Descrição Geral

Esta documentação detalha a aplicação desenvolvida para extrair dados de uso de uma página web estática, enviá-los para o *Firebase Realtime Database* e disponibilizar esses dados através de uma API. A aplicação é composta por três partes principais: Página de Teste, Plugin de Extração de Dados e API. A arquitetura segue princípios de Clean Architecture, SOLID e utiliza diversos padrões de projeto.

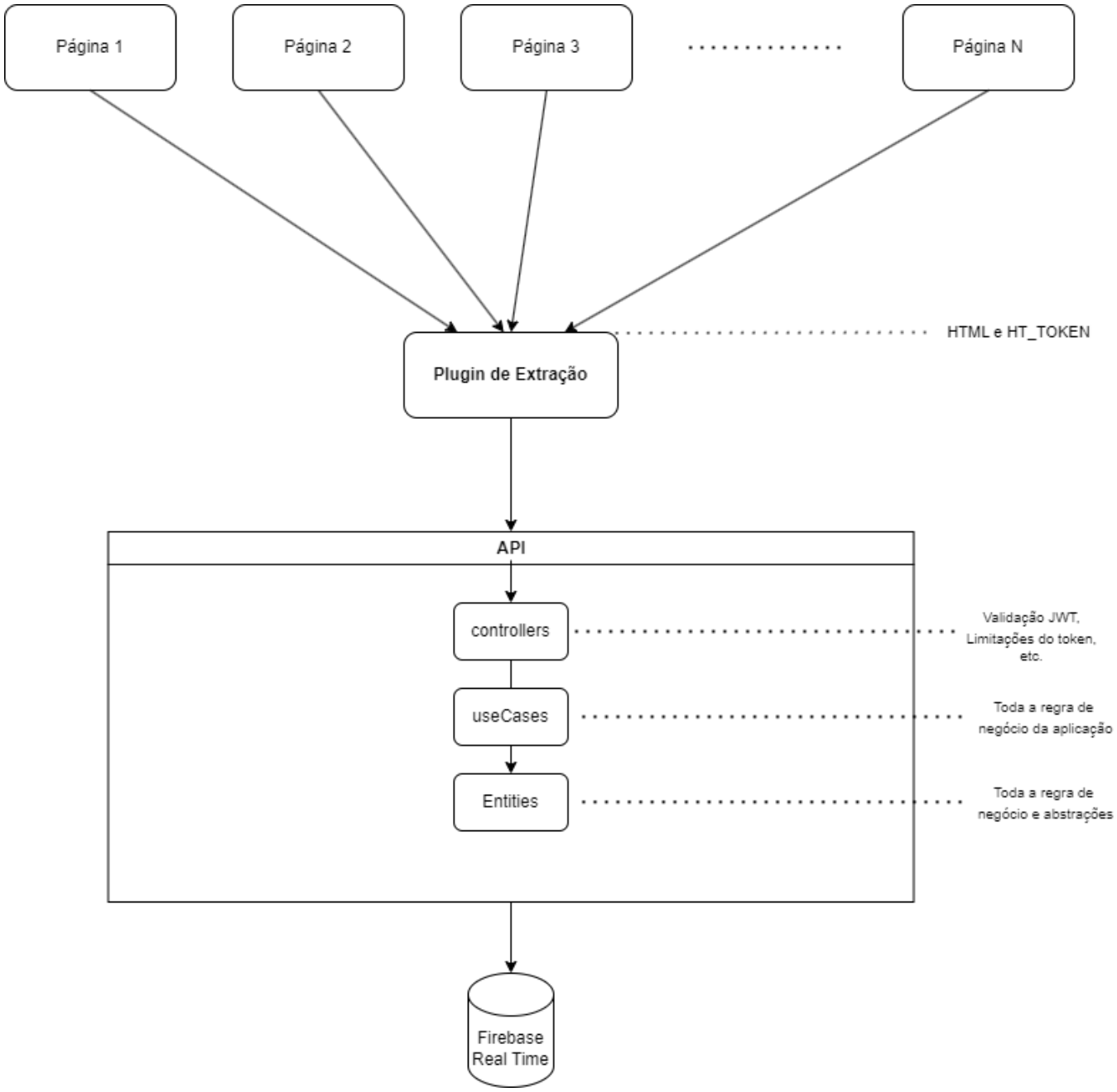
Sumário

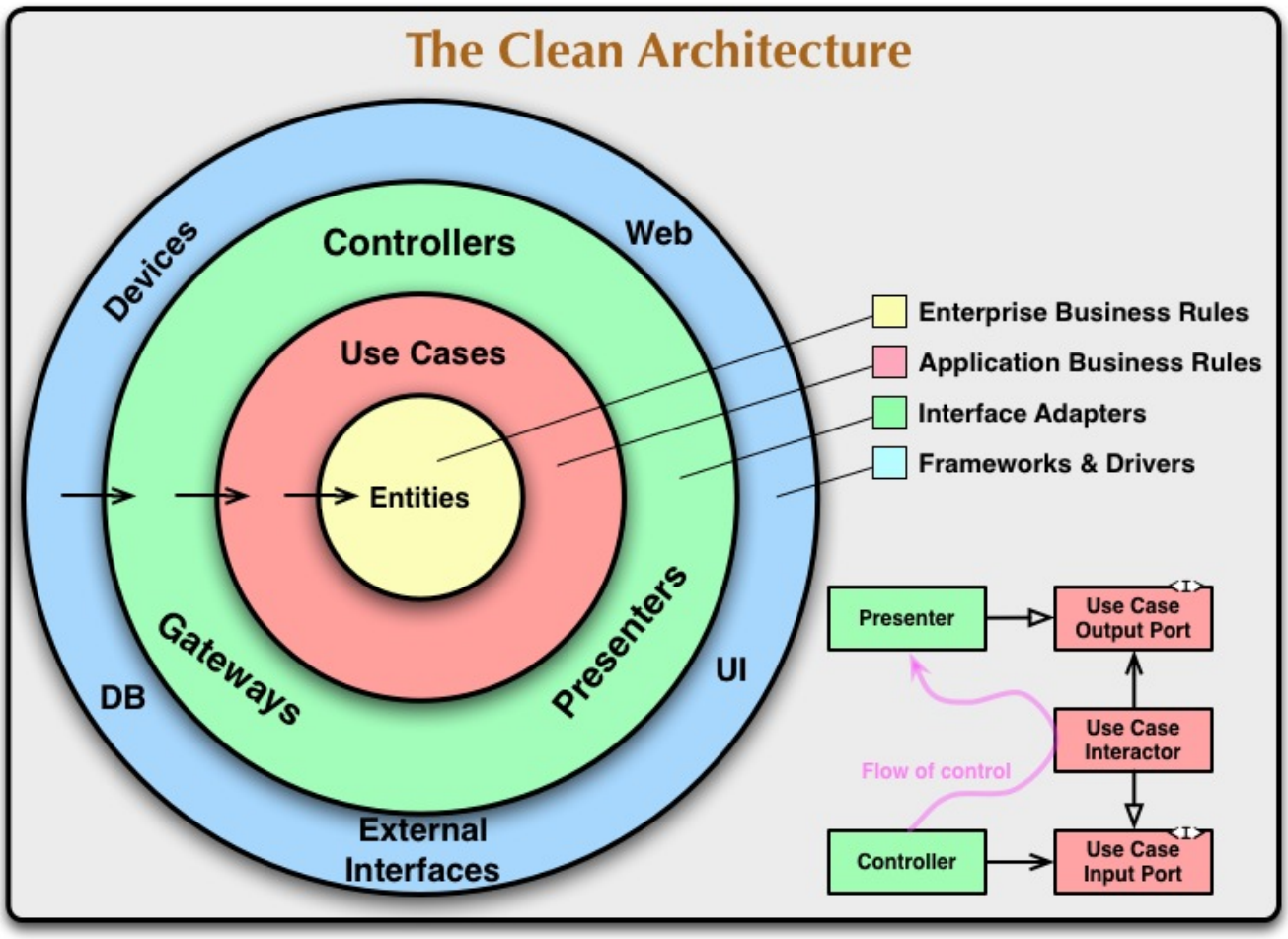
Estrutura do Repositório

A aplicação foi organizada como um monorepo com a seguinte estrutura:

```
Root/
  Page/
  Plugin/
  API/
```

1. Diagrama da Arquitetura


Página 1
Página 2
Página 3
Página N
HTML e HT_TOKEN
Plugin de Extração
API
controllers
Validação JWT, Limitações do token, etc.
useCases
Toda a regra de negócio da aplicação
Entities
Toda a regra de negócio e abstrações
Firebase Real Time



1.1. Explicação da Arquitetura

1.1.1 Introdução

A arquitetura da aplicação é projetada para promover a escalabilidade, modularidade e manutenção. Abaixo está uma descrição das principais camadas e componentes.

