sua plataforma e linguagem. Uma vez que dentro da empresa já existe uma API de serviços, o tempo de desenvolvimento também é reduzido visto que diversas funcionalidades serão reaproveitadas sem a preocupação de serem implementadas do zero.

Outra vantagem é a segurança provida pelo webservice, pois ele evita que as aplicações que estão integrando acessem o banco de dados diretamente. Dessa forma, dentro do webservice podem existir várias validações sobre a regra de negócio que está sendo executada, de forma que a alteração seja executada no banco de dados apenas caso tudo esteja correto. Se uma aplicação permite que outra acesse seu banco de dados diretamente, ele perde esse poder de controlar o que está sendo alterado, podendo assim gerar diversos bugs. Dependendo da necessidade de integração dos sistemas da empresa, esta integração pode ser feita unicamente através de webservices, o que reduziria o custo de se desenvolver uma aplicação sob medida exclusivamente para fazer as integrações.

Hoje uma das principais tecnologias utilizadas na implementação de webservices é o REST. O REST faz o uso de um URI (*Uniform Resource Identifier*) para fazer uma chamada de serviço. As URIs são as interfaces de utilização dos webservices e funcionam como um contrato que será utilizado pelos consumidores para poderem usá-las. Por exemplo, uma chamada REST para buscar os dados de um cliente que possui id igual a 2 poderia ser: `http://www.teste.com.br/clientes/2`. REST é a abreviação para



*Representational State Transfer*, e foi descrito por Roy Fielding, que é um dos principais criadores do protocolo HTTP. Ele não é um protocolo nem um padrão, mas sim um estilo arquitetural baseado no HTTP.

As URIs que uma API disponibiliza também são conhecidas como seus endpoints. Os recursos gerenciados por uma aplicação, que são identificados unicamente por meio de seus endpoints, podem ser manipulados de várias formas. Geralmente é possível criar um registro através de um endpoint, atualizar um registro, excluí-lo dentre outras operações. Alguns endpoints podem ser somente de consulta, por exemplo. Quando um cliente faz uma requisição HTTP para um serviço, além do endpoint desejado ele também deve informar o tipo de manipulação que deseja fazer no recurso. O tipo de manipulação é definido de acordo com o método do protocolo HTTP que é informado. O protocolo HTTP possui vários métodos, cada um destinado a um tipo de operação, enquanto isso o serviço REST deve identificá-los de forma a manipular o recurso da forma correta.

## Node.js

---

O Node.js foi criado por Ryan Dahl mais 14 colaboradores em 2009. Ele foi criado na tentativa de resolver o problema de arquiteturas bloqueantes. Plataformas como .NET, Java, PHP, Ruby ou Python paralisam um processamento enquanto realizam um processo de I/O no servidor. Esta paralisação é o chamado modelo bloqueante (Blocking-Thread). Neste tipo de arquitetura cada requisição que chega no servidor é enfileirada e processada uma a uma, não sendo possível múltiplos processamentos delas. Enquanto uma requisição é processada, as demais ficam ociosas em espera. Exemplos de tarefas de entrada e saída de dados que bloqueiam as demais são tarefas como enviar e-mail, fazer consulta no banco de dados e leitura em disco. Para amenizar

o problema gerado por essa espera, esses servidores criam várias threads para darem vazão à fila de espera. Quando se aumenta muito o número de acessos ao sistema, é preciso fazer upgrade nos hardwares de forma a ser possível rodar mais threads, gerando um custo maior.

**Figura 2 – Node.js**



Fonte: <https://nodejs.org/en/>.

O Node.js possui uma arquitetura não bloqueante (non-blocking thread) e apresenta uma boa performance em consumo de memória, utilizando ao máximo o poder de processamento dos servidores. Nele as aplicações são single-thread, ou seja, cada aplicação possui um único processo. Assim ele utiliza bastante a programação assíncrona, com o auxílio das funções de callback do JavaScript. Quando uma requisição está fazendo uma ação de I/O, ele a deixa executando em background e vai processando outras requisições. Assim que o processamento do I/O termina, é disparado um evento e o callback correspondente a requisição, ou seja, a função que deve ser executada com resultado da resposta, é colocada na fila para ser executada assim que possível. Em uma arquitetura bloqueante, o jeito de lidar com essa concorrência seria criar múltiplas threads para lidar com as diversas requisições. O problema é que dentro de cada thread o tempo gasto com espera de resposta não é aproveitado, deixando aquela parte da CPU ociosa.

O Node.js foi criado utilizando o V8, que é um motor JavaScript de código aberto, criado pela Google e utilizado no seu navegador, o Google Chrome. Ele compila

e executa o código em JavaScript além de manipular a alocação de memória para objetos, descartando o que não é mais preciso. Sendo assim, podemos dizer que o Node.js é basicamente um interpretador de código JavaScript. A diferença do Node.js para a codificação habitual do JavaScript é que o JavaScript sempre foi uma linguagem cliente-side, sendo executada somente no lado do usuário, em seu browser. Com o Node.js é possível executar o código JavaScript no servidor. Ele mantém um serviço rodando no servidor, que faz a interpretação e execução de códigos JavaScript. O V8 do Google é multiplataforma, sendo possível sua instalação em vários sistemas operacionais.

A criação do Node.js está muito ligada com a crescente utilização das SPAs, nas quais seu desenvolvimento é feito principalmente em JavaScript, pois a partir do momento em que é possível trabalhar com a mesma linguagem de programação tanto no backend quanto no frontend, muitas empresas se interessaram pela tecnologia devido a possibilidade do reaproveitamento de conhecimento.

### Node.js Event Loop

---

O Node.js é uma plataforma baseada em eventos. Isso significa que tudo que acontece no Node.js é uma reação a um evento. Ele segue a mesma filosofia de orientação de eventos do JavaScript, porém nele não há eventos de componentes do HTML, como por exemplo click do mouse. Uma transação processada passa por várias callbacks. Esse processo é abstraído do desenvolvedor, sendo realizado por uma biblioteca open-source desenvolvida em C, chamada libuv, que provê o mecanismo chamado Event Loop.

O Node.js trabalha dessa forma porque operações de I/O são muito lentas. Abrir um arquivo de 1GB em um editor de texto pode fazer com que o computador trave

por um tempo razoável. Operações de rede também são lentas e fazem com que o processo tenha que esperar uma resposta. É por isso que o Node.js trabalha com assincronismo, permitindo que seja desenvolvido uma aplicação orientada a eventos, graças ao Event Loop.

O Event Loop basicamente é um loop infinito, que a cada iteração verifica se existem novos eventos em sua fila de eventos. Os eventos entram nessa fila quando são emitidos durante a utilização da aplicação. O módulo responsável por emitir eventos é o Events. A maioria das bibliotecas do Node.js herdam deste módulo as funcionalidades de eventos, como os métodos emit e listen. Quando um código emite um evento, ele é enviado para a fila de eventos, para que o Event Loop possa executá-lo e posteriormente retornar seu callback. Esta callback pode ser executada através de uma função de escuta, chamada on(). A figura abaixo ilustra o funcionamento do Event Loop.

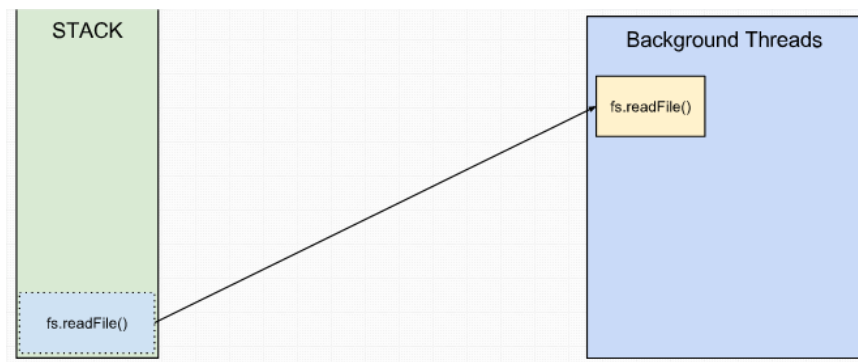
**Figura 3 – Node.js Event Loop.**



Fonte: [https://imasters.com.br/?attachment\\_id=50021](https://imasters.com.br/?attachment_id=50021).

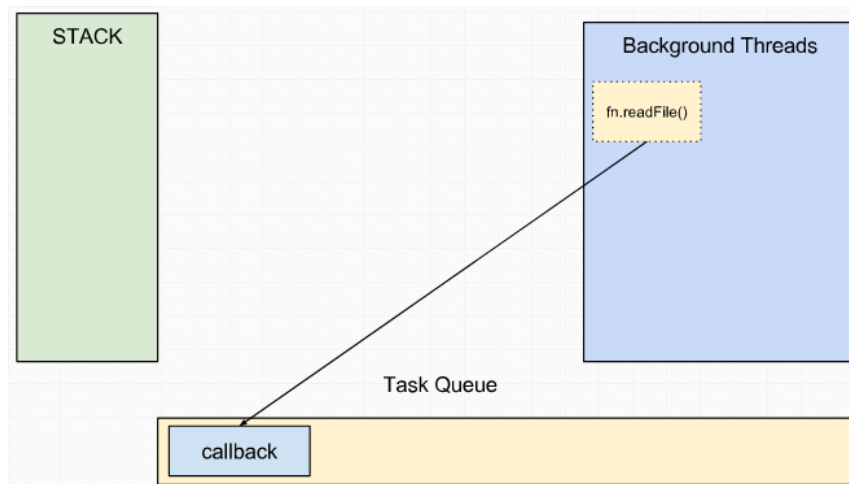
O V8 que é a base do Node.js é single-threaded. Dessa forma, quando são executadas ações de I/O que demandaram tempo, como `http.get` ou `fs.readFile`, o Node.js envia essas operações para outra thread do sistema, permitindo que a thread do V8 continue executando o código da aplicação. O Event Loop possui uma stack e sempre que um método é chamado ele entra na stack para aguardar seu processamento, pois é executada apenas uma função por vez. Após a outra thread do sistema executar a tarefa I/O, ele envia essa tarefa para a Task Queue, ou seja, ele volta aquela task para a fila de execução do Node.js, para que possa dar continuidade a seu processamento. A sequência na imagem abaixo ilustra esse fluxo.

