

Alunos: Amanda Nelva Almeida Martins,
João Ricardo de Souza Teixeira,
Monique Evelin Miranda Domingos

RA.: 0018739, 0002028, 0076892

Foi realizado o desenvolvimento do cenário proposto nº 2, que é um aplicativo para organizar tarefas do dia a dia, com funcionalidades de adicionar, marcar como concluída e remover tarefas. Abaixo iremos descrever o funcionamento do modelo, controlador e visão no contexto deste aplicativo.

Código base

App.js

```
const model = new TaskModel();  
const view = new TaskView();  
const controller = new TaskController(model, view);
```

Controller.js

```
class TaskController {  
  constructor(model, view) {  
    this.model = model;  
    this.view = view;  
  
    this.view.bindAddTask(this.handleAddTask.bind(this));  
    this.view.bindCompleteTask(this.handleCompleteTask.bind(this));  
    this.view.bindRemoveTask(this.handleRemoveTask.bind(this));  
  
    this.view.render(this.model.getTasks());  
  }  
  
  handleAddTask(description) {  
    if (description) {  
      this.model.addTask(description);  
      this.view.render(this.model.getTasks());  
    }  
  }  
  
  handleCompleteTask(index) {  
    this.model.completeTask(index);  
    this.view.render(this.model.getTasks());  
  }  
}
```

```
}

handleRemoveTask(index) {
  this.model.removeTask(index);
  this.view.render(this.model.getTasks());
}
}
```

View.js

```
class TaskView {
  constructor() {
    this.taskInput = document.getElementById('taskInput');
    this.addTaskBtn = document.getElementById('addTaskBtn');
    this.pendingTasks = document.getElementById('pendingTasks');
    this.completedTasks = document.getElementById('completedTasks');
  }

  render(tasks) {
    this.pendingTasks.innerHTML = '';
    this.completedTasks.innerHTML = '';

    tasks.forEach((task, index) => {
      const taskItem = document.createElement('li');
      taskItem.textContent = task.description;

      // Checkbox para marcar a tarefa como concluída
      const checkbox = document.createElement('input');
      checkbox.type = 'checkbox';
      checkbox.checked = task.status === 'completed';
      checkbox.addEventListener('change', () =>
this.handleCompleteTask(index, checkbox.checked));

      // Botão de remover tarefa
      const removeBtn = document.createElement('button');
      removeBtn.textContent = 'Remover';
      removeBtn.addEventListener('click', () =>
this.handleRemoveTask(index));

      taskItem.prepend(checkbox);
      taskItem.appendChild(removeBtn);

      // Verificação do status das tarefas
      if (task.status === 'pending') {
```



```
        this.pendingTasks.appendChild(taskItem);
    } else {
        this.completedTasks.appendChild(taskItem);
    }
    });
}

bindAddTask(handler) {
    this.addTaskBtn.addEventListener('click', () => {
        handler(this.taskInput.value);
        this.taskInput.value = '';
    });
}

bindCompleteTask(handler) {
    this.handleCompleteTask = handler;
}

bindRemoveTask(handler) {
    this.handleRemoveTask = handler;
}
}
```

Model.js

```
class TaskModel {
    constructor() {
        this.tasks = [];
    }

    addTask(description) {
        this.tasks.push({ description, status: 'pending' });
    }

    completeTask(index) {
        if (this.tasks[index]) {
            this.tasks[index].status = 'completed';
        }
    }

    removeTask(index) {
        this.tasks.splice(index, 1);
    }
}
```

```
getTasks() {  
    return this.tasks;  
}  
}
```

Index.html

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <title>Gerenciador de Tarefas</title>  
</head>  
<body>  
    <h1>Gerenciador de Tarefas</h1>  
    <input type="text" id="taskInput" placeholder="Digite uma tarefa">  
    <button id="addTaskBtn">Adicionar</button>  
  
    <h2>Tarefas Pendentes</h2>  
    <h4>(Marque a checkbox para concluir a tarefa)</h4>  
    <ul id="pendingTasks"></ul>  
  
