

Introdução

Este documento possui o objetivo de detalhar os elementos e componentes presentes no template da interface GUI, recomenda-se conhecimento básico em Javascript ou react.

Como Clonar repositório


Com git clone `git@github.com:syscsoftware/gui-template.git`, acesse o diretório com `cd gui-template` e depois `cd sysc-template`. Instale as dependências usando `npm install` e inicie o projeto com `npm start`.

A ordem sugerida de alteração é:

VIEW -> HOOKS -> COMPONENTES "Pai" -> COMPONENTES "Filhos"

Ou seja, verifique quais **views** vai utilizar no seu projeto e analise quais **hooks** e componentes elas precisam , em seguida **atualize os hooks com suas apis/dados** e caso o return da sua API seja muito diferente do esperado pelo hook e não for possível adaptar mude os componentes que necessitam deste hook.

Aconselha-se modificar o arquivo main(App.js) apenas caso sua aplicação não use nível de acesso ou login

 Componentes personalizados

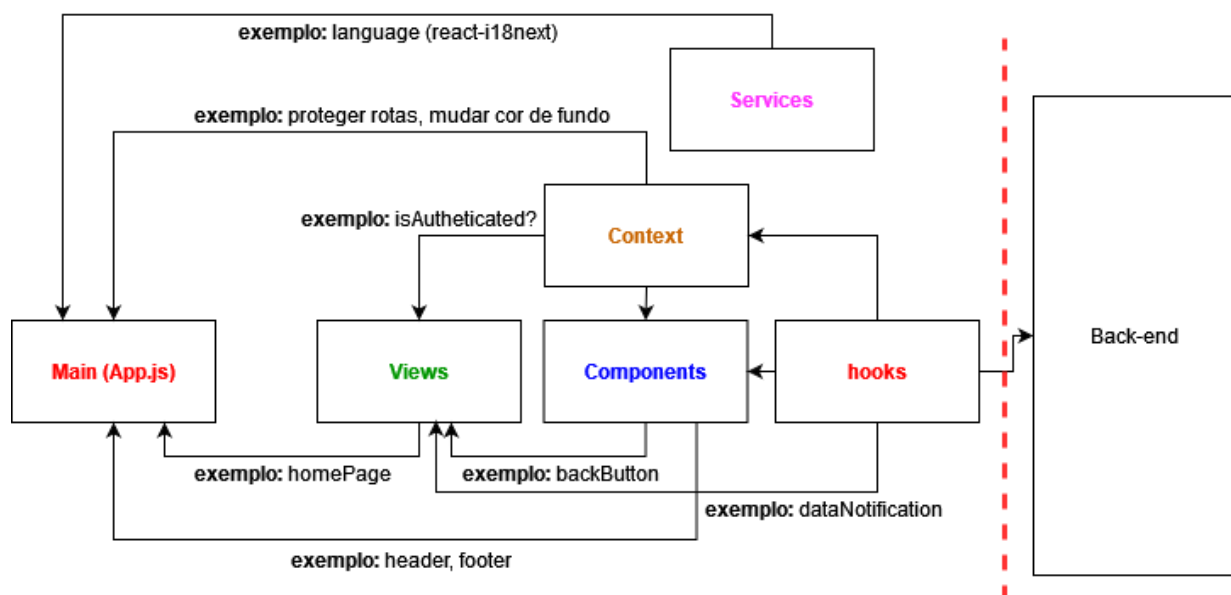
 Hooks personalizados

BREVE EXPLICAÇÃO DO REACT

A aplicação inicia no **App.js**, é este “main” que importará todas as Views, Componentes e Itens necessários para o funcionamento.

- **Views**: São as páginas da aplicação, importam os componentes necessários e ocupam 100% do espaço entre header e footer
- **Componentes**: Diferentes tipos de componentes, variam desde UI de formulário até um componente que serve para proteger as rotas e redirecionar caso não esteja logado
- **Hooks**: Servem para trazer dados tanto para **front->back** ou **back-> front (pode ser por API)**
- **Context**: Serve para deixar um estado disponível para todo o código sem precisar usar parâmetros, normalmente usado para autenticação


Fluxo dos componentes do projeto gui-template.