**Figura 4 – Node.js Event Loop – Parte 1.**



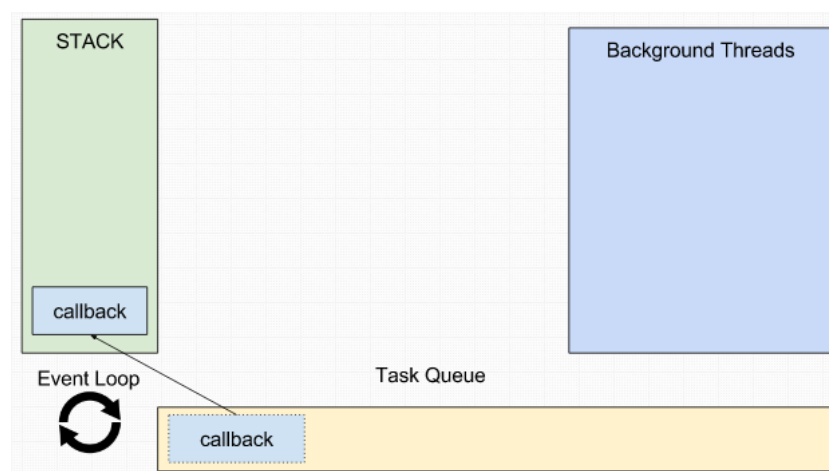
Fonte: <http://walde.co/2016/11/27/node-js-o-que-e-esse-event-loop-afinal/>.

Figura 5 – Node.js Event Loop – Parte 2.



Fonte: <http://walde.co/2016/11/27/node-js-o-que-e-esse-event-loop-afinal/>.

Figura 6 – Node.js Event Loop – Parte 3.

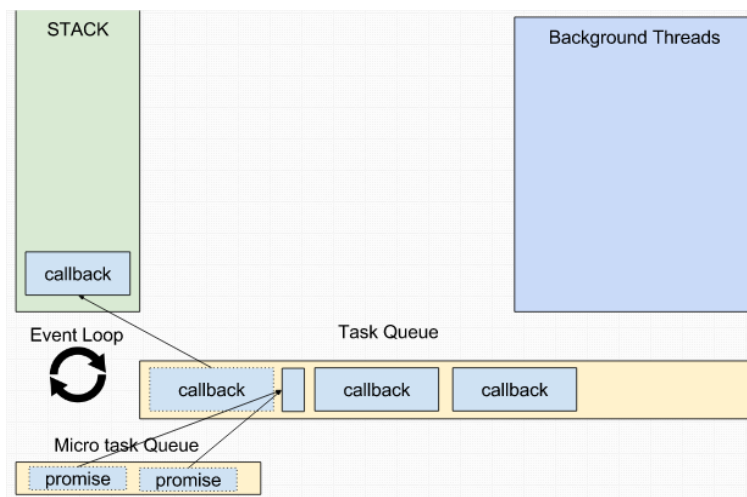


Fonte: <http://walde.co/2016/11/27/node-js-o-que-e-esse-event-loop-afinal/>.

Na Task Queue há dois tipos de tasks, as micro tasks e as macro tasks. Somente as macro tasks devem ser processadas em um ciclo do Event Loop. Exemplo de macro tasks são setTimeout, I/O e setInterval. As micro tasks são tarefas que devem ser executadas rapidamente após alguma ação, sem a necessidade de inserir uma nova task

na Task Queue. Após o Event Loop processar uma macro task da Task Queue, ele deve processar todas as micro tasks disponíveis antes de chamar outra macro task. A figura abaixo ilustra esse comportamento.

**Figura 7 – Event Loop Macro e Micro Tasks.**



Fonte: <http://walde.co/2016/11/27/node-js-o-que-e-esse-event-loop-afinal/>.

## Módulos do Node.js

Um módulo no Node.js é o mesmo que uma biblioteca no JavaScript. É um conjunto de funções que podem ser incluídas em uma aplicação. O Node.js segue o CommonJS, uma especificação de ecossistemas para o JavaScript. Assim é possível incluir um módulo que está em outro arquivo utilizando a função chamada `require`. É possível criar um módulo e importá-lo em outro arquivo facilmente. No próximo capítulo veremos uma outra forma de realizar exportação e importação de arquivos. A figura abaixo ilustra o formato de exportar e importar módulos. A primeira parte da imagem mostra a exportação de uma função chamada `testFunction`, que retorna uma string,

enquanto na segunda parte da imagem é realizada a importação deste módulo e posteriormente a chamada da função criada, imprimindo o resultado no terminal.

**Figura 8 – Node.js**

```
JS test-module.js > ...
1  exports.testFunction = function () {
2    return "Test function done";
3  };

JS index.js > ...
1  var test = require('./test-module');
2  console.log(test.testFunction());
```

O Node.js possui vários módulos nativos muito úteis. Vamos aqui sobre alguns deles, como o HTTP, File System e Events.

O Node.js possui o módulo nativo chamado HTTP, que permite transferir dados através do protocolo HTTP (Hyper Text Transfer Protocol). Este módulo consegue criar um servidor HTTP capaz de escutar portas do servidor e enviar respostas de volta ao cliente. Uma requisição do tipo abaixo retornaria como resposta 2017 July.

`http://localhost:8080/?year=2017&month=July`

Geralmente as aplicações Node não utilizam este módulo diretamente, é mais comum a utilização do Express, que provê várias facilidades para trabalhar com HTTP. No próximo capítulo falaremos sobre este framework. A figura abaixo ilustra uma utilização do HTTP do Node.js, que faz o parse de informações enviadas na URL.



**Figura 9 – Node.js HTTP.**

```
var http = require('http');
http.createServer(function (req, res) {
  if ((req.method === 'GET') && (req.url === '/test')) {
    res.write('GET /test');
  } else {
    res.write('Hello World!');
  }
  res.statusCode = 200;
  res.end();
}).listen(8080);
```

Outro módulo muito utilizado do Node.js é o File System. Ele permite trabalhar com arquivos, fazendo ações como ler, criar, atualizar, excluir e renomear arquivos. Para ler um arquivo é utilizado a função `readFile()`. Para criar um arquivo existe o método `appendFile()`, que cria um arquivo caso ele não exista, e se existir ele adiciona o conteúdo ao mesmo. Já o método `writeFile()` sobrescreve o arquivo caso ele exista e se não existir ele é criado. Para apagar um arquivo é utilizada a função `unlink()` e para renomear um arquivo a função `rename()`. No capítulo 3 deste módulo essas funções serão utilizadas para o desenvolvimento de uma API.

A figura abaixo ilustra um exemplo de leitura e retorno de conteúdo. Neste exemplo é realizada a leitura de um arquivo e caso ocorra algum erro durante a leitura, como por exemplo o fato de arquivo não existir, é impressa a mensagem de erro no terminal, caso contrário se a leitura tiver sido realizada com sucesso, o conteúdo do arquivo é impresso no terminal.

**Figura 10 – Node.js File System.**

```
var fs = require('fs');
fs.readFile('./test-file.txt', 'utf-8', function (err, data) {
  if (err) {
    console.log(err.message);
  } else {
    console.log(data);
  }
});
```

O módulo Events permite criar, disparar e escutar eventos. No Node.js qualquer ação é um evento, como por exemplo uma conexão realizada ou um arquivo aberto. O objeto EventEmitter permite que sejam adicionadas callbacks para os eventos. Para disparar um evento basta chamar o método emit e para escutar um evento o método on. A figura abaixo ilustra este comportamento. É criado um objeto chamado eventEmitter, e a partir dele é implementada uma função para ser executada quando o evento “testEvent” for disparado. Na próxima linha é disparado esse evento, realizando assim a impressão da string no terminal.

**Figura 11 – Node.js File System.**

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

eventEmitter.on('testEvent', function () {
  console.log('Test event done');
});

eventEmitter.emit('testEvent');
```

Fonte: [https://www.w3schools.com/nodejs/nodejs\\_events.asp](https://www.w3schools.com/nodejs/nodejs_events.asp).

O Node.js possui diversos módulos nativos além dos aqui citados, como por exemplo para enviar e-mails e fazer upload de arquivos. Caso deseje aprofundar

melhor no assunto, sugiro a documentação oficial e o site w3schools, nos links <https://nodejs.org/api/index.html> e <https://www.w3schools.com/nodejs/default.asp>.

## NPM

---

NPM é a sigla para *Node Package Manager*, é o gerenciador de pacotes do Node. Ele é um repositório online para publicação de projetos de código aberto, que podem ser utilizados por outros desenvolvedores em seus projetos. Através de sua ferramenta de linha de comando ele interage facilmente com o repositório online, auxiliando na instalação dos pacotes, gerenciamento de versão e gerenciamento de dependências. Ele possui milhares de bibliotecas publicadas, algumas bem famosas e que são utilizadas por muitas aplicações. Com o NPM é possível adicionar bibliotecas a uma aplicação através do seguinte comando:

```
npm install nome-da-biblioteca
```

Esse comando irá buscar no repositório a biblioteca desejada, fará o download da biblioteca para sua aplicação, colocando-a dentro de uma pasta chamada `node_modules`, na raiz da aplicação. Depois disso é possível importar a biblioteca dentro do código e utilizá-la normalmente como se fossem módulos internos do projeto. Caso deseje utilizar a biblioteca através da linha de comando, é possível instá-la globalmente, bastando adicionar a flag `-g` no final do comando de instalação. Para desinstalar a biblioteca basta executar o comando:

```
npm uninstall nome-da-biblioteca
```

Outro ponto importante é que com ele é permitido gerenciar as dependências do projeto. Grande parte dos frameworks JavaScript funcionam em conjunto com o NPM. Uma aplicação que utiliza o NPM como gerenciador de dependências deve possuir

um arquivo chamado `package.json`. Dentro deste arquivo estarão listadas todas as dependências do projeto. Ao executar o comando `npm install` dentro do diretório do projeto, ele irá verificar as dependências do projeto neste arquivo e irá instalar as que ainda não estão no `node_modules`. Ao lado do nome do pacote também é informado sua versão que será baixada. Caso na frente da dependência exista um símbolo `^`, ele irá verificar se existe uma versão mais nova que a especificada, e se sim fará a atualização. Para instalar uma versão específica de uma dependência, basta inserir um `@` após o nome do pacote com o número da versão, por exemplo:

