UNIVERSIDADE FEDERAL DE MINAS GERAIS

VINÍCIUS GABRIEL SILVA FERREIRA

Jenny

Protótipo de Aplicação Web para Auxílio na Organização Financeira de Jovens Adultos

Belo Horizonte, MG

2026

VINÍCIUS GABRIEL SILVA FERREIRA

Jenny

Protótipo de Aplicação Web para Auxílio na Organização Financeira de Jovens Adultos

Trabalho de Conclusão de Curso apresentado ao curso de Sistemas de Informação da Universidade Federal de Minas Gerais, como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Orientador: Thiago Ferreira de Noronha
Coorientador: José Fonseca

Tipo de Pesquisa: Tecnológica

Belo Horizonte, MG
2026

Resumo

O presente projeto de monografia em sistemas de informação aborda a necessidade crescente de organização financeira da Geração Z, propondo o desenvolvimento de Jenny, um protótipo de aplicação web que foca na gestão financeira pessoal e na gerência de despesas em grupo. Este relatório detalha a concepção e o desenvolvimento do projeto, cobrindo a definição de requisitos funcionais, a metodologia de desenvolvimento, as definições arquiteturais, o processo de implementação, os testes e o resultado do processo. Com o objetivo específico de alcançar um MVP (*Minimum Viable Product* – produto mínimo viável) ao fim.

Palavras-chave: gestão financeira pessoal, aplicação web, simplificação de dívidas, algoritmo de fluxo, grafos, fluxo a custo mínimo, scrum, React, TypeScript, PostgreSQL, API REST.

Abstract

The present Information Systems monograph project addresses the growing need for financial organization of Generation Z, proposing the development of Jenny, a web application prototype that focuses on personal financial management and group expense management. This report details the design and development of the project, covering the definition of functional requirements, the development methodology, architectural definitions, the implementation process, testing, and the outcome of the process. The specific goal is to achieve an MVP (Minimum Viable Product) by the end.

Keywords: personal financial management, web application, debt simplification, flow algorithm, graphs, minimum-cost flow, scrum, React, TypeScript, PostgreSQL, REST API.

Sumário

Lista de Ilustrações

Lista de Tabelas

Lista de Algoritmos

1 Introdução

As finanças desempenham um papel crucial na vida de todos, especialmente na sociedade capitalista em que vivemos, influenciando diretamente a conquista de diversos objetivos de vida, como a aquisição de bens ou a busca por estabilidade financeira. Essa área afeta a qualidade de vida, o bem-estar psicológico e outros aspectos do nosso cotidiano.

A relevância do tema financeiro é ainda mais acentuada para jovens adultos (15 a 29 anos), que frequentemente possuem renda reduzida, seja por falta de renda fixa, por estarem em formação acadêmica ou por estarem iniciando sua trajetória profissional. Para muitos, essa é a fase em que têm seu primeiro contato com a necessidade de organização financeira, sendo que 55% dos jovens da Geração Z são os principais responsáveis por suas próprias finanças, conforme pesquisas do Serasa (2025a).

Nesse contexto, a pesquisa também revela que a Geração Z representa o grupo com maior crescimento na negociação de dívidas (SERASA, 2025b). Mais de 1,5 milhão de indivíduos dessa geração negociaram dívidas nos primeiros sete meses do ano, elevando sua representatividade de 9,9% para 13,3% do total de consumidores. Esse dado indica um crescente interesse e engajamento da Geração Z nesse quesito. Por serem uma geração altamente conectada e que digitaliza grande parte de sua vida, o uso de aplicativos surge como uma oportunidade para facilitar o processo de gestão financeira.

Este trabalho tem como objetivo geral o desenvolvimento de um protótipo de aplicação web para auxílio na organização financeira de jovens adultos. Aplicando princípios da engenharia de software (SOMMERVILLE, 2019) num contexto prático ao colocar o aluno na posição de um engenheiro de software.

O objetivo específico deste trabalho é implementar o MVP de um aplicativo e para a sua criação foram definidos os seguintes passos gerais:

- Compreensão aprofundada dos requisitos iniciais e das regras de negócio.
- Definição das tecnologias utilizadas no projeto, bem como o planejamento inicial da arquitetura utilizada.
- Realização da modelagem de dados inicial, indicando como os dados serão definidos e relacionados.
- Implementação dos casos de uso a partir das definições.

Os resultados das decisões tomadas estão detalhados nas seções subsequentes deste relatório. Não há intenção de transformar esse aplicativo em um produto comercial. Sua implementação não tem como objetivo a publicação em nenhuma plataforma de distribuição, pois tem como intuito exclusivamente acadêmico.

2 Referencial Teórico

O mercado de aplicações para gestão financeira pessoal conta com diversas ferramentas consolidadas que servem como referência para os requisitos funcionais esperados pelos usuários. Além dos aplicativos específicos, existem ainda aplicativos mais generalistas com alto grau de customização nos quais os usuários podem criar suas próprias soluções pessoais.

2.1 Alternativas de Mercado

O Organizze (ORGANIZZE, 2026) é um aplicativo principalmente útil quando o usuário possui contas em diversos bancos e deseja centralizar a gerência delas. Seu principal ponto é que, além das features base, nele é possível conectar os respectivos bancos, automaticamente vendo as movimentações bancárias de forma centralizada.

O Mobills (MOBILLS, 2026) também traz a maioria das funcionalidades base do mercado, como a categorização de ganhos e gastos. A aplicação também permite acompanhar as evoluções de faturas dos cartões de crédito e contém conteúdos educativos sobre saúde financeira.

O Splitwise (SPLITWISE, 2026) não é um app de organização financeira por si só, mas sim um de compartilhamento de despesas entre grupos de pessoas. Nele é possível criar um evento e lá controlar seus respectivos gastos. Indicando o que foi pago, por quem e como dividir.

O Notion (NOTION LABS, 2026) é um exemplo de aplicação mais generalista, na qual o usuário pode criar seu próprio dashboard customizado para diversos usos, como, por exemplo, a organização financeira. Essa solução é indicada para usuários mais "hardcore" que desejam uma solução extremamente personalizada que atenda às suas necessidades específicas.

2.2 Diferenciais e Objetivos

É fundamental destacar que o protótipo em questão possui um impacto em uma área mais específica. Ele é voltado para jovens adultos, considerando que este grupo etário demonstra facilidade no uso de aplicativos e tende a ter um orçamento mais controlado, além da frequente necessidade de dividir despesas.

Esses e outros pontos foram considerados na definição dos requisitos e diretrizes de UX do protótipo, que se propõe a oferecer uma experiência de usuário mais intuitiva e simplificada, com foco na organização financeira pessoal e no compartilhamento de despesas entre grupos de pessoas.

3 Resultados Esperados

Ao final deste trabalho, pretende-se obter um conjunto de resultados que se desdobram em duas frentes complementares: o produto gerado, que é o artefato de software, e o processo documentado, que representa o conhecimento de engenharia de software aplicado.

O Produto é o resultado principal e mais concreto do projeto, entregue na forma de um protótipo funcional e demonstrável da aplicação de gestão financeira. Este artefato de software deverá funcionar em navegadores, se adaptando aos diferentes tamanhos de tela, desde celulares até monitores maiores. Além disso, deverá atender a todos os requisitos funcionais definidos no escopo, sendo composto por:

- Cadastro e login de usuários.
- Registro, edição e exclusão de transações (despesas).

- Registro, edição e exclusão de grupos de gastos, em que usuários podem gerenciar suas transações em conjunto, dividindo gastos.
- Simplificação de dívidas, alcançando o número mínimo de transações entre usuários do grupo.

A documentação do processo é outro resultado entregue. Além da aplicação prática, a documentação da jornada de engenharia para construir o projeto também é de grande importância. Este resultado se materializa nos seguintes artefatos e conhecimentos:

- Documentação Abrangente do Projeto: Um conjunto de documentos que narram as decisões técnicas e o planejamento do trabalho, incluindo:
  - O detalhamento da arquitetura de software escolhida, com suas motivações e consequências.
  - O Product Backlog, que serviu como guia para o desenvolvimento nas sprints.
- Repositórios de Código-Fonte: dois repositórios Git contendo os códigos-fonte do projeto, bem como seu histórico de edição.

4 Metodologia

Ao longo do tempo, diversas metodologias de desenvolvimento de software foram criadas, cada uma com suas características e abordagens específicas. Entre elas, destacam-se as metodologias ágeis, que se tornaram populares devido à sua flexibilidade e capacidade de adaptação às mudanças. Dentre as metodologias ágeis, o Scrum se destaca como uma das mais utilizadas, oferecendo um framework estruturado para gerenciar projetos complexos de forma iterativa e incremental, sendo o escolhido para o desenvolvimento do protótipo.

O pontapé inicial para definição da metodolgia do projeto consistiu da definição dos requisitos funcionais e não funcionais do protótipo. Eles foram definidos a partir da análise de mercado e das necesidades do usuário, e serviram como base para a criação do backlog do produto. A partir disso, o desenvolvimento seguiu o ciclo padrão do Scrum, com a realização de sprints curtas e entregas incrementais de funcionalidades.

4.1 Requisitos Funcionais e Não Funcionais

Com a análise do estado da arte do mercado foram definidas quais funções um usuário normalmente espera de aplicativos nesse setor, e a partir disso foram criados os requisitos funcionais e não funcionais (SOMMERVILLE, 2019). Após a definição dos diferenciais, eles foram moldados para cumprir o propósito do aplicativo, resultando nos seguintes requisitos, separados por cada tela do aplicativo:

4.1.1 Tela de Cadastro de Usuário

Requisitos Funcionais (RF)

- **RF-CAD-01:** O sistema deve permitir que um novo usuário se cadastre fornecendo nome completo, e-mail, data de nascimento e uma senha.

- **RF-CAD-02:** O sistema deve validar se o e-mail fornecido já não está em uso por outro usuário.
- **RF-CAD-03:** O sistema deve validar se o formato do e-mail é válido (ex: usuario@dominio.com).
- **RF-CAD-04:** O sistema deve validar se a data de nascimento fornecida está no passado.
- **RF-CAD-05:** O sistema deve exigir que a senha tenha um critério mínimo de segurança (ex: 8 caracteres, contendo letras e números).
- **RF-CAD-06:** O sistema deve ter um campo de confirmação de senha para garantir que o usuário digitou a senha desejada corretamente.

Requisitos Não Funcionais (RNF)

- **RNF-CAD-01:** As senhas dos usuários devem ser armazenadas no banco de dados de forma criptografada (hashed).
- **RNF-CAD-02:** a criptografia da senha deve conter um *salt* para que senhas iguais não geram o mesmo hash.
- **RNF-CAD-03:** A tela deve ser responsiva e se adaptar a dispositivos móveis e desktops.

4.1.2 Tela de Login de Usuário

Requisitos Funcionais (RF)

- **RF-LOG-01:** O sistema deve permitir que um usuário cadastrado faça login usando seu e-mail e senha.
- **RF-LOG-02:** O sistema deve validar as credenciais fornecidas e conceder acesso apenas se forem correspondentes a um usuário existente.
- **RF-LOG-03:** Em caso de credenciais inválidas, o sistema deve exibir uma mensagem de erro clara, sem especificar se o erro foi no e-mail ou na senha.
- **RF-LOG-04:** Após o login bem-sucedido, o sistema deve redirecionar o usuário para a tela de perfil do usuário.

Requisitos Não Funcionais (RNF)

- **RNF-LOG-01:** A sessão do usuário deve ser mantida de forma segura utilizando pares de tokens jwt.
- **RNF-LOG-02:** A sessão do usuário deve ser mantida utilizando cookies gerenciados pelo backend e pelo navegador, sem armazenamento dessas informações no estado do frontend da aplicação.

4.1.3 Tela de Perfil

Requisitos Funcionais (RF)

- **RF-PER-01:** O sistema deve exibir as informações do usuário logado: nome, e-mail e data de nascimento.
- **RF-PER-02:** O sistema deve permitir que o usuário edite seus dados exibidos.
- **RF-PER-03:** O sistema deve permitir que o usuário altere sua senha, exigindo a senha atual e a confirmação da nova senha.
- **RF-PER-04:** O sistema deve fornecer uma opção de "logout" (sair) da conta.

Requisitos Não Funcionais (RNF)

- **RNF-PER-01:** As alterações no perfil devem ser refletidas em tempo real na interface após a confirmação.
- **RNF-PER-02:** A interface deve ser clara e intuitiva, facilitando a localização das opções de edição e logout.

4.1.4 Tela de Times

Requisitos Funcionais (RF)

- **RF-GRU-01:** O sistema deve permitir que o usuário crie um novo time, definindo um nome e adicionando membros (por e-mail, nome de usuário).
- **RF-GRU-02:** O sistema deve exibir uma lista de cards contendo os times que um usuário participa.
- **RF-GRU-03:** O sistema deve permitir que o usuário edite um time.
- **RF-GRU-04:** O sistema deve permitir que o usuário exclua um time.
- **RF-GRU-05:** O sistema deve permitir que o usuário acesse a lista de pagamentos de um time a partir dessa tela.

Requisitos Não Funcionais (RNF)

- **RNF-GRU-01:** O sistema deve garantir que apenas membros convidados tenham acesso às informações e transações do time.
- **RNF-GRU-02:** As permissões de acesso e edição devem ser equivalentes entre membros de um mesmo time.

4.1.5 Tela de Transações de Time

Requisitos Funcionais (RF)

- **RF-TRG-01:** O sistema deve permitir ao usuário registrar uma nova transação, informando: descrição, valor e quais os participantes dela.

- **RF-TRG-02:** O sistema deve permitir ao usuário visualizar uma lista de transações paginadas por mês, ordenadas por padrão da mais recente para a mais antiga.
- **RF-TRG-04:** O sistema deve permitir a edição de qualquer informação de uma transação previamente registrada.
- **RF-TRG-05:** O sistema deve permitir a exclusão de uma transação, mediante confirmação.
- **RF-TRG-06:** O sistema deve calcular e exibir automaticamente o balanço de dívidas entre os membros para o mês atual (quem deve para quem e qual o valor).

Requisitos Não Funcionais (RNF)

- **RNF-TRG-01:** A listagem de transações deve carregar em menos de 3 segundos, mesmo com um histórico de 1000 transações.
- **RNF-TRG-02:** A interface para adicionar uma nova transação deve ser rápida e otimizada para poucos cliques.
- **RNF-TRG-03:** O sistema deve garantir a consistência dos dados, assegurando que o balanço do grupo seja atualizado corretamente após cada operação de CRUD (Criar, Ler, Atualizar, Deletar).

4.2 Desenvolvimento Ágil

O processo será norteado por uma metodologia de desenvolvimento ágil (BECK et al., 2001), o Scrum (SCHWABER, 2004). Esta escolha se deve à sua natureza iterativa e incremental, que permite flexibilidade para adaptações, entregas de valor frequentes e uma gestão transparente do progresso. A metodologia será adaptada para a escala do projeto, aproveitando os princípios e a estrutura do Scrum sem incorrer em sobrecarga processual.

O Scrum é um framework leve que ajuda equipes a gerenciar e solucionar problemas complexos de forma adaptativa, entregando produtos de alto valor. Ele é fundamentado no empirismo, que afirma que o conhecimento vem da experiência e da tomada de decisões com base no que é conhecido (SCHWABER; SUTHERLAND, 2020).

O trabalho em Scrum é organizado em ciclos chamados Sprints, que são períodos de tempo fixos (geralmente de 1 a 4 semanas) em que um "Incremento" de produto usável e potencialmente liberável é criado. Essa será a divisão desse projeto, com a adição de uma "sprint 0" mais longa, na qual serão realizadas as definições arquiteturais, o aprendizado das tecnologias necessárias, bem como o preenchimento da base do backlog do produto, gerenciado na ferramenta Jira (ATLASSIAN, 2026). A partir disso, as sprints subsequentes serão compostas pelo ciclo padrão de desenvolvimento do Scrum, entregando um incremento na solução ao final de cada uma.

