



Quem se prepara, não para.

# Arquitetura de Aplicações Web

5º período

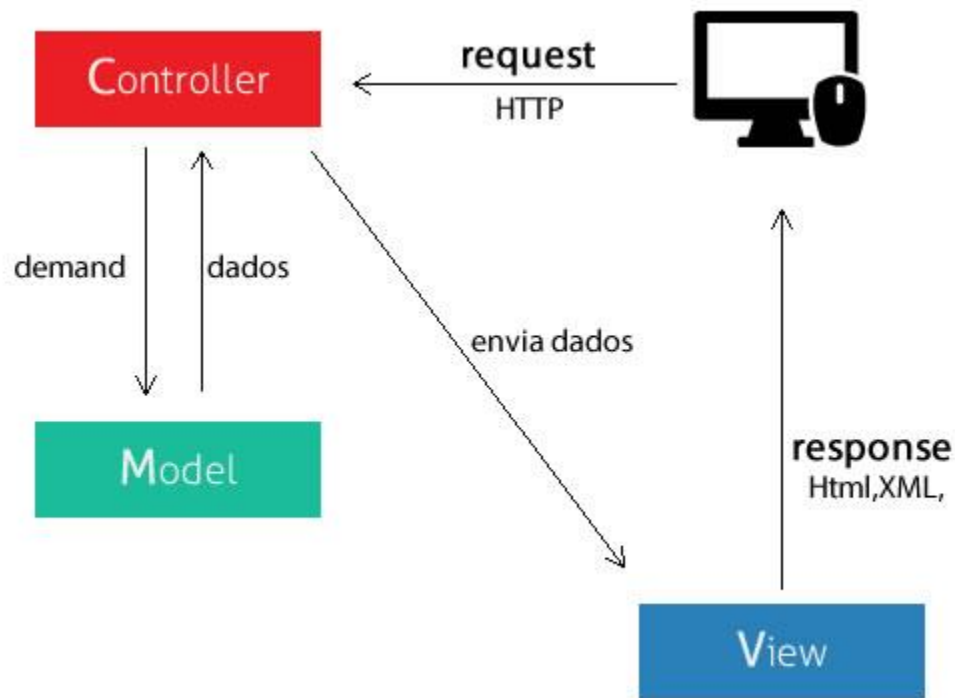
Professora: Michelle Hanne

# Sumário

- Revisão MVC
- NPM
- Outras bibliotecas
- Promises
- Criando API com Node
- Atividade Learnyounode

# Padrão de Arquitetura MVC

*Model-view-controller* (MVC) é um padrão de arquitetura de software que divide a aplicação em três camadas: **Model**, **View** e **Controller**.



# Modelo MVC

## M (MODEL)

A camada *Model* (modelo) é responsável pela leitura, escrita e validação dos dados. Nesta camada são implementadas as regras de negócios. Sempre que você pensar em manipulação de dados, pense em model.

