# Aplicação de Design de Software e Diagramas UML

## Dashboard PicMoney - Análise C-Level

**Documento Técnico de Arquitetura e Design**

Engenharia de Software e Arquitetura de Sistemas

2025

# SUMÁRIO

# 1. INTRODUÇÃO

O Dashboard PicMoney é uma aplicação web *full-stack* projetada para revolucionar a forma como a liderança C-Level da startup visualiza e interage com dados críticos de negócios. Desenvolvido para fornecer análises estratégicas, operacionais e financeiras em tempo real (simulado), o sistema capacita CEOs, CFOs e CTOs a monitorar Indicadores-Chave de Performance (KPIs), identificar tendências de mercado e tomar decisões ágeis e fundamentadas.

## 1.1 Contexto do Projeto

A PicMoney opera no dinâmico mercado de cupons digitais, utilizando Realidade Aumentada e Georreferenciamento para conectar parceiros comerciais a consumidores de forma inovadora. O desafio deste projeto foi criar um dashboard capaz de processar, integrar e visualizar o grande volume de dados gerados pelas transações, cadastros de usuários e interações geolocalizadas.

Este documento detalha as decisões de design de software, a arquitetura de sistema e a aplicação de diagramas UML que estruturam a solução. O objetivo não é apenas apresentar o produto final, mas justificar as escolhas técnicas que garantem que o sistema seja robusto, escalável, performático e de fácil manutenção, alinhado aos objetivos estratégicos da PicMoney.

# 2. ARQUITETURA DE SOFTWARE

Para atender aos requisitos de responsividade, personalização por perfil (CEO, CFO, CTO) e atualização em tempo real, optamos por uma arquitetura de microsserviços desacoplada, composta por um frontend moderno e um backend leve e eficiente.

## 2.1 Visão Geral

A solução é dividida em três camadas principais:

- **Frontend (Cliente):** Uma *Single Page Application* (SPA) desenvolvida em **React.js**. Esta escolha se justifica pela sua alta performance de renderização através do DOM virtual, ecossistema robusto e excelente gerenciamento de estado, crucial para um dashboard interativo. A interface consome dados de uma API e atualiza os componentes de visualização (gráficos, KPIs) dinamicamente.
- **Backend (Servidor):** Uma API RESTful desenvolvida em **Node.js** com o framework **Express.js**. O Node.js foi escolhido por sua natureza assíncrona e *non-blocking I/O*, ideal para lidar com múltiplas requisições de dados simultâneas e simulação de dados em tempo real sem comprometer a performance.
- **Comunicação e Dados:** A comunicação entre frontend e backend ocorre exclusivamente via chamadas **RESTful** utilizando o formato **JSON**, garantindo um baixo acoplamento entre as camadas. Isso permite que o frontend e o backend evoluam de forma independente. O **SQLite** foi selecionado como banco de dados devido à sua simplicidade de configuração, portabilidade e eficiência para os volumes de dados simulados neste projeto.

# 3. PADRÕES DE DESIGN APLICADOS

A aplicação de Padrões de Design (Design Patterns) foi fundamental para resolver problemas comuns de desenvolvimento, promover o reuso de código e garantir uma arquitetura flexível.

- **Singleton (Backend):** Este padrão foi aplicado na gerenciamento da conexão com o banco de dados (SQLite). Garantimos que apenas uma instância da conexão com o banco seja criada e compartilhada por toda a aplicação backend. Isso economiza recursos (evitando abrir e fechar conexões desnecessariamente) e previne problemas de concorrência no acesso ao arquivo do banco de dados.
- **Factory (Backend):** Utilizamos o conceito de Factory para a geração de dados simulados. Uma "fábrica" é responsável por criar diferentes tipos de *mocks* (transações, usuários, pedestres) com base em parâmetros definidos. Isso abstrai a lógica de criação de dados e facilita a adição de novos cenários de simulação no futuro, sem impactar o código existente.
- **Observer (Frontend):** Essencial para a reatividade do dashboard. O React, especialmente através da **Context API** e dos *hooks* (useState, useEffect), implementa nativamente variações deste padrão. Os componentes de gráfico (ex: LineChart, BarChart) atuam como "Observadores" que "assinam" um estado global (o "Sujeito", como os filtros de data ou perfil). Quando o usuário altera um filtro, o estado é atualizado, e todos os componentes observadores são notificados automaticamente para re-renderizar com os novos dados.
- **Facade (Frontend):** Implementamos o padrão Facade através de um módulo de serviço centralizado (ex: services/api.js). Este módulo atua como uma fachada, simplificando a interface de comunicação com o backend. Os componentes React não precisam saber detalhes complexos sobre fetch, headers HTTP, tratamento de URLs ou formatação de body. Eles apenas chamam funções simples e declarativas (ex: api.getFinancialData(filters)), tornando o código dos componentes mais limpo e facilitando a manutenção das chamadas de API em um único lugar.

# 4. PRINCÍPIOS DE DESIGN APLICADOS (SOLID)

Os princípios SOLID nortearam o design da aplicação, resultando em um código mais limpo, modular e fácil de manter.

- S - Single Responsibility Principle (Princípio da Responsabilidade Única):
  No frontend, cada componente React possui uma única responsabilidade. Por exemplo, o componente KPICard é responsável apenas por exibir um indicador e sua variação. O FilterPanel é responsável apenas por gerenciar as opções de filtro. No backend, cada rota (endpoint) da API lida com uma única ação (ex: buscar dados financeiros, buscar dados operacionais).
- O - Open/Closed Principle (Princípio Aberto/Fechado):
  O sistema foi desenhado para ser "aberto para extensão, mas fechado para modificação". Por exemplo, para adicionar um novo gráfico ao dashboard, basta criar um novo componente de gráfico (extensão) e adicioná-lo ao layout, sem a necessidade de alterar os componentes de filtro ou de busca de dados existentes.
- L - Liskov Substitution Principle (Princípio da Substituição de Liskov):
  Embora a herança clássica seja menos comum em React, o princípio é aplicado na composição de componentes. Componentes "base" (como um ChartContainer genérico) podem ser compostos por gráficos específicos (LineChart, PieChart), e a aplicação continua funcionando de forma coesa, pois ambos compartilham a interface de props esperada.
- I - Interface Segregation Principle (Princípio da Segregação de Interfaces):
  Evitamos "props gigantes". Cada componente recebe apenas as propriedades (props) de que realmente necessita. Por exemplo, um componente de gráfico que exibe apenas dados de usuários não recebe props relacionadas a finanças, mesmo que o componente "pai" tenha acesso a ambas.
- D - Dependency Inversion Principle (Princípio da Inversão de Dependência):
  Os componentes de alto nível (como as páginas do dashboard) não dependem de implementações concretas de busca de dados. Eles dependem de abstrações (a "fachada" api.js ou hooks customizados, ex: useData). Isso inverte a dependência: os detalhes de implementação (fetch, axios) dependem da abstração (a interface do serviço), permitindo trocar a biblioteca de HTTP ou a fonte de dados (de API para mock local) sem alterar os componentes da UI.

