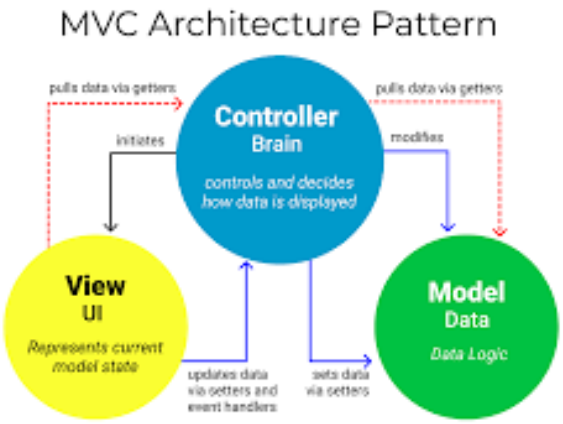
A aplicação em questão é um software back-end web, que possui as seguintes características:

Trata-se de um back-end porque gerencia todas as operações não visíveis ao usuário, como o processamento, manipulação e validação de dados, além da execução da lógica de negócios. Ele também estabelece conexões com bancos de dados para armazenar e recuperar informações, bem como com servidores. Além disso, o back-end define a arquitetura que estrutura esses processos, como ocorre na aplicação de biblioteca descrita, que armazena e gerencia dados de livros, clientes e empréstimos, apresentando apenas os resultados finais ao usuário. A aplicação é classificada como web porque permite a interação com o usuário, transmitindo informações da máquina para ele. Como ela solicita e retorna dados em resposta às ações do usuário, caracteriza-se como uma aplicação web.

A arquitetura adotada foi o padrão MVC (Model-View-Controller), conforme ilustrado no diagrama.



Descrição dos Endpoints Disponíveis

A API oferece os seguintes endpoints para gerenciamento de usuários e tarefas:

Usuários

- GET /usuario
    - **Descrição:** Retorna uma lista com todos os usuários cadastrados.

Resposta esperada: Lista em formato JSON contendo os dados dos usuários.

- POST /usuario

    - **Descrição:** Cadastra um novo usuário.

    - **Dados de entrada (JSON):**

        {

          "id": 1,

         "nome": "João Silva"

        }

    - **Resposta esperada:**"Usuário cadastrado com sucesso."

- PUT /usuario/put/{id}

    - **Descrição:** Atualiza os dados de um usuário existente com base no ID fornecido.

    - **Resposta esperada:** Confirmação da atualização (dependendo da implementação).

- DELETE /usuario/{id}

    - **Descrição:** Remove um usuário específico com base no ID.

    - **Resposta esperada:** Confirmação da exclusão (dependendo da implementação).

Tarefas

- GET /tarefa/usuario/{id}

    - **Descrição:** Retorna todas as tarefas associadas a um usuário específico, identificado pelo ID.

    - **Resposta esperada:** Lista em formato JSON contendo as tarefas do usuário.

POST /tarefa

- **Descrição:** Cadastra uma nova tarefa vinculada a um usuário existente.

- **Dados de entrada (JSON):**


```
{
  "id": 101,
  "titulo": "Estudar Spring Boot",
```

```
"descricao": "Assistir aula e fazer anotações",
"status": "Pendente",
"usuario": {
  "id": 1,
  "nome": "João Silva"
 }
}
```


- **Resposta esperada:** "Tarefa cadastrada com sucesso."

PUT /tarefa/status/{id}

- **Descrição:** Atualiza o status de uma tarefa existente com base no ID.

- **Dados de entrada (Exemplo):**"Concluída"

- **Resposta esperada:** true (se a atualização for bem-sucedida) || false (se ocorrer falha).

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8080/usuario' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "id": 3,
  "nome": "pedraaaa",
  "email": "pedraaaaaaaaaaa"
}'
```

Request URL

```
http://localhost:8080/usuario
```

Server response

Code	Details
200	Response body

```
ID já existente.
```

Download

Responses

Curl

```
curl -X 'DELETE' \
  'http://localhost:8080/usuario/3' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:8080/usuario/3
```

Server response

Code	Details
200	Response body

```
true
```

Download

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:8080/tarefas' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "id": 2,
  "titulo": "pequeno principe",
  "descricao": "aaaa",
  "status": "aaaa",
  "usuario": 0
}'
```

Request URL

```
http://localhost:8080/tarefas
```

Server response

Code	Details
200	Response body

```
Tarefa cadastrada com sucesso.
```

Download

Os padrões de design são soluções reutilizáveis para problemas comuns no desenvolvimento de software. Eles ajudam a tornar o código mais organizado, flexível e fácil de manter. A seguir, vamos explorar os principais 23 padrões de design, divididos em três categorias: criacionais, estruturais e comportamentais.

