

# **Entendendo o Javascript e o NodeJS**

**EC021 - Tópicos Avançados II  
Sistemas Distribuídos**

# Por que aprender Javascript?

“Any application that can be written in JavaScript, will eventually be written in JavaScript.” —Jeff Atwood



# Por que aprender Javascript?

- JavaScript é uma das linguagens de programação mais abertas e com a maior comunidade da atualidade.
- JavaScript é uma linguagem muito flexível que permite aos desenvolvedores escrever o estilo de código que preferem, mas não os isola de outros estilos de codificação e seus efeitos colaterais.
  - Isso significa que você pode escrever JavaScript seguindo o paradigma orientado a objetos, mas também misture outros estilos com facilidade (às vezes de forma não intencional).
  - Alguns desenvolvedores afirmam que esta é uma fraqueza da linguagem, que ela é indefinida e que a linguagem não sabe o que quer ser. Enfim tudo é permitido e você nunca irá escrever código 'puro'. Isto pode ser visto como um benefício e oportunidade (como eu) porque você pode escrever seu código da maneira que mais se adequa em cada situação!

# Por que aprender Javascript?

- Quanto útil é o JavaScript?
  - O JavaScript está em TODOS OS LUGARES e a parte incrível é que ele está apenas “se aquecendo”!
  - O JavaScript começou muito fraco e não muito útil (quando a linguagem começou a ganhar espaço, os desenvolvedores de JavaScript eram chamados de 'script kiddies' e não eram respeitados de forma alguma), em seguida, o JavaScript começou a evoluir!
- O JavaScript tinha um problema, muito grande, era restrito ao navegador, mas com o surgimento do NodeJS esta história mudou.

# Por que aprender Javascript?

- Levando tudo em conta, é seguro dizer que o JavaScript tem um futuro bonito a frente. Os padrões JavaScript estão em constante evolução já transformaram o que já foi considerado uma linguagem de programação infantil em uma potência de desenvolvimento de software.
- JavaScript tem a comunidade mais ativa, novas coisas são constantemente testadas e adotadas em pequenos intervalos de tempo!
- A Web, a plataforma na qual o JavaScript é uma parte vital, está se tornando cada vez mais importante e também com JavaScript.
- O JavaScript é suportado e importante para muitas empresas de tecnologia e também empresas de outros setores.

# Por que aprender Javascript?

- As 10 linguagens mais populares do GitHub:



# Conclusão

- JavaScript não é perfeito, mas neste momento é um forte concorrente.
- O JavaScript simplesmente pode fazer tudo, escolher o JavaScript como sua principal linguagem de programação o deixa aberto para mudar para qualquer plataforma: web, desktop, smartphone, IoT, etc.
  - Você não pode dizer exatamente o mesmo para a maioria das outras linguagens de programação.
- É possível ver que os desenvolvedores de JavaScript continuarão em alta demanda.

# Plataformas que suportam Javascript

- O JavaScript foi inventado por Brendan Eich em 1995 e tornou-se um padrão ECMA em 1997.
- ECMA-262 é o nome oficial do padrão. ECMAScript é o nome oficial da linguagem. (Fonte: W3C)
- Vamos considerar aqui as principais plataformas de clientes (prioritariamente navegadores) que aceitam Javascript:



# Plataformas que suportam Javascript

Engine	ECMA	Browser
V8	6	Chrome (Partial Support)
SpiderMonkey	6	Firefox (Partial Support)
Chakra	6	Edge (Partial Support)
Nitro	6	Safari (Partial Support)
V8	6	Opera (Partial Support)
V8	5	Chrome 23
SpiderMonkey	5	Firefox 21
JavaScript 1.8.5	5	Firefox 4
Nitro	5	Safari 6
V8	5	Opera 15
Chakra	5	Edge 12
Chakra	5	IE 10

# Engine Javascript

- Escrever código para a web às vezes parece um pouco mágico, pois escrevemos uma sequência de caracteres e as coisas acontecem em um navegador. Mas entender a tecnologia por trás da magia pode ajudá-lo a ajustar melhor seu ofício como programador. Pelo menos você pode se sentir um pouco menos incapaz ao tentar explicar o que está acontecendo nos bastidores em uma pilha JavaScript.
- Uma Engine JavaScript é um tipo de máquina virtual de processo projetada especificamente para interpretar e executar o código JavaScript.