# 5. ESTRUTURA de DIRETÓRIOS

A organização dos arquivos reflete a arquitetura modular e a separação de responsabilidades (SoC).

## 5.1 Estrutura do Frontend (React)

| Diretório | Propósito |
|---|---|
| /public | Contém assets estáticos (ícones, index.html). |
| /src | Contém todo o código-fonte da aplicação React. |
| /src/components | Componentes React reutilizáveis (ex: KPICard, ChartCard, Header). |
| /src/pages | Componentes que representam as visualizações principais (ex: DashboardCEO, DashboardCFO). |
| /src/context | Gerenciamento de estado global (ex: ProfileContext, FilterContext). |
| /src/hooks | Hooks customizados (ex: useApiData) para encapsular lógica. |
| /src/services | Módulos de fachada para serviços externos (ex: api.js). |
| /src/utils | Funções utilitárias puras (ex: formatDate, calculatePercentage). |

## 5.2 Estrutura do Backend (Node/Express)

| Diretório | Propósito |
|---|---|
| / | Arquivo principal (server.js) e configuração (package.json). |
| /routes | Define os *endpoints* da API (ex: financial.js, operational.js). |
| /controllers | Contém a lógica de negócios e interação com o banco de dados. |
| /database | Arquivo de configuração e inicialização do SQLite. |
| /simulation | Scripts e lógicas para geração de dados simulados. |

# 6. FLUXO DE DADOS

O fluxo de dados é o coração da interatividade do dashboard e segue o padrão *Observer* de forma clara.

1. **Ação do Usuário:** O usuário interage com a UI, por exemplo, selecionando um novo perfil (CEO) ou alterando um filtro de data no componente FilterPanel.
2. **Atualização de Estado:** Essa ação dispara uma função que atualiza o estado global, gerenciado pela Context API do React (o "Sujeito").
3. **Notificação e Re-renderização:** O *hook* useContext nos componentes "Observadores" (como DashboardCEO) detecta a mudança no estado.
4. **Busca de Dados (Facade):** O componente DashboardCEO, dentro de um useEffect, chama a função apropriada do serviço api.js (ex: api.getCEOKpis(newFilters)), aplicando o padrão Facade.
5. **Requisição Backend:** A fachada api.js constrói e envia uma requisição HTTP (ex: GET /api/ceo/kpis?date=...) para o backend Node.js.
6. **Processamento Backend:** O Express.js direciona a requisição para o *controller* correspondente. O *controller* consulta o banco de dados (usando a conexão Singleton), processa os dados e retorna uma resposta JSON.
7. **Resposta Frontend:** A api.js recebe a resposta JSON e a retorna para o componente DashboardCEO.
8. **Atualização da UI:** O componente atualiza seu estado local com os novos dados recebidos.
9. **Renderização Final:** O React propaga os novos dados (via *props*) para os componentes filhos (ex: KPICard, LineChart), que se re-renderizam para exibir as informações atualizadas ao usuário.

# 7. TRATAMENTO DE ERROS E RESILIÊNCIA

Um sistema C-Level não pode falhar silenciosamente. A estratégia de tratamento de erros é robusta em ambas as camadas:

- **Frontend:** Utilizamos blocos try...catch em todas as chamadas assíncronas dentro do serviço api.js. Se uma requisição falhar (ex: erro de rede ou resposta 500 do servidor), o erro é capturado e propagado para o componente, que pode então exibir uma mensagem amigável ao usuário (ex: "Não foi possível carregar os dados") em vez de quebrar a aplicação.
- **Backend:** Os *controllers* da API utilizam try...catch para encapsular a lógica de banco de dados. Qualquer erro de consulta ou processamento é capturado e um código de status HTTP apropriado é retornado (ex: 400 Bad Request para filtros inválidos, 404 Not Found para dados não encontrados, 500 Internal Server Error para falhas internas), sempre acompanhado de uma mensagem JSON descritiva para facilitar o *debug*.

# 8. SEGURANÇA E BOAS PRÁTICAS

Embora seja um ambiente de dados simulados, aplicamos boas práticas de segurança pensando em uma futura produção:

- **CORS (Cross-Origin Resource Sharing):** O backend Express.js utiliza o middleware cors para permitir requisições apenas do domínio do frontend, prevenindo que outros sites consumam a API indevidamente.
- **Helmet.js:** Utilizamos o middleware helmet no backend para adicionar cabeçalhos HTTP de segurança essenciais (como X-Content-Type-Options, Strict-Transport-Security), protegendo a aplicação contra ataques comuns como *clickjacking* e *XSS*.
- **Validação de Entradas:** O backend realiza a validação e sanitização de todos os parâmetros recebidos via req.query ou req.body antes de usá-los em consultas ao banco de dados, mitigando riscos de *SQL Injection*.

