

## Como eu faço um MVC em javascript sem frameworks

O padrão MVC (Model-View-Controller) é uma forma de organizar a estrutura de um aplicativo em três partes distintas: o modelo, a visualização e o controlador. A ideia é que cada uma dessas partes tenha uma responsabilidade específica e que o código seja dividido de uma forma que facilite a manutenção e evolução do aplicativo.

Para implementar o padrão MVC em JavaScript sem o uso de frameworks, você pode seguir os seguintes passos:

- 1 Crie o modelo: o modelo representa a camada de dados do aplicativo. Você pode criar uma classe ou função que seja responsável por gerenciar os dados do aplicativo. Por exemplo:

```
class Model {  
  constructor() {  
    this.data = []  
  }  
  
  getData() {  
    return this.data  
  }  
  
  addData(newData) {  
    this.data.push(newData)  
  }  
}
```

- 1 Crie a visualização: a visualização representa a camada de apresentação do aplicativo. Você pode criar uma função que seja responsável por renderizar os dados do aplicativo na tela. Por exemplo:

```
function renderView(data) {  
  // Renderiza os dados na tela  
}
```

- 1 Crie o controlador: o controlador representa a camada de lógica do aplicativo. Ele é responsável por fazer a ligação entre o modelo e a visualização. Você pode criar uma classe ou função que seja responsável por gerenciar as interações do usuário com o aplicativo. Por exemplo:

```
class Controller {  
  constructor(model, view) {  
    this.model = model  
    this.view = view  
  }  
  
  handleAddData(newData) {  
    this.model.addData(newData)  
    this.view.renderView(this.model.getData())  
  }  
}
```

- 1 Inicialize o aplicativo: por fim, você pode criar uma função que seja responsável por inicializar o aplicativo, criando uma instância do modelo, da visualização e do controlador. Por exemplo:

```
function init() {  
  const model = new Model()  
  const view = { renderView }  
  const controller = new Controller(model, view)  
  
  // Adiciona um evento para quando o usuário adicionar um novo dado  
  document.getElementById('add-data-button').addEventListener('click', () => {  
    const newData = document.getElementById('new-data-input').value  
    controller.handleAddData(newData)  
  })  
}
```

Note que este é apenas um exemplo de como você pode implementar o padrão MVC em JavaScript sem o uso de frameworks. É possível que em aplicações maiores e mais complexas seja necessário utilizar outras abordagens para melhorar a manutenção e escalabilidade do código.

Mostre os códigos completos, de uma aplicação completa, que use uma API da wikipédia que ao digitado um termo puxe o texto da wikipédia ou outra API parecida

Segue abaixo um exemplo completo de uma aplicação que utiliza a API da Wikipedia para buscar informações a partir de um termo digitado pelo usuário:

index.html

```
<!DOCTYPE html>  
<html>  
  <head>  
    <meta charset="UTF-8">  
    <title>Busca na Wikipedia</title>  
    <script src="script.js"></script>  
  </head>  
  <body>  
    <h1>Busca na Wikipedia</h1>
```

use mvc

Abaixo segue um exemplo de como implementar a aplicação de busca na Wikipedia utilizando o padrão MVC (Model-View-Controller):

index.html

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Busca na Wikipedia</title>
    <script src="model.js"></script>
    <script src="view.js"></script>
    <script src="controller.js"></script>
  </head>
  <body>
    <h1>Busca na Wikipedia</h1>
    <label for="search-input">Termo:</label>
    <input type="text" id="search-input">
    <button id="search-button">Buscar</button>
    <div id="results"></div>
  </body>
</html>

```

#### model.js

```

class WikipediaAPI {
  static async search(term) {
    const response = await fetch(`https://en.wikipedia.org/w/api.php?action=query&format=json&list=search&srsearch=${term}`)
    const data = await response.json()
    return data
  }

  static async getPageContent(pageId) {
    const response = await fetch(`https://en.wikipedia.org/w/api.php?action=query&format=json&prop=extracts&pageids=${pageId}`)
    const data = await response.json()
    return data
  }
}

```

#### view.js

```

class WikipediaView {
  constructor() {
    this.resultsElement = document.getElementById('results')
  }

  showResults(results) {
    this.resultsElement.innerHTML = ''

    results.forEach(result => {
      const div = document.createElement('div')
      div.innerHTML = `
        <h2>${result.title}</h2>
        <p>${result.snippet}</p>
        <button data-page-id="${result.pageid}">Ler mais</button>
      `
      this.resultsElement.appendChild(div)
    })
  }

  showPageContent(title, content) {
    this.resultsElement.innerHTML = `
      <h2>${title}</h2>
      <p>${content}</p>
    `
  }
}

```

controller.js

```
class WikipediaController {
  constructor(model, view) {
    this.model = model
    this.view = view

    this.searchButton = document.getElementById('search-button')
    this.searchInput = document.getElementById('search-input')
    this.resultsElement = document.getElementById('results')

    this.searchButton.addEventListener('click', () => this.search())
    this.resultsElement.addEventListener('click', event => this.showPageContent(event))
  }

  async search() {
    const term = this.searchInput.value
    const results = await this.model.search(term)
    this.view.showResults(results)
  }

  async showPageContent(event) {
    const pageId = event.target.getAttribute('data-page-id')
    if (pageId) {
      const pageData = await this.model.getPageContent(pageId)
      const title = pageData.query.pages[0].title
      const content = pageData.query.pages[0].extract
      this.view.showPageContent(title, content)
    }
  }
}