# Variáveis, funções e constantes

- As variáveis em JS são comumente criadas através do uso de:

```
var nomeDaVariável;
```

- Recomenda-se que os nomes das variáveis obedecem as recomendações comumente propostas para criação de variáveis nas diversas linguagens de programação como não serem utilizados caracteres especiais, não se utilizarem letras maiúsculas e etc. Assim, se quisermos criar uma variável chamada número, utilizamos o código:

```
var numero; //Declarando uma variável  
numero = 0; //Inicializando uma variável
```

# Tipos de dados

- As variáveis em JS não têm seus tipos definidos na sua criação, porém possui um conjunto restrito de valores que pode ser aceito por uma variável. Os tipos primitivos são:
  - **Numérico**: variáveis em JS podem suportar números inteiros ou decimais.
  - **Texto**: variáveis em JS que podem suportar textos. Os textos associados devem ser inseridos entre aspas.
  - **Booleano**: variáveis que podem assumir dois valores: *true* ou *false*.

# Constantes

- São variáveis declaradas com a palavra  
`const nomeDaConstante;`
- Os valores das constantes não poderão ser modificados durante o código:

```
//Declarando e inicializando uma constante  
const MAX_VALUE = 9999;
```

# Funções

- A definição da função (também chamada de declaração de função) consiste no uso da palavra chave `function`, seguida por:
  - Nome da Função.
  - Lista de argumentos para a função, entre parênteses e separados por vírgulas.
  - Declarações JavaScript que definem a função, entre chaves { }.

```
function quadrado(num) {  
    return num * num;  
}
```

# Funções

- Embora a declaração de função anterior seja sintaticamente uma declaração, funções também podem ser criadas por uma **expressão de função**. Tal função pode ser anônima; ele não tem que ter um nome. Por exemplo, a função ***quadrado*** poderia ter sido definida como:

```
var quadrado = function (num) {  
    return num * num;  
}
```

# Objetos

- Abaixo temos a declaração do objeto **wizard**:

Método

```
var wizard = {  
  name: "Gandalf",  
  mana: 45,  
  magic: function (mana) {  
    this.mana -= mana;  
  }  
}
```

Atributos



# Objetos

- Considere o código abaixo:

```
var wizard = {  
  name: "Gandalf",  
  mana: 45,  
  magic: function (mana) {  
    this.mana -= mana;  
  }  
}  
console.log(wizard.name + " - Mana: " + wizard.mana);  
wizard.magic(10);  
console.log(wizard.name + " - Mana: " + wizard.mana);
```