# 9. PERFORMANCE E OTIMIZAÇÃO

Para garantir o requisito não funcional de tempo de resposta rápido (< 2s), implementamos diversas otimizações:

- **Memoization (Frontend):** Usamos React.memo em componentes pesados (gráficos) para evitar que eles se re-renderizem se suas *props* não mudarem. O *hook* useMemo é usado para memorizar cálculos complexos ou transformações de dados que não precisam ser refeitos a cada renderização.
- **Code Splitting (Frontend):* Para um *load* inicial mais rápido, a aplicação pode ser configurada com React.lazy() e Suspense. Isso permite carregar o código de cada perfil de dashboard (CEO, CFO, CTO) apenas quando o usuário navegar para aquela seção (*lazy loading*).
- **Cache (Backend):** Para dados que mudam com pouca frequência (ex: dados históricos ou demográficos), implementamos uma estratégia simples de cache em memória no backend. Isso reduz drasticamente a carga no banco de dados e torna as respostas da API quase instantâneas.
- **Compressão:** O backend utiliza o middleware compression para comprimir as respostas JSON (gzip), reduzindo o tamanho dos dados trafegados pela rede e acelerando o *download* pelo frontend.

# 10. EXTENSIBILIDADE E MANUTENIBILIDADE

O design da aplicação foi pensado para o futuro. A arquitetura modular, combinada com os padrões (Factory, Facade) e princípios (SRP, OCP), garante que o sistema possa crescer de forma sustentável.

Adicionar um novo KPI, um novo filtro ou até mesmo um painel completo para um novo perfil (ex: CMO - Chief Marketing Officer) é um processo de adição de novos módulos, e não de alteração complexa dos módulos existentes. A documentação clara do código e o uso de diagramas UML (descritos a seguir) complementam essa estratégia, facilitando o *onboarding* de novos desenvolvedores e a manutenção evolutiva do sistema.

# 11. CONCLUSÃO GERAL

A arquitetura e o design de software aplicados ao projeto PicMoney não são apenas escolhas técnicas, mas facilitadores de negócios. Através da separação clara de responsabilidades, do uso de padrões de design comprovados e da adesão aos princípios SOLID, criamos um Dashboard Interativo que é, ao mesmo tempo, poderoso para o usuário final (C-Level) e eficiente para a equipe de desenvolvimento. O resultado é uma aplicação performática, resiliente, segura e pronta para evoluir junto com o crescimento da PicMoney.

# 12. DIAGRAMAS UML

Para documentar visualmente a arquitetura e o comportamento do sistema, foram criados dois dos principais diagramas da UML (Unified Modeling Language), conforme solicitado.

## 12.1. Diagrama de Classes

O Diagrama de Classes modela a estrutura estática do sistema, focando nos principais componentes do frontend e suas relações. Ele é essencial para entender as responsabilidades e as propriedades de cada parte da UI.

**Descrição dos Principais Componentes (Classes):**

| Classe (Componente) | Responsabilidade (Atributos/Métodos) | Relação |
|---|---|---|
| App | Orquestrador principal. Gerencia rotas e o estado global de perfil (CEO/CFO/CTO). | Agrega Dashboard, Header, FilterPanel. |
| Dashboard | "Pai" da visualização. Recebe dados filtrados e os distribui para os gráficos filhos. | Agrega KPICard, ChartCard. |
| FilterPanel | Gerencia e exibe os filtros (data, região). Modifica o estado global de filtros. | É usado por App. |
| api (Facade) | Abstrai a comunicação com o Backend. Possui métodos (ex: getKpis(), getChartData()). | É usado por Dashboard (e outros). |
| KPICard | Exibe um único indicador (ex: Receita, Custo). Recebe title, value, trend como props. | É usado por Dashboard. |
| ChartCard | Contêiner genérico para gráficos. Encapsula um gráfico (ex: LineChart). | Agrega LineChart / BarChart. |

## 12.2. Diagrama de Sequência

O Diagrama de Sequência modela o comportamento dinâmico do sistema, ilustrando a ordem das interações entre os componentes ao longo do tempo. Escolhemos detalhar o fluxo mais crítico: a atualização de um filtro pelo usuário.