```
npm install nome-da-biblioteca@1.5.3
```

Para atualizar um pacote basta executar o comando de `update`, por exemplo:

```
npm update nome-da-biblioteca
```

A partir da versão 5 do NPM foi adicionado um arquivo chamado `package-lock.json`. Ele serve para garantir a consistência das dependências entre as máquinas. Por exemplo, caso no `package.json` de quem está desenvolvendo o projeto tenha um pacote na versão 1.1.2, e esteja com `^`. Vamos supor que no momento em que o desenvolvedor instalou o pacote, esta era a versão mais atual. Vamos supor que daqui a 2 meses outra pessoa vai começar a trabalhar no projeto, e que saiu uma versão 1.1.3 do pacote com algumas correções. Ao rodar um `npm install` na sua máquina, o pacote baixado seria o 1.1.3, e assim a máquina desta pessoa ficaria diferente da do outro desenvolvedor. Isso pode gerar problemas, pois pode ser que na nova versão apareceu um bug que não tinha na anterior, por exemplo. Dessa forma o `package-lock.json` mantém a última versão que foi instalada, a localização do pacote e um código hash para verificar sua integridade. Assim garante-se que as duas máquinas estão com as dependências na mesma versão, evitando que ocorram problemas de compatibilidade. É sugerido que o `package-lock.json` seja sempre versionado, de forma a manter um

histórico e evitar que problemas de compatibilidade dificultem o desenvolvimento do projeto.

Geralmente, ao versionar os projetos no GIT ou SVN por exemplo, a pasta `node_modules` não é versionada, pois geralmente ela é bem grande. Ao baixar o projeto para uma máquina nova, basta executar o comando `npm install` que as dependências serão baixadas para a máquina em questão. Ao instalar uma nova dependência no projeto ela é automaticamente incluída no `package.json`. O arquivo `package.json` possui vários atributos, segue abaixo uma breve explicação sobre alguns deles:

- `name`: nome do pacote.
- `version`: versão do pacote.
- `description`: descrição do pacote.
- `homepage`: site do pacote.
- `author`: nome do autor do pacote.
- `contributors`: nome das pessoas que contribuíram no pacote.
- `dependencies`: lista de todas as dependências do pacote, que serão instaladas na pasta `node_modules` do projeto pelo comando `npm install`.
- `devDependencies`: dependências somente de desenvolvimento. Se o comando for executado com a flag `--production`, os pacotes listados aqui não serão instalados. Um pacote é listado aqui quando instalado com a flag `--save-dev`.
- `repository`: o tipo do repositório e a URL do pacote.
- `main`: o ponto de entrada do pacote.

- keywords: palavras-chave do pacote.

A imagem abaixo mostra um exemplo de um package.json simples.

**Figura 12 – NPM package.json.**

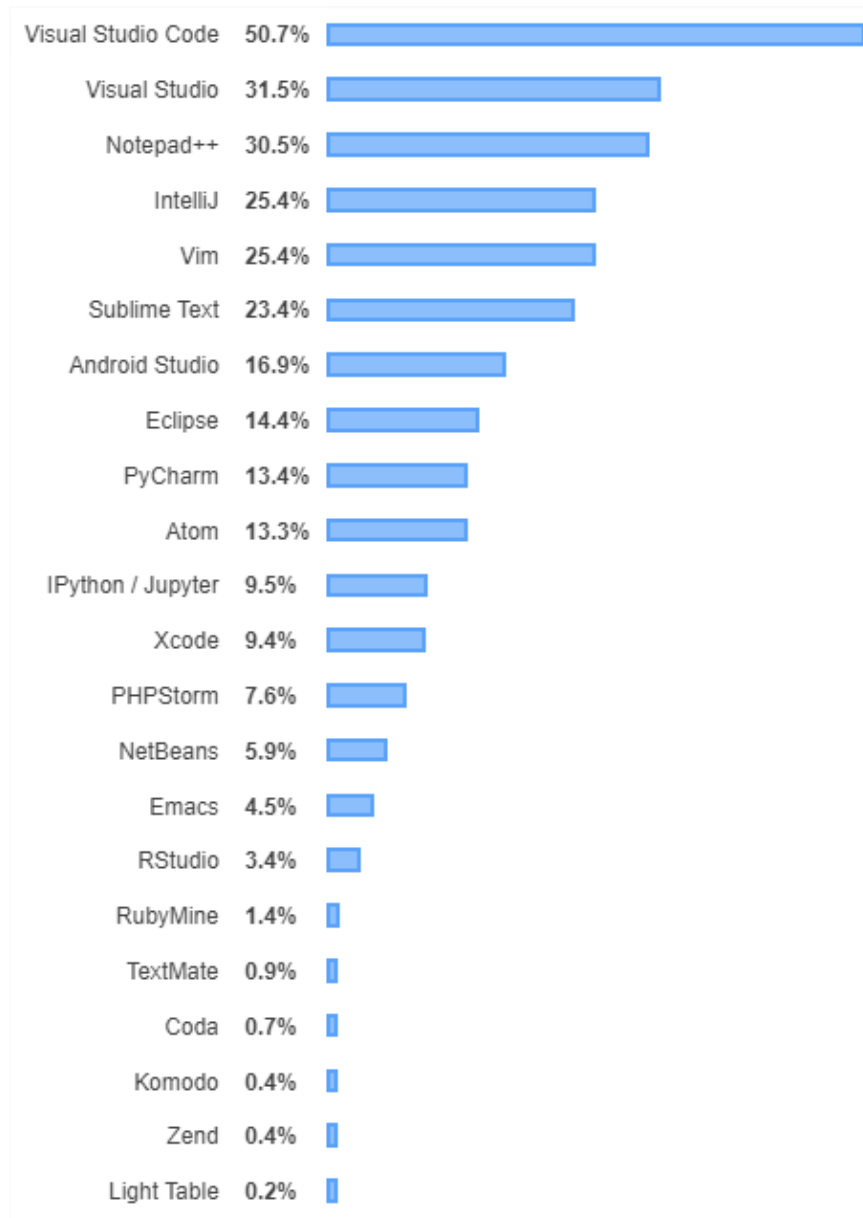
```
{
  "name": "hello-world",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

## IDE de desenvolvimento

---

Assim como as demais tecnologias baseadas em JavaScript, não existe uma IDE ou editor de texto específico que precisa ser utilizado. O desenvolvedor pode ficar à vontade para escolher o editor que ele tem maior familiaridade. A imagem abaixo mostra as IDEs mais utilizadas pelos desenvolvedores em 2019. Estes dados foram extraídos de uma pesquisa do Stack Overflow.

**Figura 13 – IDEs mais utilizadas em 2019.**

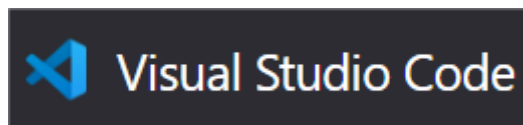


**Fonte:** <https://insights.stackoverflow.com/survey/2019>.

Como pode-se observar, o Visual Studio Code é o mais utilizado pelos desenvolvedores de forma geral. Caso você não tenha familiaridade com nenhum outro editor, sugiro que inicie por ele. O Visual Studio Code, ou VS Code, é uma ferramenta

open-source elaborada pela Microsoft. Ele possui vários plugins e integrações interessantes que podem facilitar seu trabalho. As aulas gravadas e exemplos desta disciplina utilizaram este editor, mas você pode utilizar outro caso deseje.

**Figura 14 – Visual Studio Code.**



Fonte: <https://code.visualstudio.com/>.

#### Ferramentas para consumo de endpoints

---

Uma prática muito comum dos desenvolvedores é, à medida que vão sendo desenvolvidos os endpoints, eles já vão sendo testados a partir de chamadas realizadas em uma aplicação real, ou senão a partir de ferramentas de consumo. Ferramentas para consumo de endpoints REST facilitam bastante o trabalho pois não exigem que o desenvolvedor entre na aplicação que consome o endpoint, faça todo o fluxo de telas até chegar no ponto e condição desejada para consumo do endpoint. Basta que sejam criados os consumos de cada endpoint, colocado os parâmetros, e a ferramenta irá deixar salvo para quando o desenvolvedor voltar, ele já poder fazer as chamadas rapidamente, alterando os parâmetros da forma desejada.

Existem diversas ferramentas que tem esse objetivo, nesta disciplina foi utilizada para testes a Insomnia. Ela é uma ferramenta open-source, que tem suporte para Windows, Linux e Mac, e pode ser baixada pelo próprio site (<https://insomnia.rest/>).



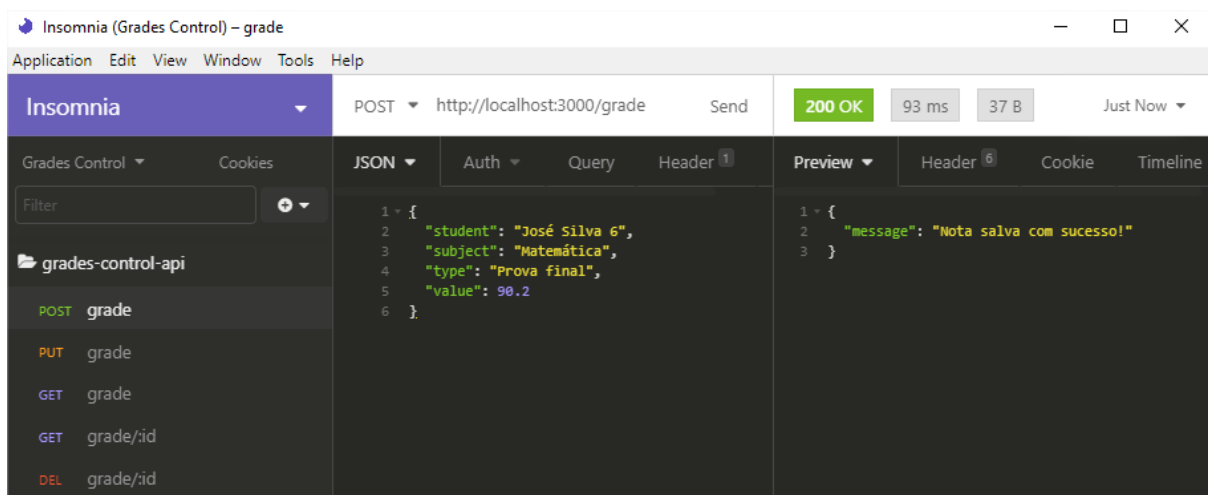
**Figura 15 – Insomnia.**



Fonte: <https://insomnia.rest/>.