# Objetos

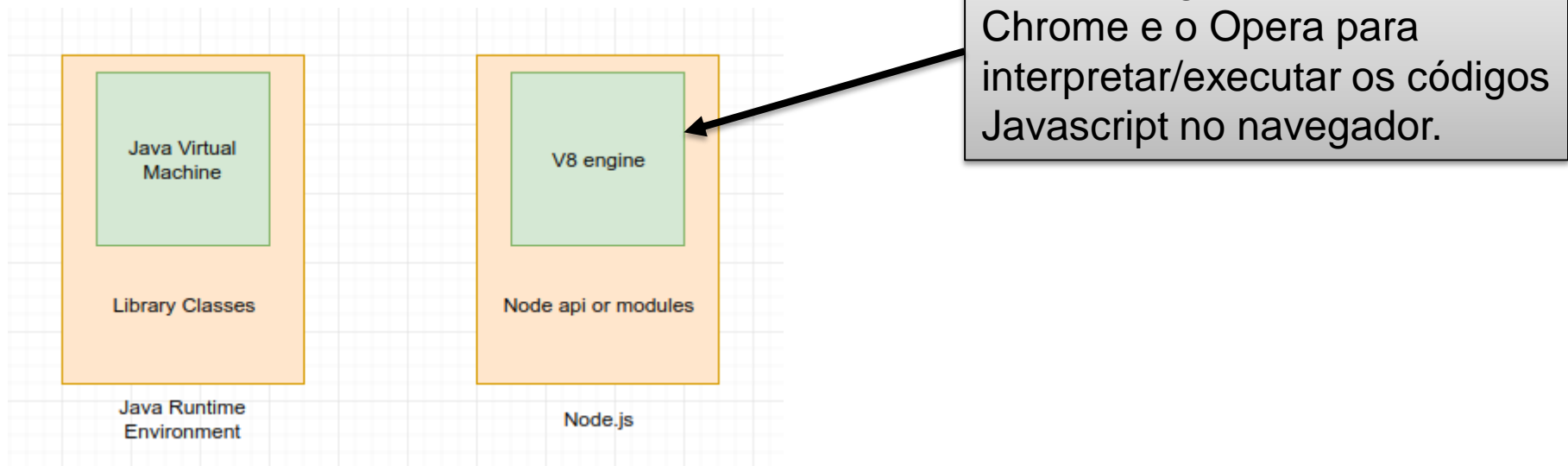
- Teremos a seguinte saída:

```
$ node index.js  
Gandalf - Mana: 45  
Gandalf - Mana: 35
```



# O NodeJS

- NodeJS é um ambiente de execução Javascript que roda do lado do servidor (backend), ou seja o NodeJS tem tudo que você precisa para executar uma aplicação escrita em Javascript.