1.1.2. Camadas e Componentes

1.1.2.1. Camada de Apresentação

- **Componentes:** Interface do usuário, controladores.
- **Responsabilidade:** Gerenciar a interação com o usuário e exibir os dados.

1.1.2.2. Camada de Lógica de Negócios

- **Componentes:** Serviços, use cases.
- **Responsabilidade:** Processar dados e aplicar regras de negócios.

1.1.2.3. Camada de Dados

- **Componentes:** Repositórios, modelos de dados.
- **Responsabilidade:** Gerenciar o acesso e a persistência dos dados.

1.1.3. Motivações de Escolha

1.1.3.1. Princípios e Padrões Utilizados

- **SOLID:** Garantir que cada módulo tenha uma única responsabilidade, seja aberto para extensão e fechado para modificação, entre outros princípios.
- **Clean Architecture:** Separar o sistema em camadas distintas para promover a independência e testabilidade.

1.1.3.2. Benefícios

- **Escalabilidade:** A arquitetura permite adicionar novas funcionalidades com impacto mínimo.
- **Manutenção:** A separação clara de responsabilidades facilita a manutenção e a evolução do sistema.

1.1.3.3. Decisões Arquiteturais

- **Tecnologias:** A escolha de tecnologias e frameworks foi feita para alinhar com as necessidades da aplicação e os padrões arquiteturais desejados.
- **Estrutura das Camadas:** A estrutura em camadas permite uma clara separação entre a lógica de apresentação, negócios e dados.

1.2. Conclusão

O diagrama e a explicação fornecem uma visão abrangente da arquitetura utilizada na aplicação, destacando as escolhas de design e os benefícios esperados.

2. Como Rodar as Aplicações

Versão recomendada do NodeJS >= 20.0.0

2.1. Página de Teste

Para rodar a Página de Teste, siga os passos abaixo:

2.1.1. **Navegue para o diretório da Página de Teste:**

```
cd Page
```

2.1.2. **Instale as dependências:**

```
npm install
```

2.1.3. **Inicie o servidor de desenvolvimento:**

```
npm run dev
```

2.1.4. **Configuração do Token:**

- **Importante:** Substitua o token no script que já está presente em `/Page/index.html`. Localize o seguinte trecho de código:

```
<script>
  var ht;
  window.addEventListener('load', () => {
    if (window.HT_TOKEN) {
      ht = window.HT_TOKEN.init({
        token: "<SEU TOKEN>"
      });
    }
  });
</script>
```

- **Substitua `<SEU TOKEN>` pelo token válido da API.**

2.2. Plugin de Extração de Dados

Para rodar o Plugin, siga os passos abaixo:

2.2.1. **Navegue para o diretório do Plugin:**

```
cd Plugin
```

2.2.2. **Instale as dependências:**

```
npm install
```

3. **Inicie o servidor de desenvolvimento:**

```
npm run dev
```

2.3. API

Para rodar a API, siga os passos abaixo:

2.3.1. **Navegue para o diretório da API:**

```
cd Api
```

2.3.2. **Instale as dependências:**

```
npm install
```

2.3.3. **Configure o Firebase:**

- **Coloque o arquivo JSON do Firebase** (contendo as credenciais de acesso) em `API/credentials/firebase.json`.

2.3.4. **Configure o JWT Secret:**

- **Crie um arquivo `.env` em `API/.env`** com a seguinte variável:

  ```
  JWT_SECRET=SUA CHAVE SECRETA
  ```

2.3.5. **Inicie o servidor de desenvolvimento:**

```
npm run dev
```

2.4. Rodar os Testes

Para rodar os testes em qualquer uma das aplicações, siga os passos abaixo:

2.4.1. **Navegue para o diretório da aplicação desejada (Page, Plugin ou API):**

```
cd [diretório_da_aplicação]
```

2.4.2. **Execute os testes:**

```
npm test
```

Certifique-se de substituir `[diretório_da_aplicação]` pelo diretório correspondente (Page, Plugin, ou API) ao executar os testes.

3. Detalhamento: Página de Teste

3.1. Descrição

A Página de Teste fornece um ambiente para testar a instalação e funcionamento do Plugin de Extração de Dados. A página foi desenvolvida usando NodeJS, TypeScript e React, e segue os padrões de acessibilidade

WCAG.