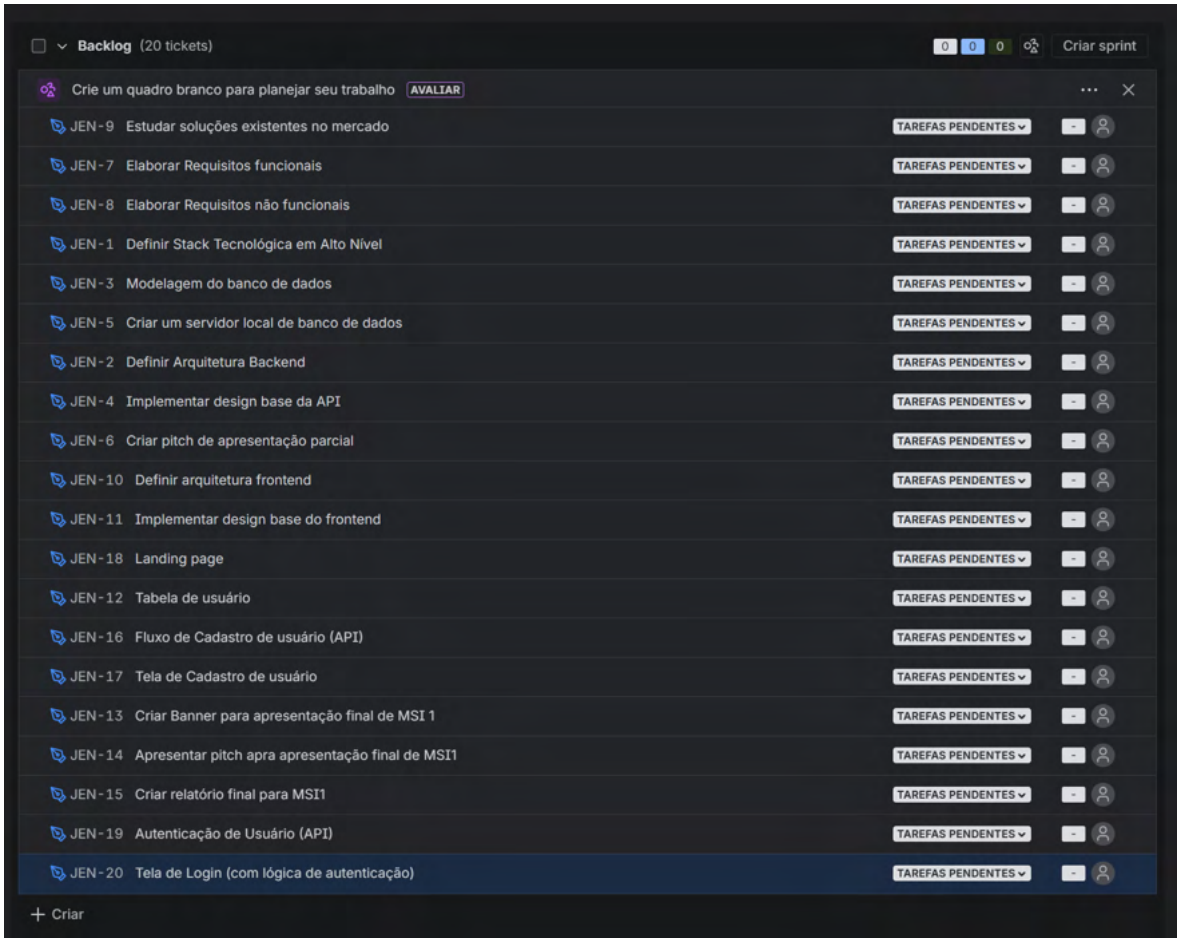

Figura 1: Backlog inicial de tarefas do projeto na ferramenta Jira, elaborado durante a sprint 0 a partir dos requisitos funcionais e não funcionais levantados.

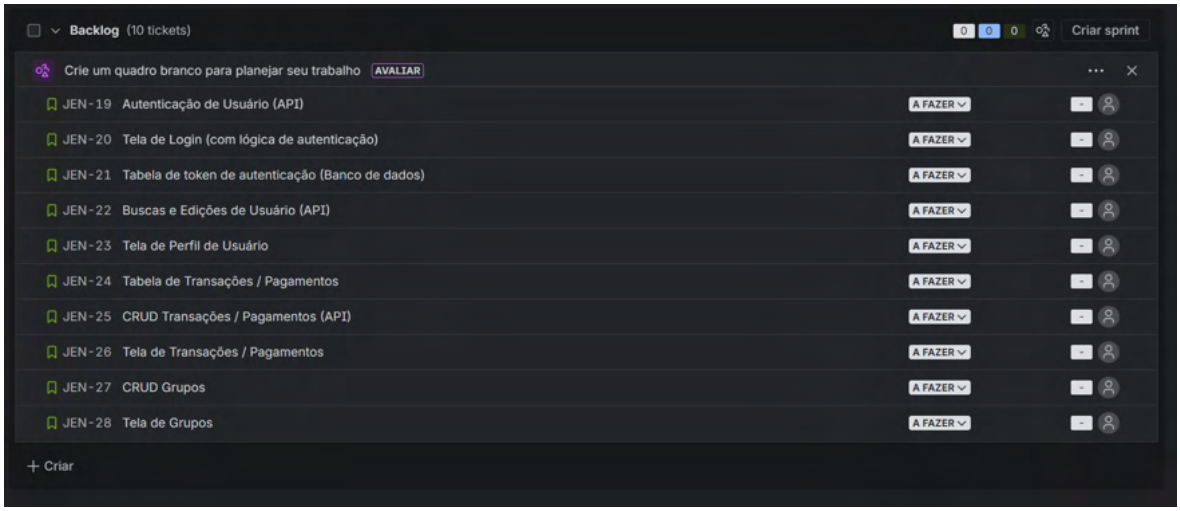

Figura 2: Backlog de histórias de usuário da segunda parte do projeto na ferramenta Jira, referente às sprints de desenvolvimento incremental das funcionalidades.

5 Arquitetura

A definição inicial de maior impacto no aplicativo foi a estruturação da sua arquitetura em alto nível, pensando no caminho dos dados desde a inserção pelo usuário até o seu armazenamento. Nesse sentido, o aplicativo foi estruturado em três seções: um frontend que lida com as interações do usuário, um backend que estrutura os acessos e realiza validações, e um banco de dados que armazena as informações. Sendo explicitados a seguir:

5.1 Banco de Dados

Para armazenamento das informações foi escolhido o uso de um banco de dados relacional PostgreSQL (POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2026). Essa escolha se deve principalmente às necessidades da feature de gerência de transações e simplificação de dívidas. Nessas features, será necessária a leitura com alta performance de um alto número de transações ao longo do tempo. Assim, são necessários recursos com alta relacionabilidade, uma forte presença de histórico e um foco em leitura. Sendo assim, um banco relacional é a escolha chave para armazenamento dessas informações (ELMASRI; NAVATHE, 2019).

O sistema de gerenciamento de banco de dados escolhido foi o PostgreSQL, uma grande referência no mercado. O motivo de sua escolha é o fato de possuir um ponto-chave: mesclar com eficiência conceitos de bancos relacionais e não relacionais, entregando a possibilidade de extrair o melhor dos dois mundos.

Na monografia em sistemas de informação 1 foi realizada a configuração local do banco de dados. Esse processo envolveu uma containerização utilizando Docker (DOCKER, 2026), o que simplificou a configuração de ambiente e possibilitou uma fácil migração. Já que retira a preocupação em relação à configuração da máquina em que o banco está implementado e também automatiza sua criação e setup através de scripts.

5.1.1 Entidades e Relacionamentos

Na concepção do projeto completo, foram utilizadas as seguintes entidades para armazenamento de informações:

Entidade Conta de Usuário (user_account)

A tabela `user_account` é a entidade central do sistema, responsável por armazenar as informações cadastrais de todos os usuários. Ela guarda os dados pessoais básicos, como nome completo (`full_name`), endereço de e-mail único (`email`), data de nascimento (`birth_date`), e a senha criptografada (`password_hash`), além de um campo opcional para a foto de perfil (`avatar`). Sendo o núcleo do aplicativo, esta tabela não possui chaves estrangeiras, mas é referenciada por praticamente todas as outras tabelas do banco (como tokens, times, membros e pagamentos), indicando a quem pertencem as ações e registros no sistema.

Entidade Tokens de Atualização (auth_refresh_token)

Focada na segurança e na manutenção das sessões, esta tabela gerencia os tokens de atualização usados no fluxo de autenticação. Ela guarda a string do token (`token`),

a data de expiração (`expires_at`), um identificador de família de tokens para prevenir ataques de roubo de sessão (`family_id`) e um status de revogação (`is_revoked`). Relaciona-se diretamente com a tabela `user_account` através da chave estrangeira `user_id`, garantindo que, se um usuário for deletado (`ON DELETE CASCADE`), todos os seus tokens ativos sejam destruídos simultaneamente.

Entidade Time (team)

A tabela `team` é responsável por registrar os grupos de divisão de despesas criados pelos usuários (como repúblicas, viagens ou grupos de amigos). Ela armazena o nome do time (`name`) e rastreia quem foi o usuário criador através do campo `created_by_user_id`, que é uma chave estrangeira ligada à tabela `user_account`. É importante notar que a exclusão do usuário criador é restrita (`ON DELETE RESTRICT`) enquanto houver grupos vinculados a ele, garantindo a integridade dos dados coletivos.

Entidade Membros de Time (team_membership)

Trata-se de uma tabela associativa (ou de junção) essencial para resolver o relacionamento de "muitos para muitos" entre Usuários e Grupos. Ela armazena apenas as chaves estrangeiras `team_id` e `user_id`, formando juntas a chave primária composta da tabela. Seu único objetivo é registrar quais usuários pertencem a quais grupos. Por usar o comportamento `ON DELETE CASCADE` em ambas as chaves, se um usuário ou um grupo for excluído do sistema, a associação entre eles é desfeita automaticamente.

Entidade Pagamentos Pessoais (payment)

Esta tabela é o coração da funcionalidade futura de gestão financeira pessoal. Ela registra todas as transações individuais, guardando a descrição (`title`), o valor (`amount`), a data em que ocorreu (`payment_date`), a categoria (`category`) e o status da transação (`status`, um enum que pode indicar se está pago, pendente, ou se falhou). Uma característica importante é que o valor é um número inteiro obrigatoriamente maior que zero, isso porque o armazenamento dos valores ocorrerá em centavos para evitar erros de arredondamento. Ela se relaciona diretamente com a `user_account` (através de `user_id`) para identificar o dono da transação. Essa tabela foi criada e mantida como uma preparação do banco para uma futura generalização de transações utilizada na feature de transações pessoais.

Entidade Pagamentos de Time (team_payment)

Onde ocorre o controle de despesas colaborativas. Esta tabela guarda os gastos feitos dentro de um grupo específico. Além dos dados comuns a uma transação (título, valor em centavos e data), ela possui relacionamentos fundamentais: vincula-se ao grupo (`team_id`) e identifica quem pagou a conta (`payer_id`, referenciando `user_account`). O grande diferencial desta tabela é o uso do campo `debtors_ids`, que é um Array de UUIDs. Este array armazena os IDs de todos os usuários do grupo que participaram daquela despesa e, portanto, devem sua parte ao `payer_id`.

Desta forma, é gerado o seguinte diagrama de entidade-relacionamento:

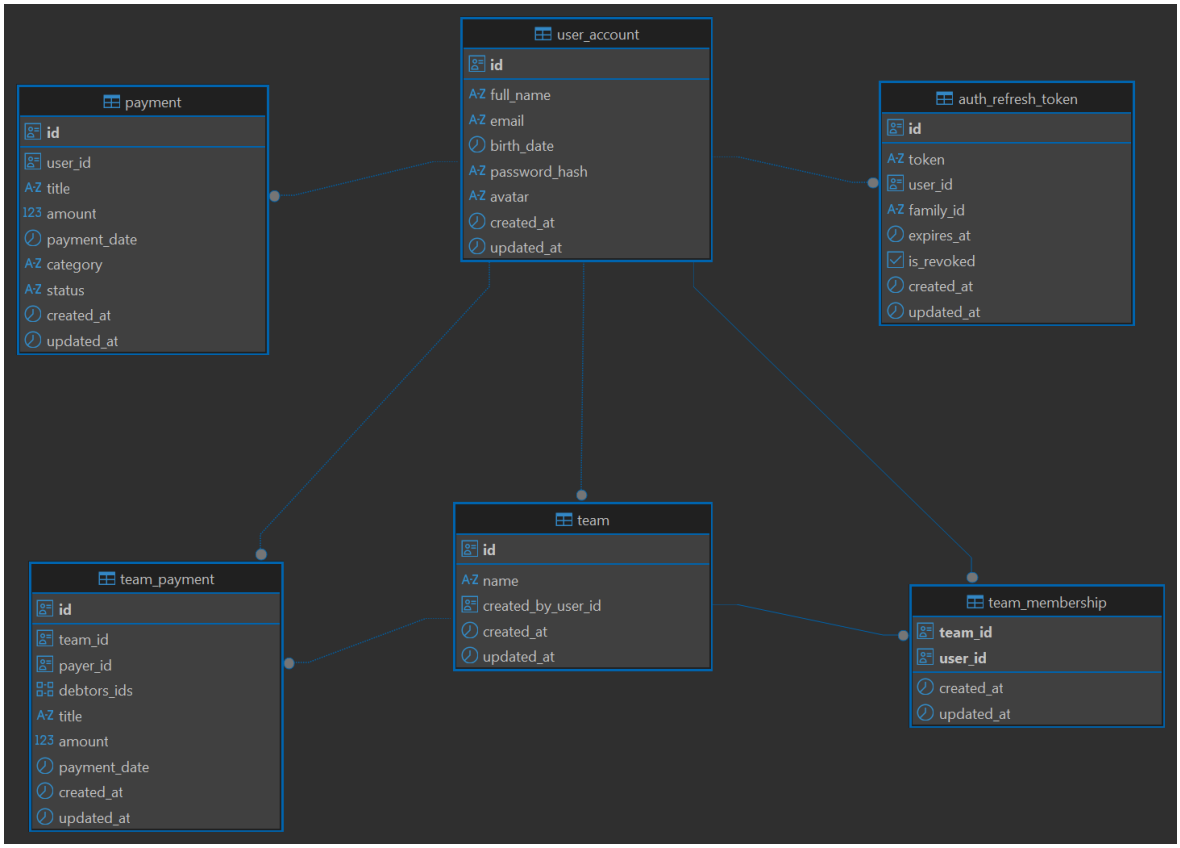

Figura 3: Diagrama entidade-relacionamento do banco de dados PostgreSQL, com as tabelas `user_account`, `auth_refresh_token`, `team`, `team_membership`, `payment` e `team_payment` e os relacionamentos entre elas.

5.2 Backend

Já o backend, principal responsável pela garantia das regras de negócio, é composto por uma API RESTful (FIELDING, 2000) em Node (OPENJS FOUNDATION, 2026) na arquitetura MVC (Model-View-Controller). O framework que dita os padrões nela é um dos mais utilizados para construção de APIs no ambiente web: o Express (EXPRESS, 2026). Sua leveza, velocidade, flexibilidade e estrutura voltada a eventos permitem que seja uma boa opção para a estruturação da API.

O padrão MVC é extremamente limpo e simples de aplicar; ele organiza o sistema em três camadas bem definidas. O *model* gerencia os dados e as regras de negócio. A *view* apresenta os dados ao "usuário" (no nosso caso, o agente externo que realizou a chamada pelo protocolo HTTP utilizando os padrões REST). E o *controller* age como uma camada intermediária, realizando a coordenação (KRASNER; POPE, 1988).

Nesse contexto, foi escolhido o Sequelize (SEQUELIZE, 2026) para realizar a representação do banco de dados em modelos na API. Essa biblioteca nos permite mapear as tabelas do banco em classes e utilizar seus métodos internos, bem como *queries* nativas para realização de operações no banco. Para adaptação dessa biblioteca aos princípios de tipagem contidos no typescript, foi utilizada uma biblioteca de terceiros, recomendada pelo time da Sequelize: o sequelize-typescript.