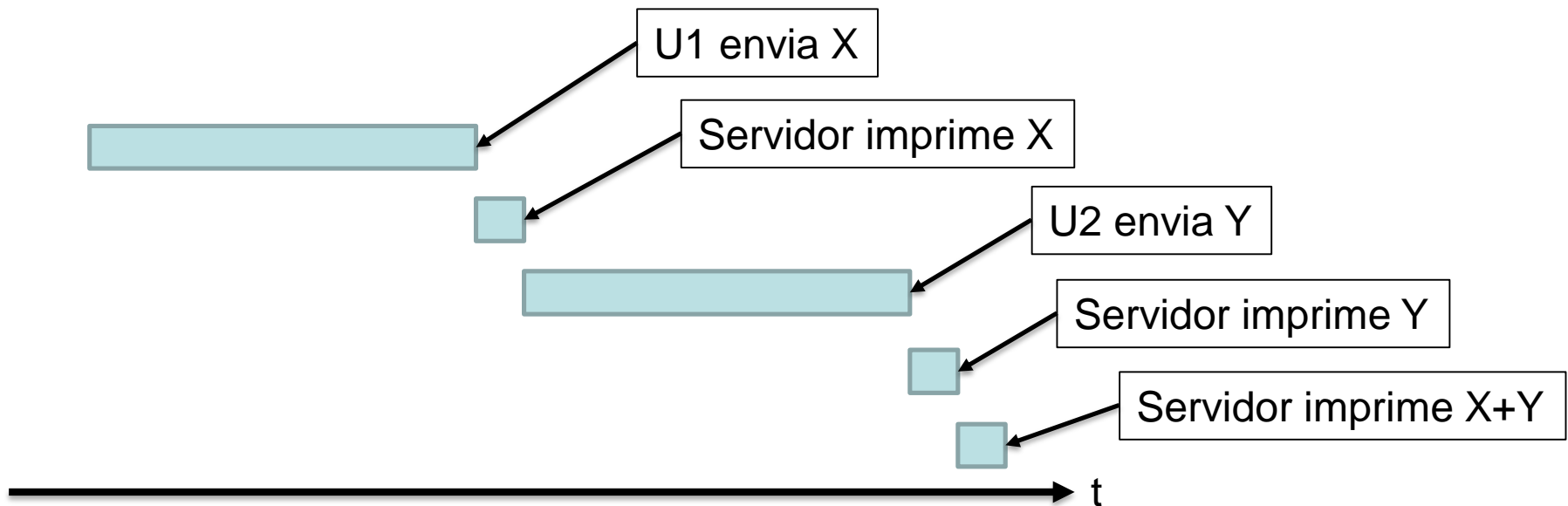


# O NodeJS

- De acordo com o site oficial do NodeJS:
  - Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine.
  - Node.js uses an **event-driven, non-blocking I/O model** that makes it lightweight and efficient.
  - Node.js' package ecosystem, **NPM**, is the largest ecosystem of open source libraries in the world.
- Em poucas palavras o objetivo principal do NodeJS é fornecer uma maneira fácil de se desenvolver sistemas distribuídos que sejam facilmente **escaláveis**.

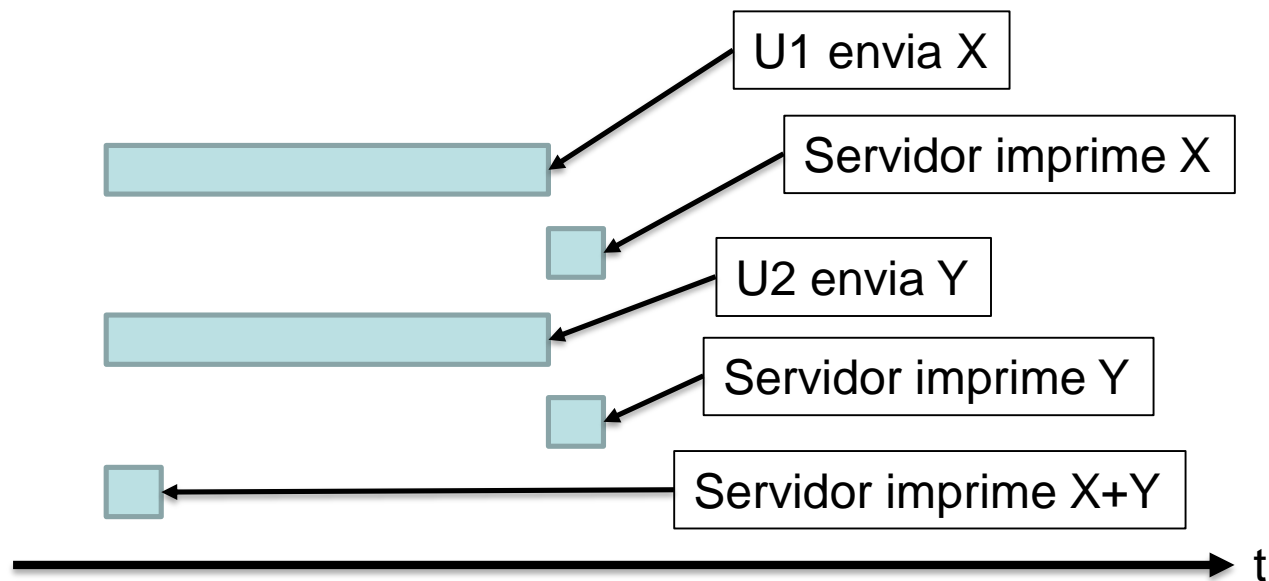
# O NodeJS

- E o que seria o modelo de E/S não bloqueante?
- Imagine um cenário onde você e um colega desejam acessar a mesma URL. Em um modelo onde as E/S são bloqueantes o fluxo ficaria conforme abaixo:



# O NodeJS

- Este mesmo cenário utilizando o modelo de E/S não bloqueante, ficaria assim:

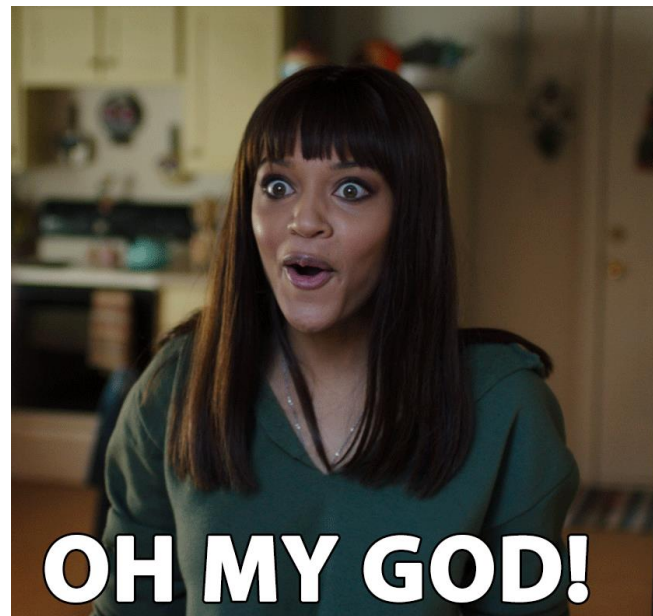


# O NodeJS

- Se isso pode estragar minha lógica, então qual a vantagem?
- Este modelo de E/S não bloqueante elimina a necessidade de se implementar servidores multithreads, já que um mesmo servidor é capaz de atender várias requests ao mesmo tempo.
- E sim, é possível, dentro deste modelo sincronizar alguns eventos para atender à algum requisito lógico.

# O NodeJS

- Ou seja, assim como é possível criar processos assíncronos em ambientes totalmente síncronos (por exemplo, utilizando Threads em Java), também é possível criar fluxos sincronizados no modelo assíncrono do NodeJS.





# NPM

- O NPM (Node Package Manager) é o gerenciador de pacotes e dependências do NodeJS, assim como temos em outras linguagens (Maven, Pip, Gradle, etc).
- Possui 3 funções definidas:
  - Site
    - Onde podem ser criados, modificados e compartilhados novos pacotes NodeJS.
  - Command Line Interface (CLI)
    - Onde a partir do terminal é possível interagir com o repositório de pacotes.
  - Registro
    - Grande base de dados sobre todo o software do Javascript.

# NPM - Comandos

- Para criarmos uma aplicação baseada em NodeJS devemos primeiramente indicar que determinado diretório será um repositório de um projeto NodeJS.
- Para isto utilizaremos o comando abaixo:  

```
npm init
```
- Você deverá entrar com algumas informações sobre o projeto a partir da CLI e, assim que finalizar será criado no diretório um arquivo **package.json**.
- Este arquivo tem a função de armazenar todas as informações relevantes sobre o seu projeto como nome, autor, repositório git e **dependências**.

# NPM - Comandos

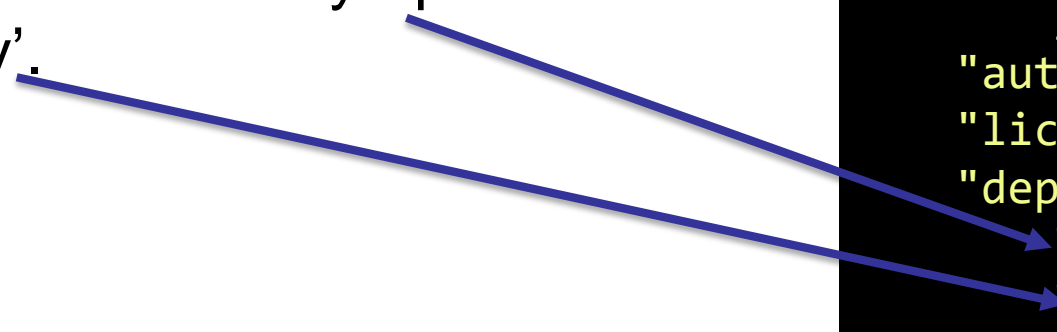
- É importante ressaltar a importância deste arquivo (package.json) em um projeto NodeJS, pois o mesmo é responsável por gerenciar todas as dependências inerentes ao seu projeto, o que faz com que não seja necessário armazenar nenhuma dependência nos seus repositórios de controle de versão, apenas este arquivo.
- Caso você tenha recebido um projeto NodeJS somente com os arquivos de código fonte e o arquivo package.json, é possível baixar todas as dependências de uma vez só fazendo com que o NPM leia este arquivo e recupere todas as dependências, utilizando o comando abaixo:

**`npm install`**

# NPM - Comandos

- O formato de um arquivo package.json está ilustrado ao lado:
- É possível notar que este projeto possui 2 dependências: 'mysql' e 'restify'.

```
{  
  "name": "projeto1",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": ""  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "mysql": "^2.16.0",  
    "restify": "^7.2.1"  
  }  
}
```



# NPM - Comandos

- Todas as dependências instaladas ficarão em uma pasta chamada '**node\_modules**' no diretório raiz do projeto.
- Caso seja necessário incluir uma nova dependência no projeto podemos utilizar o comando abaixo na CLI:

```
npm install xxxxx
```

- Para executar o nosso projeto utilizaremos o comando abaixo na CLI:

```
node xxxxx.js
```

# Resumo

Comando	Resultado
npm init	Inicializa seu diretório como repositório NodeJS
npm install	Lê o arquivo package.json e instala todas as dependências contidas nele
npm install xxxxx	Instala o pacote xxxxx no seu projeto e inclui o mesmo no package.json
node xxxxx.js	Executa o arquivo xxxxx.js no ambiente de execução do NodeJS



# Middleware

- Em SD as metas definem características cruciais para a sua implementação.
- Estas metas estão relacionadas diretamente ao negócio ao qual o sistema se aplica e tem relação direta com as estimativas de custo, esforço e viabilidade.
- As quatro metas essenciais são:
  - Acesso à recursos
  - Transparência na distribuição
  - Abertura do sistema
  - Escalabilidade

# Middleware

- Considere um sistema a ser desenvolvido em um ambiente onde há diversidade de redes de acesso, dispositivos, limitações e organização da comunicação.
- Como criar um sistema que atenda todas estas metas e ainda possua suporte para funcionar em um ambiente como este?
  - Utilizando middlewares!
- O papel do MIDDLEWARE é funcionar como uma interface responsável por interceptar diferenças operacionais mascarando uma integração de várias linguagens e padrões de comunicação através de processos bem definidos. Porém não ficará a cargo dele processar este tipo de dado.