**Descrição da Sequência (Atualização de Filtro):**

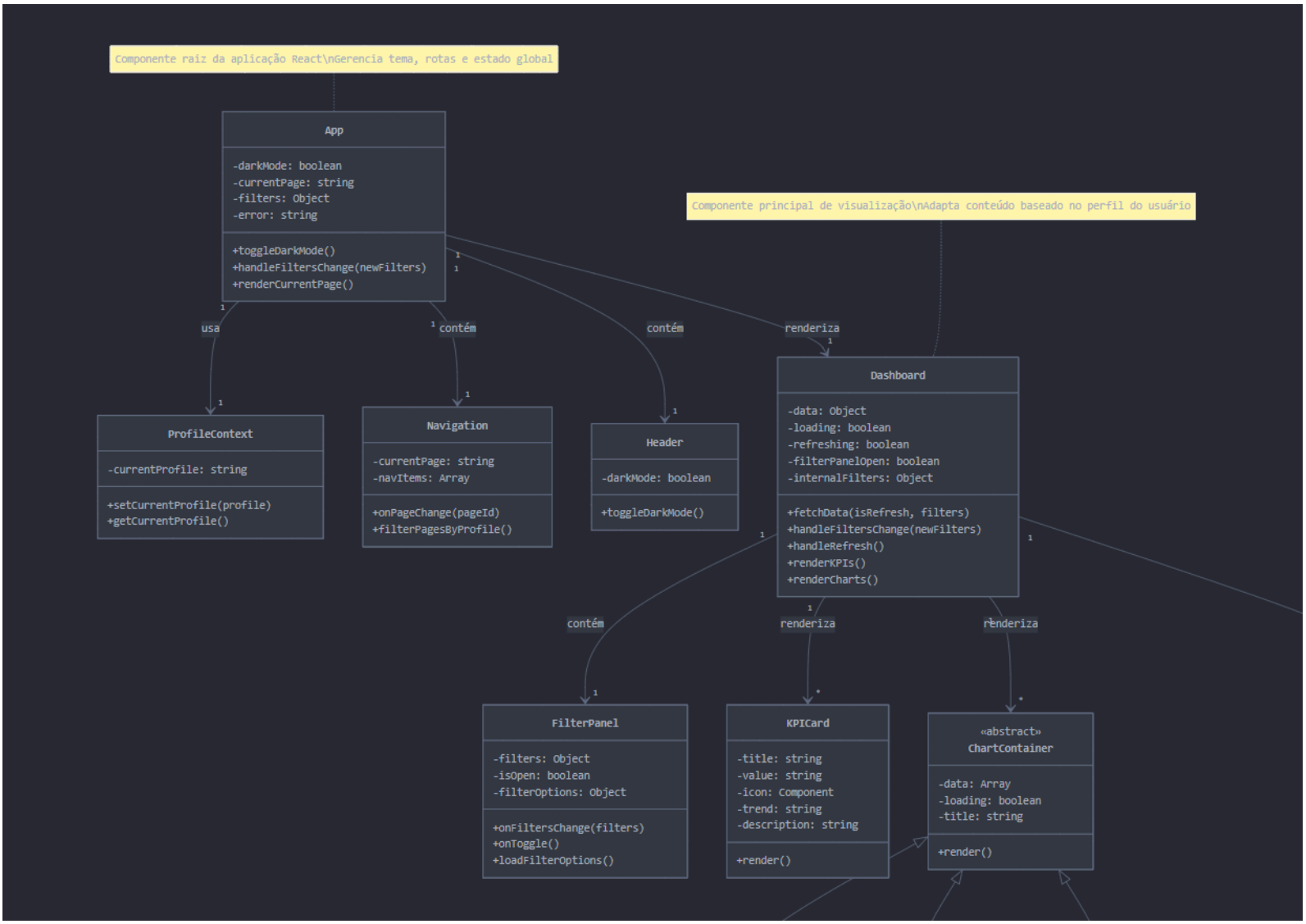
| Ordem | Objeto (Componente) | Ação / Mensagem |
|---|---|---|
| 1 | Usuário | selectFilter("Últimos 30 dias") |
| 2 | FilterPanel | updateGlobalState(newState) (Chama o Context) |
| 3 | Dashboard | (Detecta mudança no estado) useEffect() |
| 4 | Dashboard | fetchData(newState) (Chama a fachada) |
| 5 | api (Facade) | GET /api/data?filter=... (Envia req. p/ Backend) |
| 6 | Backend (Controller) | processRequest() |
| 7 | Backend (Database) | query(sql) |
| 8 | Backend (Database) | [data] (Retorna dados) |
| 9 | Backend (Controller) | json(data) (Retorna resposta) |
| 10 | api (Facade) | jsonData (Recebe dados) |
| 11 | Dashboard | setData(jsonData) (Atualiza estado local) |
| 12 | Dashboard | (props) (Envia novas props para filhos) |
| 13 | ChartCard / KPICard | re-render() (Atualiza a UI) |