■ Componentes personalizados

■ Hooks personalizados

Introdução	1
1. Componentes	5
1.1 AlertBox (UI)	5
1.2 BackButton (UI + Self-sufficient)	6
1.3 BarChart (UI)	7
1.4 CRUDForm (UI)*Atualize com base sua model	9
1.5 CRUDTable (UI)*Atualize com base sua model	11
1.6 Dashboard (UI + NEED CONFIGURE HOOK)	13
1.7 DataTable (UI)	15
1.8 Footer (UI + Self-sufficient)	17
1.9 ForgotPasswordModal (UI)	18
1.10 Header (UI + Self-sufficient + NEED CONFIGURE HOOK)	21
1.11 LanguageSwitcher (UI + Self-sufficient)	24
1.12 LineGraph (UI + Self-sufficient + NEED CONFIGURE HOOK)	26
1.13 LoginForm (UI + NEED CONFIGURE HOOK)	29
1.14 ModalComponent (UI)	33
1.15 NotificationToast (UI)	36
1.16 PieChart (UI)	39
1.17 ProtectedRoute (Just Logic)	42
1.18 RegisterForm (UI)	45
1.19 Scores (UI)	49
1.20 SidebarMenu (UI)	52
1.21 Theme (UI + Self-sufficient)	55
2. Views	58
2.1 LoginPage (NEED CONFIGURE HOOK)	58
2.2 RegisterPage (NEED UPDATE CONTEXT + HOOK)	60
2.3 HomePage	63
2.4 GraphPage	65
2.5 DashboardPage	66
2.6 CRUDPage (NEED CONFIGURE HOOK)	68
3. Hooks	72
3.1 useAuthLogin	72
3.2 useCRUDData	74
3.3 useDashboardData	77
3.4 useForgotPassword	80
3.5 useGraph	81
3.6 useNotifications	82

 Componentes personalizados

 Hooks personalizados

3.7 useWindowSize	83
4. Contexts *Importante de para login e nível de acesso	84
AuthContext	84
Fluxo de Autenticação	86
6. Services *Importante de para tradução de textos	87
7. Sugestão de API's do Backend	88
1. API: Recuperação de Senha (Forgot Password)	89
2. API: Validação de Token	90
3. API: Atualização de Senha	91
4. API: Notificações	92
5. API: Dados do Gráfico	94
6. API: Login	95
7. API: Get Item	96
8. API: Criar Item	98
9. API: Atualizar Item	99
10. API: Deletar Item	100
11. API: Dados para Scores	101
12. API: Dados para Gráficos (Pie + Bar)	102
13. API: Dados para TABELA	103
14. API: Dados para Alertas	104

1. Componentes

1.1 AlertBox (UI)

12/11/2024, 12:42:40 **ALERTA:** Verifique a temperatura do servidor.

12/11/2024, 12:42:40 **EMERGENCIA:** O uso de memória está alto!

12/11/2024, 12:42:40 **INATIVIDADE:** O sistema está inativo há mais de 10 minutos.

Componente para exibir uma lista de alertas, com tipo, mensagem e timestamp.

Dependências


- React
- AlertBox.css (estilos do componente)


Props

- `alerts` (array): Lista de objetos com as propriedades:
 - `timestamp` (string)
 - `type` (string)
 - **Opções: alerta, emergencia, inatividade**
 - `message` (string)

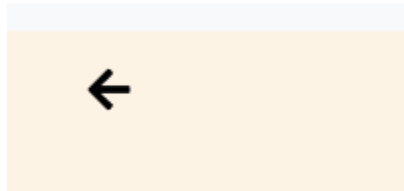
Exemplo de Uso (dentro de uma view ou component)

```
<AlertBox alerts={[{ timestamp: '2024-11-12 10:00', type: 'error', message: 'Erro.' }]} />
```

 Componentes personalizados

 Hooks personalizados

1.2 BackButton (UI + Self-sufficient)



Componente para criar um botão de navegação que retorna à página anterior no navegador.

