Curso de Desenvolvimento WEB

Boas Práticas e Estruturação de Projeto

Introdução

Introdução

Esse documento tem como objetivo passar a vocês instruções sobre como proceder no desenvolvimento em grupo de um *software*.

É importante que os padrões sejam seguidos para facilitar o entendimento mútuo do código escrito e facilitar a documentação e aprendizado de novos membros do grupo que possam vir a desenvolver nesse mesmo projeto em momentos futuros.

1. Programação em Inglês

Programação em Inglês

- Um dos maiores erros no desenvolvimento de *software* inicialmente é o uso da linguagem nativa na declaração de variáveis e funções do seu código.

- O inglês é considerado a linguagem internacional e, devido a isso, é dada como a linguagem padrão para codificação de aplicações e a documentação dessas. O seguimento dessa tendência implica não somente numa maior abrangência da sua aplicação em um mercado internacionalmente integrado quanto também melhora sua capacidade de entendimento de qualquer tipo de código utilizado ao longo da sua carreira como desenvolvedor, uma vez que todos eles também serão escritos em inglês.

- Programando em inglês garante que todo o seu código esteja padronizado em uma única língua, uma vez que todas as linguagens de programação existentes têm seus comandos e palavras-chave escritos em inglês.

Programação em Inglês

- Se algum dia você pretende trabalhar fora do Brasil, ou em uma empresa que realiza trabalho remoto internacionalmente, é importante que os seus repositórios do GitHub estejam sempre codificados e documentados em inglês, facilitando a percepção do seu trabalho por pessoas de qualquer origem.

- Além disso, diversas ferramentas e *frameworks* utilizam da semântica inglesa americana para prever nomeação de tabelas em banco de dados. Por exemplo, um *framework*, ao identificar um modelo cujo nome seja **Person**, automaticamente identifica que o mesmo pertence a entradas na tabela **people**, o seu variante plural.

- Outro motivo está na restrição da declaração de variáveis, funções e outras estruturas para nomes sem acentuação que representam exatamente o seu significado morfológico. A exemplo, a palavra **cotação**, caso utilizada para nomear uma variável, teria de ser escrita **cotacao**, enquanto na codificação em inglês que não possui acentuação, essa seria chamada de **price**, mantendo tanto seu sentido semântico quanto sua escrita inalteradas.

2. *CamelCase*

CamelCase

- ***CamelCase*** é uma norma implícita de programação quanto à nomeação de variáveis e funções.

- Consiste no uso da variação de letra minúscula para maiúscula para representar o espaçamento entre as palavras.

- É largamente utilizada no desenvolvimento de *software* e por grande parte das linguagens de programação modernas, bem como *frameworks* e bibliotecas de todos os espectros da programação.

- Seu uso foi tão amplamente aceito pela comunidade que existe uma regra do **ESLint** para garantir que todas as variáveis sejam declaradas seguindo esse padrão, em vez do uso do caractere *underline* (_), por exemplo.

CamelCase

- Alguns exemplos de sua utilização:
  - user_password -> userPassword
  - product_owner -> productOwner
  - get_user_by_email() -> getUserByEmail()

CamelCase

- O *CamelCase* pode ser separado em duas vertentes: **lowerCamelCase** e **UpperCamelCase**.

- Em geral, utilizamos o **lowerCamelCase** para declaração de variáveis e métodos, enquanto o **UpperCamelCase** é utilizado para nomeação de classes e modelos de banco de dados. Para outros casos específicos, estabeleçam um padrão com o restante da equipe para que o projeto siga somente uma estrutura de nomeação.

3. Desestruturação de Objetos

Desestruturação de Objetos

- A **Desestruturação de Objetos** é uma expressão JavaScript que permite a atribuição de chaves de um objeto a nova variáveis locais de forma simples e elegante.

- É uma das boas práticas no desenvolvimento de aplicações JavaScript, além de muito útil para tratamento de requisições POST via formulário, conforme o exemplo a seguir.