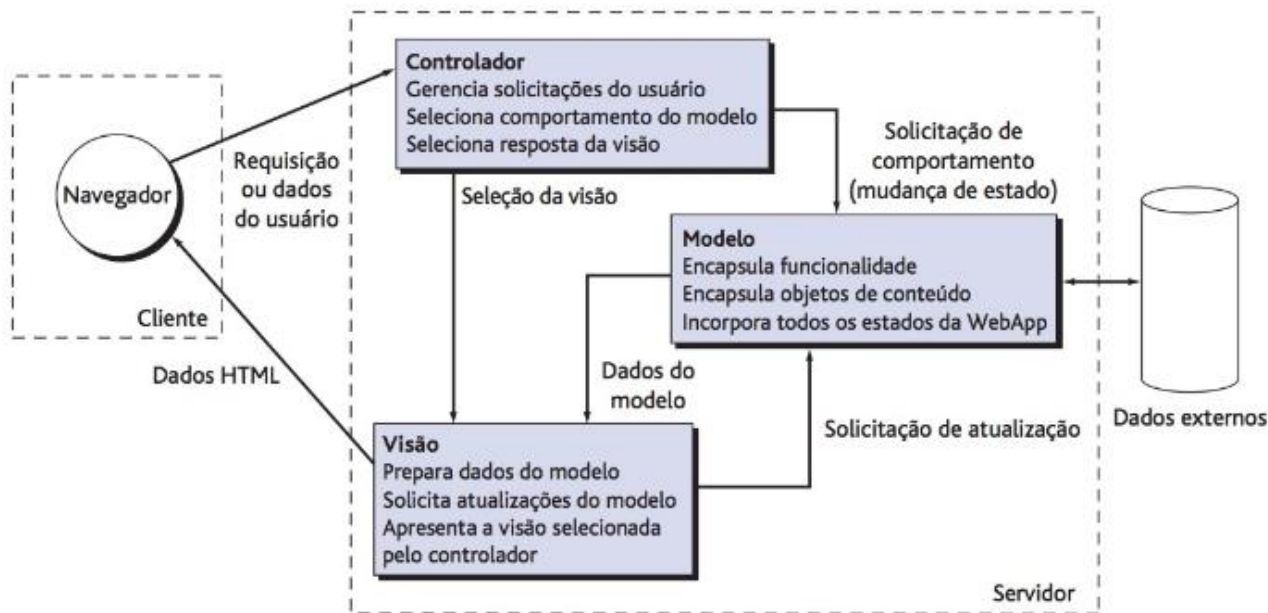
## V (VIEW)

A camada *View* (visão) é responsável pela interação com o usuário. Nesta camada são apresentados os dados ao usuário. Os dados podem ser entregues em vários formatos, dependendo do que for preciso, como páginas HTML, arquivos XML, documentos, vídeos, fotos, músicas, entre outros.

## C (CONTROLLER)

A camada *Controller* (controlador) é responsável por lidar com as requisições do usuário. Ela gerencia as ações realizadas, fala qual Model e qual View utilizar, para que a ação seja completada.

# Modelo MVC



**FIGURA 17.8** A arquitetura MVC.

Fonte: Adaptado de [Jac02b].

# Modelo MVC

Em uma **WebApp**, a visão é atualizada pelo controlador com dados do modelo baseados nas informações fornecidas pelos usuários.

As solicitações ou os dados do usuário são manipulados pelo controlador. O controlador também seleciona o objeto visão aplicável, de acordo com a solicitação do usuário. O objeto-modelo pode acessar dados armazenados em um banco de dados (repositório de dados local ou um conjunto de arquivos independentes). Os dados trabalhados no modelo devem ser formatados e organizados pelo objeto de visão apropriado e transmitidos do **servidor de aplicações** de volta para o navegador instalado no cliente para exibição na máquina do usuário.

# Modelo MVC

## O diálogo das camadas

**View:** Fala Controller! O usuário acabou de pedir para acessar o Facebook! Pega os dados de login dele aí. **Controller:** Blz. Já te mando a resposta. Ai model, meu parceiro, toma esses dados de login e verifica se ele loga. **Model:** Os dados são válidos. Mandando a resposta de login. **Controller:** Blz. View, o usuário informou os dados corretos. Vou mandar pra vc os dados dele e você carrega a página de perfil. **View:** Vlw. Mostrando ao usuário...



# Modelo MVVM

- O padrão de projeto **Model-View-ViewModel (MVVM)** foi originalmente criado para aplicativos **Windows Presentation Foundation (WPF)** usando **XAML** para separar a interface do usuário (UI) da lógica de negócios e aproveitando ao máximo o data binding (*a vinculação de dados*).
- Aplicações arquitetadas desta forma têm uma camada **ViewModel** distinta que não possui dependências de sua interface de usuário.
- Como as classes **ViewModel** de um aplicativo não têm dependências sobre a camada de interface do usuário, você pode facilmente trocar uma interface de usuário iOS por uma interface Android e escrever testes contra a camada ViewModel.