Dependências

- **React**
- **react-router-dom** (para navegação)
- **react-bootstrap** (para o componente **Button**)
- **react-icons** (para o ícone **FaArrowLeft**)

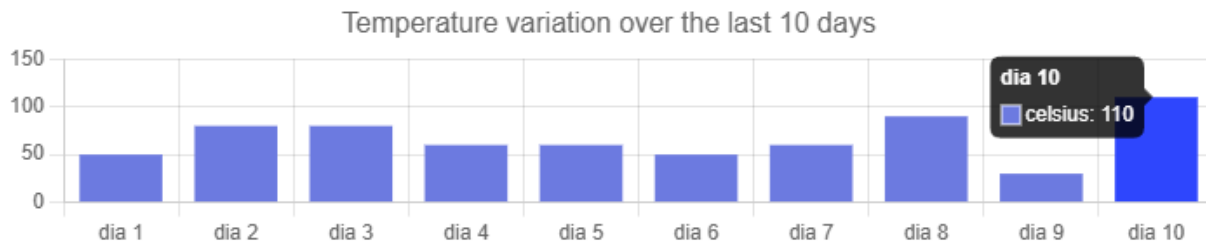
Função

- **useNavigate**: Hook do **react-router-dom** usado para navegação.
- **handleBack**: Função que usa **navigate(-1)** para voltar à página anterior.

Exemplo de Uso (dentro de uma view ou component)

```
<BackButton />
```

1.3 BarChart (UI)



Componente para renderizar um gráfico de barras utilizando o Chart.js, com suporte a tradução de títulos e legendas.

Dependências

- React
- react-chartjs-2 (para o componente `Bar` do Chart.js)
- chart.js (para criação e personalização do gráfico)
- react-i18next (para tradução de textos)

Props

- `data` (array): Lista de objetos com as propriedades:
 - `label` (string): Rótulo da barra.
 - `value` (number): Valor numérico para a barra.

Funcionalidade


- O gráfico é configurado com título e barras baseadas nas `data` passadas.
- O título e a legenda são traduzidos com `useTranslation` do `react-i18next`.


■ Componentes personalizados

■ Hooks personalizados

Exemplo de Uso

```
<BarChart data=[{ label: 'Jan', value: 30 }, { label:  
'Feb', value: 45 }] />
```

 Componentes personalizados

 Hooks personalizados

1.4 CRUDForm (UI)*Atualize com base sua model



The image shows a light gray rectangular box representing a form. Inside, there are two labels: 'Name' and 'Status'. Below 'Name' is a text input field containing 'Tipo 2 - Item A'. Below 'Status' is another text input field containing 'Ativo'. At the bottom of the box, there are two buttons. The left button has a green checkmark icon and the text 'Update item'. The right button has a red 'X' icon and the text 'Cancel'.

OBS> Descrição baseada em Item2Form (modifique de acordo tua model/tabela)


Componente de **formulário** para **criar** ou **editar** um item. Utiliza `react-bootstrap` para o layout e `react-i18next` para tradução. O formulário permite submeter dados ou cancelar a operação.

Dependências

- **React**
- **react-bootstrap** (para o componente `Form` e `Button`)
- **react-icons** (para ícones de check e cancelamento)
- **react-i18next** (para tradução de textos)

Props

- **onSubmit** (function): Função chamada ao submeter o formulário com os dados.
- **itemToEdit** (object, optional): Dados do item a ser editado, **preenche os campos do formulário**.
- **onCancel** (function, optional): Função chamada ao clicar em cancelar (apenas se `itemToEdit` estiver presente).

 Componentes personalizados

 Hooks personalizados

Funcionalidade




- O formulário **pode ser usado tanto para criar quanto para editar um item.**
- Ao editar, os campos são preenchidos com os dados do item.
- Ao submeter, os dados são passados para a função **onSubmit**.
- O botão de **cancelar** aparece apenas quando há um item para editar.

Exemplo de Uso

Dentro de uma **view** ou de outro **componente** chame este componente e em **onCancel**, chame os hooks associados

```
<Item2Form  
  
  onSubmit={{(data) => console.log('Form submitted', data)}}  
  itemToEdit={{ id: '1', name: 'Item 1', status: 'Active' }}  
  onCancel={() => console.log('Form cancelled')}}  
  
>
```

1.5 CRUDTable (UI) **Atualize com base sua model*

ID	Name	Status	Actions
1	Tipo 2 - Item A	Ativo	 
2	Tipo 2 - Item B	Inativo	 

OBS> Descrição baseada em Item2Table (modifique de acordo tua model/tabela)


Componente de tabela para listar itens, com botões de **edição** e **exclusão**. Traduz os títulos da tabela e as ações usando `react-i18next`.


