Integrantes: Arthur Santana, Bruna Leticia, Gilson Conceição, Henrick Custodio, Jessica Camily, Leonardo Santos, Maria Eduarda Acyole e Rafael Oliveira

Pesquisa - Padrão de Projeto MVC

1. Introdução ao Padrão MVC

O modelo de arquitetura MVC (Model-View-Controller) é muito usado no desenvolvimento de sites e aplicativos modernos. Ele organiza o código em três partes principais: o **Model,** o View e o **Controller**. Cada parte tem uma função bem definida, o que torna o trabalho em equipe mais fácil e permite que o projeto cresça de forma mais organizada.

O MVC é usado em frameworks populares como Ruby on Rails e Django, e também é aplicado em bibliotecas JavaScript, como o React (que usa o conceito de controle de estado), ajudando a criar aplicações mais estruturadas e eficientes.

2. Breve Histórico e Importância

O modelo MVC foi criado na década de 1970 por Trygve Reenskaug, inicialmente para aplicações de desktop. Porém, com o crescimento das aplicações web e a necessidade de separar o que é mostrado ao usuário (frontend) da parte que lida com dados e lógica (backend), o MVC foi adaptado para esse novo contexto.

Esse modelo ajuda os desenvolvedores a organizar o código de maneira que facilite tanto o desenvolvimento quanto a manutenção. Cada parte do MVC tem uma função específica, o que torna o trabalho mais eficiente, principalmente em equipes. Além disso, essa separação permite que cada parte do sistema evolua sem atrapalhar as outras.

3. Classes Model

O Model representa os dados da aplicação e gerencia a comunicação com o banco de dados. Ele é responsável por criar, ler, atualizar e deletar (CRUD) informações, tornando-o uma peça central na persistência de dados.

Exemplo Prático de Model

Abaixo está um exemplo prático de uma classe Model para um sistema de gerenciamento de usuários:

```
// Classe de modelo de usuário
class Usuario {
  constructor(id, nome, email) {
    this.id = id;
    this.nome = nome;
    this.email = email;
  }

  static async buscarTodos() {
    // lógica para buscar todos os usuários no banco de dados
  }

  static async buscarPorId(id) {
    // lógica para buscar um usuário por ID no banco de dados
  }
}

module.exports = Usuario;
```

Aqui, a classe Usuario tem métodos para buscar todos os usuários e buscar por ID, demonstrando como o Model se comunica com o banco de dados para gerenciar dados da aplicação.

4. Classes View

A View é responsável por exibir os dados da aplicação ao usuário de forma amigável. Ela não tem acesso direto ao banco de dados ou à lógica de negócios, pois recebe as informações processadas pelo Controller e as renderiza.

Exemplo Prático de View

```
// Rota para exibir detalhes de um usuário específico
app.get('/usuarios/:id', async (req, res) => {
  const id = req.params.id;
  const usuario = await Usuario.buscarPorId(id);
  res.render('usuario', { usuario });
```

});

Neste exemplo, a View exibe as informações do usuário, sem saber como ou de onde elas foram obtidas. Esse é um exemplo claro de separação de responsabilidades, pois a View só exibe os dados que recebe.

5. Classes Controller

O Controller é o intermediário entre o Model e a View. Ele interpreta as requisições dos usuários, aplica a lógica necessária e determina o que deve ser mostrado na View.

Exemplo Prático de Controller

```
// Controlador de usuários
class UsuarioController {
  async buscarTodos(req, res) {
    const usuarios = await Usuario.buscarTodos();
    res.render('usuarios', { usuarios });
  }

  async buscarPorId(req, res) {
    const id = req.params.id;
    const usuario = await Usuario.buscarPorId(id);
    res.render('usuario', { usuario });
  }
}

module.exports = UsuarioController;
```

Aqui, o UsuarioController faz a ponte entre o usuário e os dados: recebe a requisição, utiliza o Model para buscar os dados, e então exibe o resultado na View.

6. Vantagens do MVC

O modelo MVC traz várias vantagens práticas para o desenvolvimento de aplicações. Veja as principais, com exemplos para facilitar o entendimento:

- Separação de Responsabilidades: O MVC divide o código em partes que cuidam de funções específicas (dados, interface e lógica de controle), o que facilita o trabalho em equipe. Cada pessoa pode focar em uma parte do sistema sem se preocupar com as outras.

- Maior Escalabilidade: Como cada parte é independente, o sistema pode crescer de forma mais organizada. Por exemplo, uma equipe pode criar novas telas ou interfaces para o usuário (View) sem mudar a lógica de dados (Model).