Além disso, o backend utiliza como modelo e estratégia de testes a pirâmide de

testes, proposta por Cohn (2009). Essa estratégia sugere uma distribuição ideal de testes automatizados, com maior quantidade de testes de unidade na base, menos testes de integração no meio e a menor quantidade de testes ponta a ponta (*end-to-end*) no topo (VOCKE, 2018). A biblioteca de testes utilizada também é largamente ativa no mercado, e uma em que o desenvolvedor já possui experiência, o Jest (JEST, 2026).

5.3 Frontend

O frontend, responsável pela interação com o usuário, é composto por uma aplicação web responsiva construída em React (META PLATFORMS, 2026). O React é uma biblioteca para construção de interfaces de usuário interativas e dinâmicas, dividindo os elementos visuais em componentes independentes para melhor gerência de estado e otimização da performance de renderização e de resposta.

A arquitetura escolhida para organizar o frontend foi a Feature-based Architecture. Nela, ao invés de organizar os componentes apenas por tipo (ui, routes, pages, helpers). Acima de tudo, eles são agrupados por features (auth, home, profile) e dentro dessas features são separados por tipos.

Isso mantém uma certa localidade de referência entre os componentes. Partindo do princípio de que componentes de uma mesma feature interagem com mais frequência entre si, mantêm um mesmo motivo de mudança e essas mudanças geram consequências nos mais próximos; faz todo sentido que esses componentes fiquem juntos dentro de suas respectivas features, promovendo modularidade, manutenibilidade e escalabilidade.

Além disso, essa arquitetura faz com que cada feature seja um módulo autocontido com mínimo impacto em outras partes da aplicação, já que quase todas as necessidades são supridas por componentes internos. Isso permite uma alta modularidade e a fácil aplicação de padrões mais avançados de performance e hospedagem, como a separação em microfrontends.

A ferramenta de build utilizada nesse projeto é o Vite (VITE, 2026). Ele agiliza o desenvolvimento e a configuração, criando um ambiente base de build fácil de modificar, já embutindo vários *setups*, como o *linter* e um ambiente de teste automatizado, o Vitest (VITEST, 2026).

Em relação aos testes automatizados, como o funcionamento de um componente isolado é muito menos importante e difere bastante do funcionamento desse componente como parte integral de uma interface, a filosofia utilizada é a do troféu de testes (*Testing Trophy*) (DODDS, 2021). Ela prioriza a realização de testes de integração (entre componentes de uma página, por exemplo) em detrimento dos testes unitários e *end-to-end*.

Para agilização do desenvolvimento foi escolhida uma biblioteca de componentes, a Shadcn (SHADCN, 2026). Ela foi escolhida por estar no perfeito equilíbrio entre uma biblioteca de *ui headless* (sem estilos, apenas os componentes com suas interações, mas extremamente customizável) e uma biblioteca de *ui* em caixa preta (componentes com interação e estilos bem definidos, prontos para uso, mas baixa possibilidade de customização). E faz isso ao adicionar apenas o componente necessário com código aberto, estilo e interação prontos na pasta aberta de componentes compartilhados do projeto, permitindo que os desenvolvedores editem e customizem o que for necessário a qualquer momento.

6 Desenvolvimento

No desenvolvimento do projeto foram usados diversos padrões de projeto. Um exemplo é o uso dos princípios SOLID (MARTIN, 2002) para a construção de classes e funções, garantindo que cada uma delas tenha uma única responsabilidade, seja aberta para extensão, mas fechada para modificação, e assim por diante. Além disso, foram aplicados padrões de projeto (GAMMA et al., 1994) como o Singleton, para garantir que certas classes tenham apenas uma instância; o Factory Method, para criar objetos sem especificar a classe exata; e o Observer, para permitir que objetos sejam notificados sobre mudanças em outros objetos.

A refatoração (FOWLER, 2004) também foi um ponto constante no desenvolvimento do projeto, com o objetivo de melhorar a legibilidade, manutenibilidade e desempenho do código (MARTIN, 2008). Isso envolveu a reorganização de funções, a simplificação de estruturas complexas e a eliminação de duplicações, sempre mantendo o comportamento observável original do sistema.

Algumas features merecem explicações à parte, seja por sua complexidade, integração ou importância dentro da aplicação. São elas: a feature de autenticação que integra cadastro, login e lógica de acesso; e a feature de simplificação de dívidas que implementa um algoritmo de fluxo para obtenção da menor quantidade de transações monetárias dentro de um time.

6.1 Autenticação e Segurança

Autenticação é uma feature extremamente chave no sistema que lida com uma das partes mais sensíveis dele, assim, optou-se por uma arquitetura própria completamente sob controle no lugar de uma biblioteca externa. Para a stack, foi escolhido um sistema híbrido de sessão de tokens jwt (JONES; BRADLEY; SAKIMURA, 2015).

No lado do cliente, os tokens foram armazenados em cookies http-only (BARTH, 2011). Esse local foi escolhido pois esses cookies são gerenciados apenas pelo navegador: ele salva quando o backend responde e envia em toda requisição automaticamente, sem nenhum acesso por parte do JavaScript do frontend. O fato do frontend não conseguir acessar o conteúdo do cookie diminui a chance de ataques XSS (Cross-Site Scripting) (OWASP FOUNDATION, 2026) em que atacantes tentam roubar credenciais armazenadas no cliente para se passar pelo usuário.

Assim, dois tipos de token foram orquestrados: tokens de acesso e tokens de atualização. O token de acesso é um JWT de vida curta (15 minutos), que funciona como o crachá em cada chamada à API e é validado só pela assinatura, sem tocar no banco. O token de atualização vive mais tempo (7 dias), fica persistido no banco e serve apenas para pedir um novo token de acesso quando o atual expira.

O ponto de maior risco na orquestração escolhida é a possibilidade de vazamento do token de atualização, já que, com ele, um atacante conseguiria tentar acessos por dias antes da validade expirar. Para defesa desse cenário, foi criado um sistema de rotação do token ligado à sua família. Assim que um token de atualização é usado para renovar a sessão (a cada 15 minutos), ele é marcado como revogado e um novo da mesma família é emitido no lugar. Assim, mesmo que esse token seja roubado, ele ainda será invalidado quando o usuário legítimo renovar sua sessão. Nesse momento, o sistema detecta a tentativa com o token inválido e revoga todos os acessos daquela

família de tokens por precaução.

Em resumo, o fluxo de autenticação do usuário se dá da seguinte forma: no login, o backend confere a senha contra o hash, gera o par de tokens, salva o refresh no banco e devolve os dois dentro de cookies HTTP-only; o front guarda só os dados do usuário (nome, email, avatar, etc.), nunca os tokens. A partir disso, ao acessar uma rota protegida, o navegador manda o cookie junto e um middleware no backend valida a assinatura do token de acesso.

Após o tempo da sessão, o token de acesso irá expirar. Nesse momento, a requisição volta com um erro HTTP 401 (não autorizado) (FIELDING; NOTTINGHAM; RESCHKE, 2022), e é esse 401 que dispara todo o mecanismo de renovação. No front há um interceptador que escuta esses erros: ao ver um 401, ele segura a requisição que falhou (e enfileira outras que chegarem nesse intervalo, para não disparar vários refreshes simultâneos), chama o endpoint de refresh, recebe os cookies renovados e refaz a requisição original.

Para o usuário, tudo isso é invisível; ele não percebe que o token foi trocado. Além disso, se o usuário ficar até sete dias sem acessar o sistema, como o token de autenticação ainda estará válido, a renovação também acontecerá automaticamente. Só quando o próprio refresh falha, porque expirou ou foi invalidado, é que o sistema considera o usuário realmente deslogado, limpa o estado e redireciona para o login.

Um ponto digno de nota é a possibilidade de acesso à mesma conta por múltiplos dispositivos, permitida pela independência entre famílias de tokens. A criação de um token de atualização juntamente com uma família está atrelada à realização de um novo login, que por sua vez está atrelado ao navegador utilizado pelo usuário (já que o cookie é gerenciado pelo navegador). Ou seja, caso o usuário realize um login em um novo navegador ou dispositivo, será criado um novo token de uma nova família com acessos, expirações e ciclos de vida completamente independentes, permitindo que o usuário acesse a mesma conta de múltiplos dispositivos ao mesmo tempo. Para limitar o excesso de sessões, foi programado um limite de 3 sessões simultâneas por usuário.

6.2 Simplificação de Dívidas

A principal característica do projeto é a simplificação de dívidas. Isso, pois em grupos de pessoas que compartilham despesas, é comum que uma única pessoa pague algo em nome de várias. Por exemplo: alguém paga a conta inteira de um restaurante que será dividida entre todos, outra pessoa paga as compras do mês, uma terceira adianta o valor de uma corrida de aplicativo, etc. Cada um desses pagamentos feitos no grupo possui três elementos:

- um valor total da despesa que foi paga.
- um pagador: a pessoa que efetivamente desembolsou o dinheiro
- um conjunto de devedores: as pessoas entre as quais o valor da despesa é dividido

6.2.1 Definição do Problema

Em grupos com histórico muito grande de despesas, forma-se uma rede grande de dívidas entre os membros: a deve para b, que deve para c, que por sua vez já havia pago algo que beneficiou a, e assim por diante. Nesse contexto, acertar essas contas da

maneira mais simples com cada um pagando individualmente a fração devida exige um número enorme de transferências entre usuários, com várias delas sendo potencialmente desnecessárias. Por exemplo: se a deve 10 a b e b deve 10 a c, não faz sentido realizar duas transferências: basta a pagar 10 diretamente a c.

Reduzir esse número de transferências é importante, pois cada uma delas representa um esforço físico do usuário para realizá-las. Seja mais facilmente com um Pix ou com um esforço maior entregando papel-moeda manualmente, ainda é um trabalho a ser feito. Diminuir a quantidade de trabalho facilita a vida do usuário, ainda mais quando se trata de dinheiro, em que erros literalmente podem custar caro.

E assim definimos o problema a ser resolvido: Dado o histórico de despesas de um grupo, determinar um conjunto de transferências de dinheiro que quite todas as dívidas utilizando o menor número possível de transferências.

Além do problema, definimos também o seguinte limitador: Após a simplificação, nenhuma pessoa poderá realizar uma transferência para outra com um valor maior do que a dívida original entre essas duas pessoas. Esse limitador foi definido pois simplifica consideravelmente a complexidade da solução e também contribui para a interpretabilidade do resultado final, evitando perguntas como "por que estou pagando 30 reais a fulano sendo que inicialmente só devia 5?".

Para melhorar a visualização, será definido um exemplo considerando um grupo com 3 pessoas: A, B e C. Elas foram a um churrasco e dividiram os pagamentos da seguinte forma: A comprou os drinks, pagando 30 reais e dividindo igualmente 10 reais para cada. B fez o tropeiro, pagando 60 reais numa divisão de 20 para cada. Já C comprou as carnes, pagando 90 reais e dividindo 30 para cada. Nesse exemplo temos a seguinte tabela de gastos:

Pagante	**Valor**	**Devedores**
A	30	A: 10 B: 10 C: 10
B	60	A: 20 B: 20 C: 20
C	90	A: 30 B: 30 C: 30

Tabela 1: Tabela de dívidas do exemplo do churrasco: cada linha traz um pagamento com seu pagante, o valor total desembolsado e a cota devida por cada participante (devedores).

6.2.2 Modelagem do Problema

Histórico de Despesas

A rede de divisões de dívidas descrita inicialmente é complicada, mas pode ser drasticamente simplificada por uma observação: para acertar as contas, não importa qual despesa originou cada dívida. Formalmente, define-se para cada membro i o seu saldo líquido b_i como a diferença entre o total que recebeu (como pagador) e o total que deve (como devedor):

$$b_i = \underbrace{\sum_{\text{pagamentos em que } i \text{ é pagador}} \text{valor}}_{\text{recebe}} - \underbrace{\sum_{\text{pagamentos em que } i \text{ é devedor}} \text{cota}_i}_{\text{deve}}.$$

A partir dessa definição, cada pessoa se encaixa em uma das situações:

- Saldo positivo: a pessoa pagou mais do que consumiu, ou seja, tem dinheiro a receber. É uma credora.
- Saldo negativo: a pessoa consumiu mais do que pagou, ou seja, tem dinheiro a pagar. É uma devedora.
- Saldo zerado: a pessoa já está quite e pode ser ignorada; não participa de mais nenhuma transação.

Com isso, podemos definir outro axioma no problema. Como todo dinheiro que sai de alguém entra em outra pessoa, a soma de todos os saldos é necessariamente zero; o total a pagar é exatamente igual ao total a receber. Ou seja:

$$\sum_i b_i = 0$$

No exemplo, a pessoa A tem saldo -30, a pessoa B tem saldo 0 e a pessoa C tem saldo 30.

Montagem da Rede

Para acertar as dívidas foi usada a teoria de fluxo em redes (CORMEN et al., 2009; KLEINBERG; TARDOS, 2006). O problema é modelado sobre um grafo dirigido:

$$G = (V, A)$$

Cada pessoa é um vértice da rede. Esses vértices são conectados por arestas direcionadas que representam as dívidas da seguinte forma:

Para cada pessoa i, j do grupo, é criada uma aresta direcionada de i para j se i deve algum dinheiro para j; a capacidade dessa aresta é a quantidade de dinheiro devida e o peso é um valor constante 1, já que a transferência de dinheiro entre pessoas é igualmente custosa. A tabela do exemplo se traduz no seguinte grafo:

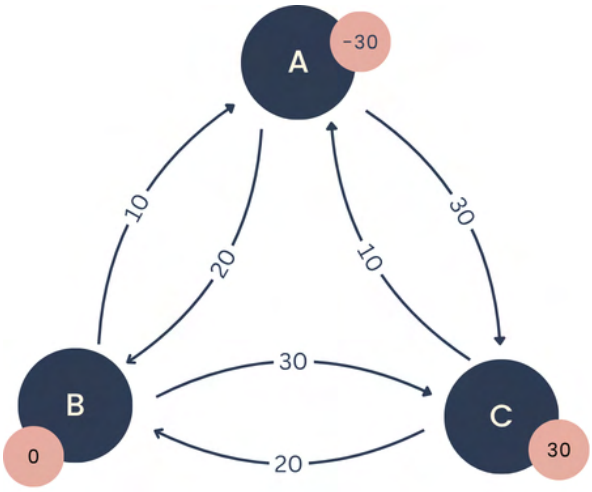

Figura 4: Rede inicial de dívidas do exemplo, um grafo dirigido em que cada vértice é um membro do grupo e cada aresta representa, com seu valor, a dívida de uma pessoa para outra.

Sobre essa estrutura foram criados dois pontos artificiais que não correspondem a pessoas reais, uma fonte e um sumidouro. A fonte é um reservatório de onde todo o dinheiro a ser movimentado "nasce". O sumidouro é um ralo para onde todo o dinheiro deve "escoar".

Assim, é criado um vértice de fonte s (*source*) ligado a todos os devedores por arestas de capacidade igual ao total devido no grupo e peso constante 0. É também criado um vértice de sumidouro t (*sink*). Todo recebedor é ligado a t por arestas com capacidade igual ao total devido no grupo e peso constante 0. No exemplo, essa rede é vista da seguinte forma:

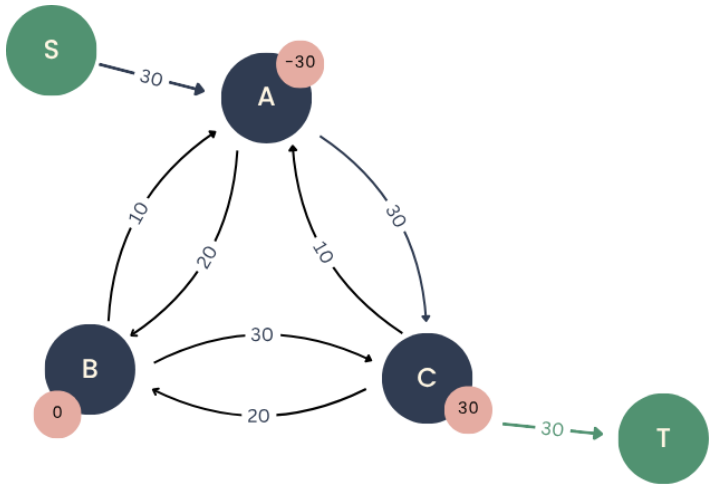

Figura 5: Rede de dívidas do exemplo acrescida dos vértices artificiais de fonte s e sumidouro t, ligados respectivamente aos devedores e aos credores para modelar o acerto de contas como um problema de fluxo a custo mínimo.