A imagem abaixo mostra o ambiente a tela do Insomnia. Para criar uma requisição basta clicar no botão de adicionar na coluna da esquerda. É possível organizar as requisições em pastas, podendo assim manter salvo endpoints de projetos diferentes de uma forma mais organizada. Ao criar uma requisição, basta escolher um nome para ela e o tipo do método HTTP. Após criada, na coluna do meio da imagem é possível alterar a URL, e na parte de baixo colocar informações a serem enviadas no body da requisição. Por exemplo, ao executar um POST, é comum que os dados sejam enviados no body em formato JSON. Para executar a requisição basta apertar as teclas Ctrl e Enter, ou senão clicar no botão “Send”. O resultado da execução é mostrado na coluna da direita, juntamente com o status HTTP retornado.

**Figura 16 – Ambiente do Insomnia.**



## Capítulo 2. Express

---

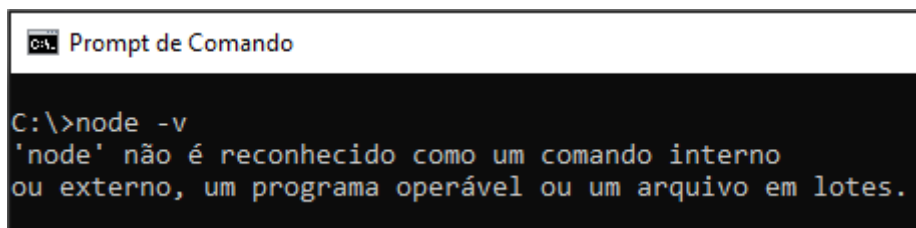
ExpressJS é um framework web para o Node.js. Ele provê várias funcionalidades que fazem o desenvolvimento de aplicações mais rápido e fácil ao ser comparado com o desenvolvimento somente com o Node.js. Em seu site ele se autodescreve com um framework web rápido, flexível e minimalista para o Node.js. Existem várias vantagens na utilização do Express para os projetos do back-end. Ele facilita o roteamento da aplicação, baseado nos métodos HTTP e URLs. Roteamento refere-se à definição de endpoints (URIs) e como eles respondem às solicitações do cliente.

### Instalação

---

Antes de instalar o Express, é necessário ter o Node.js instalado no computador. Para verificar se você já o tem instalado na máquina, abra o terminal e execute o comando `node -v`. Se o retorno for igual parecido com o da imagem abaixo, indica que você ainda não o tem instalado. Se mostrar o número da versão, indica que ele já está instalado.

**Figura 17 – Verificação da instalação do Node.js.**



```
C:\>node -v
'node' não é reconhecido como um comando interno
ou externo, um programa operável ou um arquivo em lotes.
```

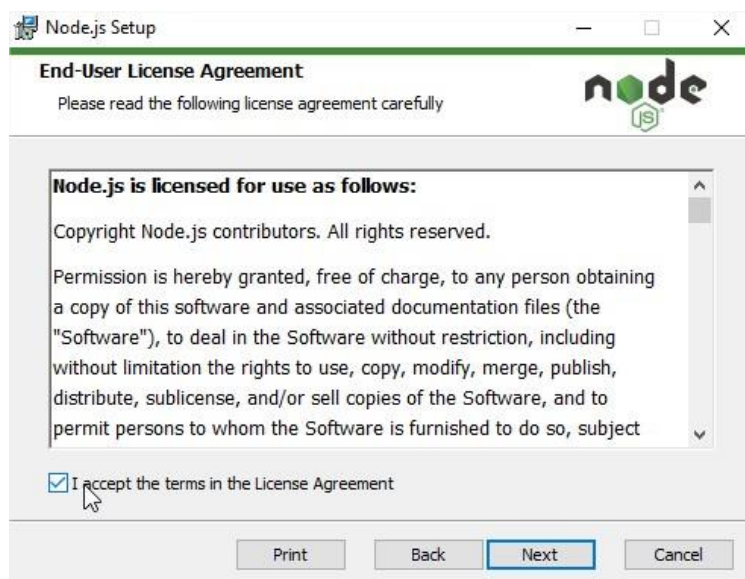
Para realizar a instalação, basta baixar o arquivo de instalação correspondente ao seu sistema operacional diretamente no site do Node.js (<https://nodejs.org/en/>) e seguir o passo a passo da instalação. A instalação é bem simples e consiste em ir

avanzando no programa de instalação. Durante a instalação o Windows pode pedir permissão, forneça ela para poder continuar. As imagens abaixo mostram o passo a passo do instalador no Windows.

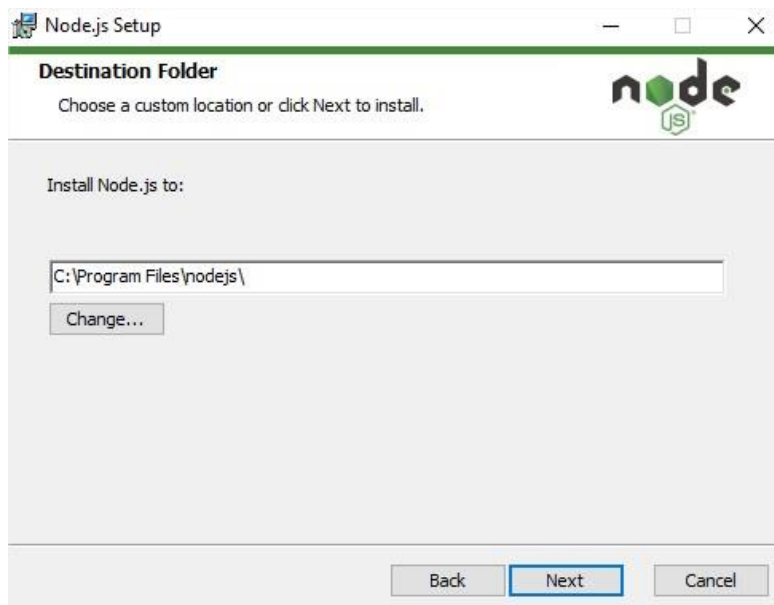
**Figura 18 – Passo 1 da instalação do Node.js.**



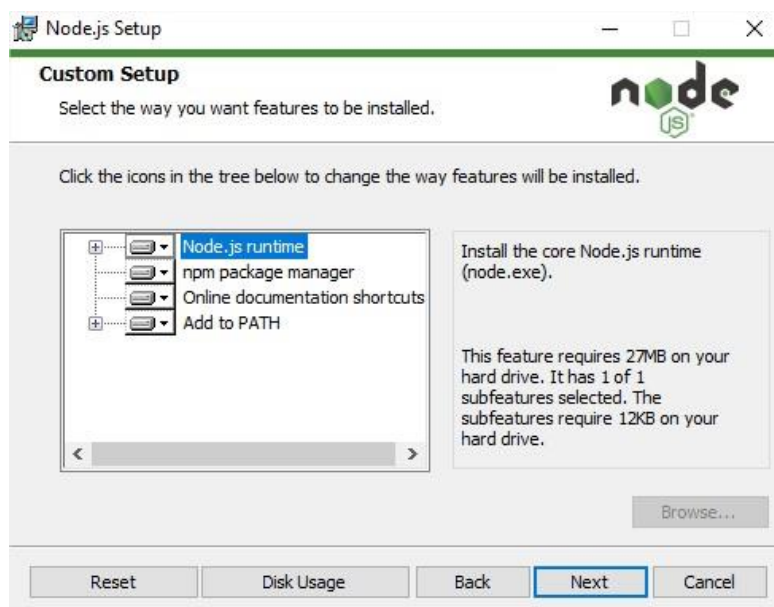
**Figura 19 – Passo 2 da instalação do Node.js.**



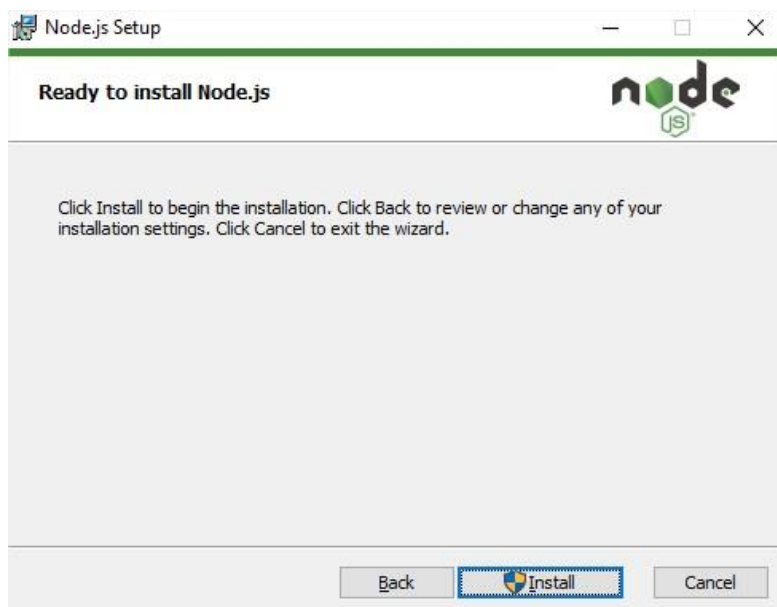
**Figura 20 – Passo 3 da instalação do Node.js.**



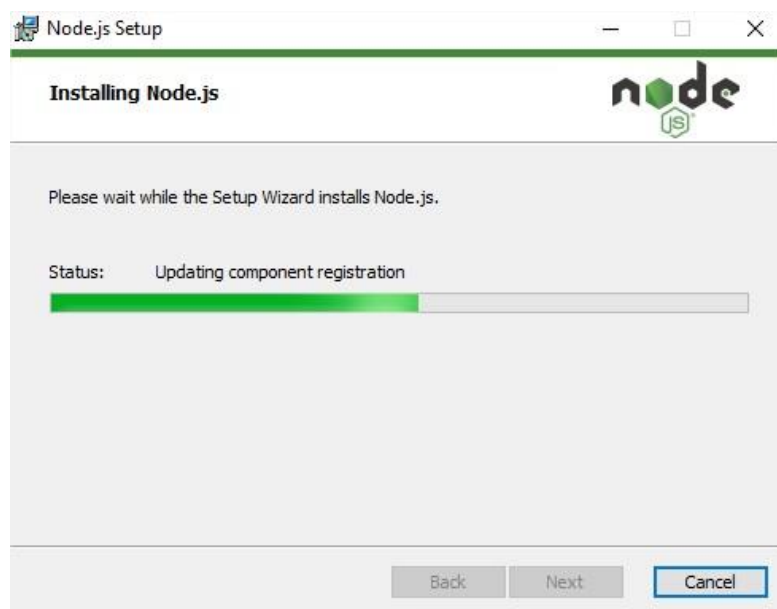
**Figura 21 – Passo 4 da instalação do Node.js.**



