<Cifra: Gestão Financeira>
Documento de Arquitetura de Software

Versão <1.0>

Histórico da Revisão

Data	Versão	Descrição	Autor
22/10/2025	1.0	Versão inicial do documento, detalhando a arquitetura monolítica MVC.	Marco Tulio e Lucas Stoppa

Índice Analítico

Documento de Arquitetura de Software

1. Introdução

1.1 Finalidade

Este documento oferece uma visão geral arquitetural abrangente do sistema, usando diversas visões arquiteturais para representar diferentes aspectos do sistema. O objetivo deste documento é capturar e comunicar as decisões arquiteturais significativas que foram tomadas em relação ao sistema.

1.2 Escopo

Cifra é uma plataforma web de organização financeira pessoal. O sistema permitirá aos usuários cadastrarem e categorizar receitas e despesas, definir orçamentos, consultar históricos financeiros e visualizar seu saldo de forma consolidada, visando facilitar o controle financeiro diário.

1.3 Definições, Acrônimos e Abreviações

- MVC (Model-View-Controller): Padrão de arquitetura que separa a aplicação em três componentes lógicos: Dados (Model), Interface (View) e Controle (Controller).
- SQLite: Sistema gerenciador de banco de dados relacional contido em uma biblioteca C, persistindo dados em um arquivo único (orcabem.sqlite).
- EJS (Embedded JavaScript): Motor de visualização usado para renderizar HTML dinâmico no servidor, permitindo o uso de componentes reutilizáveis (partials).
- Node.js: Ambiente de execução JavaScript server-side.
- Partials: Trechos de código de interface reutilizáveis (botões, menus) localizados em views/partials.

1.4 Referências

Documentações e vídeo aulas sobre a o que usamos para criar o programa.

2. Requisitos e Restrições da Arquitetura

Requisito	*Solução*
Linguagem	JavaScript (para backend e frontend dinâmico) e EJS (para templates de visualização).
Plataforma	A aplicação é um monolito web construído sobre Node.js com o framework Express.js. O servidor é responsável por renderizar as páginas HTML usando EJS.

Segurança	A segurança será garantida por autenticação de usuário (sessão/token) gerenciada pelo authController.js e verificada pelo middleware auth.js em rotas protegidas.
Persistência	Banco de dados relacional **SQLite** (arquivo orcabem.sqlite). A modelagem de dados e associações são geridas via ORM na pasta models.
Armazenamento	Arquivos de mídia (comprovantes/fotos) são armazenados localmente na pasta uploads, reduzindo dependência de serviços externos.

3. Visão de Casos de Uso

A arquitetura foi projetada para suportar os seguintes casos de uso centrais, que mapeiam diretamente para os componentes do sistema:

- **HU Autenticação:** Gerenciar cadastro e login de usuários (mapeado para authController.js, authRoutes.js, login.ejs, cadastro.ejs).
- **HU01 e HU02: Gerenciar Transações (Receita/Despesa)**: Permite ao usuário registrar, editar e excluir transações (mapeado para transacaoController.js, transacaoRoutes.js, transacoes.ejs).
- **HU03: Visualizar Saldo Atual**: Apresenta a visão consolidada das finanças (mapeado para dashboardController.js, dashboardRoutes.js, dashboard.ejs).
- **HU07: Exportar Relatório Financeiro**: Permite ao usuário gerar relatórios (mapeado para relatorioController.js, relatorioRoutes.js, relatorios.ejs).
- **HU11: Adicionar Categoria**: Permite ao usuário gerenciar categorias (mapeado para categoriaController.js, categoriaRoutes.js, categorias.ejs).
- **HU12: Configurar Metas de Economia**: Permite ao usuário definir e acompanhar metas (mapeado para metaController.js, metaRoutes.js, metas.ejs).
- **HU21: Gerenciar Contas:** Permite ao usuário gerenciar suas contas (mapeado para contaController.js, contaRoutes.js, contas.ejs).

4. Visão Lógica

4.1 Visão Geral

O sistema utiliza uma arquitetura MVC (Model-View-Controller) estrita:

Camada de Apresentação (Views): Responsável pela interface do usuário, utilizando templates EJS e componentes parciais (views/partials) para reutilização de código visual.
Camada de Controle (Controllers): Orquestra o fluxo da aplicação, recebendo requisições das Rotas, processando regras de negócio e interagindo com os Modelos.
Camada de Dados (Models): Define a estrutura das tabelas e relacionamentos (associations.js), abstraindo o acesso ao arquivo SQLite.

Rotas (routes): Recebem a requisição HTTP.