Resolução da Rede

Para resolver essa rede, será necessário empurrar dinheiro da fonte até o sumidouro. Cada unidade que sai da fonte por um devedor representa parte de uma dívida sendo paga; cada unidade que chega ao sumidouro por um credor representa parte de um crédito sendo recebido.

Como a capacidade máxima das arestas ligadas da fonte aos devedores e dos credores ao sumidouro é igual ao total devido e a capacidade das arestas internas do grafo é limitada pelas dívidas que cada um tem entre si. Fazer todo o dinheiro possível escoar da fonte ao sumidouro equivale a quitar todas as dívidas. E pela própria montagem da rede, é garantido que sempre será possível escoar todo o dinheiro dessa forma.

Mas não basta quitar as dívidas; queremos fazê-lo com o menor número de transferências. Isso significa que queremos escoar todo esse dinheiro utilizando a menor quantidade possível de arestas. Assim, o problema do acerto de contas se traduz em um problema clássico: fluxo a custo mínimo (AHUJA; MAGNANTI; ORLIN, 1993).

Definição Formal do Problema

Assim, podemos modelar como um problema de fluxo a custo mínimo (*minimum-cost flow*) sobre um grafo dirigido:

$$G = (V, A)$$

Vértices

$$V \;=\; \{s,\, t\} \cup M,$$

Em que M é o conjunto de membros do time, s é uma fonte artificial e t é um sumidouro artificial.

Arestas

- Arestas de dívida: para cada dívida de i para j, uma aresta (i, j) com capacidade u_{ij} igual ao valor devido e custo $c_{ij} = 1$, representando que uma transação monetária é necessária para realizar aquele pagamento. Dívidas paralelas na mesma direção são acumuladas em uma única aresta.
- Arestas da fonte: para cada devedor líquido i (com $b_i < 0$), uma aresta (s, i) com capacidade $|b_i|$ e custo 0.
- Arestas para o sumidouro: para cada credor líquido i (com $b_i > 0$), uma aresta (i, t) com capacidade b_i e custo 0.

As arestas artificiais (incidentes a s ou t) têm custo nulo por não corresponderem a transações reais; apenas as arestas de dívida contribuem para o custo total, que assim mede o número de transações realizadas. A ideia é empurrar um fluxo total

$$F \;=\; \sum_{i\,:\,b_i>0} b_i$$

Da fonte s até o sumidouro t. Como cada aresta de dívida tem custo unitário, minimizar o custo total do fluxo equivale a minimizar a quantidade de fluxo que atravessa arestas de dívida, a fazer isso favorecendo caminhos curtos e, portanto, a reduzir o número de transferências resultantes.

$$\textbf{minimizar} \quad \sum_{(i,j)\,\in\,A} c_{ij}\, x_{ij}$$

$$\textbf{sujeito a} \quad \sum_{j\,:\,(i,j)\in A} x_{ij} \;-\; \sum_{j\,:\,(j,i)\in A} x_{ji} \;=\; \begin{cases} F, & \text{se } i = s, \\ -F, & \text{se } i = t, \\ 0, & \text{caso contrário,} \end{cases} \qquad \forall\, i \in V,$$

$$0 \;\le\; x_{ij} \;\le\; u_{ij}, \qquad \forall\, (i,j) \in A,$$

em que:

- x_{ij} é o fluxo (quantia transferida) na aresta (i, j);
- c_{ij} é o custo unitário da aresta, $c_{ij} = 1$ para arestas de dívida e $c_{ij} = 0$ para arestas artificiais;

- u_{ij} é a capacidade da aresta (o valor máximo transferível, isto é, o Pix máximo daquela relação de dívida);
- a primeira restrição é a conservação de fluxo: o fluxo líquido que sai de cada vértice é $+F$ na fonte, $-F$ no sumidouro e 0 nos demais
- a segunda restrição limita o fluxo de cada aresta à sua capacidade.

A garantia de viabilidade decorre da construção: atribuindo a cada aresta de dívida o fluxo igual à sua capacidade total, o balanço de fluxo de cada vértice resulta exatamente em b_i. Logo, sempre existe um fluxo viável que satura todas as arestas da fonte e do sumidouro, isto é, de valor F.

6.2.3 Modelagem da Solução

O algoritmo escolhido para resolver o fluxo a custo mínimo é o de caminhos mínimos sucessivos (*Successive Shortest Path*, SSP) (BUSACKER; GOWEN, 1960; EDMONDS; KARP, 1972). A ideia central é incremental: enquanto houver fluxo a ser enviado, encontra-se no grafo residual o caminho de menor custo da fonte ao sumidouro e empurra-se por ele a maior quantidade de fluxo possível (o gargalo). Como cada caminho escolhido é de custo mínimo, o fluxo acumulado é, a cada iteração, de custo mínimo para o seu valor, propriedade essa que se mantém até atingir o fluxo total F.

Grafo Residual e Arestas Reversas

Para cada aresta (i, j) com capacidade u_{ij}, fluxo x_{ij} e custo c_{ij}, o grafo residual contém:

- a capacidade residual direta $u_{ij} - x_{ij}$, com custo c_{ij}
- uma aresta reversa (j, i) com custo $-c_{ij}$, que permite cancelar fluxo previamente enviado.

A aresta reversa é essencial: ela possibilita que o algoritmo "desfaça" decisões anteriores quando um caminho melhor é descoberto, sendo o que garante a otimalidade do custo. Na implementação, o fluxo é mantido de forma antissimétrica ($x_{ji} = -x_{ij}$), de modo que o fluxo líquido em cada aresta de dívida pode ser lido diretamente ao final. É também essa estrutura que permite que dívidas em sentidos opostos entre dois membros (por exemplo, A deve a B e B deve a A) sejam corretamente compensadas em uma única transferência líquida.

Encontrando o Caminho Mais Curto: Dijkstra com Potenciais

Encontrar a rota mais barata entre dois pontos de uma rede é um problema conhecido, resolvido pelo algoritmo de Dijkstra (DIJKSTRA, 1959). De forma simplificada, ele explora a rede a partir da fonte sempre expandindo primeiro o ponto que, até o momento, foi alcançado pelo menor custo acumulado, como se fosse uma mancha de tinta que se espalha mais rápido pelos caminhos mais baratos, garantindo que, ao tocar um ponto pela primeira vez, o faz pelo trajeto de custo mínimo.

Como as arestas reversas possuem custo negativo, o algoritmo de Dijkstra não poderia ser aplicado diretamente. Utiliza-se, então, a técnica de potenciais de Johnson (JOHNSON, 1977; TOMIZAWA, 1971): mantém-se um potencial π_v por vértice e substitui-se o custo real pelo custo reduzido.

$$c_{ij}^{\pi} \;=\; c_{ij} + \pi_i - \pi_j \;\geq\; 0,$$

Que é garantidamente não negativo. Após cada execução de Dijkstra, os potenciais são atualizados com as distâncias mínimas encontradas ($\pi_v \leftarrow \pi_v + \text{dist}(v)$), preservando a invariância de não-negatividade na iteração seguinte. Os potenciais iniciam em zero, o que é válido, pois, no grafo inicial, nenhuma aresta de custo negativo possui capacidade residual positiva.

Dijkstra foi utilizado nesse contexto por uma razão de projeto: a possibilidade de customizar custos de transação no futuro. Se no futuro quisermos, por exemplo, que certas pessoas prefiram pagar umas às outras, basta atribuir custos diferentes às arestas; o algoritmo de caminhos mínimos passará a escolher as rotas levando esses custos em conta, sem nenhuma outra alteração.

6.2.4 Resolvendo o problema

Concluída a execução do fluxo a custo mínimo, o grafo contém o fluxo ótimo em cada aresta, bastando traduzir novamente para o domínio do problema. As arestas artificiais conectadas à fonte e ao sumidouro são ignoradas, pois não representam transações reais. As arestas reversas que carregam fluxo não positivo também são ignoradas.

Para cada aresta de dívida real (i, j) com fluxo líquido positivo $x_{ij} > 0$, emite-se uma transferência: o membro i deve pagar x_{ij} ao membro j. Essa lista de transferências é retornada ao cliente. Ela representa a minimização do número de transações sob a restrição definida.

6.2.5 Pseudocódigo

Assim, o algoritmo de caminhos mínimos sucessivos se define pelo pseudocódigo apresentado no Algoritmo 1.

Algoritmo SuccessiveShortestPath(G, s, t, F)
  para cada $v \in V$ **faça** $\pi_v \leftarrow 0$
  $f \leftarrow 0$
  enquanto $f < F$ **faça**
    $(\text{dist}, \text{prev}) \leftarrow \text{Dijkstra}(G, s, \pi)$
    se $\text{dist}[t] = \infty$ **então pare** // sem caminho aumentante
    para cada $v \in V$ com $\text{dist}[v] < \infty$ **faça**
      $\pi_v \leftarrow \pi_v + \text{dist}[v]$ // atualiza potenciais
    $\delta \leftarrow F - f$
    para cada (i, j) no caminho $s \rightsquigarrow t$ **faça**
      $\delta \leftarrow \min\big(\delta,\ u_{ij} - x_{ij}\big)$ // gargalo
    para cada (i, j) no caminho $s \rightsquigarrow t$ **faça**
      $x_{ij} \leftarrow x_{ij} + \delta$
      $x_{ji} \leftarrow x_{ji} - \delta$ // fluxo antissimétrico
    $f \leftarrow f + \delta$
  retorne x

Algoritmo 1: Caminhos mínimos sucessivos (*Successive Shortest Path*). Resolve o problema de fluxo a custo mínimo de forma incremental: a cada iteração, busca no grafo residual o caminho de menor custo da fonte s ao sumidouro t e empurra por ele o máximo de fluxo possível (o gargalo), repetindo até escoar o fluxo total F. Os potenciais π_v de cada vértice são atualizados a cada passo, mantendo os custos reduzidos não-negativos e permitindo o uso de Dijkstra.

A busca do caminho de menor custo executada a cada iteração é detalhada no Algoritmo 2.

Algoritmo Dijkstra(G, s, π)
  para cada $v \in V$ **faça** $\text{dist}[v] \leftarrow \infty$
  $\text{dist}[s] \leftarrow 0$
  enquanto $\exists$ vértice não-visitado com $\text{dist} < \infty$ **faça**
    $u \leftarrow \underset{v \text{ não-visitado}}{\arg\min}\ \text{dist}[v]$
    marca u como visitado
    para cada (u, v) com $u_{uv} - x_{uv} > 0$ **faça**
      $c^{\pi}_{uv} \leftarrow c_{uv} + \pi_u - \pi_v$
      se $\text{dist}[u] + c^{\pi}_{uv} < \text{dist}[v]$ **então**
        $\text{dist}[v] \leftarrow \text{dist}[u] + c^{\pi}_{uv}$
        $\text{prev}[v] \leftarrow u$
  retorne $(\text{dist}, \text{prev})$

Algoritmo 2: Busca de caminho de custo mínimo (Dijkstra com potenciais de Johnson). A partir da fonte s, expande sempre o vértice não visitado de menor distância acumulada e relaxa as arestas com capacidade residual positiva usando o custo reduzido $c^{\pi}_{uv} = c_{uv} + \pi_u - \pi_v$, garantidamente não-negativo. Retorna as distâncias mínimas dist e os predecessores prev, que reconstroem o caminho de s a t.

6.2.6 Complexidade

Seja $n = |V|$ o número de vértices e $m = |A|$ o número de arestas. Cada iteração do laço principal envia ao menos uma unidade do fluxo total F e executa um Dijkstra. Na implementação, por se tratar de grafos pequenos (poucos membros por time), utiliza-se a versão $O(n^2)$ de Dijkstra, resultando em complexidade $O(F \cdot n^2)$ no pior caso. Para grafos maiores, a substituição por uma fila de prioridades reduziria cada busca a $O(m \log n)$ (CORMEN et al., 2009).

6.2.7 Resultados da Simplificação

No exemplo, isso significa encontrar o seguinte fluxo entre A e C:

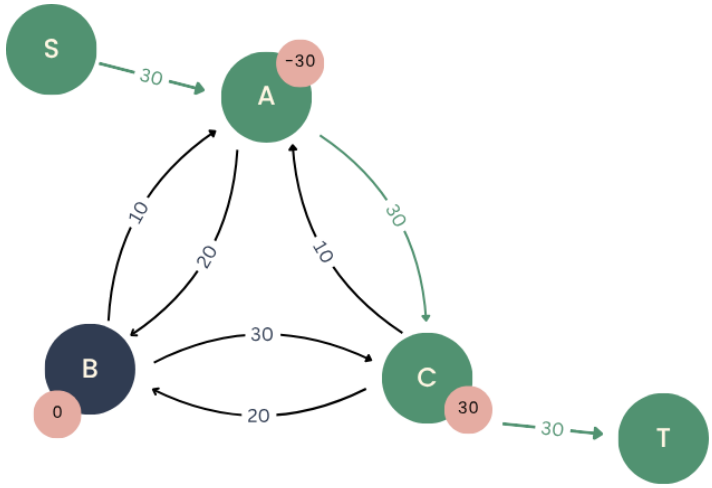

Figura 6: Rede do exemplo com o fluxo a custo mínimo destacado, indicando por quais arestas e em que quantidade o dinheiro escoa da fonte s ao sumidouro t.

O que retorna a seguinte rede simplificada:

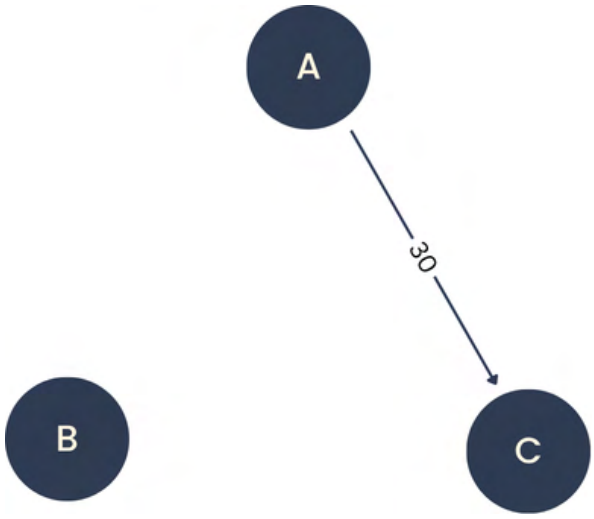

Figura 7: Rede final de transações simplificadas do exemplo, contendo apenas as transferências mínimas necessárias para quitar todas as dívidas: uma única transferência de A para C.