- Para mais informações do uso dessa técnica e consultas futuras, peço que leiam esse tópico do *Mozilla Developer Network*: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Operators/Atribuicao_via_desestruturacao.

Desestruturação de Objetos

```
const req = {
    body: {
        id: 123456789,
        email: 'tulio.ao@gmail.com',
        name: 'Túlio',
        lastName: 'Araújo',
        subject: 'Assunto do E-mail',
        body: 'Olá! Esse é o corpo do e-mail'
    }
};

// Recebe os campos de req.body em variáveis locais
// com o mesmo nome de cada parâmatro do objeto
const {
    id,
    email,
    name,
    lastName: middleName, // Novo nome para lastName
    subject,
    body
} = req.body;

// Variáveis declaradas: id, email, name, middleName, subject, body
console.log(id); // 123456789
console.log(email); // tulio.ao@gmail.com
console.log(name); // Túlio
console.log(lastName); // Undefined
console.log(middleName); // Araújo
console.log(subject); // Assunto do E-mail
console.log(body); // Olá! Esse é o corpo do e-mail
```

4. *Template Strings*

Template Strings

- As *Template Strings* são literais *string* que permitem escrita em multi-linha e interpolação de *strings*.

- É uma ferramenta JavaScript muito utilizada para construção de *strings* interpoladas com variáveis, a exemplo do texto a ser enviado por e-mail na seção de contato do AgroWeb.

- Utiliza acentos graves (crases) ao invés de aspas simples, podendo ser fechada em linhas diferentes.

- Capaz de encapsular expressões dentro de si com a sintaxe ${expression}, tornando o código mais legível.

- O exemplo a seguir ilustra os casos de uso mais comuns, que serão utilizados por nós. Uma descrição mais completa pode ser lida no tópico: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/template_strings.

Template Strings

Strings multi-linha

```
console.log('texto string linha 1\n\
texto string linha 2');
// texto string linha 1
// texto string linha 2

console.log(`texto string linha 1
texto string linha 2`);
// texto string linha 1
// texto string linha 2
```

Interpolação de Expressões

```
const a = 5;
const b = 10;
console.log('Quinze é ' + (a + b) + ' e não ' + (2 * a + b) + '.');
// Quinze é 15 e não 20.
console.log(`Quinze é ${a + b} e não ${2 * a + b}.`);
// Quinze é 15 e não 20.
```

5. Comentários

Comentários

- Comentários são fundamentais para facilitar o entendimento do código e melhorar sua legibilidade em um projeto.

- Para descrições simples, é suficiente o comentário padrão JavaScript denotado por meio de barras duplas //.

- Porém, para comentários em de funções ou definição de rotas do *framework* **ExpressJS**, por exemplo, é mais interessante o uso dos comentários em bloco de definidos pelos padrões de documentação do JavaScript: **JSDoc**.

- Esse comentário em blocos pode ser facilmente escrito pelo atalho existente no editor **Atom**, entre outros, que consiste em escrever /** + tab.

- A seguir está um exemplo de como deve ser documentado dois trechos de código comuns, bem como uma *cheatsheet* sobre o JSDoc: https://devhints.io/jsdoc.

Comentários

```
/**
 * GET Home Page
 */
router.get('/', (req, res) => {
    // Comentário Simples
    res.render('home');
});
/**
 * POST Login Request
 */
router.post('/login', (req, res) => {
    ...
});
/**
 * POST Logoff Request
 */
router.post('/logoff', (req, res) => {
    ...
});
```

```
/**
 * Função de Soma
 * @param {number} a - primeiro elemento da soma
 * @param {number} b - segundo elemento da soma
 * @returns {number} Soma dos elementos
 */
const sum = (a, b) => {
    return (a + b);
};

/**
 * Função de Soma Assíncrona
 * @param {number} a - primeiro elemento da soma
 * @param {number} b - segundo elemento da soma
 * @returns {Promise} Objeto Promise que representa a soma
 */
const promiseSum = (a, b) => {
    return new Promise((resolve, reject) => {
        resolve(a + b);
    });
};
```

5. *RESTful Routing*

RESTful Routing

- ***RESTful Routing*** é uma padronização de nomes para rotas convencionais de um fluxo **CRUD** (**C**reate, **R**ead, **U**pdate, **D**estroy), ou seja, é o mapeamento de cada uma das funções do CRUD para as rotas HTTP da aplicação WEB.