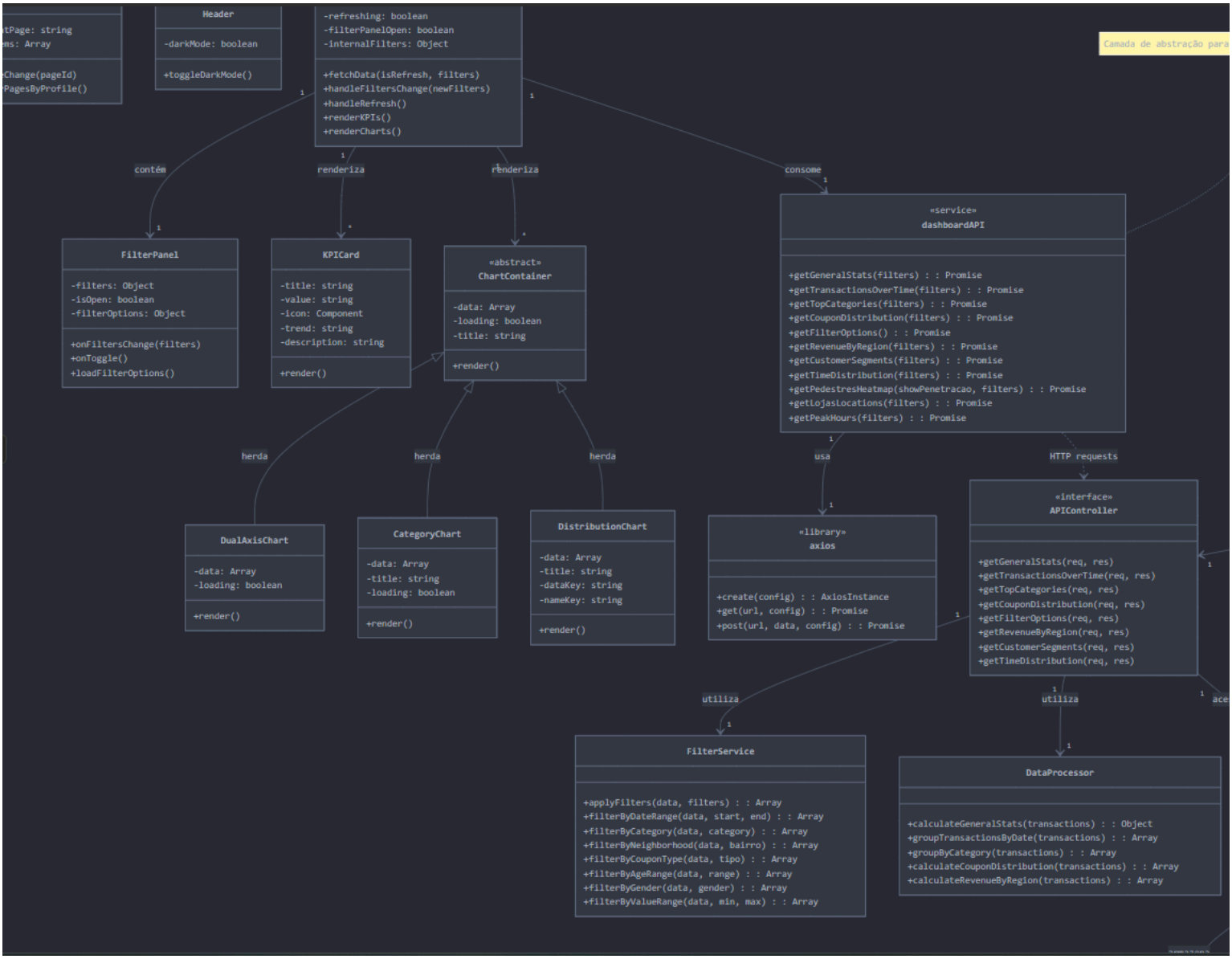
# 13. CONCLUSÃO DOS DIAGRAMAS UML

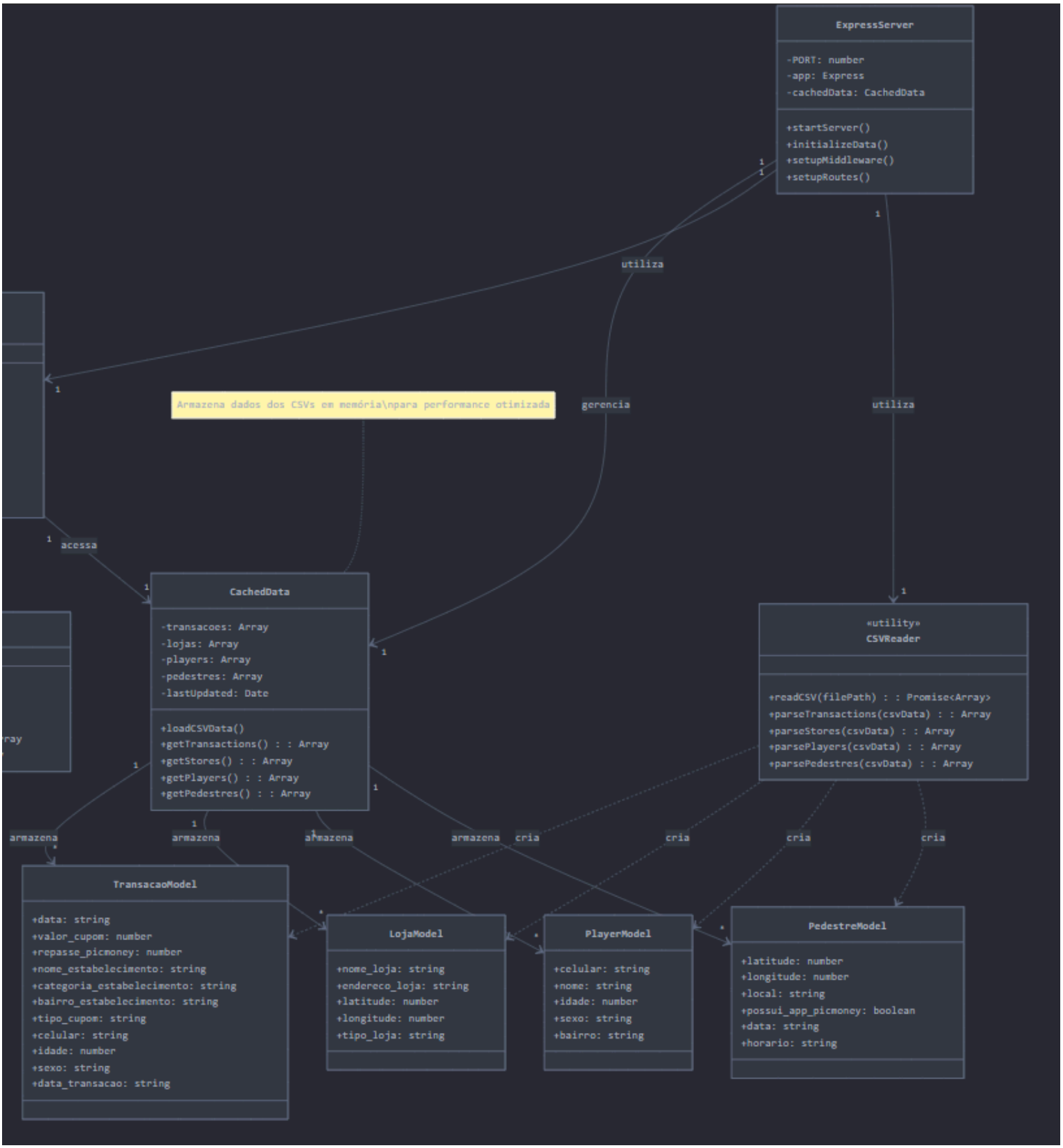
Os diagramas UML apresentados (Classes e Sequência) são ferramentas vitais para o sucesso do projeto. O Diagrama de Classes fornece um "mapa" claro da arquitetura do frontend, permitindo que a equipe entenda rapidamente quais componentes existem, quais suas responsabilidades e como eles se conectam.

O Diagrama de Sequência, por sua vez, é crucial para validar o fluxo de dados mais importante da aplicação. Ele transforma a lógica de atualização do dashboard, que pode ser complexa, em uma série de etapas visuais e fáceis de seguir, garantindo que a comunicação entre frontend, backend e banco de dados ocorra na ordem correta e como esperado. Juntos, esses diagramas servem como uma documentação viva, facilitando a comunicação da equipe, o *onboarding* de novos membros e a identificação de gargalos ou pontos de melhoria no design do sistema.