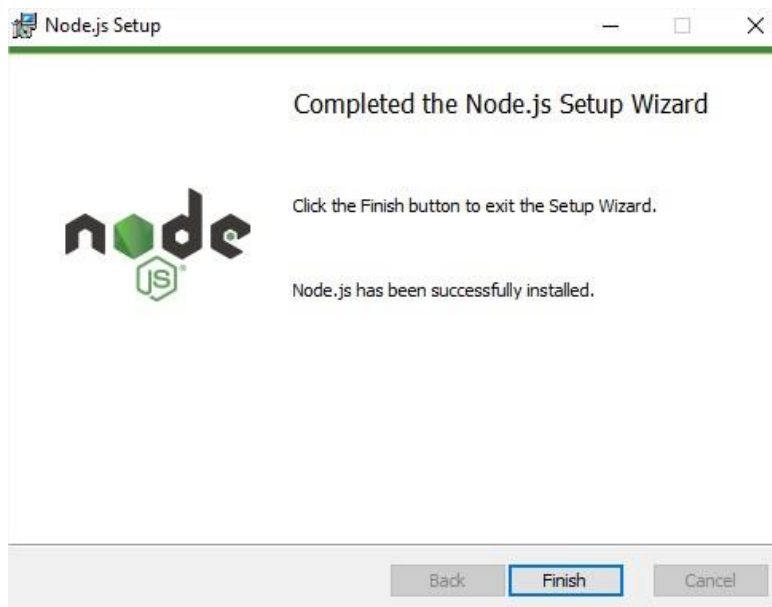
**Figura 22 – Passo 5 da instalação do Node.js.**



**Figura 23 – Passo 6 da instalação do Node.js.**



**Figura 24 – Passo 7 da instalação do Node.js.**



Para conferir se a instalação deu certo, basta executar novamente o comando `node -v` no terminal para verificar a versão instalada. Nesta disciplina foi utilizada a versão 12.14.0. Verifique também se o NPM foi instalado através do comando `npm -v`. A instalação dele vem junto do instalador do Node.js. A imagem abaixo mostra a verificação da instalação do Node.js e NPM.

**Figura 25 – Verificação instalação do Node.js e NPM.**

```
C:\> node -v
v12.14.0

C:\> npm -v
6.13.4

C:\>
```

Para instalar o Express, primeiro é necessário iniciar um projeto Node e depois o Express será adicionado como dependência neste projeto. Ao longo dessa disciplina criaremos uma API chamada `grades-control-api`, que terá por objetivo controlar o cadastro de alunos e lançar suas notas. Cria uma pasta no local que ficará seu projeto e execute o comando abaixo:

```
npm init
```

Ele irá fazer uma série de perguntas, inicialmente pode apertar enter para todas as perguntas, pegando assim o valor padrão. No fim do processo seu projeto estará criado, para instalar o Express basta executar o comando abaixo dentro da pasta do projeto:

```
npm install express
```

Esse comando irá fazer a instalação do Express no projeto utilizando o NPM. Você pode observar que foi criado um registro para o Express no `package.json`, além de uma pasta chamada `express` dentro da pasta `node_modules`. A versão do Express utilizada nesta disciplina foi a 4.17.1. A imagem abaixo mostra o estado atual do `package.json` para este projeto.

**Figura 26 – Package.json do projeto.**

```
{
  "name": "grades-control-api",
  "version": "1.0.0",
  "description": "",
  "main": "src/app.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

## Hello World

---

Uma rota define a forma como a aplicação responde a requisição de um cliente para determinado endpoint, que é composto por uma URI (Uniform Resource Identifier) e o método HTTP da requisição, como GET ou POST por exemplo. O Express define uma rota da seguinte forma:

```
app.method(path, handler)
```

No comando acima, o `app` é uma instância do Express, *method* é um método HTTP, *path* o caminho que a rota irá responder e *handler* a função que será executada quando a rota for atingida.

Vamos agora criar um exemplo simples para conferir se o ambiente está funcionando. Dentro do projeto criado na seção anterior, crie uma pasta na raiz do projeto chamada `src` e lá dentro um arquivo chamado `app.js`. Ele será o arquivo raiz da nossa aplicação, contendo a definição das rotas. A localização deste arquivo é opcional, nos exemplos desta disciplina será utilizada esta organização. Dentro desse arquivo, digite o código da abaixo:

**Figura 27 – Hello World Express.**

```
1  const express = require('express')
2  const app = express()
3  const port = 3000
4
5  app.get('/', (req, res) => res.send('Hello World!'))
6
7  app.listen(port, () => console.log(`App listening on port ${port}!`))
```

No código acima podemos contar que na primeira linha é realizada a importação do Express, que já foi instalado no projeto utilizando o NPM, através do



método `require`. Na linha 2 é criado um objeto do Express chamado `app`, que irá ser utilizado para configurar a aplicação. Este objeto possui os métodos para configurar as rotas de requisições HTTP e configurar os `middlewares`, por exemplo. Na linha 3 é criada uma variável que mais à frente será a porta utilizada pela API. Na linha 5 é definida uma rota na raiz, ou seja, quando alguém acessar o endereço `http://localhost:3000` através de um GET será retornado o texto “Hello World!”. A linha 7 inicia a API de fato, utilizando para isso o método `listen` do Express. O primeiro parâmetro define a porta que será utilizada e o segundo é a função que será executada quando a inicialização for realizada.

Para testar se o projeto está funcionando, basta executar o comando abaixo dentro da pasta do projeto:

```
node src/app.js
```

Se tudo estiver correto, o texto “App listening on port 3000!” será impresso no console, e ao acessar o endereço <http://localhost:3000> em um browser será retornado o texto “Hello World!”.

### Configurações iniciais

---

Nos exemplos anteriores utilizamos o método `require` para realizar a importação do Express, para que ele pudesse ser utilizado no projeto. Essa forma de importação de módulos é realizada através do CommonJS. Uma outra forma de realizar esta importação de módulos é através dos módulos do ECMAScript 6, que são considerados o padrão do JavaScript para exportação e importação de módulos, e é muito utilizado nos frameworks e bibliotecas front-end. O Node.js recentemente inaugurou uma forma de possibilitar a utilização deste formato, bastando para isso habilitar uma flag. A tendência é que no futuro não seja preciso habilitar esta flag para

que possa funcionar, porém por enquanto é preciso. Para rodar através da linha de comando, basta colocar a flag “—experimental-modules” depois do comando do node. Nas versões mais recentes, inclusive na utilizada nas gravações desta disciplina, não é necessário passar essa flag mais para iniciar. Caso fosse preciso, ficaria assim:

```
node --experimental-modules src/app.js
```

Nos exemplos desta disciplina foi utilizado os módulos do ES6 para exportar e importar módulos, mas nada impede que sua aplicação utilize o CommonJS. Além desta flag, também é preciso incluir uma propriedade chamada “type” com o valor “module” dentro do package.json da aplicação, ficando como a imagem abaixo.

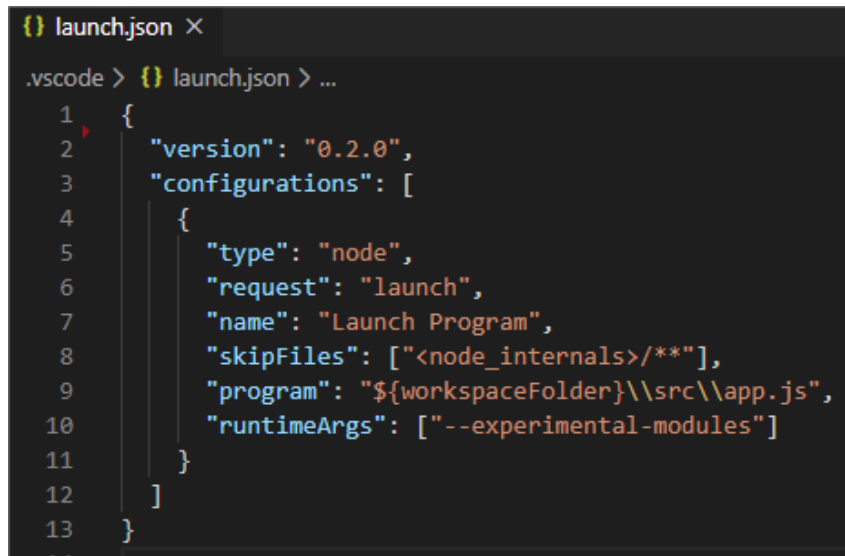
**Figura 28 – Package.json com propriedade type.**

```
{ } package.json > ...
1  {
2    "name": "grades-control-api",
3    "version": "1.0.0",
4    "description": "",
5    "main": "src/app.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "author": "",
10   "license": "ISC",
11   "dependencies": {
12     "express": "^4.17.1"
13   },
14   "type": "module"
15 }
```

Para que o processo de debug da aplicação possa ser realizado, é preciso acrescentar a flag “experimental-flags” no comando que é executado quando o projeto é iniciado pelo Visual Studio Code, ao invés de ser iniciado pela linha de comando. Para realizar essa configuração, edite o arquivo chamado launch.json que fica dentro da pasta

chamada ".vscode" na raiz do projeto, com o conteúdo da imagem abaixo. A propriedade "runtimeArgs" é quem fará o papel da flag.

**Figura 29 – Launch.json.**



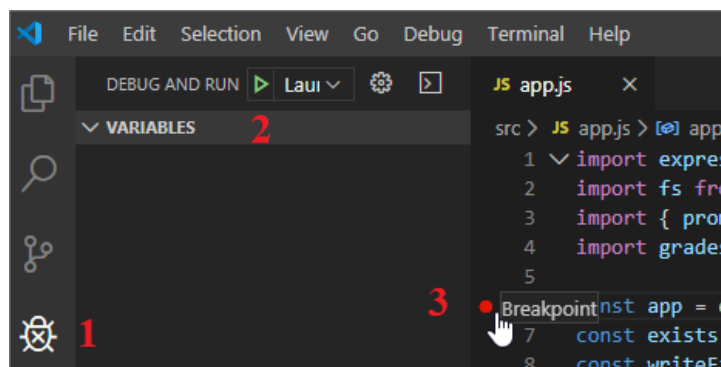
```