    <h2>Tarefas Concluídas</h2>  
    <ul id="completedTasks"></ul>  
  
    <script src="./js/model.js"></script>  
    <script src="./js/controller.js"></script>  
    <script src="./js/view.js"></script>  
    <script src="./js/app.js"></script>  
</body>  
</html>
```

1. Modelo (TaskModel)

O conceito do modelo propõe que ele é responsável por armazenar e gerenciar os dados da aplicação, além de fornecer métodos para manipulação desses dados. Temos abaixo as principais funções:

- **constructor:** Inicializa o vetor (array) **tasks** vazio para armazenar as tarefas.

- **addTask(description):** Adiciona uma nova tarefa ao array de tarefas, definindo inicialmente o status como "pending" (pendente).
- **completeTask(index):** Marca a tarefa como "completed" (concluída), alterando seu status.
- **removeTask(index):** Remove uma tarefa do array com base no índice.
- **getTasks():** Retorna todas as tarefas armazenadas no modelo.

O modelo não se comunica diretamente com o usuário ou com a interface, ao invés disso, fornece ao controlador as ferramentas necessárias para atualizar e consultar as informações.

2. Visão (TaskView)

A visão é responsável por exibir os dados ao usuário e capturar interações. Ela não contém as regras de negócio, apenas atualiza a interface com base nos dados recebidos. Temos abaixo as principais funções:

- **constructor:** Recebe as referências dos elementos HTML, como o campo de entrada de texto (taskInput), botão de adicionar tarefa (addTaskBtn), e campos separados para exibir as tarefas pendentes (pendingTasks) e concluídas (completedTasks).
- **render(tasks):** Atualiza a interface com base na lista de tarefas fornecida. Também limpa os contêineres (pendingTasks e completedTasks).
 - Para cada tarefa:
 - Cria um elemento `` com o texto da tarefa.
 - Adiciona uma **checkbox** para alternar o status da tarefa entre "pending" e "completed".
 - Adiciona um botão **"Remover"** para excluir a tarefa.
 - Insere o elemento na lista correspondente (pendentes ou concluídas).
- **bindAddTask(handler):**
 - Associa o evento de clique do botão "Adicionar" à função **handler**, que recebe o valor do campo de entrada.
- **bindCompleteTask(handler) e bindRemoveTask(handler):**
 - Associam os eventos de **checkbox** e botão "Remover" às funções **handler**, permitindo que o controlador gerencie essas ações.

Em resumo, a visão é notificada pelo controlador para atualizar a interface sempre que houver mudanças nos dados. Ela também captura as ações do usuário (como adicionar ou concluir uma tarefa) e delega essas ações ao controlador.

3. Controlador (TaskController)

O controlador é responsável por lidar com a interação entre o modelo e a visão. Ele processa as ações do usuário, manipula os dados no modelo e solicita à visão que atualize a interface. No contexto MVC, utilizamos vários **handlers**, que são funções ou métodos que lidam com eventos. Esses "handlers" são responsáveis por executar uma ação específica quando algo acontece, como a interação do usuário com a interface.

No código, os métodos `handleAddTask`, `handleCompleteTask`, e `handleRemoveTask` no controlador são exemplos de handlers. Eles recebem os eventos (como cliques ou alterações) e processam as informações para realizar as mudanças necessárias no modelo ou visão.

Também temos vários exemplos de utilização de **funções de callback**. Uma função de callback é uma função que é passada como argumento para outra função, para que ela seja chamada posteriormente, geralmente em resposta a algum evento ou quando uma operação é concluída. Callbacks permitem que o código seja mais dinâmico e responsivo.

Neste trabalho utilizamos principalmente para lidar com eventos do DOM, como cliques, mudanças ou submissões de formulário. Temos abaixo as principais funções do controlador:

- **constructor:** Recebe o modelo e a visão como parâmetros e os armazena como propriedades. Também configura os "handlers" (e funções de callback) para as interações da visão, vinculando-as aos métodos do controlador. Inicializa a interface chamando **this.view.render** com a lista de tarefas obtida do modelo.
- **handleAddTask(description):** Adiciona uma nova tarefa no modelo chamando **model.addTask(description)**. Após adicionar, solicita à visão que atualize a interface com as tarefas atualizadas.
- **handleCompleteTask(index):** Atualiza o status de uma tarefa no modelo chamando **model.completeTask(index)**. Solicita à visão que atualize a interface com as tarefas atualizadas.

- **handleRemoveTask(index):** Remove uma tarefa no modelo chamando `model.removeTask(index)`. Solicita à visão que atualize a interface com as tarefas atualizadas.

O controlador é como o intermediário entre a visão e o modelo:

1. Captura ações do usuário através dos "handlers" definidos.
2. Manipula os dados no modelo de acordo com a ação.
3. Atualiza a interface da visão para refletir as mudanças.

Lógica do sistema e interação entre Modelo, Controlador e Visão

Fluxo de uma tarefa adicionada:

1. O usuário digita uma descrição e clica no botão "Adicionar".
2. A visão (TaskView) captura o evento e executa o handler configurado no `bindAddTask`, que chama `handleAddTask` no controlador.
3. O controlador adiciona a tarefa ao modelo (`TaskModel.addTask`) e solicita à visão que atualize a interface com os dados atualizados (`TaskView.render`).

Fluxo de conclusão de uma tarefa:

1. O usuário marca a checkbox de uma tarefa pendente.
2. A visão executa o handler configurado no `bindCompleteTask`, chamando `handleCompleteTask` no controlador.
3. O controlador atualiza o status da tarefa no modelo (`TaskModel.completeTask`) e solicita à visão que atualize a interface.

Fluxo de remoção de uma tarefa:

1. O usuário clica no botão "Remover" ao lado de uma tarefa.
2. A visão executa o handler configurado no `bindRemoveTask`, chamando `handleRemoveTask` no controlador.
3. O controlador remove a tarefa do modelo (`TaskModel.removeTask`) e solicita à visão que atualize a interface.