[http://www.macoratti.net/16/09/net\\_mvvm1.htm](http://www.macoratti.net/16/09/net_mvvm1.htm)



- Ao instalar o Node.js, três programas são instalados:
  - O node, executador de arquivos JavaScript;
  - O npm e seu irmão mais novo npx
- O **NPM** (***Node Package Manager***) é um gerenciador de pacotes tipo Ruby Gems (ruby), NuGet (.NET), pip (python), maven/gradle (Java)
- A ideia do npm é:
  - Reutilizar programas JavaScript (📁 pacotes)
  - Gerenciar as dependências do seu projeto Node.js
  - Tornar seus programas/utilitários (i.e., pacotes) disponíveis para a comunidade

# O que é um pacote

- É um “programa” Node.js
- Pode ser privado ou público (padrão)
- Quando é público, qualquer um pode instalá-lo e ver seu código fonte, caso ele esteja no GitHub, por exemplo.
- Uma pasta em seu computador é considerada um pacote se ela possui um arquivo chamado package.json:

```
{  "name": "bespoke-math",  
  "version": "1.2.0",  
  "dependencies": {  
    "katex": "^0.6.0"  
  }  
}
```

# O que é um pacote

- Instalando pacotes com o npm
- Para instalar um pacote no diretório atual, usamos:

```
$ npm install <nome-do-pacote>
```

- Se quisermos instalar um pacote de forma global (acessível de qualquer lugar, como um programa executável):

```
$ npm install -g <nome>
```

- Ou então **npm install --global <nome>**

# CommonJS

- Cada módulo tem o seu escopo e pode:
  - **require** coisas de outros módulos
  - **module.exports** suas próprias coisas
- Exemplo:

matematica.js

```
function fft(sinal) {  
  // transforma fourier  
  return ....;  
}  
  
module.exports = {  
  fft  
}
```

principal.js

```
const mat = require('matematica.js')  
  
console.log(mat.fft([...]))
```

# Express

- Se intitulam um web framework para Node.js:
  - Rápido
  - Não opinativo (unopinionated)
  - Minimalista
  - Site oficial: <http://expressjs.com/>
- Instalando:
- `$ npm install express --save`

## Servidor “*hello world*” com Express

```
import express from 'express'
const app = express()

app.get('/', (req, res) => {
  res.send('Hello World!')
})

const server = app.listen(3000, () => {
  const host = server.address().address;
  const port = server.address().port;

  console.log(`Listening at http://${host}:${port}`);
});
```

# Express - Rotas

- O express facilita a especificação da ação a ser tomada dependendo da URL solicitada
- Uma rota é um verbo HTTP (GET, POST etc.) e uma URL
- A cada rota é associada uma callback:

```
// GET / (página inicial)
app.get('/', (request, response) => {
  response.render('index')    // desenha a view 'index'
})
// POST /contato
app.post('/contato', (request, response) => {
  // envia um email usando os dados enviados
})
```

# Gerando HTML Dinamicamente

- Queremos poder gerar HTML dinamicamente. Para isso, precisamos de uma linguagem que facilite isso, ao mesmo tempo que possibilite a separação de código HTML do código nessa linguagem
- Para ser “não opinativo”, o Express não impõe uma linguagem específica:
- **ejs (.ejs, era o formato original do Express)**
- **handlebars (.hbs, [site oficial](#))**
- **pug (.pug, [site oficial](#))**



# Gerando HTML Dinamicamente (EJS)

- Gerando HTML dinamicamente com ejs
- Usamos ejs (ou qualquer outro templating engine) em 3 passos:
  - Configuramos o express
  - Escrevemos arquivos HTML no formato ejs
  - Para determinadas rotas, renderizamos views
- (1) Para configurar o Express para usar ejs:
- `app.set('view engine', 'ejs')`
- (2) Escrevemos arquivos no formato `.ejs` em vez de `.html`. Trecho de um arquivo, e.g., `equipe.ejs`:

```
<ul>
  <% for (let i = 0; i < users.length; i++) { %>
    <li><%= users[i].nome %></li>
  <% } %>
</ul>
```

