Curso: Ciência da Computação

Disciplina: Engenharia de Software e Arquitetura de Sistemas

Professora: Lucy Mari

Integrantes:

Leonardo Santos da Silva – 24026495

Lucas de Lima Gutierrez – 2402601

Lucas Silva Maciel – 24025942

Thiago Akira Higa Mitami – 24026254

Entrega 2

Aplicação de Design de Software

O design de software foi aplicado de forma planejada para garantir organização, reuso e manutenção fácil do código. Desde o início do projeto, buscamos dividir o sistema em camadas bem definidas, isolando responsabilidades e evitando dependências desnecessárias entre os módulos.

A aplicação é dividida em três grandes partes:

- Frontend (React): Responsável pela interface gráfica do sistema, exibe os dados do dashboard e envia requisições HTTP para o backend. A arquitetura de componentes foi pensada para permitir reutilização e manutenção independente.

- Backend (Python / Flask): Atua como intermediário entre o frontend e o banco de dados. Ele processa as requisições vindas do React, realiza cálculos (como o ticket médio) e retorna os dados prontos em formato JSON.

O backend foi organizado em três arquivos principais:

- **models.py** → define os modelos de dados;
- **services.py** → contém as regras de negócio;
- **routes.py** → define as rotas da API REST.

Essa separação garante coesão e isolamento de responsabilidades.

Banco de Dados (SQLite): Responsável pela persistência dos dados. A comunicação com o banco foi encapsulada dentro de funções específicas, evitando que o frontend ou as rotas acessem os dados diretamente.

O design aplicado também se baseia nos princípios de design orientado a objetos, como encapsulamento, abstração e baixo acoplamento, garantindo que cada módulo tenha um papel bem definido no sistema.

Em resumo, o design do dashboard foi pensado para ser modular, extensível e de fácil manutenção, permitindo que novos recursos (como novos relatórios ou métricas) possam ser adicionados sem comprometer as partes já implementadas.

Conjunto SOLID

Os princípios SOLID foram aplicados para manter a qualidade do código, facilitar futuras modificações e evitar a criação de dependências rígidas entre os módulos do sistema.

S - Single Responsibility Principle (Princípio da Responsabilidade Única): Cada classe ou módulo deve ter apenas uma responsabilidade, ou seja, uma razão para mudar. No projeto, esse princípio foi aplicado separando a lógica de negócio, o controle de rotas e o modelo de dados em arquivos diferentes:

- **models.py** → define a estrutura dos dados (ex.: classe Transacao, Cupom, Usuario);
- **services.py** → implementa cálculos e funções de regra de negócio (ex.: função calcular_ticket_medio());
- **routes.py** → lida apenas com as rotas HTTP e respostas da API.

Essa separação torna o código mais organizado e facilita testes, manutenção e futuras expansões. Por exemplo, se for necessário mudar a forma como o ticket médio é calculado, basta editar services.py, sem interferir nas rotas ou na estrutura do banco.

O - Open/Closed Principle (Aberto para Extensão, Fechado para Modificação): O código deve estar aberto para ser estendido, mas fechado para ser modificado.

No projeto, esse princípio é visto na estrutura das rotas da API. Novos endpoints podem ser adicionados no arquivo routes.py sem alterar as rotas existentes. Cada rota é independente, permitindo extensões sem alterar o comportamento atual do sistema. Isso garante estabilidade e reduz o risco de bugs quando novas funcionalidades são adicionadas.

L - Liskov Substitution Principle (Princípio da Substituição de Liskov): Objetos de uma classe derivada devem poder substituir objetos da classe base sem quebrar o código. Embora o Dashboard use uma estrutura mais simples, esse princípio se aplica em funções genéricas que tratam dados de diferentes tipos de entidades (usuários, cupons, transações).

Por exemplo, funções que realizam operações de leitura no banco de dados podem receber diferentes classes de modelo sem precisar alterar a lógica interna. Se futuramente for criada uma classe CupomPremium herdando de Cupom, ela deve funcionar nas mesmas rotinas de persistência, sem precisar alterar o restante do código.

I - Interface Segregation Principle (Princípio da Segregação de Interfaces): Uma classe não deve ser forçada a depender de métodos que não utiliza.

No caso do projeto, isso foi aplicado indiretamente no frontend e backend. O frontend consome apenas as rotas necessárias para exibir os dados do dashboard, não precisa conhecer toda a estrutura do backend. Da mesma forma, o backend não expõe informações desnecessárias, retornando apenas os dados úteis em cada endpoint. Isso reduz o acoplamento entre as partes do sistema e evita dependências desnecessárias.

D - Dependency Inversion Principle (Princípio da Inversão de Dependência): Módulos de alto nível não devem depender de módulos de baixo nível; ambos devem depender de abstrações. No projeto, a camada de rotas (routes.py) não depende diretamente do banco de dados, ela depende das funções do serviço (services.py), que atuam como abstração da lógica de dados.