- A convenção de rotas para cada recurso de sua aplicação facilita o seu desenvolvimento e a sua facilidade de conversão em uma **API** (**A**pplication **P**rogramming **I**nterface), ou seja, a sua mesma aplicação WEB pode, com pequenas alterações, aceitar requisições de um aplicativo mobile e integrá-los em um mesmo sistema.

RESTful Routing

- As operações CRUD são as possíveis manipulações de um recurso (usuário, produto, etc.) de sua aplicação por meio de formulários específicos, sendo elas: Create, Read, Update e Destroy.

- Por outro lado, as rotas do padrão *REST* são definidas em 7 operações, sendo elas:

    1. **Index:** Lista todas as entradas de um recurso.
    2. **New:** Exibe o formulário de criação de recurso.
    3. **Create:** Requisição do formulário anterior para criar uma nova entrada do recurso.
    4. **Read:** Exibe os detalhes de uma entrada.
    5. **Edit:** Exibe o formulário de edição de recurso preenchido com uma entrada específica.
    6. **Update:** Requisição do formulário anterior para atualizar uma entrada do recurso.
    7. **Destroy:** Requisição para deletar uma entrada do recurso.

RESTful Routing

RESTful Routes

A table of all 7 RESTful routes

Name	Path	HTTP Verb	Purpose	Mongoose Method
Index	/dogs	GET	List all dogs	Dog.find()
New	/dogs/new	GET	Show new dog form	N/A
Create	/dogs	POST	Create a new dog, then redirect somewhere	Dog.create()
Show	/dogs/:id	GET	Show info about one specific dog	Dog.findById()
Edit	/dogs/:id/edit	GET	Show edit form for one dog	Dog.findById()
Update	/dogs/:id	PUT	Update particular dog, then redirect somewhere	Dog.findByIdAndUpdate()
Destroy	/dogs/:id	DELETE	Delete a particular dog, then redirect somewhere	Dog.findByIdAndRemove()

RESTful Routing

- Como visto no exemplo anterior, o recurso utilizado foi **dogs**, ou seja, existe uma tabela chamada **dogs** no banco de dados cujo fluxo de CRUD é implementado por meio das 7 rotas REST.

- O campo **:id** nas rotas é uma parametrização do identificador do recurso utilizado. Ele é facilmente implementado do roteador do ExpressJS e veremos como isso é feito com detalhes em nossa próxima aula.

- A última coluna representa os métodos utilizados para fazer as modificações pertinentes no banco de dados da aplicação, sendo esse uma implementação do **MongoDB**. No nosso caso, estamos utilizando o **Firebase Firestore** e, portanto, iremos alterar esses métodos para corresponder à nossa arquitetura.

- Além disso, pode-se notar que existem dois novos verbos HTTP além de GET e POST, sendo eles os verbos **PUT** e **DELETE**, que em realidade são métodos POST acompanhados de um cabeçalho utilizado para abstração da requisição que representam (Update e Destroy, respectivamente), funcionando de forma similar mas facilitando a configuração das rotas na nossa aplicação.