Dependências

- **React**
- **react-bootstrap** (para o componente `Table` e `Button`)
- **react-icons** (para os ícones de edição, exclusão e ativação)
- **react-i18next** (para tradução de textos)

Props

- **data** (array): Lista de objetos a serem exibidos na tabela. Cada item deve conter:
 - `id` (string|number): Identificador único.
 - `name` (string): Nome do item.
 - `status` (string): Status do item.
- **onEdit** (function): Função chamada ao clicar no botão de edição, recebe o item como argumento.

 Componentes personalizados

 Hooks personalizados

- **onDelete** (function): Função chamada ao clicar no botão de exclusão, recebe o **id** do item.
- **onActivate** (function, optional): Função chamada ao clicar no botão de ativação (comportamento ainda comentado no código).

Funcionalidade

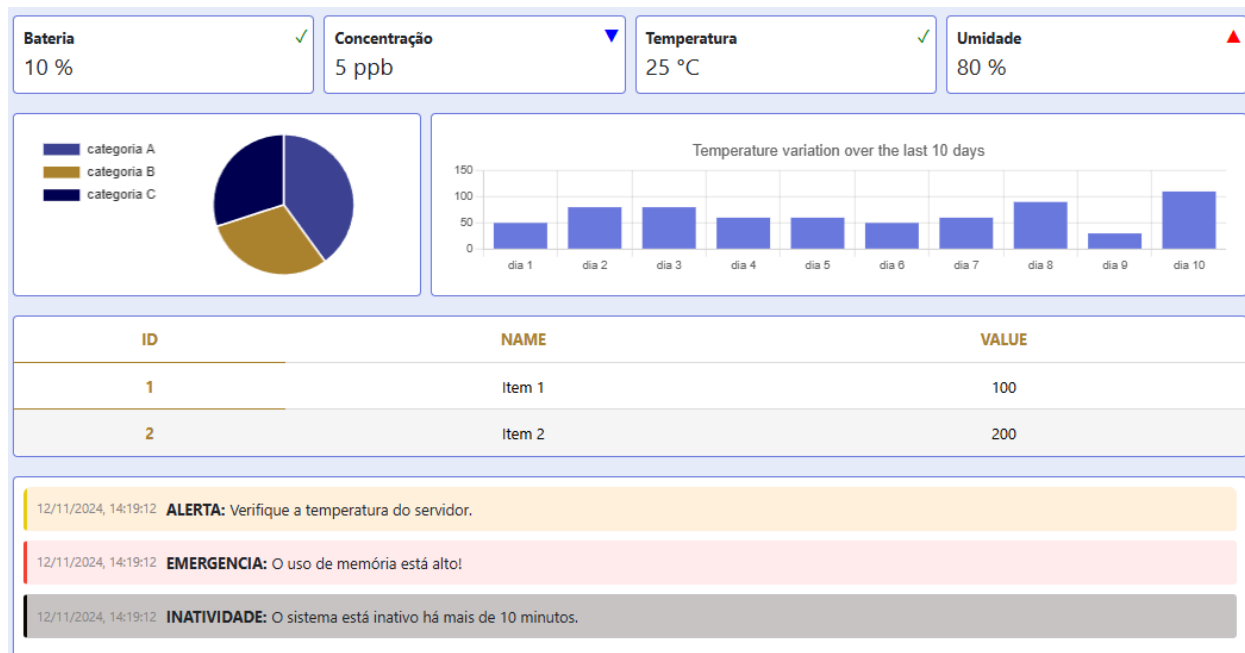
- Exibe os dados em uma tabela com colunas para **ID**, **Nome**, **Status** e **Ações**.
- Botões de edição e exclusão para cada item.
- Suporte a tradução para os títulos das colunas e para as ações.

Exemplo de Uso

Dentro de uma **view** ou de outro **componente** chame este componente e com em **onEdit**, **onDelete**, **onActivate** chame os hooks associados

```
<Item2Table  
  
  data={[{ id: 1, name: 'Item 1', status: 'Ativo' }]}  
  
  onEdit={(item) => console.log('Edit item', item)}  
  
  onDelete={(id) => console.log('Delete item', id)}  
  
  onActivate={(id) => console.log('Activate item', id)}  
  
>
```


1.6 Dashboard (UI + NEED CONFIGURE HOOK)



Componente principal para exibir um painel com gráficos, pontuações, tabela de dados e alertas. Utiliza hooks para obter dados e ajustar o layout conforme o tamanho da tela.