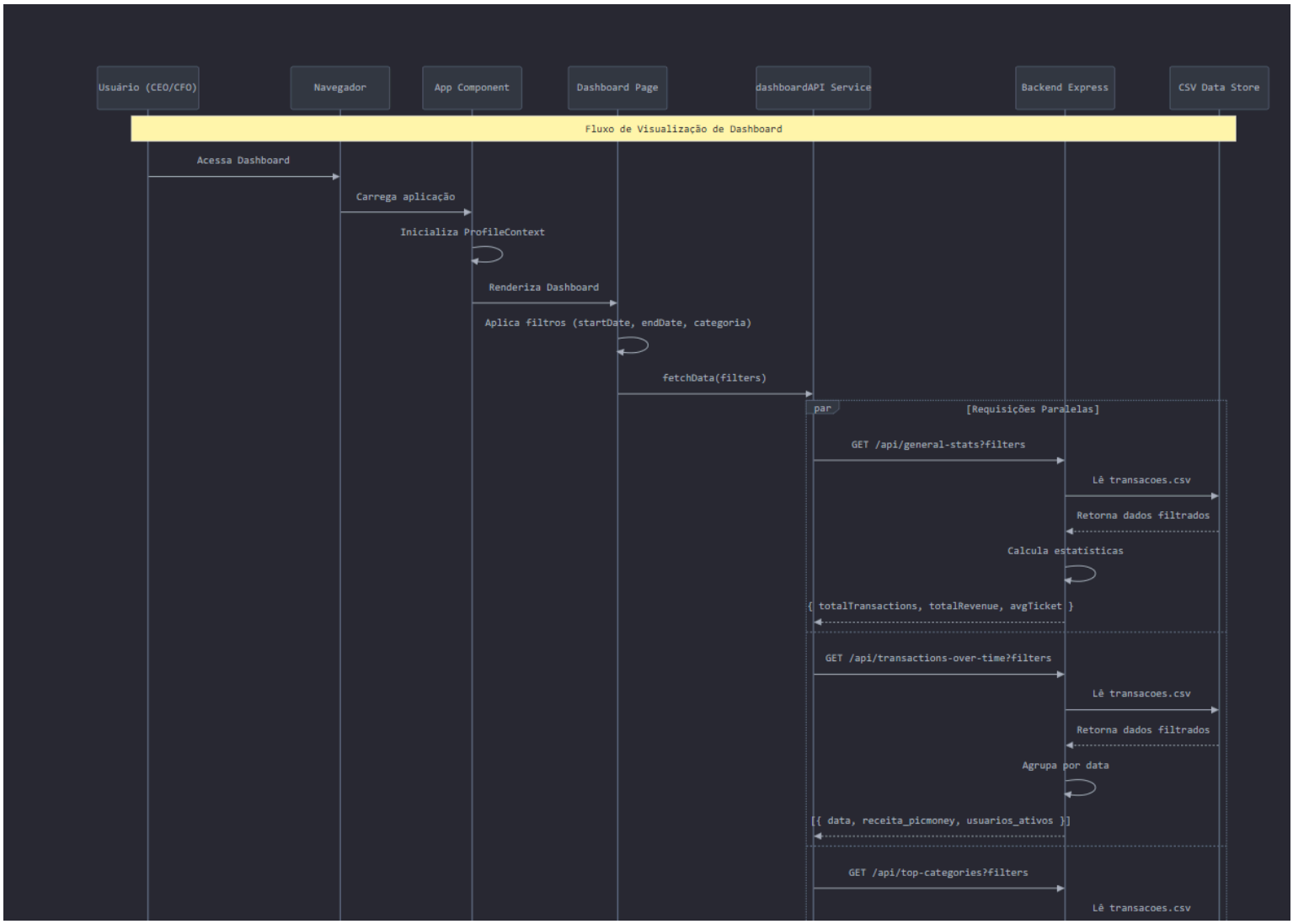
## Diagrama de Classes

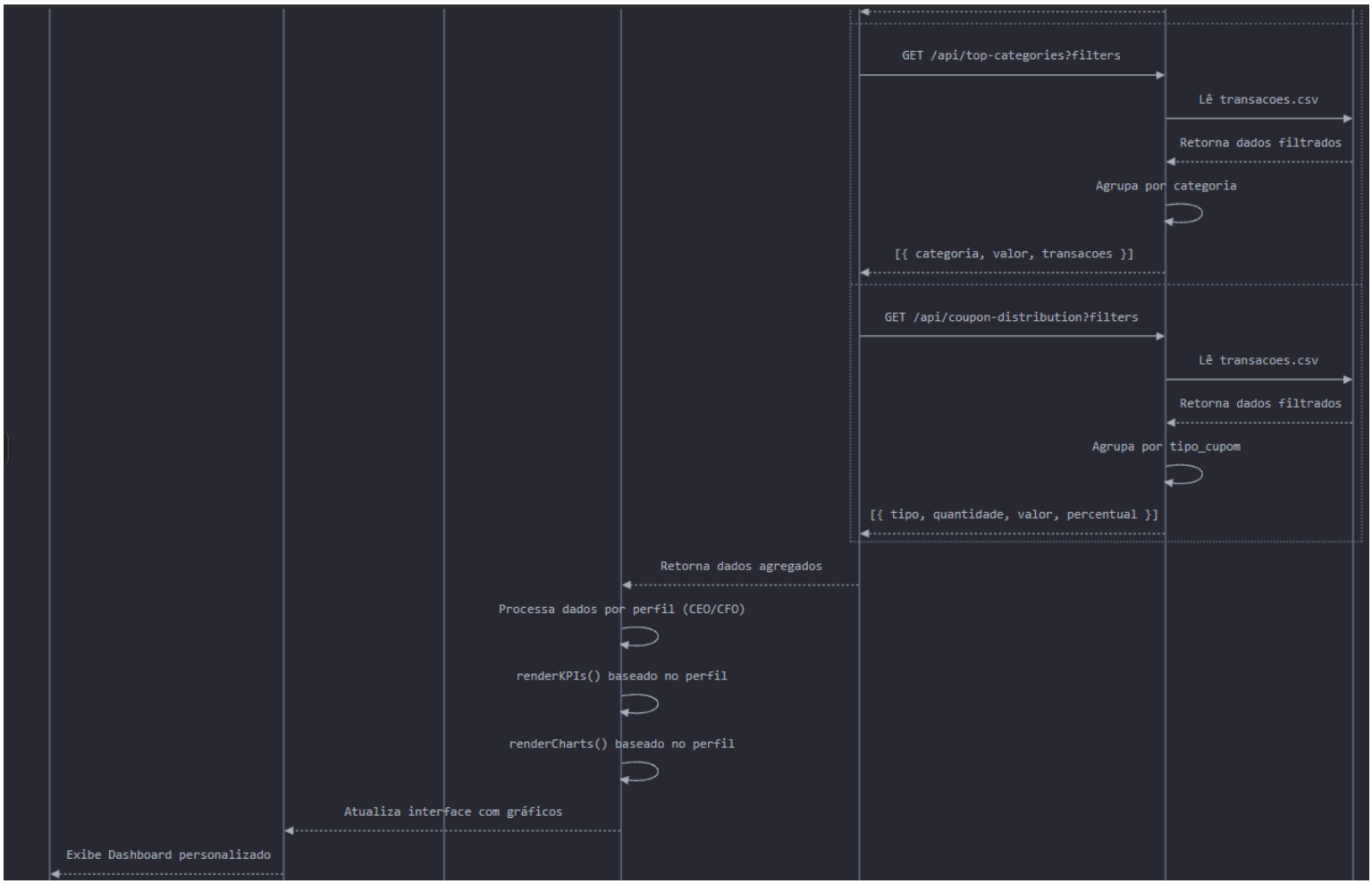


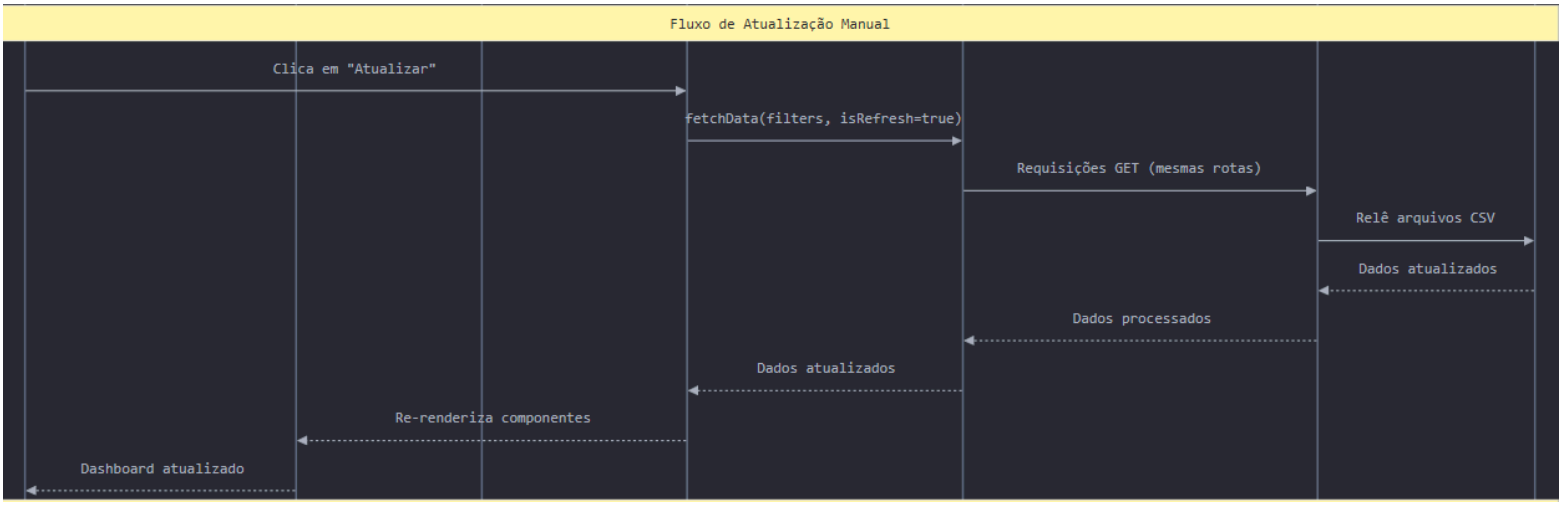

tPage: string
ens: Array
Change(pageId)
PagesByProfile()
Header
-darkMode: boolean
+toggleDarkMode()
-refreshing: boolean
-filterPanelOpen: boolean
-internalFilters: Object
+fetchData(isRefresh, filters)
+handleFiltersChange(newFilters)
+handleRefresh()
+renderKPIs()
+renderCharts()
Camada de abstração para
contém
renderiza
renderiza
consome
FilterPanel
-filters: Object
-isOpen: boolean
-filterOptions: Object
+onFiltersChange(filters)
+onToggle()
+loadFilterOptions()
KPICard
-title: string
-value: string
-icon: Component
-trend: string
-description: string
+render()
«abstract»
ChartContainer
-data: Array
-loading: boolean
-title: string
+render()
«service»
dashboardAPI
+getGeneralStats(filters) : : Promise
+getTransactionsOverTime(filters) : : Promise
+getTopCategories(filters) : : Promise
+getCouponDistribution(filters) : : Promise
+getFilterOptions() : : Promise
+getRevenueByRegion(filters) : : Promise
+getCustomerSegments(filters) : : Promise
+getTimeDistribution(filters) : : Promise
+getPedestresHeatmap(showPenetracao, filters) : : Promise
+getLojasLocations(filters) : : Promise
+getPeakHours(filters) : : Promise