4.2 Pacotes de Design Significativos do Ponto de Vista da Arquitetura

A estrutura de diretórios do projeto reflete diretamente a divisão em pacotes lógicos:

- **Pacote controllers**
  - **Descrição:** Contém a lógica de negócio principal. Orquestra a interação entre os models (dados) e as views (apresentação).
  - **Componentes Significativos:** transacaoController.js, dashboardController.js, authController.js.
- **Pacote models**
  - **Descrição:** Define as entidades do sistema e a lógica de persistência (conexão e associações com o banco de dados).
  - **Componentes Significativos:** Usuario.js, Transacao.js, Conta.js, Categoria.js, Meta.js, associations.js.
- **Pacote routes**
  - **Descrição:** Mapeia as URLs da aplicação (ex: /dashboard, /transacoes) para os métodos nos controllers.
  - **Componentes Significativos:** dashboardRoutes.js, transacaoRoutes.js, authRoutes.js.
- **Pacote views**
  - **Descrição:** Camada de apresentação. Contém os arquivos de template.ejs que geram o HTML final para o usuário.
  - **Componentes Significativos:** dashboard.ejs, transacoes.ejs, login.ejs, partials/.
- **Pacote public**
  - **Descrição:** Armazena arquivos estáticos que não precisam de processamento no servidor, como CSS e JS de cliente.
  - **Componentes Significativos:** css/style.css, js/main.js.
- **Pacote middlewares**
  - **Descrição:** Contém funções que interceptam requisições para fins específicos, como autenticação.
  - **Componentes Significativos:** auth.js.

5. Visão de Processos (opcional)

A implantação é simplificada e autocontida (Monolito):

- **Nó Único (Servidor):** A aplicação Node.js (app.js), o banco de dados (orcabem.sqlite) e os arquivos de mídia (uploads) residem no mesmo servidor ou contêiner.
- **Vantagem:** Essa arquitetura elimina a latência de rede entre a aplicação e o banco de dados e facilita a migração do sistema (basta copiar a pasta do projeto).
- **Benefício:** Elimina latência de rede interna e simplifica drasticamente a configuração de infraestrutura.

6. Visão de Implantação

A persistência é realizada via SQLite, um banco de dados *serverless*.

- Inicialização: O arquivo `seed.js` é responsável por popular o banco com dados iniciais (catálogos, admin) na primeira execução.
- Estrutura: As tabelas são criadas automaticamente baseadas nos arquivos da pasta `models`.
- Segurança: A integridade é reforçada por backups diários automáticos armazenados na pasta `backups/`.

7. Visão de Dados (opcional)

A perspectiva de dados é central para o Cifra e é definida pelos arquivos no pacote models. As entidades principais são:

- **Usuario**: Armazena dados de autenticação.
- **Conta**: Armazena as contas financeiras do usuário (ex: Carteira, Banco).
- **Categoria**: Armazena as categorias de transações (ex: Alimentação, Salário).
- **Transacao**: Armazena as receitas e despesas.
- **Meta**: Armazena as metas de economia.

O arquivo associations.js define os relacionamentos (inferidos):

- Um Usuario tem muitas Contas, muitas Categorias, muitas Transacoes e muitas Metas.
- Uma Transacao pertence a uma Conta e a uma Categoria.

8. Volume e Desempenho

As características de desempenho foram definidas com base nos Requisitos Não Funcionais (RNFs) das Histórias de Usuário:

- Tempo de Resposta:
  - Registro de transações (HU01/HU02): < 300 ms.
  - Atualização de saldo (HU03): < 200 ms.
  - Geração de relatórios/gráficos: < 500 ms.
- Volume de Dados:
  - O sistema com SQLite suporta confortavelmente o uso individual ou de pequenas empresas (até 100.000 transações anuais) sem degradação de performance.
  - O armazenamento de arquivos (`uploads/`) é limitado apenas pelo disco físico do servidor.

9. Qualidade

Item	*Descrição*	*Solução*
Escalabilidade	Capacidade do sistema de crescer (em usuários e dados) sem degradação.	A arquitetura Node.js permite escalar verticalmente (aumentando CPU/RAM). Para escalar horizontalmente, seria necessário migrar o SQLite para um banco Cliente-Servidor (como PostgreSQL), o que é facilitado pelo uso do ORM nos models.
Confiabilidade	Capacidade do sistema de operar sem falhas e tratar erros adequadamente.	O sistema possui rotinas de recuperação de falhas via backupController.js, garantindo que o usuário possa restaurar seus dados financeiros em caso de corrupção do arquivo principal.
Disponibilidade	Tempo que o sistema permanece acessível e operacional.	A disponibilidade depende do servidor de hospedagem, mas a arquitetura leve permite reinicialização do serviço em segundos em caso de queda (app.js).
Portabilidade	Facilidade de mover a aplicação para diferentes ambientes.	**Alta.** Como o banco de dados é um arquivo e as dependências estão no package.json, o sistema pode ser movido para qualquer sistema operacional (Windows/Linux) apenas copiando a pasta.

Segurança	Proteção contra acesso não autorizado e integridade dos dados.	Implementada via autenticação (authController.js) e autorização (middleware auth.js), além de práticas padrão de segurança web (ex: proteção contra XSS/CSRF).