RESTful Routing

```
var express = require("express");
var router  = express.Router();

/**
 * GET Index - Show all dogs
 */
router.get('/', (req, res) => {
    ...
});

/**
 * GET New - Show form to create new dog
 */
router.get('/new', (req, res) => {
    ...
});

/**
 * POST Create - Add new dog to DB
 */
router.post('/', (req, res) => {
    ...
});
```

```
/**
 * GET Show - Show more info about one dog
 */
router.get('/:id', (req, res) => {
    // The URL id is passed as a request param
    console.log(`:id is ${req.params.id}`);
    ...
});


/**
 * GET Edit - Show the dog edit form
 */
router.get('/:id/edit', (req, res) => {
    ...
});

/**
 * PUT Update - Update the dog in DB
 */
router.put("/:id", (req, res) => {
    ...
});
```

RESTful Routing

```
/**
 * DELETE Destroy - Removes the dog from DB
 */
router.delete("/:id", (req, res) => {
    ...
});

module.exports = router;
```

RESTful Routing

- Na próxima aula, iremos detalhar o funcionamento das rotas REST e como essas serão implementadas em nossos formulários *front-end*.

- Porém, a partir da descrição feita, considerem a seguinte metodologia para estruturação das rotas de sua aplicação: Todas as rotas da aplicação serão estruturadas com orientação aos recursos dessa.

- Utilizando o projeto AgroWeb como exemplo, teremos rotas para: **Produtos**, **Cotações** e **Usuários**. Uma excessão a essa metodologia são as rotas para fluxo de *login* e *logoff.*