- Facilidade de Manutenção: Com cada parte bem definida, é mais fácil localizar e corrigir problemas. Por exemplo, se houver um erro na exibição de dados, pode-se corrigir diretamente na View, sem precisar mexer no Model ou no Controller.

- Facilidade de Compreensão para Novos Desenvolvedores: Como cada camada tem uma função bem clara, o código fica mais fácil de entender, ajudando novos desenvolvedores a se adaptarem mais rapidamente à equipe.

Essas vantagens tornam o MVC uma estrutura sólida para criar aplicações que crescem de forma organizada e são fáceis de manter.

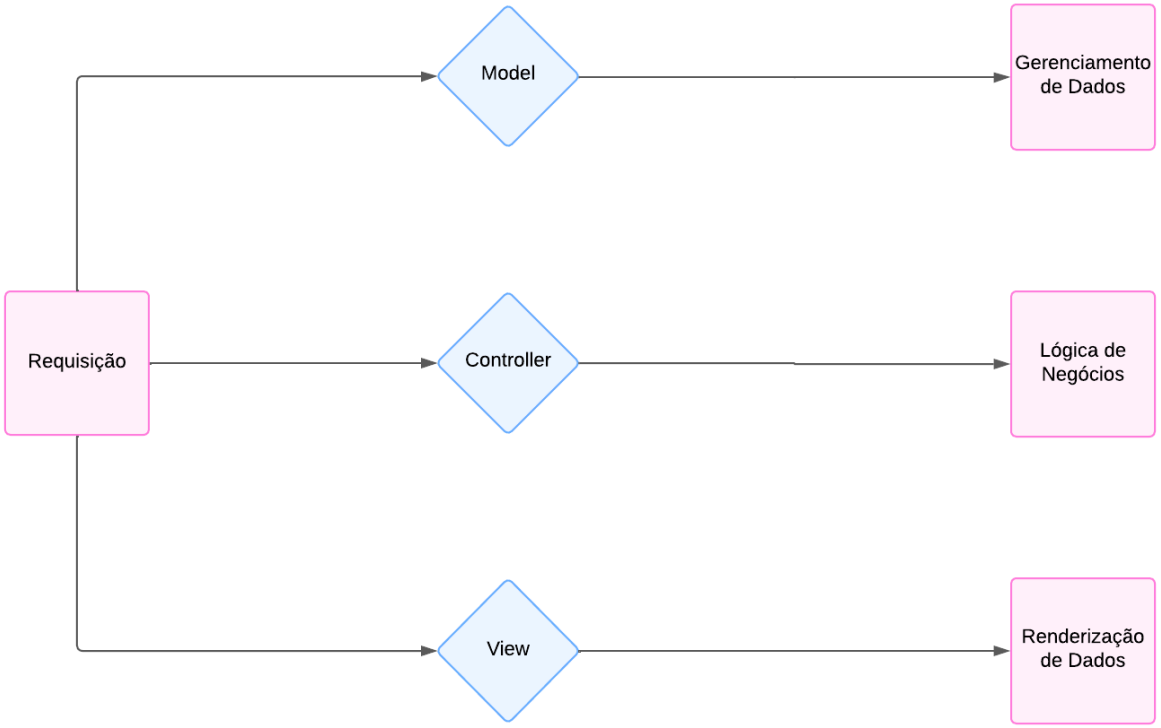
7. Integração do MVC

A integração dos três componentes (Model, View e Controller) forma uma aplicação que funciona de forma organizada, com cada parte desempenhando uma função específica:

1. O *usuário* faz uma ação, como acessar a página de perfil.
2. O **Controller** recebe essa ação, pede ao **Model** para buscar as informações no banco de dados.
3. O **Model** retorna os dados, que são enviados para a **View**, que então exibe a resposta para o usuário.

Cada parte do processo é bem definida, tornando a aplicação mais eficiente e fácil de manter.

Diagrama Simples



Este fluxo ilustra como o MVC mantém cada componente focado em sua responsabilidade específica.

8. Conclusão

O padrão MVC é uma estrutura importante para criar aplicações web organizadas e que podem crescer facilmente. A separação de responsabilidades facilita a manutenção do código e permite que as equipes trabalhem de forma mais eficiente.

Atualmente, o MVC é muito usado em vários frameworks e continua sendo uma escolha popular na indústria de software, devido à sua eficiência.