Esses arquivos são chamados de views e devem ficar dentro de uma pasta que configurarmos (valor padrão: `./views`)

```
app.set('views', 'arquivos_ejs');
```

# Gerando HTML Dinamicamente (EJS)

- Gerando HTML dinamicamente com ejs
- Usamos ejs (ou qualquer outro templating engine) em 3 passos:
  - Configuramos o express
  - Escrevemos arquivos HTML no formato ejs
  - Para determinadas rotas, renderizamos views
- (1) Para configurar o Express para usar ejs:
- `app.set('view engine', 'ejs')`
- (2) Escrevemos arquivos no formato .ejs em vez de .html. Trecho de um arquivo, e.g., `equipe.ejs`:

```
<ul>  
  <% for (let i = 0; i < users.length; i++) { %>  
    <li><%= users[i].nome %></li>  
  <% } %>  
</ul>
```

# Gerando HTML Dinamicamente (EJS)

- (3) Ao definirmos os handlers das nossas rotas, chamamos `response.render` e passamos o nome do arquivo da view que deve ser usado (sem a extensão):

```
app.get('/equipe', function(request, response) {  
  response.render('equipe', contexto);    // vai pegar arquivos_ejs/equipe.ejs  
});
```

- É possível (e muito comum) disponibilizar dados para a view e podemos fazer isso usando o segundo parâmetro de `response.render`:

```
response.render('equipe', {  
  users: [  
    { nome: 'TJ Holowaychuk', foto: 'tj.jpg' },  
    { nome: 'Douglas Wilson', foto: 'dcw.jpg' }  
  ]  
});
```

DICA:

<http://expressjs.com/en/guide/using-template-engines.html>

# Callback

A programação assíncrona não usa o retorno da função para informar que a função foi finalizada. Ela trabalha com o “estilo de passagem de continuação (continuation-passing style), CPS”. Em definição, uma função escrita neste estilo recebe como argumento uma “continuação explícita”. Quando a função resulta um valor, ela “retorna” pela chamada da função de continuação com este valor via argumento. As funções de continuação explícitas também são conhecidas como *Callback*.

# Promises

Basicamente, uma vez que uma *promise* é chamada, ela inicia em **pending state (pendente)**. Isto significa que a função **caller** (que chamou a promise) continua sua execução enquanto espera pela promise terminar seu próprio processamento e retornar ao caller com algum feedback. Quando esse retorno acontece, a **promise** é **retornada em resolved state ou rejected state**.

# Promises

Quando precisamos realizar várias chamadas assíncronas, podemos ter um callback hell: várias callbacks aninhadas

- Dificulta a leitura e escrita
- Suscetível a erros do programador
- Trata erros apenas por callback, dificultando a legibilidade/manutenibilidade do código
- Soluções

uso de promessas explicitamente ou com `async/await`

# Promises

Uma promise é um objeto “**thenable**”, i.e., podemos invocar **.then**, passando uma função que só será chamada quando a promessa for cumprida (com êxito ou falha):

- **.then(callbackSuccess, callbackError)** pode receber 2 funções
- ...ou podemos **usar .catch** para tratar o erro de uma “**promise chain**” de forma genérica

É possível criar objetos do tipo **Promise** de forma que nós definimos quando elas estão resolvidas (com sucesso ou falha)

# Problemas com Promessas

- Promessas com `.then` encadeados reduzem (mas não acabam) com callback hell 🔥
- Há possibilidade de `.catch` não capturar exceção.
- Caso `.catch` seja atrasado (eg, devido a alguma espera na criação da promessa - exemplo).
- É difícil escrever um fluxo condicional em uma cadeia de promessas.
- Depurar ainda fica um pouco difícil com promessas.
- É possível aumentar a legibilidade, se o código parecer síncrono.