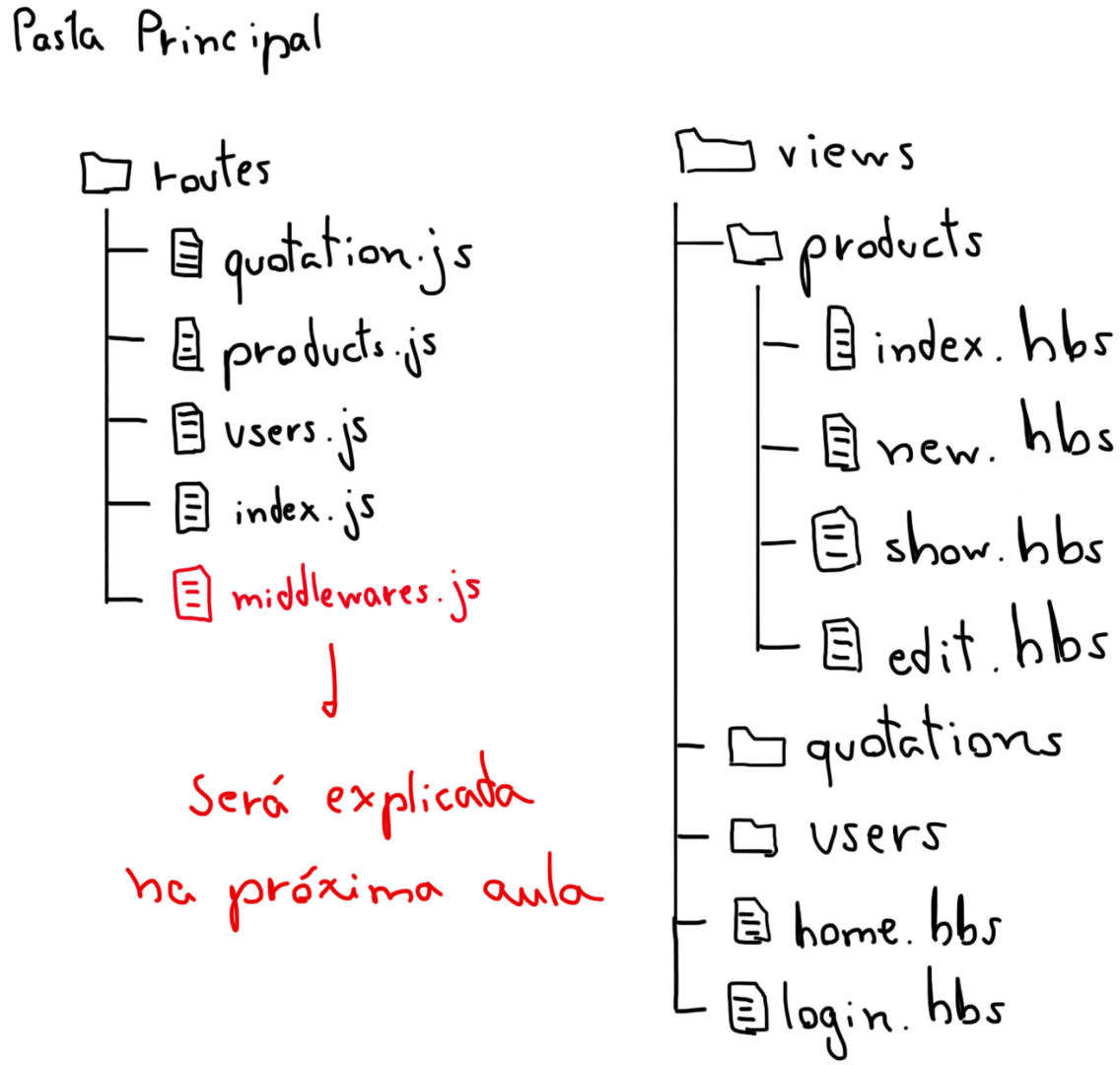
RESTful Routing

- Dado isso, a lógica de *back-end* para controle do acesso de cada uma dessas rotas será feita por meio do que chamamos de *middlewares*, métodos intermediários que fazem a autenticação da sessão de usuário antes de permitir que esse acesse uma determinada parte da aplicação, verificando se o mesmo possuí a permissão necessária para determinado recurso.

- Esses métodos constituem grande parte da lógica do roteamento em uma aplicação ExpressJS e, por isso, serão explicados presencialmente.

- Sobre a estruturação do projeto, esse padrão permite uma fácil organização das pastas, de forma que cada recurso possuirá seu próprio arquivo de rotas e as *views* desses recursos serão separadas também, conforme o esquema a seguir.

RESTful Routing



6. Folhas de Estilo e *Scripts*

Folhas de Estilo e *Scripts*

- Assim como foi explicado, nossa aplicação utiliza o pré-processador de CSS **SASS**, de forma que todas as folhas de estilo podem ser compiladas em um único arquivo **style.css**.

- Como sugestão e boa prática, é recomendado que todas as folhas de estilo customizadas criadas por vocês sejam organizadas de forma intuitiva e incluídas no arquivo principal **style.sass** com o uso da diretiva **@import**.

- No cabeçalho da sua *View* será incluída, então, somente o arquivo **style.css**.

Folhas de Estilo e *Scripts*

- Na parte de *scripts*, muitas páginas criadas por vocês estão realizando a importação por meio de **CDNs** (**C**ontent **D**elivery **N**etworks), o que facilita o desenvolvimento da aplicação mas reduz a velocidade de carregamento das páginas, pois buscam o conteúdo em outros servidores WEB.

- Para conteúdo amplamente utilizado no projeto e carregado diversas vezes, é interessante manter uma cópia local que acelera o seu processamento e garante a estabilidade da sua aplicação, uma vez que não depende da disponibilidade do recurso externamente. Um exemplo disso é o *script* do **JQuery**, que deve ser importado de um arquivo local.

7. Conclusão

Conclusão

- Para facilitar o desenvolvimento em grupo da aplicação, SIGAM os padrões estabelecidos pelo projeto, mesmo que esses difiram dos citados aqui. Para facilitar, sigam as normas do **ESLint**: Usem *arrow functions*, *const* e *let*, nomenclatura *CamelCase*, *template strings*, desestruturação de objetos, comentários no padrão JSDoc, etc.

- No quesito de estruturação da aplicação, sigam os padrões estabelecidos pelo ***RESTful Routing***. Isso será um norte para criação de novas funcionalidades e integração dessas à aplicação de forma modular.

- Organizem os *scripts* e folhas de estilo de forma homogênea ao longo da aplicação, evitando mudanças súbitas de padrão entre as *views*.