3.2. Funcionalidades

- Alternância entre temas Dark Mode e Light Mode
- Validação de código da W3C
- Acessibilidade
- Testes unitários

3.3. Estrutura do Diretório

```
/Page
  /cypress
  /public
  /src
    /assets
      /Images
    /components
      /Button
      /Container
      /Footer
      /Header
      /ProfilePhoto
      /Section
      /SwitchThemeMode
    /containers
      /AboutContainer
      /HomeContainer
      /PostsContainer
      /ProjectsContainer
    /core
      /providers
      /utils
    /screens
      /AboutScreen
      /HomeScreen
      /PostsScreen
      /ProjectsScreen
    /styles
      Global.ts
      Theme.ts
    App.tsx
    main.tsx
```

3.4. Padrões de Projeto Utilizados e Motivações

3.4.1. Componentização: Organização da UI em componentes reutilizáveis.

- *Motivação: Facilitar a reutilização, manutenção e teste de componentes. Além de promover uma estrutura mais organizada e legível. Cada componente é responsável por uma parte específica da interface, o que*

torna o código mais modular e fácil de entender.

- *Exemplo na Aplicação: Componentes como Button, Header, Footer, Container são usados para encapsular partes específicas da UI, garantindo que possam ser reutilizados em diferentes partes da aplicação.*

3.4.2. Hooks: Utilização de custom hooks, separando responsabilidades

- *Motivação: Foi utilizado Custom hooks para permitir a separação das responsabilidades relacionadas ao estado e lógica de negócios da interface de usuário. Além disso, com as custom hooks, foi possível a reutilização de lógica entre diferentes componentes sem duplicação de código.*
- *Exemplo na Aplicação: Custom hooks são usados para gerenciar a lógica de alternância de temas entre Dark Mode e Light Mode. Isso permite que a lógica de mudança de tema seja reutilizada por diferentes componentes, mantendo a consistência e simplificando a manutenção.*

3.4.3. Providers: Gerenciamento de estado global e contexto

- *Motivação: Houve a necessidade de gerenciamento do esttado global e o contexto da aplicação. Dessa forma, os providers permitiram a passagem de dados e funções entre componentes sem a necessidade de prop drilling, facilitando o gerenciamento do estado e a comunicação entre componentes em diferentes níveis da árvore de componentes.*
- *Exemplo na Aplicação: Providers são utilizados para gerenciar o tema da aplicação (Dark Mode e Light Mode). Um ThemeProvider fornece o estado do tema e funções para alterá-lo, permitindo que qualquer componente na árvore de componentes acesse e modifique o tema conforme necessário.*

3.4.4. Resumo:

- Os Design Paterns utilizados são essenciais para manter a aplicação escalável, modular e de fácil manutenção, garantindo que cada parte dos sistema seja claramente definida e facilmente testável.

3.5. Regras Isoladas (Seguindo SOLID)

A página de teste foi desenvolvida seguindo os princípios do SOLID, assegurando que cada componente e container tenha responsabilidades bem definidas e que o código seja modular e de fácil manutenção.

3.5.1. Single Responsibility Principle (SRP):

- *Motivação: Cada componente deve ter uma única responsabilidade. Isso facilita a compreensão, manutenção e teste do componente.*
- *Exemplo na Aplicação:*
    - O componente Button é responsável apenas por renderizar um botão.
    - O componente Header lida exclusivamente com a renderização do cabeçalho da página.

3.5.2. Open/Closed Principle (OCP):

- *Motivação: Os componentes devem estar abertos para extensão, mas fechados para modificação. Isso permite adicionar novas funcionalidades sem alterar o código existente.*

- *Exemplo na Aplicação:*
    - O componente SwitchThemeMode pode ser estendido para suportar novos temas sem modificar sua implementação atual.

3.5.3. Liskov Substitution Principle (LSP):

- *Motivação: Subclasses devem poder substituir suas classes base sem alterar o comportamento esperado do programa.*
- *Exemplo na Aplicação:*
    - Componentes que herdam de um componente base devem manter a compatibilidade e o comportamento esperado, garantindo que qualquer instância de um componente derivado funcione conforme o esperado.