No contexto do aplicativo, esse resultado é exibido na tela de simplificação de dívidas. Com o seguinte exemplo:

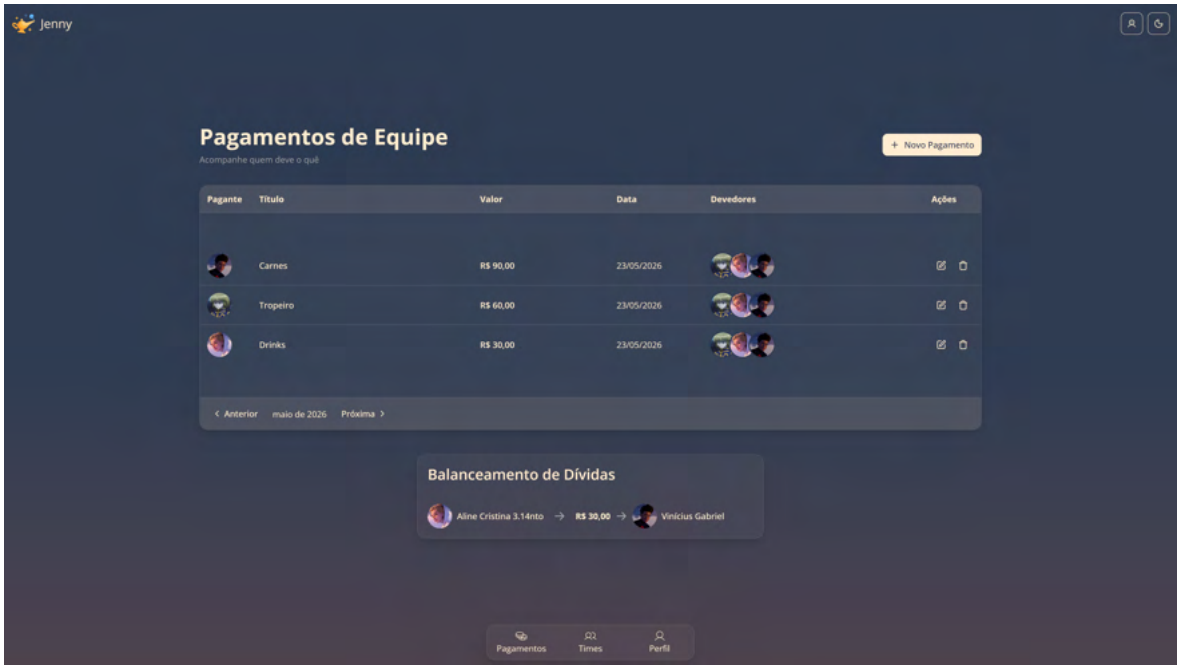

Figura 8: Exemplo da tela de pagamentos de um time exibindo ao usuário a lista de dívidas já simplificada pelo algoritmo (desktop, tema escuro).

7 Inteligência Artificial

O uso de inteligência artificial no projeto se deu de algumas formas. Uma forma mais prática foi o uso na implementação de código. Outras formas mais teóricas foram pesquisas e refinamentos relacionados a princípios de desenvolvimento de software. Além disso, essas ferramentas foram utilizadas na geração de documentação do projeto, como na criação de requisitos e no auxílio na descrição e explicação das features e dos processos no relatório.

A forma mais teórica foi o uso de agentes de pesquisa para contextualização e aprofundamento de tópicos de desenvolvimento. Ela se deu no uso de ferramentas como o *deep research* do Gemini (GOOGLE, 2026) para a escrita de pequenos artigos sobre áreas de necessidade no desenvolvimento em que o desenvolvedor estava "enferrujado" ou desatualizado, como "configuração de um ambiente de testes em React usando Vitest". Também foi utilizada como passo inicial para a busca de informações que baseiam decisões arquiteturais, com a posterior validação de fontes e pesquisa detalhada manual.

A forma mais prática foi o uso de agentes de código auxiliares, como o Github Copilot (GITHUB, 2026b) e o Claude (ANTHROPIC, 2026b), que auxiliaram na geração de códigos base para funções e componentes simples a partir da função de autocomplete. Essas ferramentas também foram usadas na correção de erros e bugs. Além disso, alguns agentes foram utilizados para validar e refinar soluções feitas pelo desenvolvedor, sejam de código ou não, buscando melhorar o detalhamento, performance, legibilidade, extensibilidade, entre outros princípios da engenharia de software.

Na documentação, os agentes foram utilizados para validação dos textos escritos no relatório de modo a aumentar a clareza e o entendimento. Também foram usados

na validação dos textos da interface do usuário, na definição de requisitos e descrição destes nas histórias e tasks.

7.1 Práticas Utilizadas para Trabalho com IA

No contexto do desenvolvimento de software, algumas boas práticas foram utilizadas para obtenção de melhores resultados com o uso de inteligência artificial. Essas são configurações a nível de repositório que ajudam a dar mais contexto para os agentes e a conseguir melhores resultados mais rapidamente.

7.1.1 Instructions

Instruções foram utilizadas para definir axiomas dentro do projeto. Regras imutáveis que devem ser consideradas para implementações em toda a aplicação. Essas regras são injetadas automaticamente como contexto base para qualquer requisição feita para agentes de IA (GITHUB, 2026a).

Nesses arquivos foram descritas informações a respeito do negócio, como o objetivo do projeto, seu público-alvo, as plataformas em que funciona e suas diretrizes de experiência do usuário para design. Fazendo com que os agentes considerem o negócio e tenham o usuário final como objetivo.

Além disso, a arquitetura do projeto foi documentada nestes arquivos. Contendo itens como: sua stack tecnológica com regras de uso, comandos para executar e testar, as convenções e padrões de projeto usados, regras de roteamento, regras para validação de dados e gerência de estado da aplicação, entre outros. Com o objetivo de fazer com que códigos gerados por IA sigam os padrões definidos na modelagem do produto.

7.1.2 Skills e Agents

Para injeção de contexto vinculado a tarefas específicas, foram usadas as skills. Essa é uma feature importante para evitar o sobrecarregamento de contexto nas IAs, bem como para a diminuição do uso de tokens (ANTHROPIC, 2026a).

A principal tarefa que recebeu skills foi o teste automatizado de software, especificamente os unitários e de integração. Tanto no backend quanto no frontend foram criadas skills descrevendo a metodologia de testes utilizada, juntamente com boas práticas e testes de referência.

No frontend foi adicionado o impeccable (BAKAUS, 2026), um pacote de skills e agentes focadas no desenvolvimento de interfaces e na experiência do usuário. Ele foi utilizado para planejar, desenvolver e incrementar os designs do projeto, com responsividade e interfaces limpas.

Juntamente com as skills, foi adicionado um agente de acessibilidade, que valida e melhora as interfaces de acordo com as regras da WCAG 2.1/2.2 (W3C, 2018, 2023) para alcançar uma experiência inclusiva.

8 Resultados e Discussões

A seguir, destacam-se os resultados alcançados, com uma breve descrição das telas e os prints das interfaces. Nota-se que por trás dessas telas existe toda uma estrutura

para a execução do caso de uso de maneira correta, com o backend e o banco de dados. Além disso, cada tela será apresentada com sua versão em tema claro e escuro com múltiplas variações de tamanho de tela.

8.1 Landing page

Um dos resultados alcançados foi a landing page, que apresenta o Jenny para o usuário, permitindo que ele navegue para as features de acesso inicial, cadastro e login.

Figura 9: Landing page do Jenny em desktop no tema claro: tela inicial que apresenta o aplicativo ao visitante e dá acesso às funcionalidades de cadastro e login.

Figura 10: Landing page do Jenny em desktop no tema escuro: tela inicial que apresenta o aplicativo ao visitante e dá acesso às funcionalidades de cadastro e login.

Figura 11: Landing page do Jenny em dispositivo móvel, nos temas claro (à esquerda) e escuro (à direita).

8.2 Página de Cadastro

A página de cadastro é primariamente composta por um formulário em que o usuário preenche suas informações pessoais, com validações automatizadas.

Figura 12: Página de cadastro em desktop no tema claro, com o formulário de criação de conta e as validações automáticas dos dados do novo usuário.

Figura 13: Página de cadastro em desktop no tema escuro, com o formulário de criação de conta e as validações automáticas dos dados do novo usuário.

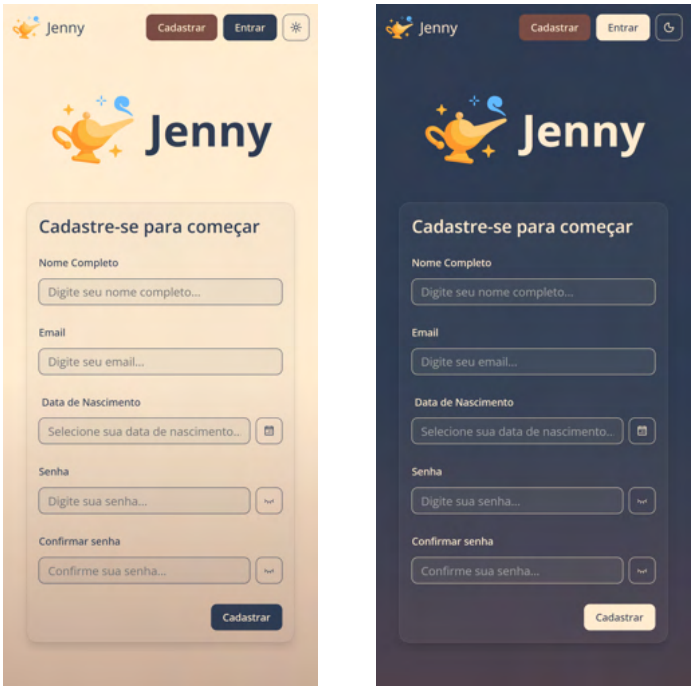

Figura 14: Página de cadastro em dispositivo móvel, nos temas claro (à esquerda) e escuro (à direita).

8.3 Página de Login

A página de login é primariamente composta por um formulário em que o usuário preenche suas informações de acesso, com validações automatizadas.

Figura 15: Página de login em desktop no tema claro, com o formulário de autenticação do usuário por e-mail e senha.

Figura 16: Página de login em desktop no tema escuro, com o formulário de autenticação do usuário por e-mail e senha.

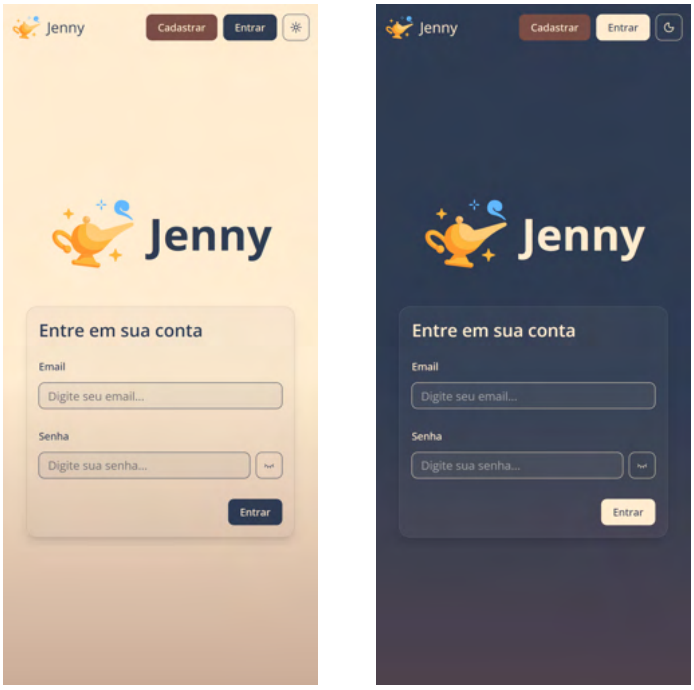

Figura 17: Página de login em dispositivo móvel, nos temas claro (à esquerda) e escuro (à direita).

8.4 Página de Perfil

A página de perfil é primariamente composta por formulários onde o usuário pode editar suas informações e alterar sua senha.

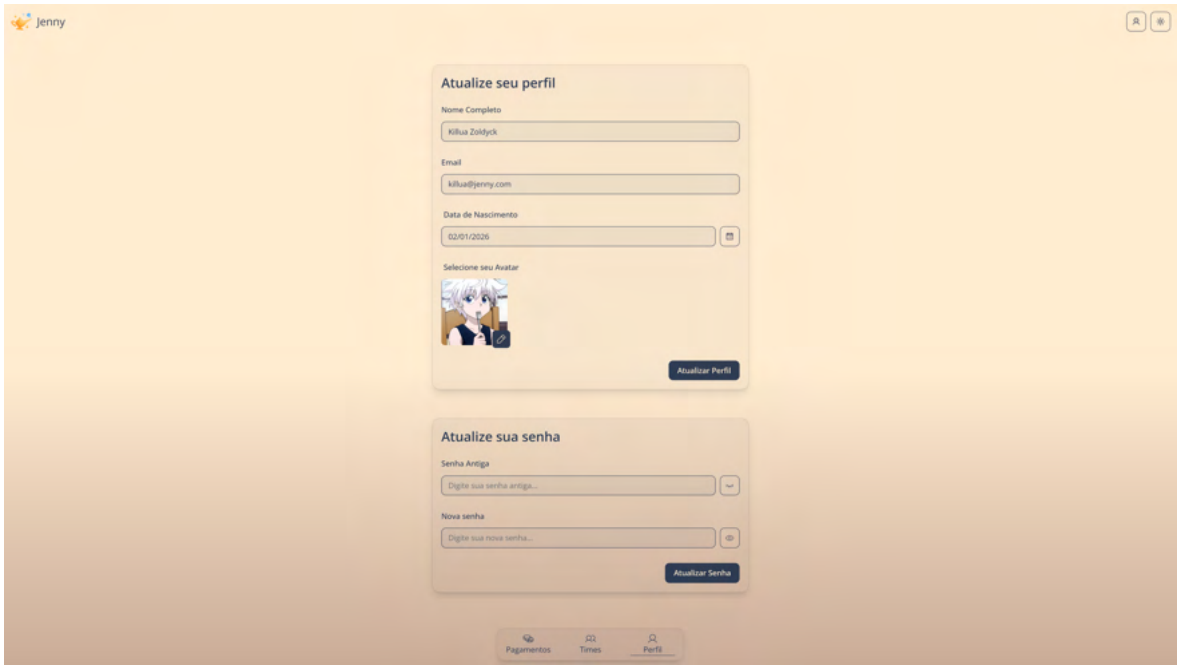

Figura 18: Página de perfil em desktop no tema claro, onde o usuário visualiza e edita seus dados pessoais e altera a senha.

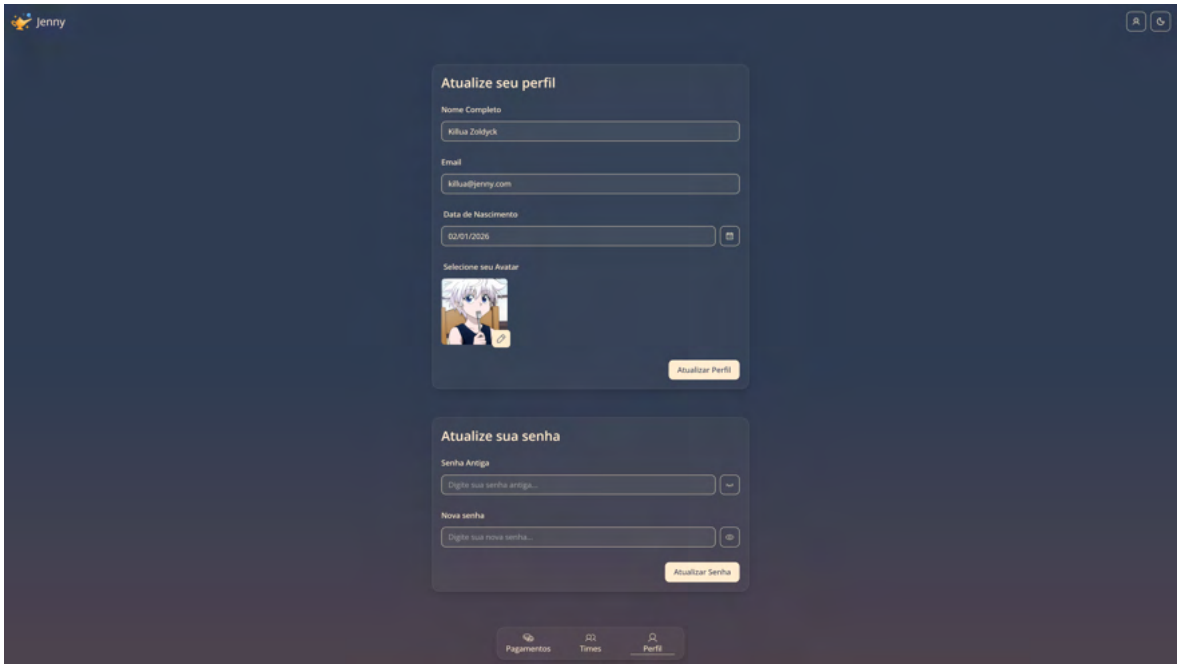

Figura 19: Página de perfil em desktop no tema escuro, onde o usuário visualiza e edita seus dados pessoais e altera a senha.