herda
herda
herda
usa
HTTP requests
DualAxisChart
-data: Array
-loading: boolean
+render()
CategoryChart
-data: Array
-title: string
-loading: boolean
+render()
DistributionChart
-data: Array
-title: string
-dataKey: string
-nameKey: string
+render()
«library»
axios
+create(config) : : AxiosInstance
+get(url, config) : : Promise
+post(url, data, config) : : Promise
«interface»
APIController
+getGeneralStats(req, res)
+getTransactionsOverTime(req, res)
+getTopCategories(req, res)
+getCouponDistribution(req, res)
+getFilterOptions(req, res)
+getRevenueByRegion(req, res)
+getCustomerSegments(req, res)
+getTimeDistribution(req, res)
utiliza
utiliza
FilterService
+applyFilters(data, filters) : : Array
+filterByDateRange(data, start, end) : : Array
+filterByCategory(data, category) : : Array
+filterByNeighborhood(data, bairro) : : Array
+filterByCouponType(data, tipo) : : Array
+filterByAgeRange(data, range) : : Array
+filterByGender(data, gender) : : Array
+filterByValueRange(data, min, max) : : Array
DataProcessor
+calculateGeneralStats(transactions) : : Object
+groupTransactionsByDate(transactions) : : Array
+groupByCategory(transactions) : : Array
+calculateCouponDistribution(transactions) : : Array
+calculateRevenueByRegion(transactions) : : Array