Modelo de design

No desenvolvimento do PicMoney Dashboard, foram adotados dois modelos complementares de design: a Arquitetura em Camadas, responsável pela

organização estrutural do sistema, e o padrão MVC (Model–View–Controller), aplicado dentro do backend para organizar o fluxo interno de dados e responsabilidades.

Arquitetura em Camadas

A Arquitetura em Camadas define a forma como o sistema é estruturado e dividido em partes. Cada camada possui uma responsabilidade específica e se comunica apenas com a camada imediatamente abaixo, garantindo baixo acoplamento e alta coesão. No nosso projeto, a aplicação foi estruturada em três camadas principais:

Camada de Apresentação (Frontend): Desenvolvida em React, é responsável pela interação com o usuário e pela exibição dos dados do dashboard. Essa camada envia requisições HTTP ao backend e renderiza as informações recebidas (como ticket médio, número de transações e relatórios).

Camada de Negócio (Backend): Desenvolvida em Python com Flask, contém as regras de negócio do sistema, como cálculos, validações e tratamento de dados. Essa camada processa as requisições do frontend e retorna respostas em formato JSON. Exemplo: a função calcular_ticket_medio() em services.py implementa a lógica de cálculo com base nas transações armazenadas.

Camada de Dados (Banco de Dados): Implementada com SQLite, responsável pela persistência das informações. O acesso aos dados é encapsulado por funções específicas, evitando que outras camadas acessem o banco diretamente. Exemplo: consultas SQL feitas apenas dentro dos métodos de models.py.

Padrão MVC (Model–View–Controller)

O MVC é um padrão de design utilizado dentro da camada de aplicação (no backend e frontend) para organizar o código e controlar o fluxo de informações entre a interface e a lógica do sistema.

Model: Representa a camada de dados e persistência. No projeto, isso corresponde ao SQLite, responsável pelo armazenamento e recuperação dos dados das transações, cupons e usuários. O model é acessado através das funções definidas no backend para consultar e manipular as informações do banco de dados.

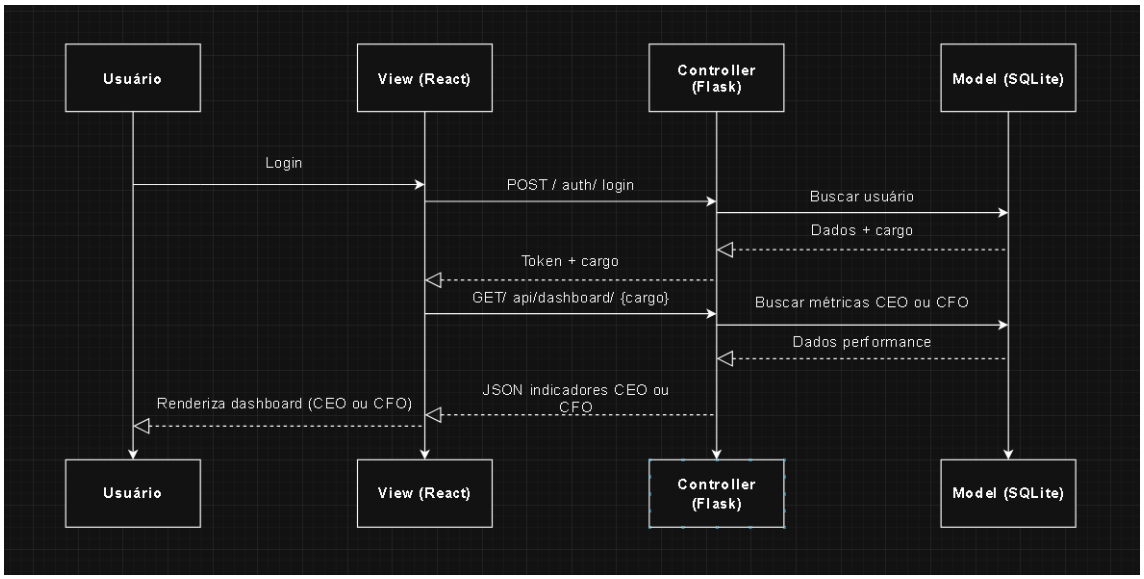
View: Responsável pela exibição das informações ao usuário. A View no dashboard é o frontend em React, que consome os dados da API Flask. Diretório: src/Entregas/frontend

Controller: Faz a ponte entre as requisições do frontend e a lógica do backend. Ele recebe a solicitação HTTP, chama a função adequada do serviço e retorna o resultado. Arquivo: routes.py. Exemplo: endpoint /ticket_medio, que chama a função calcular_ticket_medio() e devolve o resultado para o React.

Conclusão

O Dashboard produzido demonstra a aplicação prática de princípios sólidos de engenharia de software. Através do design modular, do uso dos princípios SOLID, da arquitetura em camadas, e o padrão MVC, o sistema alcança baixo acoplamento, alta coesão e facilidade de expansão. Essas práticas tornam o projeto mais profissional, escalável e pronto para evoluir, seja com novos relatórios, novos usuários ou a integração com bancos de dados mais robustos. O resultado é um sistema limpo, sustentável e alinhado com as boas práticas de desenvolvimento moderno.