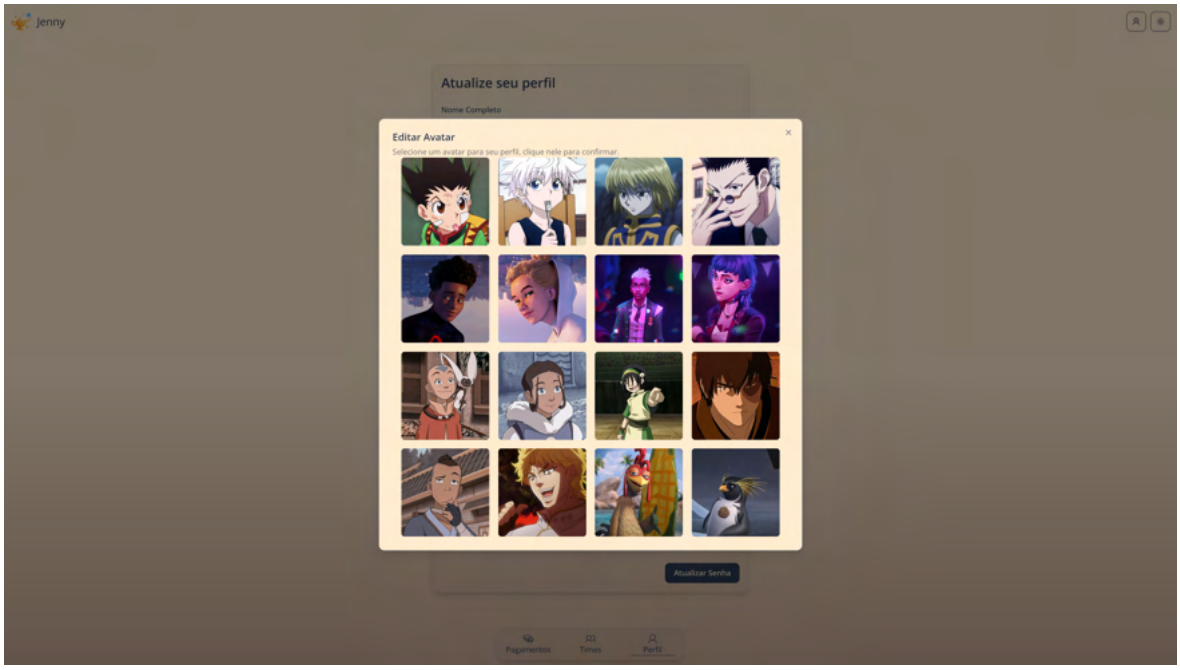

Figura 20: Modal de seleção de avatar da página de perfil em desktop no tema claro, usado para escolher a foto de perfil do usuário.

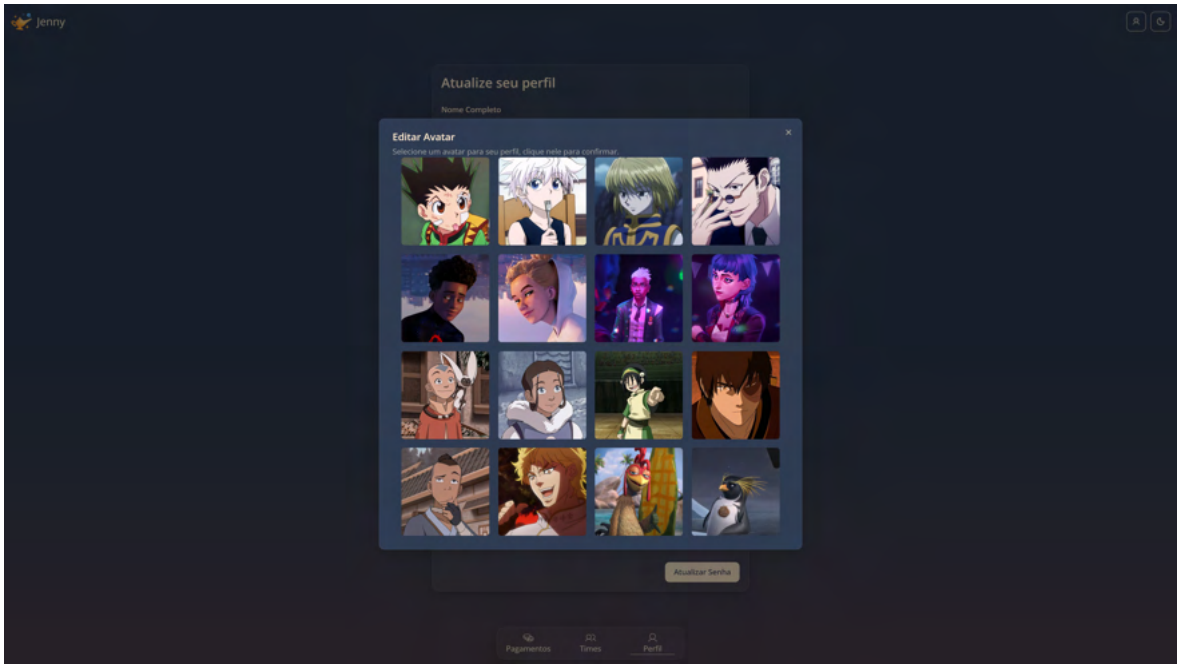

Figura 21: Modal de seleção de avatar da página de perfil em desktop no tema escuro, usado para escolher a foto de perfil do usuário.

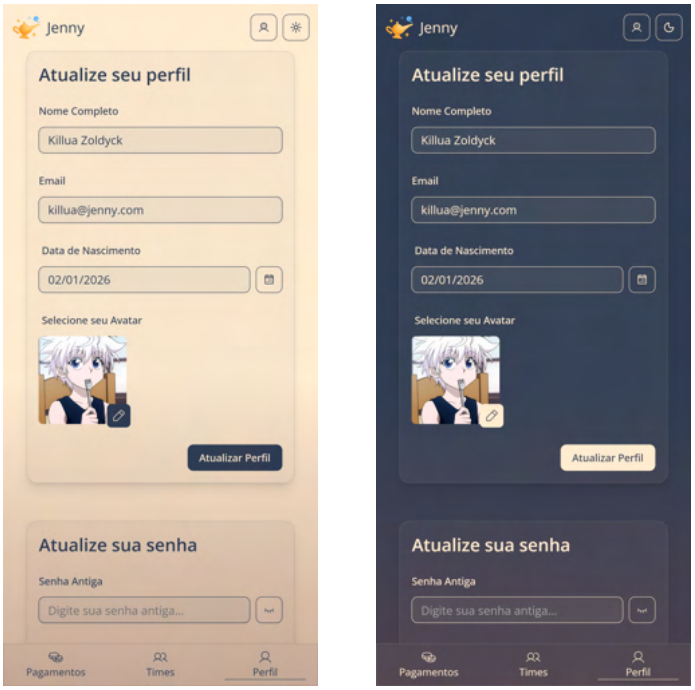

Figura 22: Página de perfil em dispositivo móvel, nos temas claro (à esquerda) e escuro (à direita).

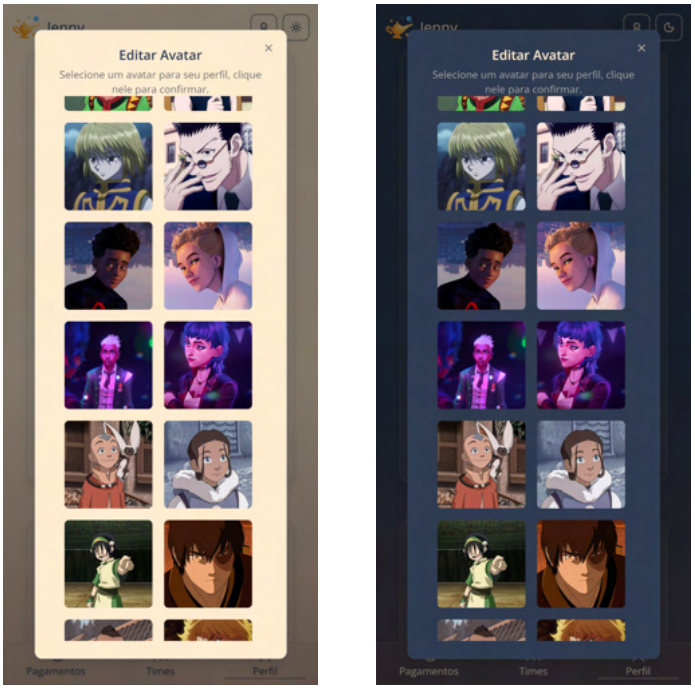

Figura 23: Modal de seleção de avatar da página de perfil em dispositivo móvel, nos temas claro (à esquerda) e escuro (à direita).

8.5 Página de Times

A página de times é primariamente composta por uma lista de cards onde estão exibidos os times de que o usuário participa; cada um deles permite ações como edição e exclusão.

A tela também permite a criação de times.

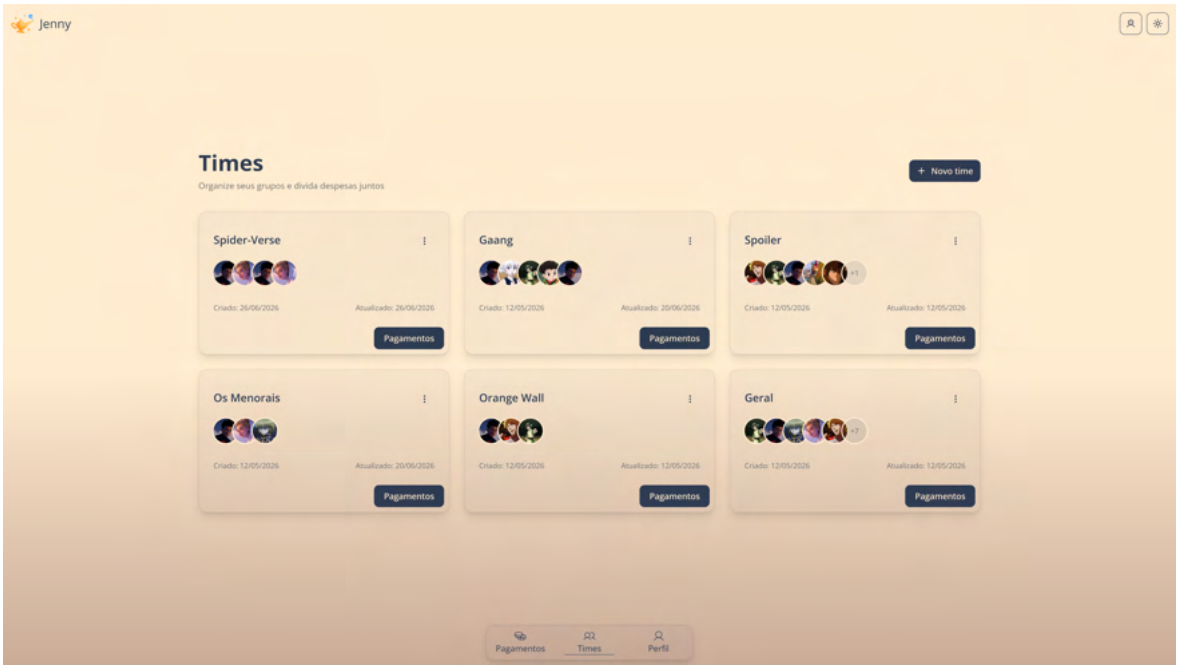

Figura 24: Página de times em desktop no tema claro, exibindo em cards os times de que o usuário participa e permitindo criar, editar e excluir times.

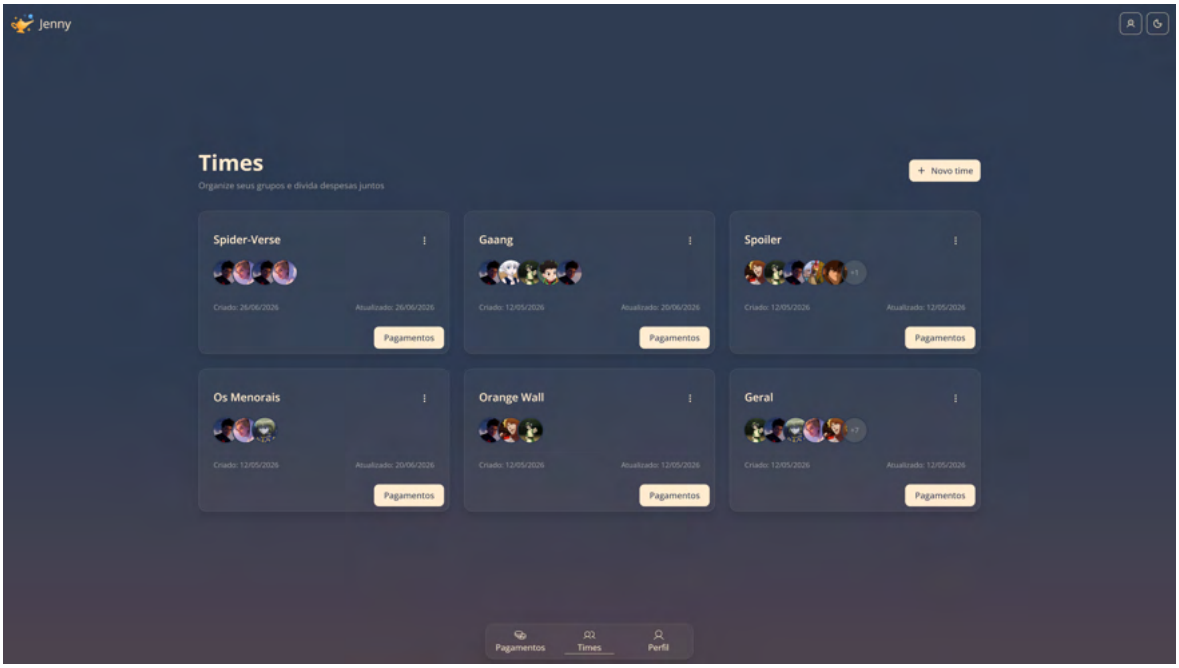

Figura 25: Página de times em desktop no tema escuro, exibindo em cards os times de que o usuário participa e permitindo criar, editar e excluir times.