const model = WikipediaAPI
const view = new WikipediaView()
const controller = new WikipediaController(model, view)
```

Neste exemplo, a classe **WikipediaAPI** é responsável por fazer as chamadas à API da Wikipedia, a classe **WikipediaView** é responsável por mostrar os resultados da busca e o conteúdo da página e a classe **WikipediaController** é responsável por fazer a ligação entre o modelo e a visualização. A função

continue responsável por fazer a ligação entre o modelo e a visualização. A função

A função **search** é responsável por executar a busca a partir do termo digitado pelo usuário e exibir os resultados na visualização. A função **showPageContent** é responsável por exibir o conteúdo completo de uma página da Wikipedia quando o usuário clica no botão "Ler mais" em um dos resultados da busca.

Neste exemplo, o modelo, a visualização e o controlador são separados em arquivos distintos para facilitar a manutenção e o entendimento do código. Além disso, é importante notar que a classe **WikipediaAPI** faz uso de funções assíncronas para realizar as chamadas à API da Wikipedia, e a classe **WikipediaView** usa o método **innerHTML** para adicionar o HTML correspondente aos resultados e ao conteúdo completo da página. Por fim, o controlador faz uso do método **addEventListener** para associar ações aos elementos da interface do usuário.

Mostre outra aplicação, tipo TODO com armazenamento local, usando MVC também e o conceito de módulos se necessários

Abaixo segue um exemplo de como implementar uma aplicação de lista de tarefas (TODO) com armazenamento local, utilizando o padrão MVC e o conceito de módulos:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Lista de Tarefas</title>
    <script src="controller.js"></script>
  </head>
  <body>
    <h1>Lista de Tarefas</h1>
    <form id="todo-form">
      <input type="text" id="todo-input">
      <button type="submit">Adicionar</button>
    </form>
    <ul id="todo-list"></ul>
    <script>
      const controller = new TodoController()
      controller.init()
    </script>
  </body>
</html>
```

model.js

```
const TodoModel = (() => {
  const storage = window.localStorage
  let todos = []

  const saveTodos = () => {
    storage.setItem('todos', JSON.stringify(todos))
  }

  const loadTodos = () => {
    const data = storage.getItem('todos')
    todos = data ? JSON.parse(data) : []
  }

  const addTodo = todo => {
    todos.push(todo)
    saveTodos()
  }

  const removeTodo = todo => {
    todos = todos.filter(t => t !== todo)
    saveTodos()
  }

  const getTodos = () => todos

  return { addTodo, removeTodo, getTodos, loadTodos }
})();
```

view.js

```

const TodoView = (() => {
  const todoForm = document.getElementById('todo-form')
  const todoInput = document.getElementById('todo-input')
  const todoList = document.getElementById('todo-list')

  const renderTodos = todos => {
    todoList.innerHTML = ''

    todos.forEach(todo => {
      const li = document.createElement('li')
      const button = document.createElement('button')
      button.innerText = 'X'
      button.addEventListener('click', () => removeTodoClicked(todo))
      li.innerText = todo
      li.appendChild(button)
      todoList.appendChild(li)
    })
  }

  const addTodo = todo => {
    const li = document.createElement('li')
    const button = document.createElement('button')
    button.innerText = 'X'
    button.addEventListener('click', () => removeTodoClicked(todo))
    li.innerText = todo
    li.appendChild(button)
    todoList.appendChild(li)
  }

  const removeTodoClicked = todo => {
    const event = new CustomEvent('removeTodo', { detail: todo })
    document.dispatchEvent(event)
  }

  todoForm.addEventListener('submit', event => {
    event.preventDefault()
    const todo = todoInput.value
    const event = new CustomEvent('addTodo', { detail: todo })
    document.dispatchEvent(event)
    todoInput.value = ''
  })

  return { renderTodos, addTodo }
})()

```

#### controller.js

```

class TodoController {
  constructor() {
    this.model = TodoModel
    this.view = TodoView

    document.addEventListener('addTodo', event => this.addTodoHandler(event))
    document.addEventListener('removeTodo', event => this.removeTodoHandler(event))
  }

  init() {
    this.model.loadTodos()
    const todos = this.model.getTodos()
    this.view.renderTodos(todos)
  }

  addTodoHandler(event) {
    const todo = event.detail
    this.model.addTodo(todo)
    this.view.addTodo(todo)
  }
}

```

```
removeTodoHandler(event) {
  const todo = event.detail
  this.model.removeTodo(todo)
  const todos = this.model.getTodos()
  this
```

```
termine o código e continue a montar a aplicação pedida: class TodoController {
  constructor() {
    this.model = TodoModel
    this.view = TodoView

    document.addEventListener('addTodo', event => this.addTodoHandler(event))
    document.addEventListener('removeTodo', event => this.removeTodoHandler(event))
  }

  init() {
    this.model.loadTodos()
    const todos = this.model.getTodos()
    this.view.renderTodos(todos)
  }


  addTodoHandler(event) {
    const todo = event.detail
    this.model.addTodo(todo)
    this.view.addTodo(todo)
  }

  removeTodoHandler(event) {
    const todo = event.detail
    this.model.removeTodo(todo)
    const todos = this.model.getTodos()
    this
```

```
this.view.renderTodos(todos) }}
```

Com esses arquivos, temos a estrutura básica do padrão MVC, onde a `TodoModel` representa os dados da aplicação, a `TodoView` representa a interface de usuário e o `TodoController` é responsável por gerenciar a lógica da aplicação.

Além disso, para facilitar o gerenciamento de módulos, podemos utilizar o conceito de IIFE (Immediately Invoked Function Expression). Com essa estrutura, é possível criar uma aplicação de lista de tarefas simples, onde o usuário pode adicionar e remover tarefas.

From ChatGPT, designed with  FancyGPT