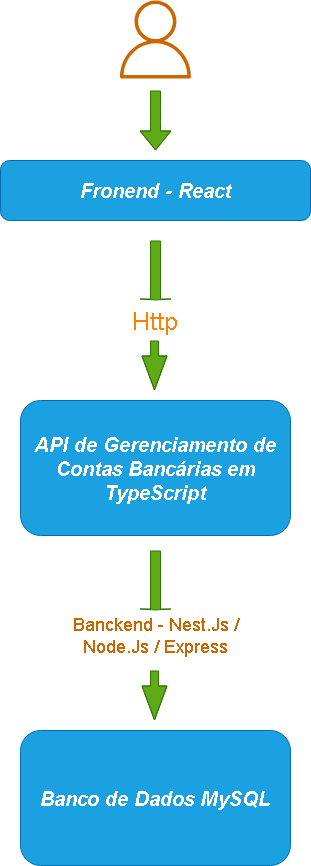
1. **Aplicação Bancária Full Stack para Gerenciamento de Contas e Transferências:**
   1. Banking application project - Gestão de Contas e Transferências
   2. Aplicação para cadastro e gerenciamento de contas bancárias, que permite o registro de usuários, criação de contas bancárias, realização de transferências entre contas e verificação do histórico de movimentações. Desenvolvida com TypeScript, utilizando Node.js e Nest.js no backend, e React.js no frontend.
2. **Tecnologias utilizadas:**
   1. Principais tecnologias e frameworks:
      1. **Backend:** Node.Js, Nest.Js, TypeScript
      2. **Frontend:** Next.Js, Axios, TypeScript
      3. **Banco de dados:** MySQL
      4. **Outras ferramentas:** Jest, Cypress, Git, Postman, Br Modelo Web (modelagem de banco de dados).
3. **Arquitetura da aplicação:**
   1. Arquitetura Backend — Nest.Js com TypeScript (Arquitetura Modular)
      1. Arquitetura escolhida por ser escalável, com baixo acoplamento e uma organização clara de código. Essa estrutura está alinhada aos princípios SOLID, especialmente o de responsabilidade única, além de aproveitar o sistema robusto de injeção de dependência fornecido pelo NestJS. Essa abordagem facilita a manutenção, os testes e a expansão da aplicação conforme novas funcionalidades são implementadas.
   2. Arquitetura Frontend — Next.Js com TypeScript (Arquitetura Modular + App Router)
      1. Arquitetura adotada por garantir uma separação clara de responsabilidades, promover a reutilização de componentes e minimizar a repetição de código. O uso do App Router permite um roteamento mais dinâmico e moderno, com suporte a Server Components e layouts aninhados. A estrutura modular facilita a escalabilidade da aplicação e conta com suporte nativo a TypeScript, trazendo mais segurança e produtividade ao desenvolvimento.
   3. A comunicação entre o frontend e o backend ocorre por meio de requisições HTTP utilizando a biblioteca Axios.
   4. O frontend e o backend são aplicações separadas, que se comunicam através da API exposta pelo backend.



1. **Instruções para rodar o projeto (setup):**
   1. **Passo a passo para rodar localmente:**
      1. Clone os repositórios do **backend** e do **frontend** no GitHub:
         1. Bankend: <https://github.com/Carloshpjacinto/banking_application_project>
         2. Frontend: <https://github.com/Carloshpjacinto/banking_application_project_ui>
      2. Acesse as pastas do **backend** e do **frontend**, e instale as dependências com o seguinte comando:
         1. npm install
      3. Crie as variaveis de ambiente no arquivo .env nas duas aplicações conforme o ambiente local (por exemplo: porta, banco de dados, JWT secret). O arquivo .env.example mostra os nomes utilizados nas variáveis do projeto.
      4. Executar os servidores:
         1. Backend: npm run start:dev
         2. Frontend: npm run dev
            1. É possível executar a aplicação com um mock server utilizando o comando: npm run dev:mock