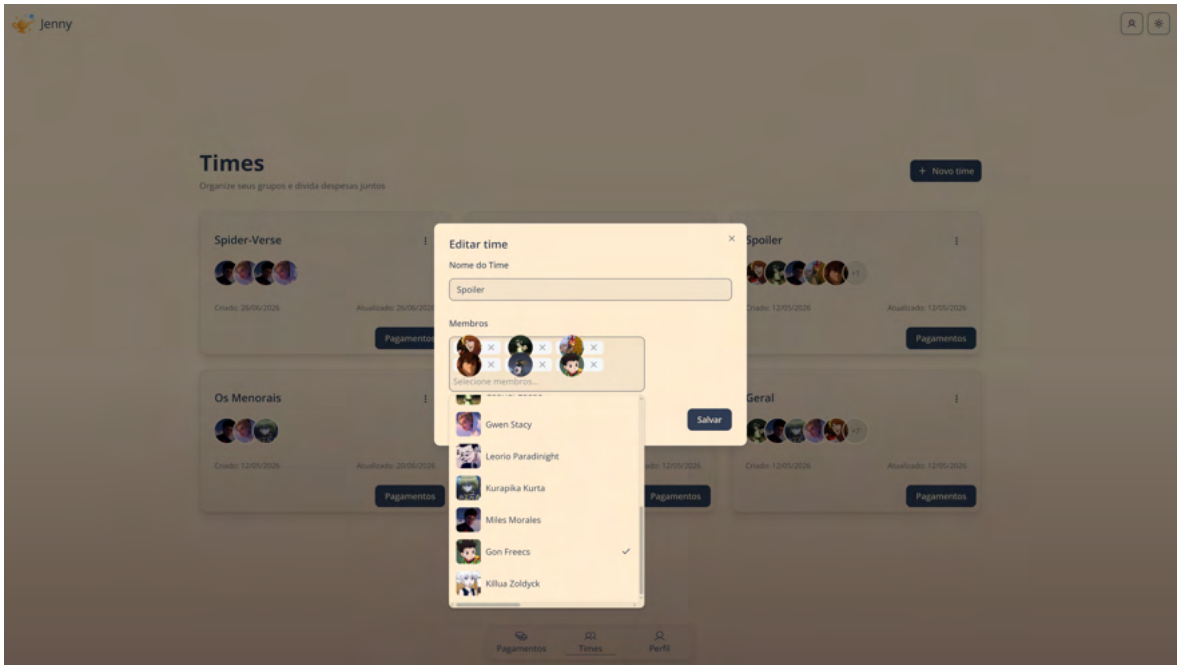

Figura 26: Edição de um time na página de times em desktop no tema claro, com o formulário para alterar o nome e os membros do time.

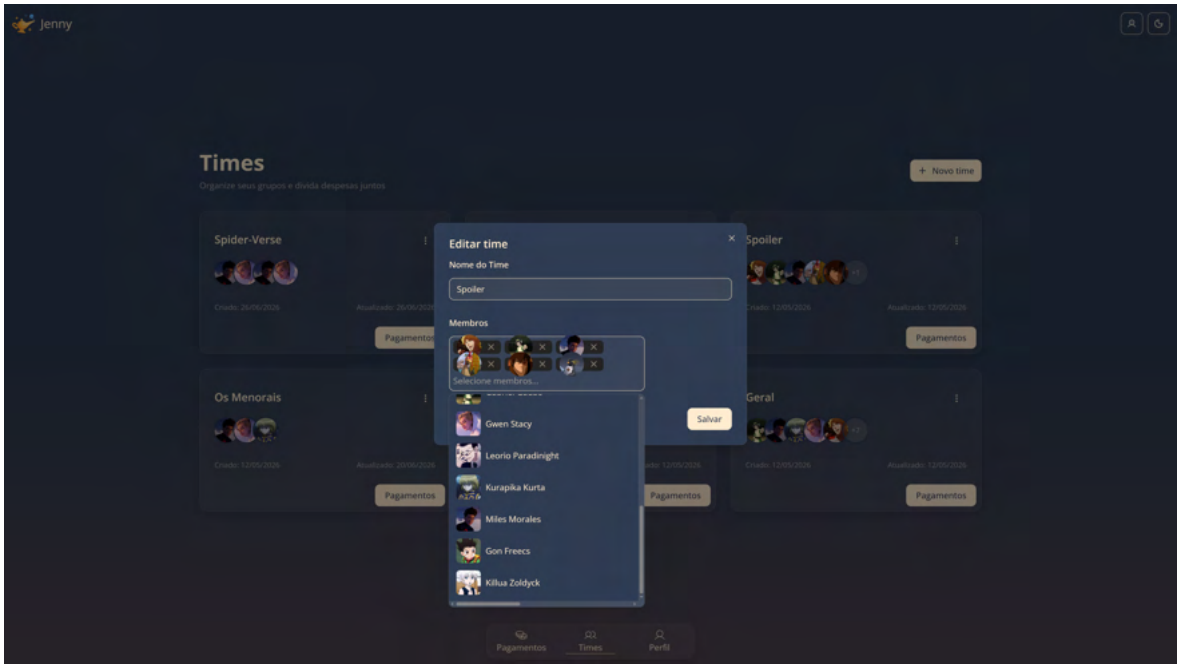

Figura 27: Edição de um time na página de times em desktop no tema escuro, com o formulário para alterar o nome e os membros do time.

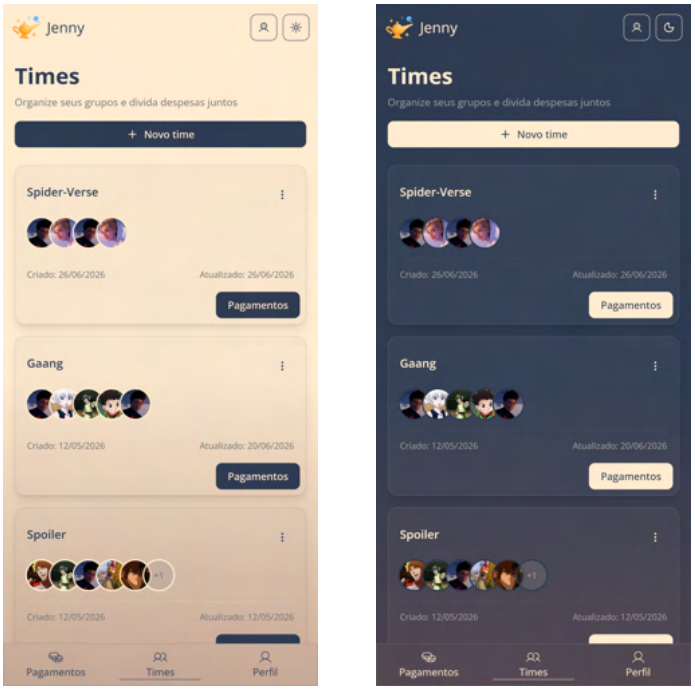

Figura 28: Página de times em dispositivo móvel, nos temas claro (à esquerda) e escuro (à direita).

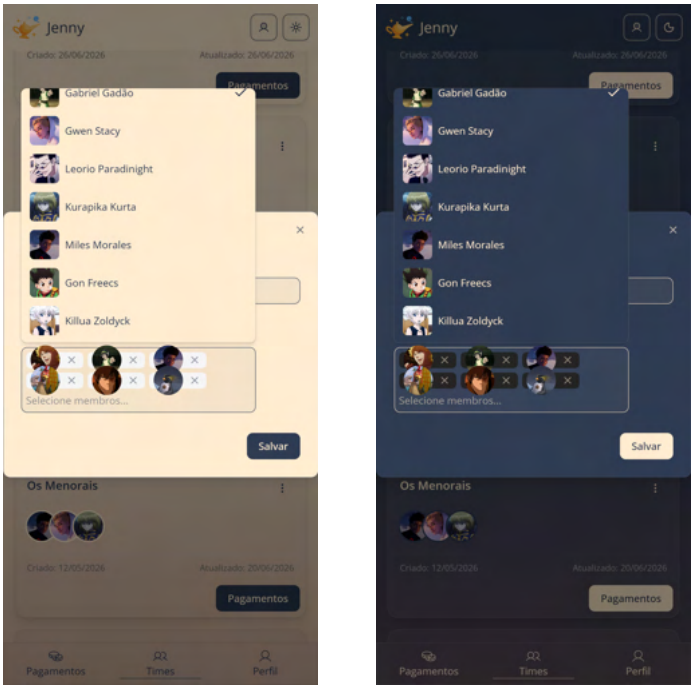

Figura 29: Edição de um time na página de times em dispositivo móvel, nos temas claro (à esquerda) e escuro (à direita).

8.6 Página de Pagamentos de Time

A página de pagamentos é primariamente composta por uma lista de itens onde estão exibidos os pagamentos ocorridos no time no presente mês; cada um deles permite

ações como edição e exclusão. A tela também permite o cadastro de novos pagamentos. Ademais, nela é exibida a lista de dívidas já simplificada.

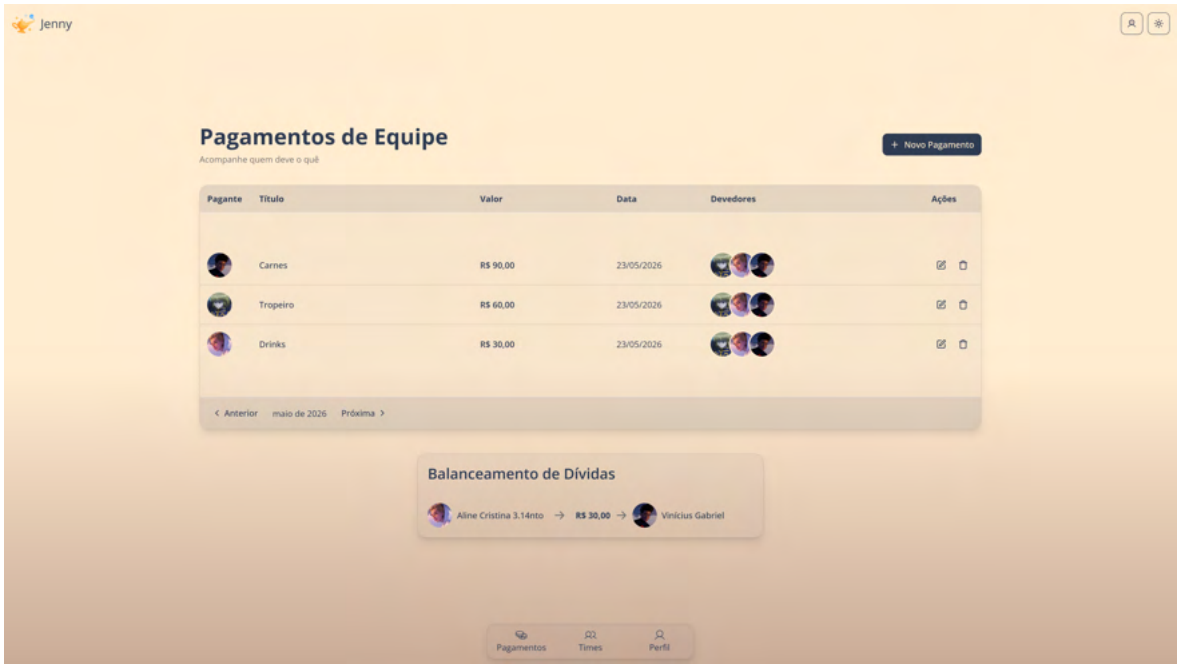

Figura 30: Página de pagamentos de um time em desktop no tema claro, com a lista de despesas do mês e o resumo das dívidas já simplificadas entre os membros.

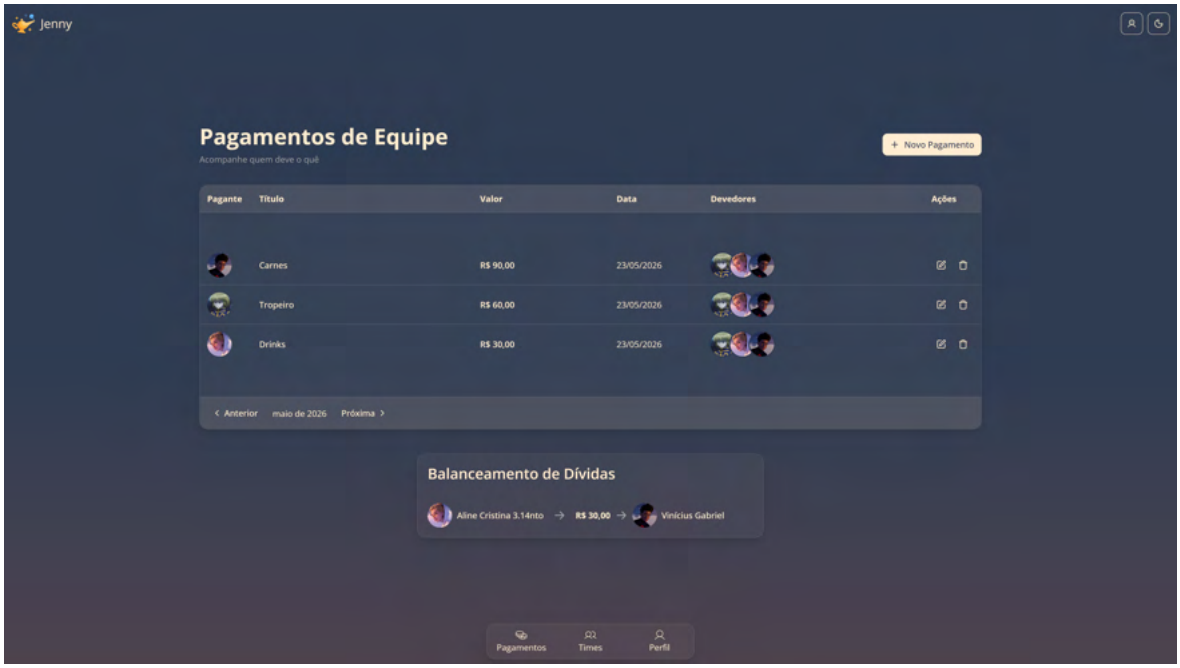

Figura 31: Página de pagamentos de um time em desktop no tema escuro, com a lista de despesas do mês e o resumo das dívidas já simplificadas entre os membros.

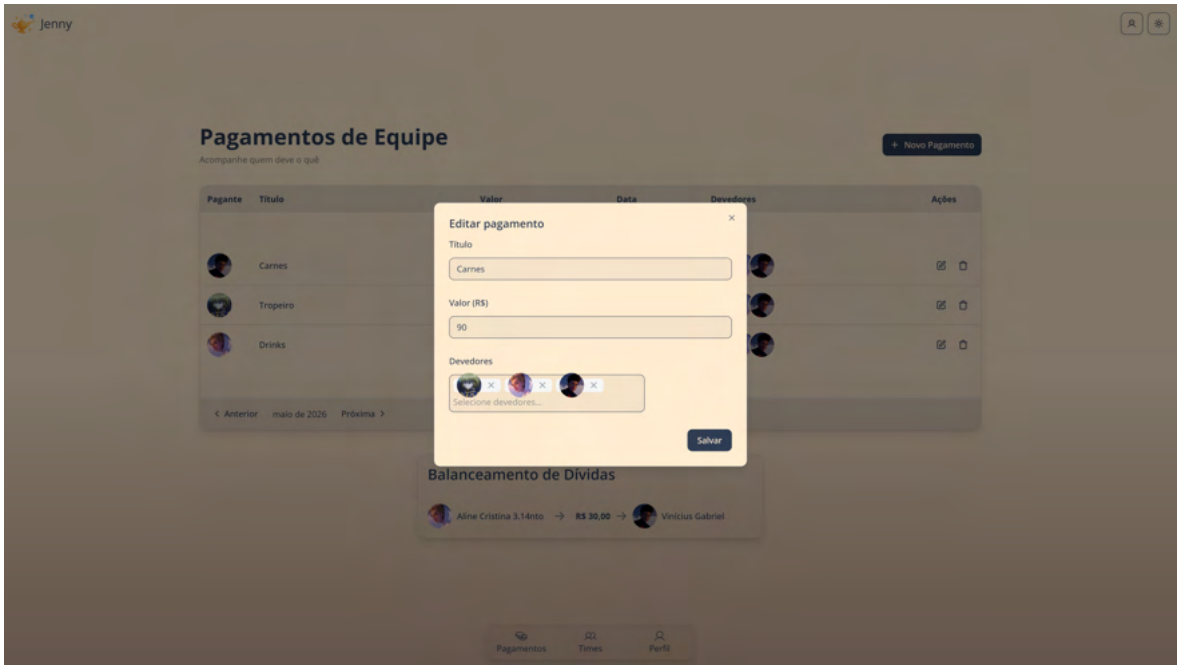

Figura 32: Edição de um pagamento na página de pagamentos em desktop no tema claro, com o formulário para alterar descrição, valor e participantes da despesa.