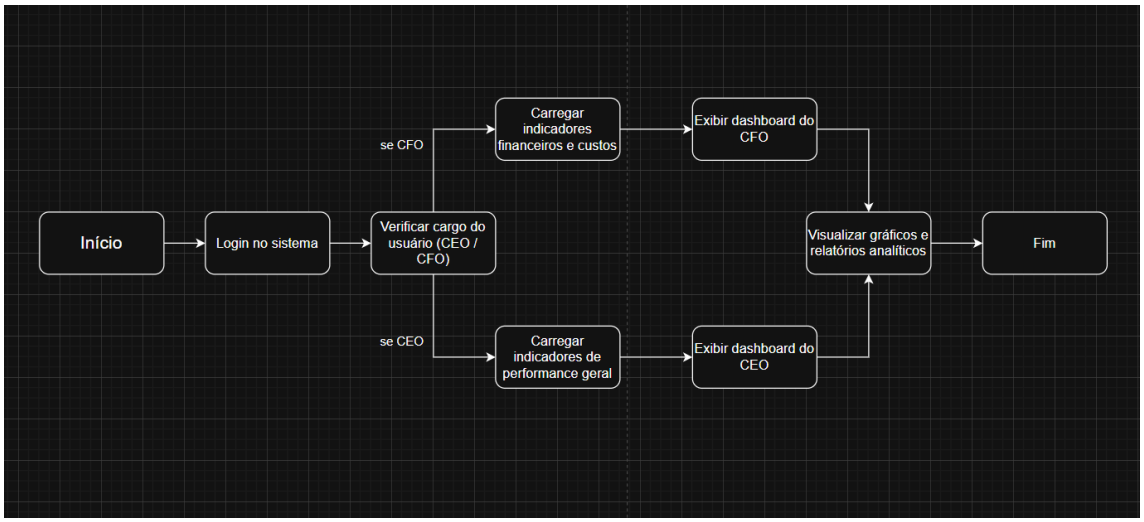
Diagrama de Sequência baseada no modelo MVC



Este diagrama ilustra o fluxo de interação entre os componentes do sistema PicMoney durante o acesso ao dashboard. O processo inicia com o login do usuário.

Após a autenticação, o sistema identifica o cargo do usuário (CEO / CFO) e carrega os indicadores específicos de performance geral. O diagrama demonstra a arquitetura MVC em ação, mostrando a comunicação clara entre View (React), Controller (Flask) e Model (SQLite) para entregar a experiência personalizada conforme o perfil do usuário.

Diagrama de Atividade



O Diagrama de Atividade representa o fluxo de ações executadas dentro do sistema NextBoard, o dashboard analítico desenvolvido para a empresa PicMoney. Esse diagrama tem como objetivo demonstrar graficamente o funcionamento lógico do sistema, desde o momento em que o administrador acessa a plataforma até a visualização dos indicadores e relatórios.

No início do fluxo, o administrador realiza o login no sistema. Em seguida, o backend realiza a verificação do cargo do usuário, identificando se o acesso é feito pelo CEO ou pelo CFO. Essa etapa define o caminho lógico do diagrama:

-Caso o usuário seja o CEO, o sistema carrega e exibe indicadores de performance geral, como crescimento de usuários, volume de transações e metas atingidas.

-Caso o usuário seja o CFO, são carregados indicadores financeiros, incluindo receitas, despesas, fluxo de caixa e rentabilidade.

Após o carregamento das informações, ambos os perfis seguem para a etapa de visualização, onde o dashboard exibe gráficos, relatórios e métricas consolidadas.

O fluxo termina com o usuário analisando os dados apresentados, o que caracteriza o objetivo final do sistema: fornecer uma ferramenta de apoio à decisão, permitindo que os administradores da PicMoney visualizem informações estratégicas de maneira clara e intuitiva.

Conclusão:

A aplicação dos diagramas UML no projeto PicMoney demonstrou ser fundamental para o planejamento e validação da arquitetura do sistema. Através do diagrama de sequência, foi possível modelar com precisão as interações entre os componentes MVC, enquanto o diagrama de atividade permitiu visualizar os fluxos de negócio para diferentes perfis de usuário (CEO e CFO).

Resultados Obtidos

-Especificação de Requisitos: Identificamos a necessidade de endpoints diferenciados (/api/ceo e /api/cfo) para atender aos diferentes indicadores requeridos por cada cargo.

-Otimização do Fluxo: O mapeamento visual permitiu identificar oportunidades de melhoria no processo de autenticação e carregamento de dados.

-Documentação Eficaz: Os diagramas servirão como guia para a equipe de desenvolvimento durante a implementação e futuras manutenções do sistema.