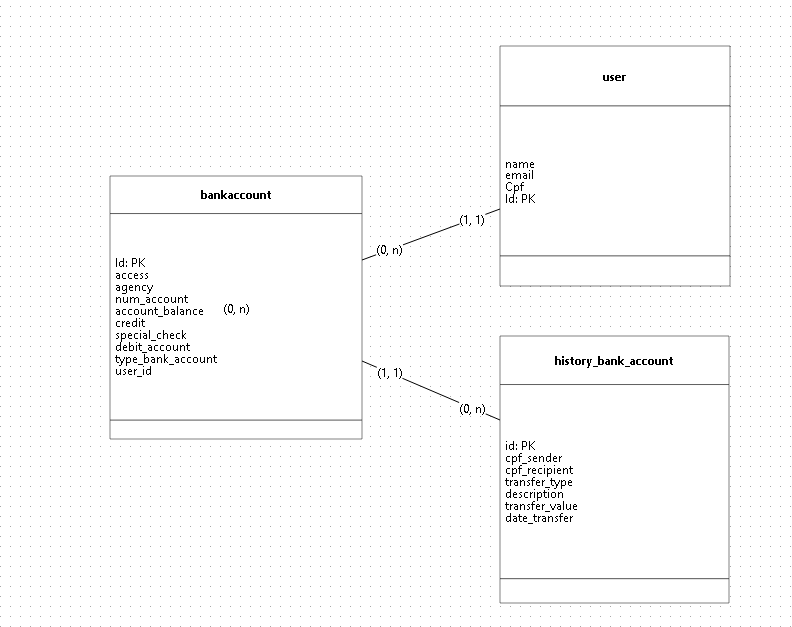
Atenção às portas da aplicação e do mock server, pois o Next.js altera automaticamente a porta da aplicação em caso de conflito.

1. **Funcionalidades principais:**
   1. Abaixo estão listadas as principais funcionalidades implementadas na aplicação**:**
      1. **Cadastro de usuário e criação de conta bancária:**  
          Permite o registro de novos usuários e a geração automática de uma conta bancária vinculada.
      2. **Login de usuário:**  
          Autenticação do usuário com verificação de credenciais para acesso à conta bancária.
      3. **Autenticação via JWT:**  
          Utilização de JSON Web Tokens para proteger rotas e garantir o acesso seguro às funcionalidades da aplicação.
      4. **Dashboard do usuário:**  
          Exibição de informações da conta bancária, como saldo, movimentações e opções de operação.
      5. **Depósito de valores na conta:**  
          Permite realizar depósitos para habilitar transferências.
      6. **Transferência de valores entre contas:**  
          Funcionalidade que possibilita a movimentação de saldo entre contas cadastradas no sistema.
2. **Principais Endpoints da API:**

|  |  |  |
| --- | --- | --- |
| **Método** | **Rota** | **Descrição** |
| POST | /auth/register | Registra os dados básicos de um novo usuário. |
| POST | /auth/register/access | Registra a senha de acesso do usuário e gera os dados da conta bancária. |
| POST | /auth/login | Autentica o usuário e retorna o token de acesso (JWT). |
| GET | /auth/profile | Retorna os dados do usuário autenticado. |
| POST | /auth/transfer | Realiza a transferência de valores entre contas cadastradas. |
| GET | /auth/bankaccounthistory | Realiza a busca do histórico da conta bancaria, pode ser filtrado por tipos de descrição. SENT (enviada), RECEIVED (recebida), DEPOSIT(deposito). |
| GET | /bankaccount/{id} | Busca e retorna a conta bancária do usuário. |

Todos os endpoints (exceto registro e login) requerem autenticação via token JWT no header da requisição.