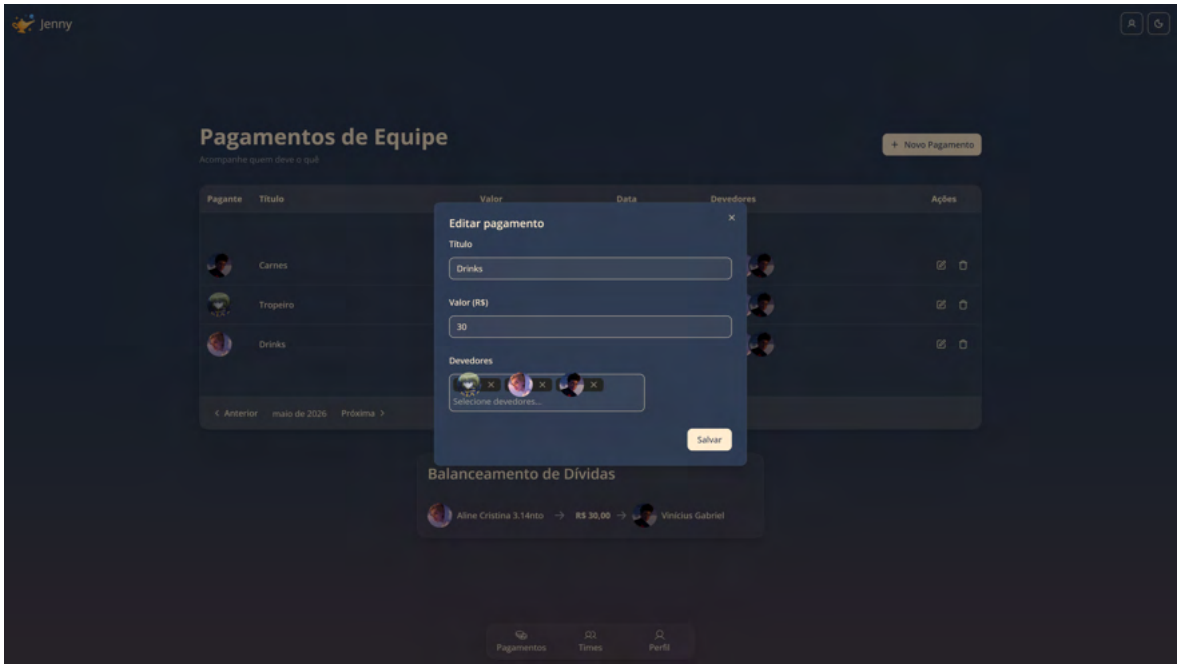

Figura 33: Edição de um pagamento na página de pagamentos em desktop no tema escuro, com o formulário para alterar descrição, valor e participantes da despesa.

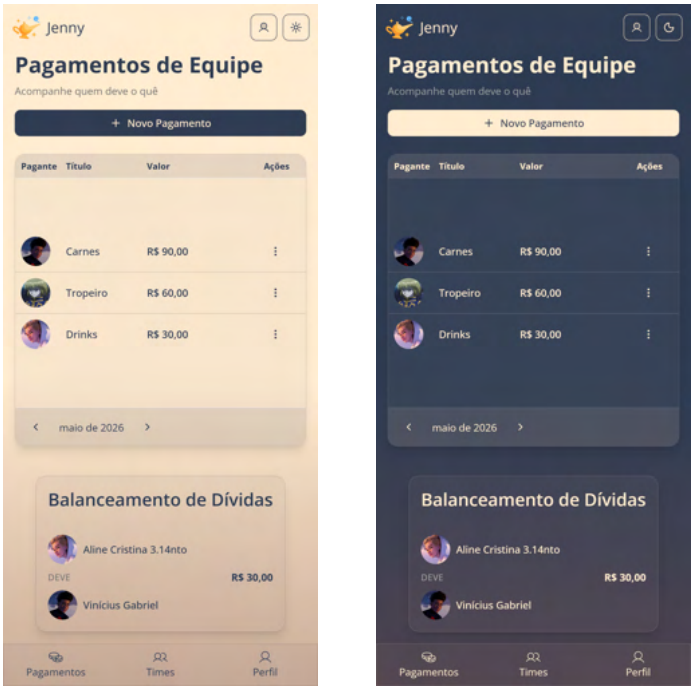

Figura 34: Página de pagamentos de um time em dispositivo móvel, nos temas claro (à esquerda) e escuro (à direita).

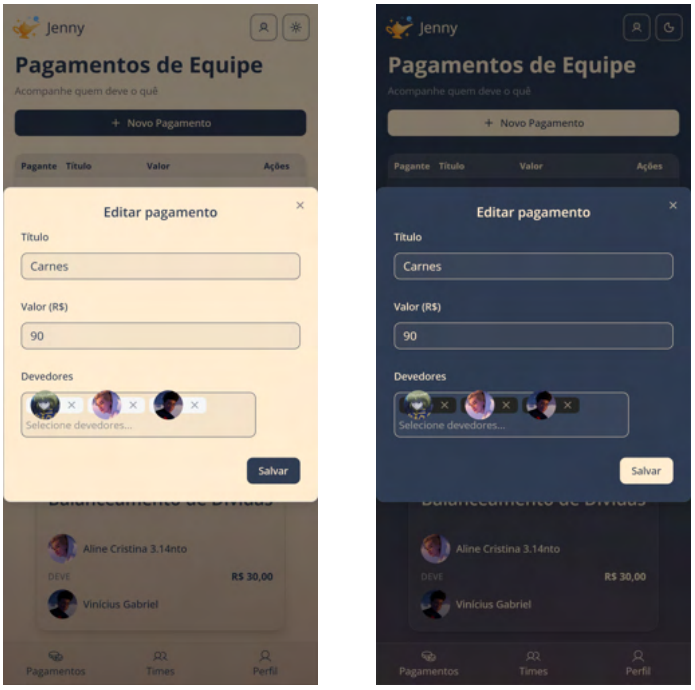

Figura 35: Edição de um pagamento na página de pagamentos em dispositivo móvel, nos temas claro (à esquerda) e escuro (à direita).

8.7 Artefatos de Código

Os códigos do projeto, bem como o histórico de edição, estão hospedados no GitHub (GITHUB, 2026c). O frontend está no seguinte repositório: `https://github.com`

`/viniciusgabrielsf/jenny-web`. O backend está no seguinte repositório `https://github.com/viniciusgabrielsf/jenny-api`. Eles também podem ser acessados a partir dos seguintes QR Codes.

Figura 36: QR Code de acesso ao repositório do frontend (jenny-web) hospedado no GitHub.

Figura 37: QR Code de acesso ao repositório do backend (jenny-api) hospedado no GitHub.

9 Conclusão

O objetivo proposto para a Monografia em Sistemas de Informação foi alcançado, com a implementação do projeto e de suas features propostas. Além disso, os subprodutos de documentação e código possuem grande valor.

Desta forma, perspectivas de trabalhos futuros incluem a abordagem da simplificação de dívidas sem restrições. Essa abordagem minimizaria completamente a quantidade de transações para qualquer caso, encontrando a solução realmente ótima. Essa abordagem também necessitaria de adição de mecanismos de interface para maximizar a interpretabilidade do algoritmo.

Por fim, os conhecimentos adquiridos ao longo da graduação em Sistemas de Informação foram essenciais para a construção da aplicação. Nela foram relacionados conhecimentos de bancos de dados, engenharia de software e algoritmos, resolvendo problemas complexos reais de forma que gerem valor para usuários.

Referências Bibliográficas

AHUJA, Ravindra K.; MAGNANTI, Thomas L.; ORLIN, James B. **Network Flows: Theory, Algorithms, and Applications**. Englewood Cliffs: Prentice Hall, 1993.

ANTHROPIC. **Agent Skills — Claude Platform Documentation**. Disponível em: <https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview>. Acesso em: 2 jul. 2026.

_____. **Claude**. Disponível em: <https://claude.ai/>. Acesso em: 2 jul. 2026.

ATLASSIAN. **Jira: Issue & Project Tracking Software**. Disponível em: <https://www.atlassian.com/software/jira>. Acesso em: 2 jul. 2026.

BAKAUS, Paul. **Impeccable: The design language that makes your AI harness better at design**. Disponível em: <https://github.com/pbakaus/impeccable>. Acesso em: 2 jul. 2026.

BARTH, Adam. **HTTP State Management Mechanism (RFC 6265)**. [S.l.]: Internet Engineering Task Force, 2011. Disponível em: <https://www.rfc-editor.org/rfc/rfc6265>. Acesso em: 2 jul. 2026.

BECK, Kent et al. **Manifesto for Agile Software Development**. 2001. Disponível em: <https://agilemanifesto.org/>. Acesso em: 2 jul. 2026.

BUSACKER, Robert G.; GOWEN, Paul J. **A Procedure for Determining a Family of Minimum-Cost Network Flow Patterns**. Bethesda: Operations Research Office, The Johns Hopkins University, 1960. Technical Paper 15.

COHN, Mike. **Succeeding with Agile: Software Development Using Scrum**. Boston: Addison-Wesley, 2009.

CORMEN, Thomas H. et al. **Introduction to Algorithms**. 3. ed. Cambridge: MIT Press, 2009.

DIJKSTRA, Edsger Wybe. A note on two problems in connexion with graphs. **Numerische Mathematik**, v. 1, p. 269–271, 1959.

DOCKER. **Docker: Accelerated Container Application Development**. Disponível em: <https://www.docker.com/>. Acesso em: 2 jul. 2026.

DODDS, Kent C. **The Testing Trophy and Testing Classifications**. 2021. Disponível em: <https://kentcdodds.com/blog/the-testing-trophy-and-testing-classifications>. Acesso em: 2 jul. 2026.

EDMONDS, Jack; KARP, Richard M. Theoretical improvements in algorithmic efficiency for network flow problems. **Journal of the ACM**, v. 19, n. 2, p. 248–264, 1972.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de Banco de Dados**. 7. ed. São Paulo: Pearson, 2019.

EXPRESS. **Express: Fast, unopinionated, minimalist web framework for Node.js**. Disponível em: <https://expressjs.com/>. Acesso em: 2 jul. 2026.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. 2000. Tese (Doutorado) – University of California, Irvine. Disponível em: <https://ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Acesso em: 2 jul. 2026.

FIELDING, Roy Thomas; NOTTINGHAM, Mark; RESCHKE, Julian. **HTTP Semantics (RFC 9110)**. [S.l.]: Internet Engineering Task Force, 2022. Disponível em: <https://www.rfc-editor.org/rfc/rfc9110>. Acesso em: 2 jul. 2026.

FOWLER, Martin. **Refatoração: Aperfeiçoando o Projeto de Código Existente**. Porto Alegre: Bookman, 2004.

GAMMA, Erich et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Boston: Addison-Wesley, 1994.

GITHUB. **Adding repository custom instructions for GitHub Copilot**. Disponível em: <https://docs.github.com/copilot/customizing-copilot/adding-custom-instructions-for-github-copilot>. Acesso em: 2 jul. 2026.

_____. **GitHub Copilot**. Disponível em: <https://github.com/features/copilot>. Acesso em: 2 jul. 2026.

_____. **GitHub: Where the world builds software**. Disponível em: <https://github.com/>. Acesso em: 2 jul. 2026.

GOOGLE. **Gemini**. Disponível em: <https://gemini.google.com/>. Acesso em: 2 jul. 2026.

JEST. **Jest: Delightful JavaScript Testing**. Disponível em: <https://jestjs.io/>. Acesso em: 2 jul. 2026.

JOHNSON, Donald B. Efficient algorithms for shortest paths in sparse networks. **Journal of the ACM**, v. 24, n. 1, p. 1–13, 1977.

JONES, Michael; BRADLEY, John; SAKIMURA, Nat. **JSON Web Token (JWT) (RFC 7519)**. [S.l.]: Internet Engineering Task Force, 2015. Disponível em: <https://www.rfc-editor.org/rfc/rfc7519>. Acesso em: 2 jul. 2026.

KLEINBERG, Jon; TARDOS, Éva. **Algorithm Design**. Boston: Pearson/Addison-Wesley, 2006.

KRASNER, Glenn E.; POPE, Stephen T. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. **Journal of Object-Oriented Programming**, v. 1, n. 3, p. 26–49, 1988.

MARTIN, Robert C. **Agile Software Development: Principles, Patterns, and Practices**. Upper Saddle River: Prentice Hall, 2002.

_____. **Clean Code: A Handbook of Agile Software Craftsmanship**. Upper Saddle River: Prentice Hall, 2008.

META PLATFORMS. **React: The library for web and native user interfaces**. Disponível em: <https://react.dev/>. Acesso em: 2 jul. 2026.

MOBILLS. **Mobills: App de controle financeiro**. Disponível em: <https://www.mobills.com.br/>. Acesso em: 2 jul. 2026.

NOTION LABS. **Notion: The AI workspace that works for you**. Disponível em: <https://www.notion.com/>. Acesso em: 2 jul. 2026.

OPENJS FOUNDATION. **Node.js**. Disponível em: <https://nodejs.org/>. Acesso em: 2 jul. 2026.

ORGANIZZE. **Organizze: Gerenciador financeiro**. Disponível em: <https://www.organizze.com.br/>. Acesso em: 2 jul. 2026.

OWASP FOUNDATION. **Cross Site Scripting (XSS)**. Disponível em: <https://owasp.org/www-community/attacks/xss/>. Acesso em: 2 jul. 2026.

POSTGRESQL GLOBAL DEVELOPMENT GROUP. **PostgreSQL: The World's Most Advanced Open Source Relational Database**. Disponível em: <https://www.postgresql.org/>. Acesso em: 2 jul. 2026.

SCHWABER, Ken. **Agile Project Management with Scrum**. Redmond: Microsoft Press, 2004.

SCHWABER, Ken; SUTHERLAND, Jeff. **The 2020 Scrum Guide**. 2020. Disponível em: <https://scrumguides.org/scrum-guide.html>. Acesso em: 2 jul. 2026.

SEQUELIZE. **Sequelize: Feature-rich ORM for modern Node.js and TypeScript**. Disponível em: <https://sequelize.org/>. Acesso em: 2 jul. 2026.

SERASA. **55% dos jovens da Geração Z são os principais responsáveis pelas suas próprias finanças, revela Serasa**. Pesquisa realizada pelo Instituto Opinion Box com 2.923 jovens de 18 a 29 anos, em julho de 2024. 2025. Disponível em: <htt

ps://www.serasa.com.br/imprensa/jovens-da-geracao-z-sao-os-principais-responsaveis-pelas-suas-proprias-financas-revela-serasa/>. Acesso em: 2 jul. 2026.

SERASA. **Geração Z é o público que mais cresce nas negociações de dívidas, revela Serasa**. 2025. Disponível em: <https://www.serasa.com.br/imprensa/geracao-z-e-o-publico-que-mais-cresce-nas-negociacoes-de-dividas/>. Acesso em: 2 jul. 2026.

SHADCN. **shadcn/ui: Build your component library**. Disponível em: <https://ui.shadcn.com/>. Acesso em: 2 jul. 2026.

SOMMERVILLE, Ian. **Engenharia de Software**. 10. ed. São Paulo: Pearson Education do Brasil, 2019.

SPLITWISE. **Splitwise: Split expenses with friends**. Disponível em: <https://www.splitwise.com/>. Acesso em: 2 jul. 2026.

TOMIZAWA, Nobuaki. On some techniques useful for solution of transportation network problems. **Networks**, v. 1, n. 2, p. 173–194, 1971.

VITE. **Vite: Next Generation Frontend Tooling**. Disponível em: <https://vite.dev/>. Acesso em: 2 jul. 2026.

VITEST. **Vitest: Next Generation testing framework**. Disponível em: <https://vitest.dev/>. Acesso em: 2 jul. 2026.

VOCKE, Ham. **The Practical Test Pyramid**. 2018. Disponível em: <https://martinfowler.com/articles/practical-test-pyramid.html>. Acesso em: 2 jul. 2026.

W3C. **Web Content Accessibility Guidelines (WCAG) 2.1**. 2018. Disponível em: <https://www.w3.org/TR/WCAG21/>. Acesso em: 2 jul. 2026.

_____. **Web Content Accessibility Guidelines (WCAG) 2.2**. 2023. Disponível em: <https://www.w3.org/TR/WCAG22/>. Acesso em: 2 jul. 2026.