3.5.4. Dependency Inversion Principle (DIP):

- *Motivação: Depender de abstrações, não de implementações concretas. Isso promove a reutilização de código e facilita a substituição de dependências.*
- *Exemplo na Aplicação:*
    - O uso de ThemeProvider para gerenciar o estado do tema, permitindo que diferentes implementações de gerenciamento de tema possam ser injetadas sem modificar os componentes que utilizam o tema.

4. Detalhamento: Plugin de Extração de Dados

4.1. Descrição

O Plugin extrai dados da página web e os envia para a API. Ele é ativado por um botão injetado na página e fornece feedback sobre a conclusão, andamento ou erros da extração.

4.2. Funcionalidades

- Validação do token
- Feedback visual das informações que serão extraídas
- Extração de dados do dispositivo, sistema operacional, origem (domínio) e contagem de mudanças de tema
- Feedback visual de conclusão
- Feedback visual de erro
- Feedback visual de token inválido
- Feedback visual de loading

4.3. Estrutura do Diretório

```
/Plugin
  /public
  /src
    /core
      /@types
      /config
      /entities
    /infra
      /api
      /browser
    /repositories
    /ui
    /usecases
    /utils
    global.d.ts
    main.ts
    vite-env.d.ts
  /test
  .env.local
  .gitignore
  index.html
  jest.config.cjs
  package.json
  tsconfig.json
  vite.config.ts
```

4.4. Padrões de Projeto Utilizados e Motivações

4.4.1. Singleton Pattern:

- *Motivação: Garantir que uma classe tenha apenas uma única instância e fornecer um ponto global de acesso a essa instância. Foi útil pela necessidade de gerenciar um recurso compartilhado, como os repositories.*
- *Exemplo na Aplicação: A classe AnalyticsDataRepository foi implementada como um Singleton para garantir que apenas uma instância seja usada para acessar e salvar dados de analytics, evitando inconsistências e duplicações.*

4.4.2. Dependency Injection:

- *Motivação: Promover a flexibilidade e a testabilidade do código ao injetar dependências em vez de criá-las internamente. Isso permite substituir facilmente as dependências por mocks ou stubs durante os testes, além de facilitar a troca de implementações em tempo de execução.*
- *Exemplo na Aplicação: Dependências como repositórios (TokenValidateRepository, BrowserAnalyticsRepository) são injetadas nas classes que as utilizam, promovendo a inversão de controle e facilitando a substituição das implementações concretas por alternativas de teste.*

4.5. Regras de SOLID e Clean Architecture

O plugin de extração de dados foi desenvolvido seguindo tanto os princípios do SOLID quanto os conceitos de Clean Architecture, garantindo um código limpo, modular e fácil de manter.

4.5.1. Single Responsibility Principle (SRP):

- *Motivação: Cada classe ou módulo deve ter uma única responsabilidade.*
- *Exemplo na Aplicação:*
    - A classe AnalyticsData é responsável apenas pela modelagem dos dados de analytics.
    - A classe TokenValidateRepository lida exclusivamente com a validação dos tokens.

4.5.2. Open/Closed Principle (OCP):

- *Motivação: O código deve ser aberto para extensão, mas fechado para modificação.*
- *Exemplo na Aplicação:*
    - A classe BrowserAnalyticsRepository pode ser estendida para suportar novos métodos de extração de dados sem alterar a implementação existente.

4.5.3. Liskov Substitution Principle (LSP):

- *Motivação: Subclasses devem poder substituir suas classes base sem alterar o comportamento esperado.*
- *Exemplo na Aplicação:*
    - Implementações de IGetBrowserAnalyticsRepository devem funcionar de maneira intercambiável sem alterar o comportamento do sistema.

4.5.4. Interface Segregation Principle (ISP):

- *Motivação: Muitas interfaces específicas são melhores do que uma interface geral única.*
- *Exemplo na Aplicação:*
    - Interfaces específicas para diferentes tipos de repositórios (IAnalyticsDataRepository, IGetBrowserAnalyticsRepository) em vez de uma interface genérica.

4.5.5. Dependency Inversion Principle (DIP):

- *Motivação: Depender de abstrações, não de implementações concretas.*
- *Exemplo na Aplicação:*
    - O uso de injeção de dependência para fornecer repositórios e serviços ao invés de criar instâncias diretamente dentro das classes.