1 {
2   "version": "0.2.0",
3   "configurations": [
4     {
5       "type": "node",
6       "request": "launch",
7       "name": "Launch Program",
8       "skipFiles": ["<node_internals>/**"],
9       "program": "${workspaceFolder}\\src\\app.js",
10      "runtimeArgs": ["--experimental-modules"]
11    }
12  ]
13 }

```

Para debugar o projeto, basta clicar no ícone de um inseto na parte esquerda do Visual Studio Code e clicar no ícone verde de inicialização na aba superior. Para incluir um breakpoint, basta clicar no lado esquerdo do número da linha que deseja realizar a depuração. Após a depuração inicializada, uma barra de opções é mostrada na parte superior. A imagem abaixo ilustra esses três passos para iniciar o processo.

**Figura 30 – Iniciar debug.**



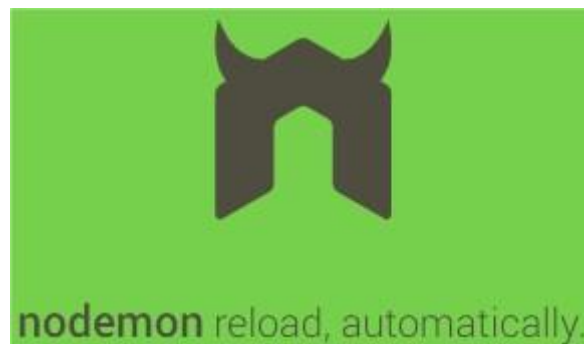
Uma biblioteca muito útil para trabalhar com desenvolvimento em Node.js é a Nodemon. Após realizar sua instalação utilizando o NPM e inicializar o projeto utilizando-a, ela então passa a monitorar o mesmo, e quando identifica alguma alteração ela automaticamente para e inicia a aplicação, facilitando assim o fluxo de desenvolvimento. Para instalar a ferramenta basta executar o comando abaixo:

```
npm install -g nodemon
```

Após instalado, basta trocar o comando *node* por *nodemon* na inicialização do projeto, ficando da forma abaixo:

```
nodemon src/app.js
```

**Figura 31 – Nodemon.**



Fonte: <https://nodemon.io/>.

É preciso informar ao Nodemon que ele deve desconsiderar alterações em alguns tipos de arquivos para efetuar a reinicialização da aplicação, como por exemplo arquivos de log. Para isso é preciso fazer a configuração da imagem abaixo no package.json do projeto.

**Figura 32 – Nodemon desconsiderar arquivos.**

```
"nodemonConfig": {  
  "ignore": [  
    "*.log",  
    "*.json"  
  ]  
},
```

Fonte: <https://nodemon.io/>.

## Rotas

---

Nesta seção vamos trabalhar mais um pouco com as rotas do Express. Vimos anteriormente como implementar uma rota simples, respondendo ao método GET na raiz da API.

Além de permitir criar rotas para cada tipo de método HTTP, como GET, POST, PUT e DELETE por exemplo, o Express também permite que seja utilizado o método “all”, interceptando assim todos os tipos de métodos. Por exemplo:

**Figura 33 – Express all.**

```
app.all('/testAll', (req, res) => {  
  res.send(req.method);  
});
```

Um caminho para uma rota pode ser uma string, um padrão de string ou uma expressão regular. O caso de uma string comum é o exemplo mais simples que estamos usando até o momento, já que a aplicação busca pelo nome exato da rota.

Uma outra forma de definir o caminho de uma rota é definindo um padrão, podendo ser utilizado os caracteres “?”, “+”, “\*” e “()”. Os caracteres “-” e “.” são interpretados normalmente, sem tratamentos especiais.

Segue abaixo a utilidade de cada um destes caracteres:

- O caractere “?” indica que a letra imediatamente anterior a ele é opcional. Assim, o Express irá responder a rotas que correspondam a string como um todo, mas que contenham ou não essa letra naquela posição.
- O caractere “+” indica que a letra imediatamente anterior a ela pode ser repetida diversas vezes naquela posição, que mesmo assim o Express irá interceptar a requisição e processá-la.
- O caractere “\*” indica que naquela posição pode ocorrer qualquer string, que desde que o restante da string esteja correta, a rota será processada.
- O caractere “()” indica que a string dentro dos parênteses será tratada como uma unidade, então no caso de no meio do caminho estiver os caracteres “(ab)?”, irá indicar que poderá ou não a parte da string “ab” estar contida ali naquela parte.

A imagem abaixo demonstra a utilização destes caracteres na definição de uma rota.

Figura 34 – Caracteres especiais.

```
app.all('/testAll', (req, res) => {
  res.send(req.method);
});

app.get('/teste?', (_, res) => {
  res.send('/teste?');
});

app.get('/buzz+', (_, res) => {
  res.send('/buzz+');
});

app.get('/one*Blue', (_, res) => {
  res.send('/one*Blue');
});

app.post('/test(ing)?', (_, res) => {
  res.send('/test(ing)?');
});
```

Uma outra forma de definir o caminho de uma rota é através de uma expressão regular. A imagem abaixo mostra um exemplo desta utilização, no qual qualquer rota terminando com “Red” seria processada.

Figura 35 – Expressão regular.

```
app.get(/.*Red$/, (_, res) => {
  res.send('/.*Red$/');
})
```

Através das rotas também é possível capturar parâmetros. Isto é muito útil pois permite que recursos possam ser identificados através da URL. Por exemplo, é possível buscar informações de determinado aluno a partir de um GET passando seu id como parâmetro na própria URL. Os parâmetros podem ser obtidos através da propriedade “params” do objeto da requisição. Ao definir uma rota que espera um parâmetro, basta

colocar um “:” antes do nome do parâmetro. A imagem abaixo exemplifica esta funcionalidade.

**Figura 36 – Parâmetros na rota.**

```
app.get('/testParam/:id', (req, res) => {  
  res.send(req.params.id);  
});
```

Uma outra característica das rotas é que possível fazer com que mais de uma função seja executada para determinada requisição. Elas são executadas na ordem que foram inseridas e a execução passa para a próxima quando o método next() é invocado. As imagens abaixo ilustram esse funcionamento de duas formas, a primeira passando os métodos diretamente e a segunda através de um array.

**Figura 37 – Métodos para tratamento de rota.**

```
app.get('/testMultipleHandlers', (_, res, next) => {  
  console.log('First method');  
  next()  
}, (_, res) => {  
  console.log('Second method');  
  res.end();  
});
```

**Figura 38 – Métodos via array para tratamento de rota.**

```
const callback1 = (req, res, next) => {  
  console.log("Callback 1");  
  next();  
};  
  
const callback2 = (req, res, next) => {  
  console.log("Callback 2");  
  res.end();  
};  
  
app.get('/testMultipleHandlersArray', [callback1, callback2]);
```



Rotas que respondem ao mesmo endereço, mudando apenas o tipo do método HTTP podem ser agrupadas sob o método “route” do Express, para facilitar a escrita e o entendimento. Segue abaixo um exemplo.

**Figura 39 – Métodos agrupados pelo route.**

```
app.route('/testRoute')
  .get((req, res) => {
    res.end();
  })
  .post((req, res) => {
    res.end();
  })
  .delete((req, res) => {
    res.end();
  })
```

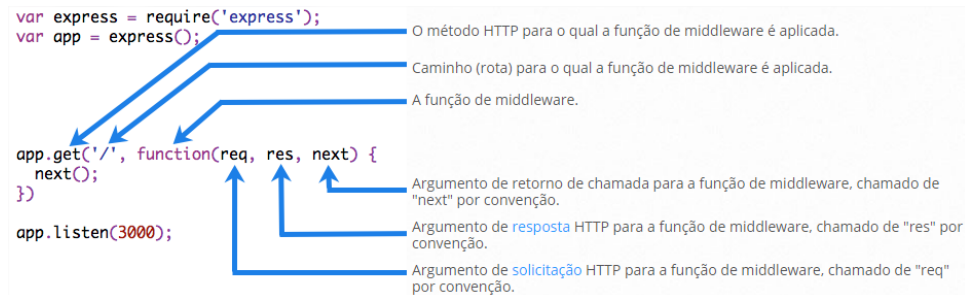
O Express também permite que as rotas possam ser subdividas em vários arquivos, facilitando assim a organização do projeto. Isso é possível a partir da utilização do objeto “Router” do Express. Na próxima seção este formato será abordado, a partir dos middlewares em nível do roteador.

## Middlewares

---

Funções de middleware são funções que têm acesso ao objeto de solicitação (req), o objeto de resposta (res) e a próxima função de middleware no ciclo da requisição e resposta do aplicativo (next). Elas podem executar qualquer código, fazer mudanças nos objetos de solicitação, encerrar o ciclo e chamar a próxima função de middleware na pilha. Ela pode ser utilizada para interceptar chamadas em específico ou qualquer chamada. Elas são as funções que são executadas quando determinada rota é atingida. A imagem abaixo ilustra a composição de um middleware.

**Figura 40 – Função de middleware.**



Fonte: <http://expressjs.com/>.

Uma função de middleware pode ser implementada no nível da aplicação ou no nível do roteador. No nível da aplicação ele pode ser configurado através de uma instância do Express, utilizando para isso as funções “use” ou uma específica de um método HTTP, como GET ou POST por exemplo. Os exemplos dos tópicos anteriores utilizam middlewares de nível de aplicação, pois são configurados no objeto instância do Express. A imagem abaixo mostra três exemplos, a primeira faz com que todas as requisições passem por ela, já que não foi construída em uma rota específica, a segunda recebe todas as requisições no caminho “/testMiddleware” independentemente do método HTTP, e a última processa somente as requisições GET na mesma rota citada.

**Figura 41 – Middleware nível da aplicação.**