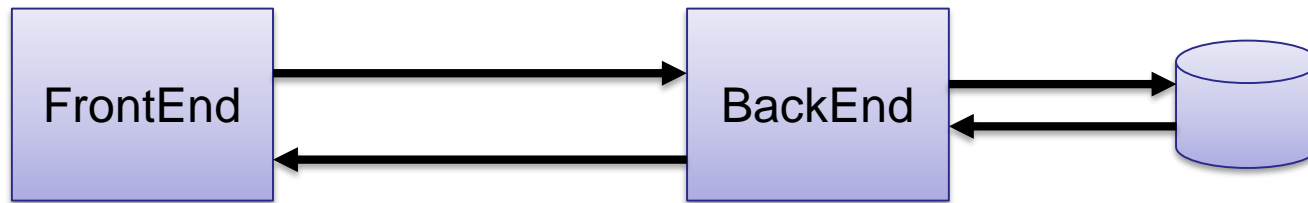


# Middleware

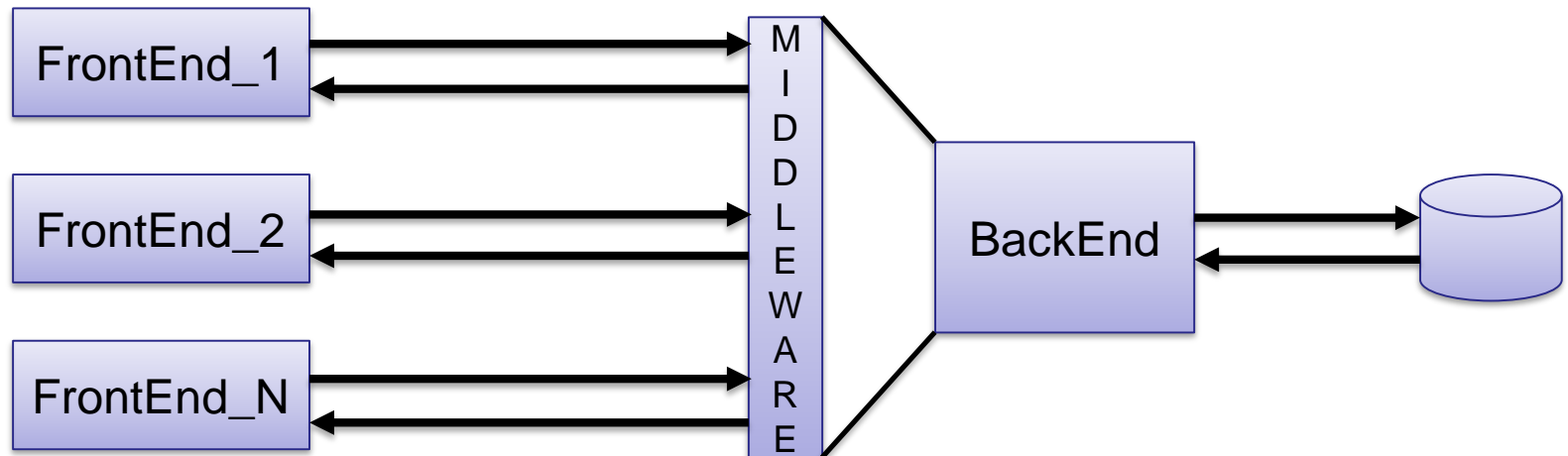
- Para este tipo de solução o middleware funcionará como uma fronteira entre o servidor e seus arredores. Nele será possível gerenciar as metas e garantir o funcionamento do sistema em um ambiente diversificado.

# Middleware

Antes



Depois



# Middleware no NodeJS

- Uma das práticas mais comuns hoje em dia para expor os serviços do backend para sistemas distribuídos é utilizando API's REST.
- *“Podemos definir REST como estilo arquitetural de SD. A idéia do REST é utilizar de maneira mais eficiente e em sua plenitude as características do protocolo HTTP, principalmente no que diz respeito à semântica do protocolo. O resultado disso ao final das contas é, além da utilização mais “correta” do protocolo, um trânsito de informações mais eficiente e, por consequência, mais rápido.”*

# Middleware no NodeJS

- Para criar uma API REST em NodeJS utilizaremos o framework **restify**.
- Este framework é uma alternativa mais leve (em relação ao clássico Express – muito utilizado em aplicações Web javascript) para desenvolvimento exclusivo de API's REST.
- O primeiro passo é instalar o módulo do restify na sua aplicação:

```
npm install restify
```

- Verifique se a dependência foi criada no arquivo package.json:

```
"dependencies": {  
  "restify": "^7.2.1"  
}
```

# Middleware no NodeJS

- Passo 1 – Importe a dependência no seu arquivo index.js:  
`var restify = require('restify');`
- Passo 2 – Crie o seu servidor:  
`var server = restify.createServer({  
 name: 'Prática 1'  
});`
- Passo 3 – Configure seu servidor para receber requests com body no formato json:  
`server.use(restify.plugins.bodyParser());`

# Middleware no NodeJS

- Passo 4 – Crie seu primeiro endpoint, defina por qual método HTTP ele será chamado e passo como um segundo argumento a chamada a uma função. Esta função será executada toda vez que o endpoint for acessado:

```
server.get(' /hello' , helloWorld) ;
```

- Passo 5 – Defina a porta onde seu servidor vai rodar:

```
var port = process.env.PORT || 5000 ;
```

- Passo 6 – Chame a função que irá subir o servidor:

```
server.listen(port, function() {  
    console.log('%s rodando' , server.name) ;  
} ) ;
```

# Middleware no NodeJS

- Passo 7 – Implemente a função 'helloWorld' que será executada sempre que acessarmos <http://localhost:5000/hello>:

```
function helloWorld(req, res, next) {  
  res.send("Hello world");  
  next();  
}
```

# Middleware no NodeJS

- Passo 8 – Execute o seu arquivo index.js:  
`node index.js`
- Passo 9 – Teste o acesso ao seu middleware pelo Postman.