# AWAIT


- await substitui o .then
- Parece síncrono, mas suspende execução até a promessa ser cumprida.
- E isso não bloqueia a execução do processo (ie, é assíncrono).

```
function espera(tempo) {  
  return new Promise(resolver => {  
    setTimeout(resolver, tempo*1000)  
  })  
}
```

```
console.log('tempo = 0')  
await espera(2)    // <--  
console.log('tempo = 2')  
// tempo = 0  
// ...  
// tempo = 2 (2s depois)
```

```
console.log('tempo = 0')  
espera(2).then(() => console.log('tempo = 2'))  
// tempo = 0  
// ...  
// tempo = 2 (2s depois)
```

# Retorno de Await

- O valor que é resolvido pela Promise é retornado pela função
-  Exemplo: pegar dados de notícias, criar um template e mostrar
- 3x operações assíncronas em sequência:

```
mostraNoticia() << formata() << dados('noticia')
```

```
const db = { /*...*/ }  
function dados(entidade) {  
  return new Promise(resolver => {  
    // faz algo assíncrono (eg, acessa banco ou ajax)  
    // e resolve (cumpre) a promessa com o resultado  
    resolver(db[entidade])  
  })  
}
```

```
// com await 🚀  
const noticia = await dados('noticias')  
const template = await formata(noticia)  
mostraNoticia(template)
```

```
// com promise.then  
dados('noticias')  
  .then(formata)  
  .then(mostraNoticia)
```

# Tratando Erros

- JavaScript possui ***try / catch***, mas eles não funcionam com **Promise**
- É necessário usar **.catch** ou passar uma **errorCallback** como 2º argumento para **.then**
- Se usarmos **await**, podemos usar ***try / catch*** sem problemas:

Se usarmos `await`, podemos usar `try / catch` sem problemas:

// com await 🍷

```
try {  
  const noticia = await dados('noticias')  
  const template = await formata(noticia)  
  mostraNoticia(template)  
} catch (erro) {  
  mostraUmaPropaganda()  
  console.error(erro)  
}
```

// com promise.catch

```
dados('noticias')  
  .then(formata)  
  .then(mostraNoticia)  
  .catch(erro => {  
    mostraUmaPropaganda()  
    console.error(erro)  
  })
```

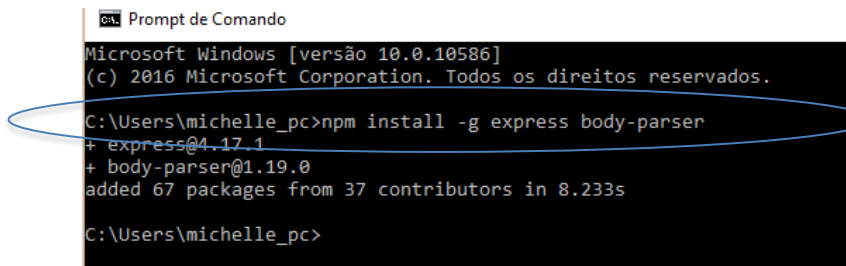
# Criando uma API Node com Express.js

O Express permite criar aplicações web simples e outras aplicações.

**Passo 1:** Para instalar acesse o terminal e digite:

**npm install -g express body-parser**

O comando **npm install -g** instala o Express globalmente no seu sistema, acrescentar dependências com o comando **body-parser**.



```
cmd Prompt de Comando
Microsoft Windows [versão 10.0.10586]
(c) 2016 Microsoft Corporation. Todos os direitos reservados.

C:\Users\michelle_pc>npm install -g express body-parser
+ express@4.17.1
+ body-parser@1.19.0
added 67 packages from 37 contributors in 8.233s

C:\Users\michelle_pc>
```

# Criando uma API Node com Express.js

## Passo 2: Criando a pasta da API

Ainda no terminal, crie um diretório com o nome da sua aplicação e vá até o diretório criado.

```
C:\> Prompt de Comando

D:\testes_node>mkdir api-node

D:\testes_node>cd api-node

D:\testes_node\api-node>_
```