1. **Estrutura do banco de dados:**
   1. **Diagrama do banco de dados:**



* 1. **Tabela user:**
     1. **Id** -> Número de identificação única do registro do usuário (chave primária.
     2. **name** -> nome do usuário, esse campo é do tipo string.
     3. **email** -> e-mail do usuário para envio de confirmação de transferência, campo do tipo string.
     4. **cpf** -> CPF do usuário para realização de transferências entre contas, campo do tipo string
  2. **Tabela bankAccount:**
     1. **id** → Número de identificação única da conta bancária (chave primária).
     2. **access** → Código de acesso da conta bancária, do tipo string.
     3. **agency** → Número da agência bancária, campo do tipo string.
     4. **num\_account** → Número da conta bancária, do tipo string.
     5. **account\_balance** → Valor disponível na conta para transferências por débito.
     6. **credit** → Valor disponível para crédito, do tipo Decimal.
     7. **special\_check** → Limite do cheque especial da conta, do tipo Decimal.
     8. **Debit\_account** → Valor total de débito da conta, do tipo Decimal.
     9. **type\_bank\_account** → Tipo da conta bancária (ex: corrente, poupança), do tipo enum.
     10. **user\_id** → Chave estrangeira que referência o usuário dono da conta (id da tabela User), do tipo number.
  3. **Tabela bankAccountHistory:**
     1. **id** → Identificador único do registro de histórico de transações (chave primária), do tipo number.
     2. **cpf\_sender** → CPF do remetente que realizou a transferência, do tipo string.
     3. **cpf\_recipient** → CPF do destinatário que recebeu a transferência, do tipo string.
     4. **transfer\_type** → Tipo da transferência realizada (por exemplo: PIX\_TRANSFER ou DEPOSIT), do tipo enum.
     5. **description**→ Situação da transferência (ex: “SENT”, “RECEIVED”, “DEPOSIT”), do tipo string ou enum.
     6. **transfer\_value** → Valor monetário transferido, do tipo decimal.
     7. **date\_transfer** → Data e hora em que a transferência foi realizada, do tipo datetime.

1. **Como rodar teste**
   1. Os testes automatizados do backend foram feitos com **Jest** por ser uma ferramenta rápida, simples de configurar e amplamente usada na comunidade JavaScript/TypeScript. Para executá-los com cobertura de código, utilize o comando: npm run test:cov
2. **Possíveis melhorias**
   1. **Refatoração da estrutura de services relacionados a type-of-transfer:**  
       É possível melhorar a organização do código criando uma função ou classe única responsável por receber e tratar os dados relacionados às transferências e seus respectivos tipos. Essa centralização facilita a manutenção, evita duplicação de lógica e melhora a escalabilidade do serviço.
   2. **Substituição do ORM atual pelo Prisma:**  
       A adoção do Prisma em vez do uso direto do Express para interações com o banco de dados pode simplificar a manipulação de dados. O Prisma oferece uma camada de abstração mais segura, com tipagem forte, validação automática e suporte a migrations, o que acelera o desenvolvimento e reduz riscos de inconsistência na base de dados.
   3. **Melhor estrutura para tratamento de erros:** Atualmente, o tratamento de exceções pode ser simplificado ou inconsistente entre os módulos. A implementação de um middleware global de erros (como um ExceptionFilter customizado no NestJS) ajudaria a capturar e padronizar mensagens de erro, além de facilitar a manutenção e o debug.
   4. **Validações mais robustas (ex: usuários e contas bancárias):** A lógica de verificação para impedir duplicações — como cadastro de usuários já existentes ou criação de contas bancárias repetidas — pode ser melhor encapsulada em validações customizadas nos DTOs ou services. Isso ajuda a prevenir inconsistências nos dados e melhora a experiência do usuário.
   5. **Atualização de dados do usuário:** Atualmente, os dados do usuário como nome e e-mail não podem ser atualizados após o cadastro. Uma rota autenticada (PATCH /users) pode ser implementada para permitir essa atualização de forma segura, com as devidas validações.
   6. **Responsividade de páginas:**  Melhorias na responsividade: algumas páginas já se comportam bem na maioria dos tamanhos de tela, mas ainda apresentam pequenos problemas em resoluções específicas que podem ser ajustados.
   7. **Verificação de dados enviados:** A aplicação, a princípio, foca nas funcionalidades de cadastro e gerenciamento de contas bancárias, envio de transferências monetárias, consulta de histórico e filtragem por tipo de movimentação. Portanto, a implementação de validações como limites de valor para transferências, verificação de CPF, e-mail e nome de usuário se enquadra como débito técnico, devendo ser analisada e corrigida futuramente.