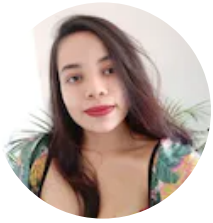


Ei, eu sou Isabel Rubim.

Construindo um simples projeto MVC do zero com JavaScript

out 13, 2021

Gostaria de mostrar um pouco do que é a arquitetura **Model View Controller (MVC)** com um aplicativo simples, no começo parece ser um conceito muito abstrato e difícil, mas com uma demostração pode tornar o aprendizado mais fácil, então vem comigo que vou te ensinar como construir um projeto do zero para te ajudar a entender um pouco da arquitetura MVC apenas com JavaScript!

O projeto que vamos construir é bem simples, vamos utilizar a **API do GitHub** para buscar seu perfil e alguns dados 😜

Mas antes deixa eu te dar uma breve introdução do que seria uma arquitetura MVC.

- **Model** ele é o manipulador de dados, no caso aqui vamos consumir a API do GitHub e tratar esse dados no model;
- A **View** é o que você quer mostrar na tela e também manipular elementos que estão diretamente ligados a visualização;
- O **Controller** é o que faz a comunicação entre o Model e a View, é ele que vai saber qual dado é transitado entre os dois.

Ok, agora que sabemos um pouco do que é MVC, mãos à obra 🔨

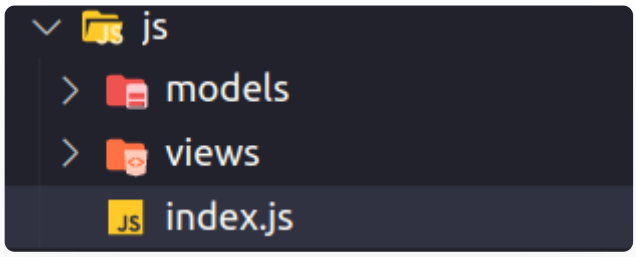
Inicialmente, vamos criar nosso HTML sendo estruturado assim:

```
<body>
  <header>
```

```
    GitHub App - Model View Controller (MVC)
  </header>

  <main class="container">
    <section class="profile">
    </section>

    <form class="filter">
      <input type="text" value="" />
      <button type="submit">Buscar</button>
    </form>

    <section class="repo">
      <h2>Repositórios favoritos</h2>
      <div class="repo-list"></div>
    </section>
  </main>

  <script type="module" src="js/index.js"></script>
</body>
```

Você deve estar pensando, porque exitem essas seções "vazias" sem conteúdo ou todo o código HTML com essas classes definidas nas tags, bem elas vão ser para referenciamos na nossa View!

Vamos criar uma pasta na raiz do nosso projeto chamada **js** que terá essa estrutura:



A pasta para as views, models e o arquivo index.js que será nosso controller.

Vamos continuar nas views, nela vamos ter 3 arquivos.

Vamos começar pelo o arquivo **base.js** que vamos ter funções e elementos que manipula as visualizações. Primeiro, vamos criar um objeto **elements** que terá referências das nossas classes definidas no HTML, para que a gente possa usar como referência quando quisermos adicionar ou tirar alguma elemento da visualização 🎨

Ele será assim:

```
export const elements = {
  profile: document.querySelector(".profile"),
  input: document.querySelector("input"),
  repos: document.querySelector(".repo-list"),
};
```

Em seguida, vamos criar um arquivo de **userView.js** dentro da pasta views, ele terá uma função **renderUser** que vai montar nosso HTML quando receber os dados do usuário do GitHub:

```
import { elements } from "./base.js";

export const renderUser = ({
  avatar_url,
  html_url,
  public_repos,
  followers,
  following,
}) => {
  const markup = `
    <div class="profile-header">
      <img src="${avatar_url}" alt="Avatar">
      <a href="${html_url}" target="_blank">Visitar perfil</a>
    </div>
    <ul class="profile-list">
      <li>Repositórios: ${public_repos}</li>
      <li>Seguidores: ${followers}</li>
      <li>Seguindo: ${following}</li>
    </ul>
  `;

  elements.profile.insertAdjacentHTML("afterbegin", markup);
};
```

A função **renderUser** utilizamos nosso objeto **elements** para referenciar a classe **profile**, utilizamos essa referência para inserir ela dentro da nossa section profile definida no

nosso HTML com o método **insertAdjacentHTML** que recebe a posição como primeiro parâmetro e como segundo o nosso HTML.

Ok, agora vamos para o model da nossa view user.

Dentro da pasta **models** vamos criar um arquivo chamado **User.js**, ele vai conter uma classe User que vai fazer uma requisição para API do GitHub e como resposta irá retornar os dados do usuário que vamos utilizar na nossa view.

```
class User {
  constructor(user) {
    this.user = user;
  }

  async getUser() {
    try {
      const apiUrl = `https://api.github.com/users/${this.user}`;
      const apiUrlStarred = `https://api.github.com/users/${this.user}/st
      const response = await fetch(apiUrl);
      const result = await response.json();

      this.avatar_url = result.avatar_url;
      this.followers = result.followers;
      this.following = result.following;
      this.public_repos = result.public_repos;
      this.html_url = result.html_url;
      this.starred_url = apiUrlStarred;
    } catch (error) {
      console.log(error);
    }
  }
}

export { User };
```

Agora que terminamos a nossa view e model para o User, vamos para o Repo que vai conter tudo relacionado a repositórios do GitHub que vamos utilizar.