Dependências

- **React**
- **react-bootstrap** (para o componente **Container**)
- **Scores**: Componente para exibir pontuações.
- **PieChart** e **BarChart**: Componentes de gráficos.
- **DataTable**: Componente de tabela de dados.
- **AlertBox**: Componente para exibir alertas.
- **useDashboardData (hook)**: Hook que possui os dados do painel.
- **useWindowSize (hook)**: Hook para monitorar o tamanho da janela.

 Componentes personalizados

 Hooks personalizados

Funcionalidade

- Obtém dados do dashboard via `useDashboardData`.
- Ajusta o layout dos gráficos dinamicamente usando `useWindowSize`.
- Exibe uma mensagem de "Carregando..." enquanto os dados estão sendo carregados.

Estrutura

- **Pontuações:** Exibidas através do componente `Scores`.
- **Gráficos:** Inclui `PieChart` e `BarChart`, que atualizam automaticamente com o redimensionamento da tela.
- **Tabela de Dados:** Exibida via `DataTable`.
- **Alertas:** Exibidos no componente `AlertBox`.

Exemplo de Uso

Certifique-se que o hook `useDashboardData` está retornando dados, não há problema se não possuir todas informações, conteúdos vazios apenas não serão mostrados, não haverá quebra do sistema.

Dentro de uma view/Página apenas chame

```
<Dashboard />
```

1.7 DataTable (UI)

ID	NAME	VALUE
1	Item 1	100
2	Item 2	200

Componente de tabela dinâmica para exibir dados tabulares. Exibe uma mensagem quando não há dados disponíveis e ajusta as colunas automaticamente com base nas chaves dos objetos do array de dados.

Dependências

- React
- DataTable.css: Arquivo de estilo customizado para o componente.


Props


- `data` (array): Array de objetos contendo os dados a serem exibidos na tabela. Cada objeto deve ter as mesmas chaves para que sejam usados como cabeçalhos.

Funcionalidade


- Renderiza uma tabela baseada em `data`.
- Se `data` estiver vazio ou for `null`, exibe uma mensagem "Nenhum dado disponível".
- Cria automaticamente o cabeçalho com as chaves do primeiro objeto do array.


Exemplo de Uso

 Componentes personalizados

 Hooks personalizados

```
<DataTable data=[{ id: 1, name: 'Item 1', status: 'Active' }]]  
>
```

 Componentes personalizados

 Hooks personalizados

1.8 Footer (UI + Self-sufficient)



© 2024 SYSC System Group. All rights reserved

Componente de rodapé que exibe o logo da empresa e um texto com copyright, suportando tradução através de react-i18next.

Dependências


- React
- Footer.css: Arquivo de estilo personalizado para o rodapé.
- react-i18next: Para tradução do texto.

Funcionalidade

- Exibe um link com o logo da empresa que **redireciona para uma página externa ao ser clicado**.
- Renderiza um texto de copyright que inclui o ano atual e uma mensagem traduzida com o react-i18next.