1. **Singleton**
   Garante que uma classe tenha apenas uma instância e fornece um ponto global de acesso a ela. Útil para gerenciar recursos compartilhados, como conexões de banco de dados.

2. **Factory Method**
   Define uma interface para criar um objeto, mas permite que as subclasses decidam qual classe instanciar. Facilita a criação de objetos sem especificar suas classes concretas.

3. **Abstract Factory**
   Fornece uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas. Ideal para sistemas que precisam ser independentes de como seus objetos são criados.

4. **Builder**
   Separa a construção de um objeto complexo da sua representação, permitindo que o mesmo processo de construção crie diferentes representações.

5. **Prototype**
   Permite criar novos objetos copiando uma instância existente, conhecida como protótipo. Útil quando a criação direta de objetos é cara em termos de

desempenho.

6. **Adapter**
   Permite que interfaces incompatíveis trabalhem juntas, convertendo a interface de uma classe em outra esperada pelos clientes.

7. **Bridge**
   Separa a abstração da implementação, permitindo que ambas evoluam independentemente. Útil para evitar a explosão de subclasses.

8. **Composite**
   Compõe objetos em estruturas de árvore para representar hierarquias parte-todo. Permite tratar objetos individuais e composições de objetos de maneira uniforme.

9. **Decorator**
   Adiciona responsabilidades adicionais a um objeto dinamicamente. Oferece uma alternativa flexível à subclasse para estender funcionalidades.

10. **Facade**
    Fornece uma interface unificada para um conjunto de interfaces em um subsistema, facilitando seu uso.

11. **Flyweight**
    Usa compartilhamento para suportar grandes quantidades de objetos de forma eficiente. Reduz o uso de memória ao compartilhar partes comuns entre objetos.

12. **Proxy**
    Fornece um substituto ou representante de outro objeto para controlar o acesso a ele. Pode adicionar funcionalidades como controle de acesso ou adiamento de inicialização.

13. **Chain of Responsibility**
    Permite que vários objetos tenham a chance de tratar uma solicitação, passando-a pela cadeia até que um objeto a trate.

14. **Command**
    Encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações, enfileirar ou registrar solicitações.

15. **Interpreter**
    Define uma representação gramatical para uma linguagem e um interpretador que usa essa representação para interpretar sentenças da

linguagem.

16. **Iterator**
    Fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.

17. **Mediator**
    Define um objeto que encapsula como um conjunto de objetos interage, promovendo o acoplamento fraco ao evitar que os objetos se refiram explicitamente uns aos outros.

18. **Memento**
    Sem violar o encapsulamento, captura e externaliza o estado interno de um objeto para que ele possa ser restaurado posteriormente.

19. **Observer**
    Define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

20. **State**
    Permite que um objeto altere seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado sua classe.

21. **Strategy**
    Define uma família de algoritmos, encapsula cada um e os torna intercambiáveis. Permite que o algoritmo varie independentemente dos clientes que o utilizam.

22. **Template Method**
    Define o esqueleto de um algoritmo em uma operação, deixando alguns passos para as subclasses. Permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura.

23. **Visitor**
    Representa uma operação a ser realizada nos elementos de uma estrutura de objetos. Permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera.

Neste projeto, alguns padrões de design foram aplicados de forma prática para estruturar e organizar melhor a aplicação. O mais evidente é o padrão MVC (Model-View-Controller), que separa claramente as responsabilidades: a Model

representa os dados da aplicação (como as classes Tarefa e Usuario), a Controller trata as requisições do cliente (como as classes TarefaController e UsuarioController), e a View, neste caso, corresponde às respostas JSON enviadas pela API. Essa separação facilita a manutenção e evolução do sistema. Outro padrão presente é o Singleton, utilizado nas classes TarefaBanco e UsuarioBanco. Ele garante que apenas uma instância de cada "banco" seja usada durante toda a execução da aplicação, permitindo um controle centralizado dos dados simulados em memória. Isso é essencial para evitar inconsistências ou duplicações quando várias partes da aplicação acessam os dados simultaneamente. Além disso, temos o uso do padrão Repository, ainda que de forma manual. As classes que simulam o banco de dados encapsulam toda a lógica de acesso aos dados, como inserções, buscas e atualizações. Essa abordagem separa a lógica de persistência dos dados da lógica de negócios da aplicação, o que facilita futuras mudanças, como a integração com um banco de dados real.