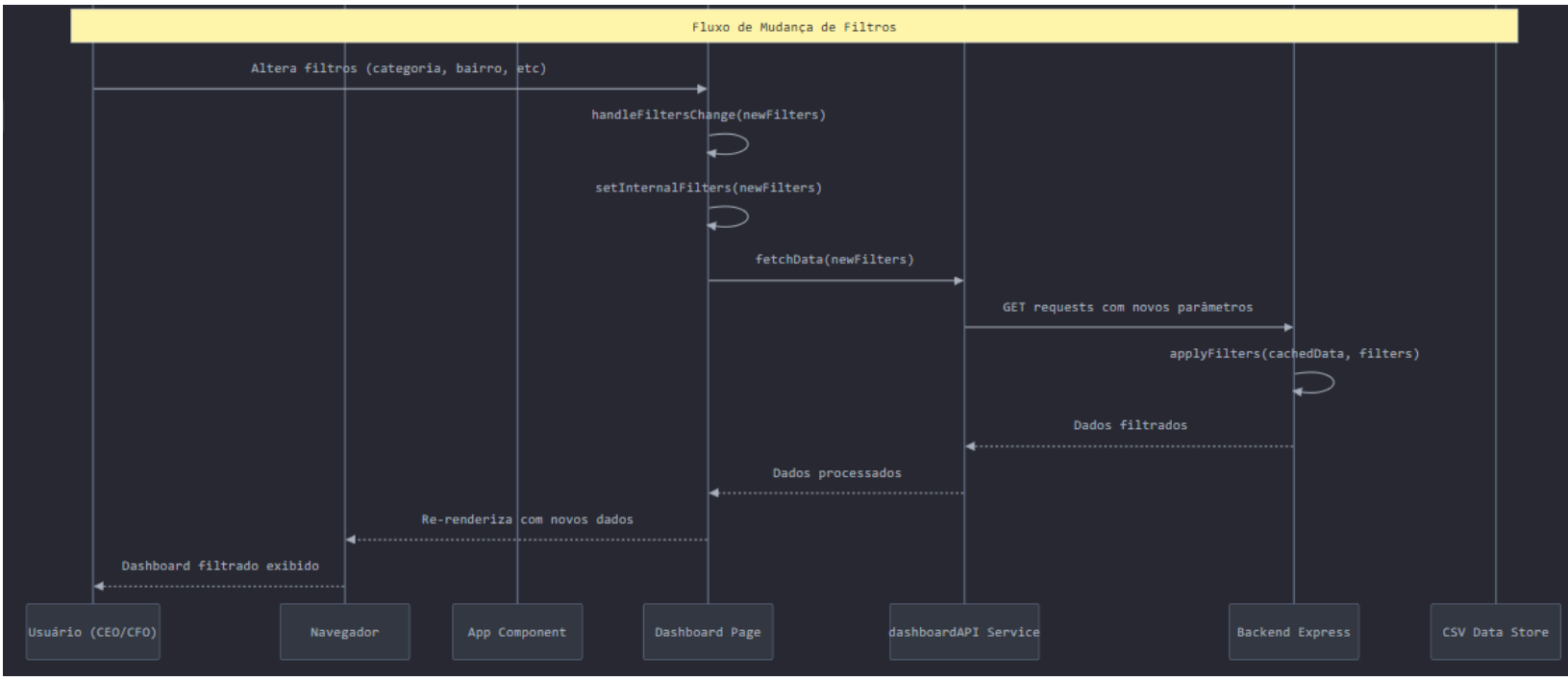

ExpressServer
-PORT: number
-app: Express
-cachedData: CachedData
+startServer()
+initializeData()
+setupMiddleware()
+setupRoutes()
utiliza
utiliza
Armazena dados dos CSVs em memória\npara performance otimizada
gerencia
acessa
CachedData
-transacoes: Array
-lojas: Array
-players: Array
-pedestres: Array
-lastUpdated: Date
+loadCSVData()
+getTransactions() : : Array
+getStores() : : Array
+getPlayers() : : Array
+getPedestres() : : Array
«utility»
CSVReader
+readCSV(filePath) : : Promise<Array>
+parseTransactions(csvData) : : Array
+parseStores(csvData) : : Array
+parsePlayers(csvData) : : Array
+parsePedestres(csvData) : : Array
armazena
armazena
armazena
armazena
cria
cria
cria
cria
TransacaoModel
+data: string
+valor_cupom: number
+repasse_picmoney: number
+nome_estabelecimento: string
+categoria_estabelecimento: string
+bairro_estabelecimento: string
+tipo_cupom: string
+celular: string
+idade: number
+sexo: string
+data_transacao: string
LojaModel
+nome_loja: string
+endereco_loja: string
+latitude: number
+longitude: number
+tipo_loja: string
PlayerModel
+celular: string
+nome: string
+idade: number
+sexo: string
+bairro: string
PedestreModel
+latitude: number
+longitude: number
+local: string
+possui_app_picmoney: boolean
+data: string
+horario: string

## Diagrama de Sequência




GET /api/top-categories?filters
Lê transacoes.csv
Retorna dados filtrados
Agrupa por categoria
[{ categoria, valor, transacoes }]
GET /api/coupon-distribution?filters
Lê transacoes.csv
Retorna dados filtrados
Agrupa por tipo_cupom
[{ tipo, quantidade, valor, percentual }]
Retorna dados agregados
Processa dados por perfil (CEO/CFO)
renderKPIs() baseado no perfil
renderCharts() baseado no perfil
Atualiza interface com gráficos
Exibe Dashboard personalizado


Fluxo de Atualização Manual
Clica em "Atualizar"
fetchData(filters, isRefresh=true)
Requisições GET (mesmas rotas)
Relê arquivos CSV
Dados atualizados
Dados processados
Dados atualizados
Re-renderiza componentes
Dashboard atualizado


Fluxo de Mudança de Filtros
Altera filtros (categoria, bairro, etc)
handleFiltersChange(newFilters)
setInternalFilters(newFilters)
fetchData(newFilters)
GET requests com novos parâmetros
applyFilters(cachedData, filters)
Dados filtrados
Dados processados
Re-renderiza com novos dados
Dashboard filtrado exibido
Usuário (CEO/CFO)
Navegador
App Component
Dashboard Page
dashboardAPI Service
Backend Express
CSV Data Store