4.5.6. Clean Architecture:

4.5.6.1. Camadas da Arquitetura:

- *Motivação: Separar as responsabilidades em camadas distintas para promover a independência da implementação e a testabilidade.*
- *Exemplo na Aplicação:*
    - Entities: Contém as entidades de domínio (AnalyticsData), que são objetos de negócios que não dependem de frameworks ou bibliotecas externas.
    - Use Cases: Contém a lógica de aplicação (GetAnalyticsDataUseCase, SaveAnalyticsDataUseCase), coordenando a interação entre as entidades e os repositórios.
    - Interface Adapters: Adaptadores e transformadores que convertem dados entre a camada de aplicação e a camada de infraestrutura
    - Frameworks and Drivers: Contém implementações específicas de frameworks e bibliotecas externas (Api, BrowserAnalyticsRepository).

4.5.6.2. Independência da Framework:

- *Motivação: A aplicação não deve depender de detalhes da framework. A framework deve ser facilmente substituível.*
- *Exemplo na Aplicação:*
    - A camada de infraestrutura pode ser trocada sem impactar a lógica de negócios ou a camada de aplicação. Por exemplo, substituir a implementação de Api ou BrowserAnalyticsRepository por outra biblioteca.

4.5.6.3. Teste Independente:

- *Motivação: Cada camada deve ser testável de forma independente, promovendo a testabilidade e facilitando a detecção de problemas.*
- *Exemplo na Aplicação:*
    - Testes unitários foram implementados para verificar a funcionalidade de cada use case, entidade e repositório de forma isolada.

4.5.7 Resumo

A adoção desses princípios e padrões de projeto garante que a aplicação seja robusta, fácil de manter e extensível, permitindo que novas funcionalidades sejam adicionadas com mínimo impacto no código existente.

5. Detalhamento: API

5.1. Descrição

A API recebe os dados extraídos e os armazena no Firebase Realtime Database. Foi desenvolvida para processar e gerenciar as solicitações de dados de analytics.

5.2. Funcionalidades

- Recepção e armazenamento de dados de analytics (/collect)
- Processamento de dados recebidos
- Integração com o Firebase Realtime Database
- Testes unitários
- Token JWT
- Restrição de acesso (5 requisições a cada 10 minutos por token)
- Listagem do histórico de extrações (/list ou /list?id=ID_DO_TOKEN)

5.3. Estrutura do Diretório

```
/API
  /src
    /controllers
    /middlewares
    /repositories
    /routes
    /services
    /utils
    app.ts
  .env
  .gitignore
  jest.config.cjs
  package.json
  tsconfig.json
```

5.4. Endpoints Disponíveis

5.4.1. **POST /collect**

O endpoint `/collect` é responsável por receber e armazenar dados de analytics extraídos. Este endpoint requer um token de autenticação, que tem um limite de 5 requisições a cada 10 minutos.

5.4.1.1. Requisição

- **Método:** `POST`
- **URL:** `/collect`
- **Cabeçalhos:**
  - `Authorization: Bearer <TOKEN>` - Token JWT para autenticação.
- **Body:**

  O corpo da requisição deve estar no formato JSON com as seguintes propriedades:

  ```
  {
    "device": "string",
    "os": "string",
  ```

```
  "sourceDomainUrl": "string",
  "themeChangeCount": number
}
```

- **device**: O tipo de dispositivo (ex: "desktop", "mobile").
- **os**: O sistema operacional (ex: "Windows", "macOS", "Android", "iOS").
- **sourceDomainUrl**: A URL do domínio de origem da página.
- **themeChangeCount**: O número de mudanças de tema (dark/light) na página.

5.4.1.2. Resposta

- **Código de Sucesso:** `200 OK`

```
{
  "message": "Dados recebidos e armazenados com sucesso."
}
```

- **Código de Erro:**
  - **400 Bad Request** - Se o corpo da requisição não estiver no formato correto.
  - **401 Unauthorized** - Se o token estiver ausente ou inválido.
  - **429 Too Many Requests** - Se o limite de requisições (5 por 10 minutos) for excedido.

```
{
  "error": "Mensagem de erro detalhada."
}
```

5.4.2. **GET /list**

O endpoint `/list` permite recuperar todas as extrações de dados salvas. É um endpoint público e não requer autenticação.

5.4.2.1. Requisição

- **Método:** `GET`
- **URL:** `/list` ou `/list?id=ID_DO_TOKEN`
  - Se nenhum parâmetro `id` for fornecido, o endpoint retorna todas as extrações salvas.
  - Se um parâmetro `id` for fornecido, o endpoint retorna os 20 últimos itens salvos para o token especificado.