```
app.use((req, res, next) => {
  console.log(new Date());
  next();
});

app.use('/testMiddleware', (req, res, next) => {
  console.log('/testMiddleware');
  if (req.method === 'GET') {
    next();
  } else {
    res.end();
  }
});

app.get('/testMiddleware', (req, res) => {
  res.send("GET /testMiddleware");
});
```

Já uma função de middleware no nível do roteador não estão vinculadas diretamente a instância do Express, mas sim a uma instância do Router. É comum que um projeto, ao invés de colocar todas as suas rotas em um mesmo arquivo, faça uma divisão destas rotas em vários arquivos, utilizando para isso o Router. Essa abordagem faz com que o projeto fique mais organizado, mantendo as rotas relacionadas juntas, mas separando em arquivos as que não tem tanta relação, facilitando assim a manutenção e evolução do projeto. Para criar um roteador, o código é o da imagem abaixo.

**Figura 42 – Criação do Router.**

```
const router = express.Router();
```

Após criado, as rotas podem ser construídas da mesma forma que no nível de aplicação, só que ao invés de utilizar a instância do Express, é utilizado a instância do Router em questão. Após a construção de todas as rotas, basta associar o roteador criado a instância do Express, passando a rota que ele irá responder. A imagem abaixo ilustra esse procedimento.

**Figura 43 – Middleware nível do roteador.**

```
import express from 'express';

const app = express()
const router = express.Router();

router.get('/', (req, res) => {
  console.log('/test');
  res.end();
});

app.use('/test', router);

app.listen(3000, async () => {
  console.log('API started!')
});
```

## Tratamento de erros

---

Tratamento de erros é uma parte muito importante de uma API. Um erro pode ser originado a partir de vários pontos, como por exemplo valores inválidos que foram passados como parâmetro. Por isso é importante que a API seja capaz de se recuperar de um erro e informar adequadamente ao usuário o que ocorreu, para que o mesmo possa tratar da melhor forma o ocorrido. O Express faz um tratamento padrão caso nenhum outro tenha sido especificado. A imagem abaixo ilustra um erro gerado sem tratamento, que no caso é então tratado pelo Express, que irá retornar um erro para o usuário.

**Figura 44 – Tratamento de erro padrão.**

```
app.get('/', function (req, res) {  
  throw new Error('Error');  
})
```

Caso o erro tenha sido gerado a partir de um código assíncrono e deseje utilizar o tratamento padrão do Express, é preciso passar o erro para o “next”, para que assim ele possa ser identificado. Caso não seja chamado o “next”, a requisição não irá retornar. A imagem abaixo ilustra este cenário.

**Figura 45 – Tratamento de erro padrão assíncrono.**

```
app.post('/', async (req, res, next) => {  
  try {  
    throw new Error('Error message');  
  } catch (err) {  
    next(err);  
  }  
});
```

O Express permite que o desenvolvedor escreva as próprias funções para tratamento de erro. Para isso, basta adicionar um quarto parâmetro na função de middleware, ficando de acordo com a imagem abaixo.

**Figura 46 – Tratamento próprio de erro.**

```
app.use(function (err, req, res, next) {  
  console.error(err.stack);  
  res.status(500).send('An error occurred!');  
});
```

O middleware para tratamento de exceção deve ser configurado por último na instância do Express, de forma que ele possa receber erros gerados em todas as definições anteriores. É permitido que exista várias funções para tratamento de erros, da mesma forma como os middlewares comuns, bastando chamar o “next” passando o objeto de erro como parâmetro, para enviar o fluxo para a próxima função. É importante observar que neste caso a última função de tratamento deverá encerrar a requisição através do objeto de resposta, caso contrário a requisição ficará pendente sem resposta. A imagem abaixo ilustra esta forma de tratar um erro em mais de uma função.

**Figura 47 – Tratamento de erro em várias funções.**

```
app.use(function (err, req, res, next) {  
  console.log('Error 1');  
  next(err);  
});  
  
app.use((err, req, res, next) => {  
  console.log('Error 2');  
  res.status(500).send('An error occurred!');  
});
```

## Gravação de logs

---

Uma funcionalidade muito importante para uma API é a gravação de logs. A partir dos logs é possível verificar como está sendo o uso dos endpoints ou até mesmo rastrear erros que ocorreram.

Uma forma simples de se fazer isso é através da utilização dos métodos do console nativo do JavaScript, como `“console.log”`, `“console.error”` e `“console.warn”`. Ao utilizar esses métodos, eles fazem a impressão no próprio terminal, o que faz com que ao fechar o terminal seja perdido o conteúdo. Para contornar isso, muitas pessoas direcionam este conteúdo para um arquivo, fazendo com que a informação seja então persistente.

Existem alguns problemas na adoção desta abordagem. Um deles é que não é possível desativar os logs, eles serão sempre impressos. Pode ter situações em que você não deseja que tudo seja capturado, mas sim somente um determinado conjunto de logs; neste caso não é possível definir o nível de log da aplicação, como por exemplo alternar entre gravar todos os logs ou somente logs de warning e error. Outro problema desta abordagem é que essas funções são síncronas, bloqueando assim a thread do Node.js enquanto estão fazendo a escrita.

Existem várias bibliotecas de log para o Node.js que tentam oferecer uma solução de log mais completa e que resolvam os problemas citados anteriormente. Uma das bibliotecas mais utilizadas para esse fim é a chamada Winston. Ela é uma biblioteca que suporta vários tipos de transporte, ou seja, com ela é possível que um mesmo log seja enviado a destinos diferentes, como por exemplo arquivos de log diferentes para um banco de dados remoto ou até mesmo para o próprio console. Ele também suporta os 7 níveis de log abaixo:

- error: 0

- warn: 1
- info: 2
- http: 3
- verbose: 4
- debug: 5
- silly: 6

Com esses níveis de log a aplicação pode, por exemplo, ao capturar um erro não esperado, utilizar o nível “error” para gravar o log e para apenas registrar uma informação, o “info”. Isso irá permitir que você possa configurar dinamicamente até qual ponto você deseja que sua aplicação registre os logs naquele momento. Se estiver em produção você pode querer capturar os “info”, mas talvez em desenvolvimento isso não seja interessante. Da mesma forma para o nível de “debug”, que você pode optar por, em algum momento, habilitar esse nível durante o desenvolvimento, e a partir daí a aplicação começaria a imprimir no log tudo que estivesse sido impresso com esse nível, facilitando o processo de depuração. Porém esse mesmo nível de “debug” poderia não ser interessante em produção, já que estaria poluindo o log com informações de debug. O nível dos logs respeita a ordem crescente, por exemplo, ao definir a aplicação com um nível de log 4, ela irá gravar nos logs os níveis 4, 3, 2, 1, 0, excluindo então os acima dele, no caso o 5 e 6.

Outra funcionalidade interessante do Winston é a possibilidade de configurar formatos de log, adicionando informações customizadas no formato desejado, como por exemplo a data e horário do registro. A imagem abaixo ilustra um exemplo de configuração e utilização desta biblioteca. Nas aulas gravadas a utilização desta biblioteca será explicada com mais detalhes.

Figura 48 – Configuração e utilização do Winston.

```
const { combine, timestamp, label, printf } = winston.format;

const myFormat = printf(({ level, message, label, timestamp }) => {
  return `${timestamp} [${label}] ${level}: ${message}`;
});

const logger = winston.createLogger({
  level: 'silly',
  transports: [
    new (winston.transports.Console)(),
    new (winston.transports.File)({ filename: 'grades-control-api.log' })
  ],
  format: combine(
    label({ label: 'grades-control-api' }),
    timestamp(),
    myFormat
  )
});

logger.error('Error log');
logger.warn('Warn log');
logger.info('Info log');
logger.verbose('Verbose log');
logger.debug('Debug log');
logger.silly('Silly log');
```

### Servindo arquivos estáticos

---

Uma funcionalidade interessante do Express é que ele permite que sejam servidos arquivos estáticos, sem que para isso tenha que se instalar uma nova ferramenta ou fazer muitas configurações. Ele já vem com uma função de middleware embutida, chamada “express.static”, que recebe como parâmetro o diretório raiz de onde estão localizados os arquivos, partindo da raiz da aplicação. Dessa forma é possível servir qualquer tipo de conteúdo estático, como imagens, páginas HTML, arquivos JavaScript e arquivos CSS, por exemplo. A imagem abaixo ilustra esse comando.



**Figura 49 – Servindo arquivos estáticos.**

```
app.use(express.static('public'));
```

No exemplo da imagem acima, caso tenha um arquivo chamado “logo.jpg” na pasta “public”, ele será acessível através da URL “http://localhost:3000/logo.jpg”. Para servir mais de uma pasta do projeto como arquivos estáticos, basta escrever essa linha de código quantas vezes for preciso com os nomes dos diretórios desejados. Também é possível criar um caminho virtual para acesso ao conteúdo, mesmo que ele não exista fisicamente. Para isso basta passá-lo como primeiro parâmetro da função. A imagem abaixo ilustra essa possibilidade.

**Figura 50 – Servindo arquivos estáticos com caminho virtual.**