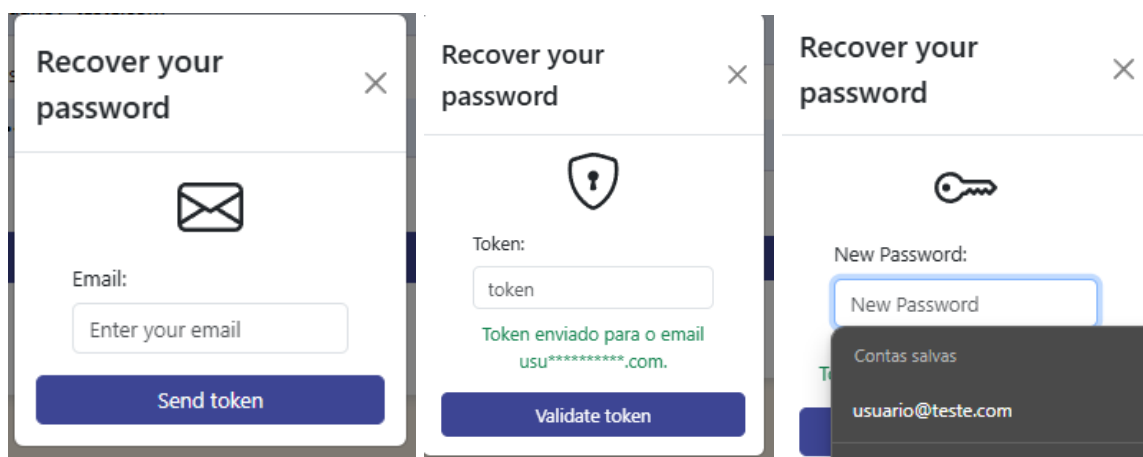
Exemplo de Uso

```
<Footer />
```

 Componentes personalizados

 Hooks personalizados

1.9 ForgotPasswordModal (UI)




Modal de recuperação de senha com vários passos (**4 passos no total**), incluindo envio de token, validação do token e atualização da senha. Suporta **feedback de erro e mensagens de sucesso em cada passo do processo**.


Dependências

- **React**
- **react-bootstrap**: Utilizado para componentes de modal, botão, formulário e spinner.
- **react-bootstrap-icons**: Ícones visuais para representar cada etapa do processo.
- **react-i18next**: Para tradução do conteúdo.
- **Theme**: Objeto de tema personalizado para controle de cores.

Props

- **showForgotModal** (boolean): Controla a visibilidade do modal.

 Componentes personalizados

 Hooks personalizados

- **setShowForgotModal** (function): Função para alternar a visibilidade do modal.
- **step** (number): Define a etapa atual (1 a 4) do processo de recuperação.
- **username** (string): E-mail do usuário para enviar o token de recuperação.
- **setUsername** (function): Função para atualizar o e-mail.
- **token** (string): Token de validação fornecido ao usuário.
- **setToken** (function): Função para atualizar o token.
- **newPassword** (string): Nova senha do usuário.
- **setNewPassword** (function): Função para atualizar a nova senha.
- **message** (string): Mensagem de sucesso ou instrução.
- **forgotError** (string): Mensagem de erro para feedback ao usuário.
- **forgotLoading** (boolean): Indica se uma ação assíncrona está em andamento.
- **handleForgotPassword** (function): Função para acionar o envio do token.
- **handleValidateToken** (function): Função para acionar a validação do token.
- **handleUpdatePassword** (function): Função para acionar a atualização da senha.

Funcionalidade

- **Etapa 1:** O usuário insere seu e-mail para envio do token de recuperação.
- **Etapa 2:** O usuário insere o token recebido para validação.
- **Etapa 3:** O usuário define a nova senha.
- **Etapa 4:** Mensagem de sucesso ao concluir o processo.

Exemplo de Uso

Dentro do LoginForm ou outro componente pai faça :

`<ForgotPasswordModal`

```
    showForgotModal={showForgotModal}

    setShowForgotModal={setShowForgotModal}

    step={step}

    username={username}

    setUsername={setUsername}

    token={token}

    setToken={setToken}

    newPassword={newPassword}

    setNewPassword={setNewPassword}

    message={message}

    forgotError={forgotError}


    forgotLoading={forgotLoading}

    handleForgotPassword={handleForgotPassword}

    handleValidateToken={handleValidateToken}

    handleUpdatePassword={handleUpdatePassword}
```

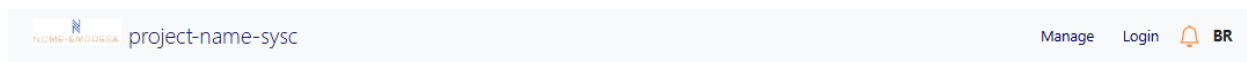
`/>`

 Componentes personalizados

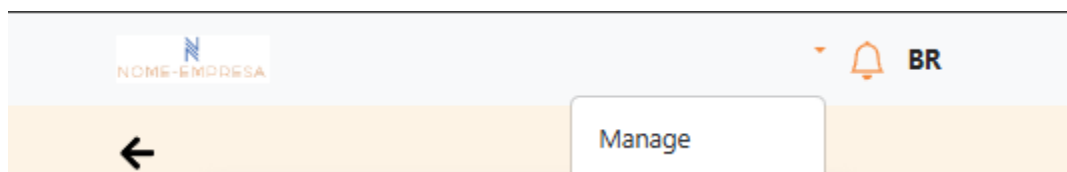
 Hooks personalizados

1.10 Header (UI + Self-sufficient + NEED CONFIGURE HOOK)

Header para dispositivos maiores que 1000px



Header para dispositivos menores que 1000px



Descrição

O componente **Header** é uma barra de navegação dinâmica que se adapta a diferentes tamanhos de tela. Inclui links para navegação, troca de idioma, notificações, e uma opção de logout, dependendo do estado de autenticação do usuário. Quando visualizado em telas menores, os links são agrupados em um menu de "hambúrguer" para facilitar o acesso.

