Relatório Técnico: Comparação de Arquiteturas Frontend

Disciplina: Engenharia de Software
Alunos: Ryan Leite, Conrado Einstein, João Marcelo Pimenta

Projeto: ToDo App Comparativo (MVC, MVP, MVVM + REST/Reativo)

1. Análise da Lógica nas Arquiteturas

Com base na implementação realizada em React, observou-se uma clara distinção na localização das regras de negócio e manipulação de estado:

- **MVC (Model-View-Controller):** A lógica concentrou-se na classe `TaskController.js`. O componente React (`TodoMVC.jsx`) atuou apenas como capturador de eventos, repassando-os imediatamente para o Controller. Foi necessário injetar a função `setTasks` no Controller para que ele pudesse, imperativamente, ordenar a atualização da View.

- **MVP (Model-View-Presenter):** A lógica residiu na classe `TaskPresenter.js`. A diferença fundamental para o MVC foi o desacoplamento: a View (`TodoMVP.jsx`) implementou uma interface de contrato (`viewInterface`), tornando-se totalmente passiva. O Presenter não manipula o estado do React diretamente, mas invoca métodos abstratos como `showTasks()`.

- **MVVM (Model-View-ViewModel):** A lógica ficou misturada ao ciclo de vida do componente em `TodoMVVM.jsx`. O próprio *State* do React (`useState`) atuou como ViewModel. O *Data Binding* foi natural: ao alterar a variável de estado `tasks`, a interface reagiu automaticamente, sem necessidade de uma classe intermediária orquestrando a atualização.

2. Integração com Backend Reativo

A arquitetura **MVVM** mostrou-se a mais simples de integrar com o padrão Reativo (Server-Sent Events).

- **Motivo:** O hook `useEffect` do React funciona nativamente como um observador. Ao conectar o `EventSource` dentro do componente, qualquer mensagem recebida atualizava o estado (`setTasks`), e o React cuidava da renderização.

- **Dificuldade no MVC/MVP:** Nestes padrões, foi necessário gerenciar o ciclo de vida da conexão (`EventSource`) dentro das classes Controller e Presenter. Isso gerou complexidade extra para garantir que a conexão fosse fechada (`dispose`) quando o componente React fosse desmontado, algo que o `useEffect` resolve de forma mais elegante no MVVM.

3. Impacto da Troca REST para Reativo no Frontend

A mudança de REST para Reativo alterou fundamentalmente o fluxo de dados (Data Flow):

- **No modo REST:** O fluxo é **Pull-based** (o cliente pede). A atualização da lista dependia de uma ação explícita (carregamento inicial ou clique no botão "Atualizar"). No código, isso se traduziu em chamadas `axios.get()` pontuais.

- **No modo Reativo:** O fluxo tornou-se **Push-based** (o servidor envia). O frontend deixou de ser um solicitante ativo para se tornar um "escutador" passivo.

- **Implementação:** No código, a lógica de `fetchTasks()` (busca única) foi substituída (ou complementada) pela instância de um `EventSource`. A lógica de *polling* (ficar perguntando ao servidor) foi eliminada, reduzindo tráfego de rede desnecessário.

4. Escolha Arquitetural para Sistemas Maiores

Para um sistema desenvolvido em **React**, a arquitetura escolhida seria o **MVVM**.

Justificativa:

Embora o MVP e MVC ofereçam excelente separação de responsabilidades (testáveis isoladamente), eles entram em conflito com a natureza declarativa do React.

Durante o desenvolvimento do TodoMVC e TodoMVP, foi necessário escrever código "artificial" (como passar setTasks para classes ou criar interfaces de View) apenas para satisfazer o padrão, lutando contra a ferramenta. O MVVM aproveita a arquitetura interna do React, resultando em menos código (boilerplate), leitura mais fluida e manutenção simplificada, pois alinha a gestão de estado da biblioteca com a lógica de negócio.