```
app.use('/images', express.static('public'));
```

No exemplo desta imagem, o conteúdo da pasta “public” poderia ser acessado a partir da URL “http://localhost:3000/images/logo.jpg”.

### Capítulo 3. Organização de projetos Node.js

---

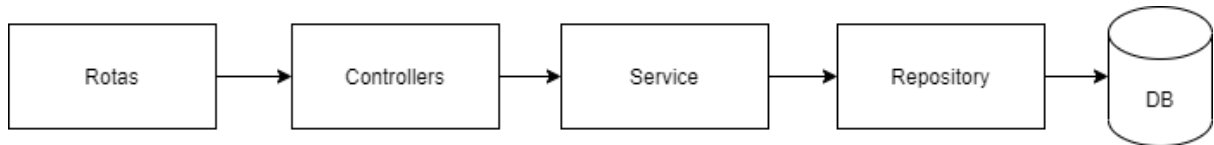
Um aspecto muito importante no desenvolvimento de software é a organização e estrutura do projeto. O Node.js não exige nenhuma estrutura específica para funcionar, fica a cargo do desenvolvedor escolher qual a melhor estrutura para seu cenário.

Em projetos pequenos, muitas vezes não se percebe os possíveis problemas que podem ocorrer com a evolução desestruturada de um projeto. À medida que ele vai crescendo, começamos a observar diversos problemas devido ao alto acoplamento, como por exemplo dificuldades na evolução do código, dificuldade do trabalho em equipe no mesmo projeto, dificuldade na reutilização de código, dificuldade de realização de testes unitários entre outros.

No desenvolvimento com APIs utilizando Node.js, um dos piores cenários de organização seria desenvolver tudo no mesmo arquivo, geralmente o arquivo de entrada do projeto, como por exemplo um “index.js”. Um outro cenário mais frequente, um pouco mais organizado, seria separar as rotas de acordo com suas entidades, organizando-as em um diretório chamado “routes”. Porém, mesmo assim, podemos notar problemas durante a evolução do projeto ao adotar esta abordagem, pois as rotas ficam sobrecarregadas, lidando até mesmo com regras de negócio e persistência de dados, por exemplo.

Uma estrutura muito utilizada para organização de APIs com Node.js é utilizar os conceitos de Route, Controller, Service e Repository. Veremos adiante um pouco sobre as responsabilidades de cada uma dessas camadas. A figura abaixo ilustra a ligação entre as camadas citadas.

**Figura 51 – Servindo arquivos estáticos com caminho virtual.**



## Route

A responsabilidade da Route é a mesma da vista anteriormente. Em um projeto Node.js utilizando o Express, por exemplo, as rotas são responsáveis por encaminhar as requisições de acordo com a rota informada, de acordo com o path e o método HTTP informado na requisição.

A única obrigação de uma route é encaminhar a requisição para o devido controller, não sendo sua obrigação nem mesmo a validação da requisição, como por exemplo se todos os parâmetros obrigatórios foram passados. A imagem abaixo ilustra um trecho de código de uma route.

**Figura 52 – Route.**

```

import express from "express";
import AccountController from "../controllers/account.controller.js";

const router = express.Router();

router.post("/", AccountController.createAccount);
router.get("/", AccountController.getAccounts);
router.get("/:id", AccountController.getAccount);
router.delete("/:id", AccountController.deleteAccount);
router.put("/", AccountController.updateAccount);
router.patch("/updateBalance", AccountController.updateBalance);

router.use((err, req, res, next) => {
  logger.error(`${req.method} ${req.baseUrl} - ${err.message}`);
  res.status(400).send({ error: err.message });
});

export default router;
  
```

## Controller

---

A responsabilidade do controller é receber a requisição direcionada pela rota, fazer as devidas validações, pegar os parâmetros informados na requisição e encaminhar para os services adequados, que realizarão a regra de negócio necessária.

Não é responsabilidade do controller efetuar regras de negócio, pois isso dificultaria a reutilização desta regra em outros locais, já que ela estaria altamente acoplada a validações e tratamentos específicos do Express, por exemplo. Outro problema de colocar regras de negócio nesta camada é que caso seja preciso trocar a tecnologia responsável pela API, não seria possível reaproveitar as regras, seria preciso refatorá-las também. A imagem abaixo ilustra um trecho de código de um controller.

**Figura 53 – Controller.**

```
async function createAccount(req, res, next) {
  try {
    let account = req.body;
    if (!account.name || account.balance == null) {
      throw new Error("Name e Balance são obrigatórios.");
    }
    account = await AccountService.createAccount(account);
    res.send(account);
    logger.info(`POST /account - ${JSON.stringify(account)}`);
  } catch (err) {
    next(err);
  }
}
```

## Service

---

A responsabilidade do service é implementar as regras de negócio da aplicação. Ele pode interagir com o banco de dados a partir do repository. Um dos benefícios que

obtemos ao concentrar as regras de negócio nesta camada é que poderemos reutilizá-las em vários outros controllers.

Outra vantagem é a facilidade de criar testes unitários nas regras de negócio, pois não seria necessário nem mesmo subir um servidor HTTP para realizar os testes, bastando chamar as funções desta camada. Outro ponto importante é que, caso seja preciso alterar uma regra de negócio, o desenvolvedor alteraria somente naquele local, sem a necessidade de ir alterando a mesma regra em diversos controllers. A imagem abaixo ilustra um trecho de código desta camada.

**Figura 54 – Service.**

```
import AccountRepository from "../repositories/account.repository.js";

async function createAccount(account) {
    return await AccountRepository.insertAccount(account);
}

async function getAccounts() {
    return await AccountRepository.getAccounts();
}
```

## Repository

---

A responsabilidade do repository é interagir com o banco de dados, nas consultas, alterações, exclusões e inserções. A vantagem de separar esta camada do service, é que ela pode ser reutilizada em vários services. Outra vantagem é que, caso seja necessário trocar a fonte de persistência de dados, a única camada impactada seria esta, enquanto nas demais a mudança seria transparente. A imagem abaixo ilustra um trecho de código desta camada.

Figura 55 – Repository.

```
import { promises as fs } from "fs";

const { readFile, writeFile } = fs;

async function getAccounts() {
  const data = JSON.parse(await readFile(global.fileName));
  return data.accounts;
}

async function getAccount(id) {
  const accounts = await getAccounts();
  const account = accounts.find(account => account.id === parseInt(id));
  if (account) {
    return account;
  }
  throw new Error("Registro não encontrado.");
}
```

## Capítulo 4. GraphQL

---

GraphQL é uma query language para APIs. É uma forma de explicitar, durante a consulta ou alteração na API, quais as informações são necessárias como resposta naquela requisição, de acordo com a necessidade.

Ele é um runtime no servidor para executar as consultas e alterações de acordo com os tipos definidos, não sendo acoplados a nenhum banco de dados em específico. Dessa forma, ele permite que o cliente solicite somente os dados necessários, podendo buscar mais dados relacionados em uma mesma requisição.

No GraphQL, a API é organizada em torno dos tipos e não dos endpoints. Ele permite que o cliente requisiute somente o que existe e provê erros de forma clara. Ele também permite adicionar campos e tipos sem afetar as consultas já existentes, evitando a criação de várias versões da API. O GraphQL não é acoplado a uma linguagem de programação em específico, o desenvolvedor provê funções para cada campo dos tipos, e o GraphQL as chama de forma concorrente.

Uma consulta pode ser realizada a partir de uma query, enquanto uma alteração é realizada a partir de uma mutation. O desenvolvedor deve fornecer, para cada uma das queries ou mutations informadas, uma forma de buscar as informações relativas a aquela solicitação. Essa forma de buscar essas informações é chamada de resolver, sendo uma função que faz as devidas ações e retorna o que é devido.

## Referências

---

EXPRESS. *Home*. 2021. Disponível em: <<https://expressjs.com>>. Acesso em: 15 fev. 2022.

FERREIRA, Rodrigo. REST: Princípios e boas práticas. *Blog Caelum*, 23 out. 2017. Disponível em: <<https://blog.caelum.com.br/rest-principios-e-boas-praticas/>>. Acesso em: 15 fev. 2022.

INSOMNIA. *Home*. 2021. Disponível em: <<https://insomnia.rest>>. Acesso em: 15 fev. 2022.

INTERFACE de Programação de Aplicações: O que é API?. *Red Hat*, 31 out. 2021. Disponível em: <<https://www.redhat.com/pt-br/topics/api/what-are-application-programming-interfaces>>. Acesso em: 15 fev. 2022.

KADLECSIK, Tamas. Understanding the Node.js Event Loop. *RisingStack*, 21 set. 2021. Disponível em: <<https://blog.risingstack.com/node-js-at-scale-understanding-node-js-event-loop/>>. Acesso em: 15 fev. 2022.

NETO, Waldemar. *Node.js: O que é esse Event Loop afinal?*. 27 nov. 2016. Disponível em: <<https://walde.co/2016/11/27/node-js-o-que-e-esse-event-loop-afinal/>>. Acesso em: 15 fev. 2022.

NODE.JS – NPM. *Tutoriaispoint*, c2021. Disponível em: <[https://www.tutorialspoint.com/nodejs/nodejs\\_npm.htm](https://www.tutorialspoint.com/nodejs/nodejs_npm.htm)>. Acesso em: 15 fev. 2022.

Node.js Modules. *W3 Schools*, c1999-2022. Disponível em: <[https://www.w3schools.com/nodejs/nodejs\\_modules.asp](https://www.w3schools.com/nodejs/nodejs_modules.asp)>. Acesso em: 15 fev. 2022.

NODEJR BRASIL. *Home*. Disponível em: <<https://nodebr.com>>. Acesso em: 15 fev. 2022.



NODEJS. Home. 2021. Disponível em: <<https://nodejs.org/en/>>. Acesso em: 15 fev. 2022.

PEREIRA, Caio Ribeiro. Entendendo o Event-loop do Node.js. *iMasters*, 1 nov. 2013. Disponível em: <<https://imasters.com.br/front-end/entendendo-o-event-loop-do-node-js>>. Acesso em: 15 fev. 2022.

QUIGLEY, James. Everything You Wanted To Know About package-lock.json But Were Too Afraid To Ask. *Medium – Coinmonks*, 12 ago. 2017. Disponível em: <<https://medium.com/coinmonks/everything-you-wanted-to-know-about-package-lock-json-b81911aa8ab8>>. Acesso em: 15 fev. 2022.

ROZLOG, Mike. REST e SOAP: Usar um dos dois ou ambos?. Tradução de Marcelo Costa. *InfoQ*, 2 out. 2013. Disponível em: <<https://www.infoq.com/br/articles/rest-soap-when-to-use-each/>>. Acesso em: 15 fev. 2022.

SOAP vs REST. What's the Difference?. *Smartbear*, 2 jan. 2020. Disponível em: <<https://smartbear.com/blog/soap-vs-rest-whats-the-difference/>>. Acesso em: 15 fev. 2022.

WANYOIKE, Michael; DIERX, Peter. A Beginner's Guide to npm, the Node Package Manager. *Sitepoint*, 9 mar. 2020. Disponível em: <<https://www.sitepoint.com/npm-guide/>>. Acesso em: 15 fev. 2022.

WEB service: o que é, como funciona, para que serve?. *Opensoft*, 7 jun. 2021. Disponível em: <<https://www.opensoft.pt/web-service/>>. Acesso em: 15 fev. 2022.

WINSTON. *GitHub*. Disponível em: <<https://github.com/winstonjs/winston>>. Acesso em: 15 fev. 2022.