Vamos começar da view do repo, dentro da pasta **views**, vamos criar um arquivo chamado **repoView.js**

```
import { elements } from "./base.js";

export const renderRepositories = (repositories) => {
  let markup = "";

  repositories.forEach(({ html_url, name }) => {
    markup += `
      <a href="${html_url}" class="repo-url" target="_blank">
        ${name}
      </a>
    `;
  });

  elements.repos.innerHTML = markup;
};
```

Com a mesma dinâmica para o userView.js, fizemos aqui para o repositório, a função **renderRepositories** vai receber uma lista de repositórios e vai montar a visualização dos repos.

Para o nosso model do repositório, vamos criar um arquivo dentro da pasta **models** chamado **Repo.js**

```
class Repo {
  constructor(repoUrl) {
    this.repoUrl = repoUrl;
  }

  async getRepositories() {
    try {
      const response = await fetch(this.repoUrl);
      this.repos = await response.json();
    } catch (error) {
      console.log(error);
    }
  }
```

```
}

export { Repo };
```

A classe Repo irá receber uma URL que vai retornar os repositórios do usuário que vamos buscar.

Estamos quase lá para finalizarmos nosso projeto, só falta um arquivo e não menos importante que é... 😈

Isso mesmo, o nosso **Controller** 🎮

Lembra do arquivo **index.js** que está na raiz da pasta **js**? Então, vamos nele!

Vamos criar duas funções controller, uma para pegar os dados do usuário do nosso **model User** e renderizar a nossa **view User** e outra para pegar os repositórios do nosso **model Repo** e enviar para a nossa **view Repo**.

Veja que aqui deixa bem claro que ele é o comunicador entre o model e a view.

```
import { User } from "./models/User.js";
import { Repo } from "./models/Repo.js";

import * as userView from "./views/userView.js";
import * as repoView from "./views/repoView.js";

import { clearUI, elements } from "./views/base.js";

const state = {};

const controlFavoriteRepositories = async (url) => {
  try {
    state.repositories = new Repo(url);

    await state.repositories.getRepositories();

    repoView.renderRepositories(state.repositories.repos);
  } catch (error) {
    console.log(error);
  }
```

```
};

const controlSearch = async (event) => {
  event.preventDefault();

  try {
    const searched = elements.input.value;

    state.user = new User(searched);

    await state.user.getUser();

    clearUI();

    userView.renderUser(state.user);

    await controlFavoriteRepositories(state.user.starred_url);

    elements.input.value = "";
  } catch (error) {
    console.log(error);
  }
};

window.addEventListener("submit", controlSearch);
window.addEventListener("load", controlSearch);
```

Vamos começar pela função **controlSearch** que tem como objetivo receber o nome do usuário que foi digitado no input e que será enviado para o nosso model User. Em seguida, com os dados do usuário, guardamos esse valor dentro de um state global definido no inicio do arquivo, para futuramente se quisermos utilizar algum dado que já fizemos requisição, podemos chamar o state e usar os valores que contém nele.

Depois chamamos o método **getUser()** que irá trazer os dados do usuário, logo depois chamamos a função **clearUI()** que foi criada no arquivo base para remover o HTML existente e montar uma nova visualização do usuário assim que for pesquisado o seu user. Dentro do arquivo **base.js** vamos criar ela assim:

```
export const clearUI = () => {
  elements.profile.innerHTML = "";
```

```
};
```

Com o bloco profile vazio, chamamos a função para renderizar a visualização do usuário pesquisado. Em seguida, chamamos o controller **controlFavoriteRepositories** passando a URL que obtemos para renderizar os repositórios favoritos do usuário pesquisado.

O controller para renderizar os repositórios segue o mesmo padrão do usuário, primeiro chamamos a classe Repo e em seguida obtemos os dados para passar para a view e assim mostrando na tela os repositórios favoritos.

Alguns detalhes para finalizarmos, no final do arquivo index.js contém dois eventos **submit** e **load**, o submit vai ser acionado assim que for pesquisado algum nome de usuário e o load para renderizar o valor default do input definido no HTML, ambos chamando a função **controlSearch**.

Coloquei meu nome de usuário como default no value do input, mas fique a vontade de colocar o seu nome de usuário do GitHub!

```
<input type="text" value="IsabelRubim" />
```

Além disso, criei uma pasta chamada **css** na raiz do projeto e um arquivo **styles.css** dentro da pasta para colocar alguns estilos que você pode copiar **aqui**. E depois chamei os estilos no HTML dentro da tag head:

```
<link rel="stylesheet" href="css/styles.css" />
```

Por último, chamamos nosso controller dentro da tag body do HTML:

```
<script type="module" src="js/index.js"></script>
```

E pronto, um aplicativo simples que consulta qualquer usuário do GitHub, livre de bibliotecas e que mostra como funciona a arquitetura Model View Controller. E aqui está os links de demostração e código fonte:

- Ver demostração
- Ver código fonte

Espero que esse tutorial tenha ajudado você a entender MVC. É um padrão facilmente utilizado em projetos reais e que pode ser um ótimo conhecimento para você.

← Voltar

Feito com 💜 Isabel Rubim.