# Criando uma API Node com Express.js

## Passo 2: Criando a pasta da API

Ainda no terminal, crie um diretório com o nome da sua aplicação e vá até o diretório criado.

```
C:\> Prompt de Comando

D:\testes_node>mkdir api-node

D:\testes_node>cd api-node

D:\testes_node\api-node>_
```

# Criando uma API Node com Express.js

**Passo 3:** Baixe o módulo do node na sua pasta. Cria uma estrutura de diretórios dentro de node\_modules.

```
npm install express
```

**Passo 4:** Criar o arquivo de configurações package.json

```
D:\testes_node\api-node>npm init -y
Wrote to D:\testes_node\api-node\package.json:

{
  "name": "api-node",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

# Criando uma API Node com Express.js

## Passo 5: Criando o arquivo app.js

Agora, dentro do diretório **api-node**, abra o seu editor de textos preferido (ex. Visual Studio e crie um arquivo chamado **app.js**. Vamos precisar usar o pacote do Express que foi instalado anteriormente para poder criar a nossa API. Vejamos como vai ficar o nosso arquivo **app.js**:



# Criando uma API Node com Express.js

```
var express = require('express');
var bodyParser = require('body-parser');
var app = express();

app.use(bodyParser.json());

app.get('/maps', function(req, res) {
  var dados = [
    {
      lat: -25.470991,
      lon: -49.271036
    },
    {
      lat: -0.935586,
      lon: -49.635540
    },
    {
      lat: -2.485874,
      lon: -43.128493
    }
  ];

  res.send(JSON.stringify(dados));
});

app.listen(8000, function() {
  console.log('Servidor rodando na porta 8000.');
```

Importa os pacotes instalados.

body-parser será o responsável por parsear as requisições via JSON.

configura o Express e atribui à variável app a ele

configura a forma como iremos tratar nossas requisições no Express.

rotear a URL /maps via método GET:

Quando o usuário navegar até a rota /maps, nosso servidor vai retornar um array de objetos contendo informações com latitude e longitude.

configurar uma porta onde nosso servidor web ficará escutando as chamadas HTTP.

# Criando uma API Node com Express.js

## Passo 6: Executar o arquivo app.js e testar no navegador

```
D:\testes_node\api-node>node app.js  
Servidor rodando na porta 8000.
```

<http://localhost:8000/maps>

← → ↻ ⓘ localhost:8000/maps

[{"lat":-25.470991,"lon":-49.271036},{"lat":-0.935586,"lon":-49.63554},{"lat":-2.485874,"lon":-43.128493}]

# Learnyounode



- <https://nodeschool.io/pt-br/>
- Workshop que ensina conceitos sobre o Node

A screenshot of a Windows PowerShell terminal window. The title bar says "Windows PowerShell". The terminal has a blue background with white text. At the top, it says "LEARN YOU THE NODE.JS FOR MUCH WIN!" and "Select an exercise and hit Enter to begin". Below this is a list of exercises: "» HELLO WORLD", "» BABY STEPS", "» MY FIRST I/O!", "» MY FIRST ASYNC I/O!", "» FILTERED LS", "» MAKE IT MODULAR", "» HTTP CLIENT", "» HTTP COLLECT", "» JUGGLING ASYNC", "» TIME SERVER", "» HTTP FILE SERVER", "» HTTP UPPERCASER", and "» HTTP JSON API SERVER". To the right of the first five exercises, the word "[COMPLETED]" is displayed. At the bottom, there are three options: "HELP", "CREDITS", and "EXIT".

Para instalar o learnyounode, faremos de forma global com o npm:

**\$ npm install -g learnyounode**

Para executar:

**\$ learnyounode**



- Fazer os exercícios de 1 ao número 6

# Referências

<https://fegemo.github.io/cefet-web/classes/js7>

<https://medium.com/balta-io/nodejs-async-await-21ca3636252a>

<https://fegemo.github.io/cefet-web/classes/ssn6>