5.4.2.2. Resposta

- **Código de Sucesso:** `200 OK`

- **Sem parâmetro `id`:**

```
[
  {
    "id": "string",
    "device": "string",
    "os": "string",
    "sourceDomainUrl": "string",
    "themeChangeCount": number,
    "createdAt": "string"
  },
  ...
]
```

- **Com parâmetro `id`:**

```
[
  {
    "id": "string",
    "device": "string",
    "os": "string",
    "sourceDomainUrl": "string",
    "themeChangeCount": number,
    "createdAt": "string"
  },
  ...
]
```

  - **id**: Identificador único da extração.
  - **device**: O tipo de dispositivo.
  - **os**: O sistema operacional.
  - **sourceDomainUrl**: A URL do domínio de origem da página.
  - **themeChangeCount**: O número de mudanças de tema.
  - **createdAt**: Data e hora em que os dados foram armazenados.

5.5. Padrões de Projeto Utilizados e Motivações

5.5.1. Repository Pattern:

- *Motivação: Separar a lógica de acesso a dados da lógica de negócios, promovendo uma interface clara para acessar e manipular dados.*
- *Exemplo na Aplicação: A classe AnalyticsDataRepository é responsável por interagir com o Firebase Realtime Database, isolando a lógica de acesso a dados da lógica de processamento de dados.*

5.5.2. Middleware:

- *Motivação: Modularizar o processamento das requisições HTTP, adicionando funcionalidades como autenticação, validação e logging de forma centralizada.*
- *Exemplo na Aplicação: Middlewares são usados para validar tokens, verificar o limite de acesso e processar as requisições antes que elas cheguem aos controladores.*

5.5.3. Service Layer:

- *Motivação: Encapsular a lógica de negócios em serviços, permitindo que os controladores se concentrem apenas na manipulação das requisições e respostas.*
- *Exemplo na Aplicação: Services são utilizados para processar e manipular dados de analytics, garantindo que a lógica de negócios esteja isolada e facilmente testável.*

5.5.5 Regras de SOLID e Clean Architecture

A API foi desenvolvida seguindo princípios SOLID e Clean Architecture para garantir um código modular, extensível e de fácil manutenção.

5.5.6. Single Responsibility Principle (SRP):

- *Motivação: Cada classe ou módulo deve ter uma única responsabilidade.*
- *Exemplo na Aplicação:*
    - Controladores lidam apenas com o processamento das requisições.
    - Serviços lidam com a lógica de negócios.

5.5.7. Open/Closed Principle (OCP):

- *Motivação: O código deve ser aberto para extensão, mas fechado para modificação.*
- *Exemplo na Aplicação:*
    - O código da API pode ser estendido com novos endpoints e serviços sem alterar o código existente.

5.5.8. Liskov Substitution Principle (LSP):

- *Motivação: Subclasses devem poder substituir suas classes base sem alterar o comportamento esperado.*
- *Exemplo na Aplicação:*
    - Qualquer implementação de repositório deve manter a compatibilidade com a interface esperada, permitindo a substituição das implementações conforme necessário.

5.5.9. Dependency Inversion Principle (DIP):

- *Motivação: Depender de abstrações, não de implementações concretas.*
- *Exemplo na Aplicação:*

- A API depende de abstrações para interagir com os repositórios e serviços, facilitando a substituição e a extensão do sistema sem alterar o código base.

5.5.10. Clean Architecture:

- *Motivação: Separar o sistema em camadas distintas para promover a separação de responsabilidades e permitir a flexibilidade de substituir partes do sistema sem afetar outras.*
- *Exemplo na Aplicação:*
  - Entidades: Representam o núcleo do sistema, encapsulando as regras de negócios e dados (por exemplo, AnalyticsData).
  - Use Cases: Contêm a lógica de aplicação específica e coordenam o fluxo de dados entre as entidades e os repositórios (por exemplo, casos de uso para processar e armazenar dados de analytics).
  - Interface Adapters: Incluem controladores e transformadores que convertem dados entre o formato necessário para a lógica de negócios e o formato requerido pela API (por exemplo, adaptadores de API).
  - Frameworks e Drivers: Incluem a implementação concreta de frameworks e bibliotecas que interagem com o sistema (por exemplo, integração com o Firebase Realtime Database).

6. Conclusão

Esta documentação fornece uma visão geral completa da aplicação, incluindo a estrutura do repositório, as funcionalidades de cada componente e os padrões de projeto utilizados