Observação ! : É possível descartar o uso de **NotificationToast** e **useNotifications**, apenas apague os componentes

Dependências

- **React** e **React Router**: Para navegação e redirecionamento.
- **react-bootstrap**: Utilizado para o layout da barra de navegação, botão e dropdown.
- **react-i18next**: Para internacionalização e tradução.
- **react-icons**: Para exibir o ícone de "hambúrguer".

Componentes personalizados

Hooks personalizados

- **AuthContext**: Contexto de autenticação customizado para verificar a autenticação e gerenciar o logout. (leia sobre contextos para um entendimento melhor)
- **NotificationToast**: Componente de notificações.
- **LanguageSwitcher**: Componente para alternar o idioma do site.
- **useNotifications**: Hook customizado para carregar as notificações do usuário.

Props

O **Header** não recebe props diretamente, mas utiliza vários hooks e contextos para gerenciar o estado e as interações do componente.

Funcionalidade

- **Links de Navegação:**
 - Links para o *Dashboard*, *Itens*, *Graph* e *Login* são renderizados dinamicamente com base no estado de autenticação.
 - Em dispositivos móveis, os links são agrupados dentro de um **Dropdown**.
- **Logout:**
 - Disponível apenas para usuários autenticados, o botão de logout limpa o contexto de autenticação e redireciona o usuário para a página de login.
- **Alternância de Idioma:**
 - Inclui o componente **LanguageSwitcher** para alternar entre idiomas.
- **Notificações:**
 - Mostra o componente **NotificationToast** para exibir notificações em tempo real.

Componentes personalizados

Hooks personalizados

Estado Local

- **isMobile** (**boolean**): Determina se a tela é menor que 1000 pixels para alternar entre a versão de desktop e a versão "hambúrguer" da navegação.


Exemplo de Uso

Este componente é esperado para ser usado no topo da aplicação, como em uma estrutura de layout global, atualmente é usado no App.js, ou seja, qualquer view/Page que seja criada receberá a header do componente "pai" App.js.

```
import Header from './components/Header/Header';
```

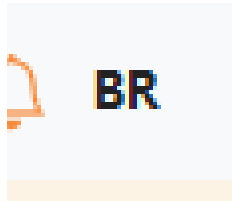
```
function App() {  
  return (  
    <div className="App">  
      <Header />  
      {/* Outros componentes da aplicação */}  
    </div>  
  );  
}
```

```
export default App;
```

 Componentes personalizados

 Hooks personalizados

1.11 LanguageSwitcher (UI + Self-sufficient)



Descrição

O componente `LanguageSwitcher` é um alternador entre os idiomas inglês (EN) e português (BR) com um único clique. Ele utiliza o hook `useTranslation` do `react-i18next` para gerenciar a alteração de idioma na aplicação.

Dependências


- `react-i18next`: Biblioteca de internacionalização que fornece o hook `useTranslation` para alternar entre os idiomas suportados.

Props

Este componente não recebe nenhuma prop diretamente, mas usa o hook `useTranslation` para controlar o idioma atual da aplicação.

Estado Local

- `isEnglish (boolean)`: Define o idioma atual exibido no componente.
 - `true`: O idioma atual é inglês.
 - `false`: O idioma atual é português.

 Componentes personalizados

 Hooks personalizados

Funcionalidade

- Alternância de Idioma:
 - O clique no componente chama a função `toggleLanguage`, que alterna entre `pt` (português) e `en` (inglês), e atualiza o estado `isEnglish` para refletir o novo idioma.

Exemplo de Uso

Inclua o componente `LanguageSwitcher` em qualquer lugar da aplicação como no cabeçalho ou rodapé.

```
import LanguageSwitcher from './components/LanguageSwitcher';

function Header() {

  return (

    <header>

    <LanguageSwitcher />

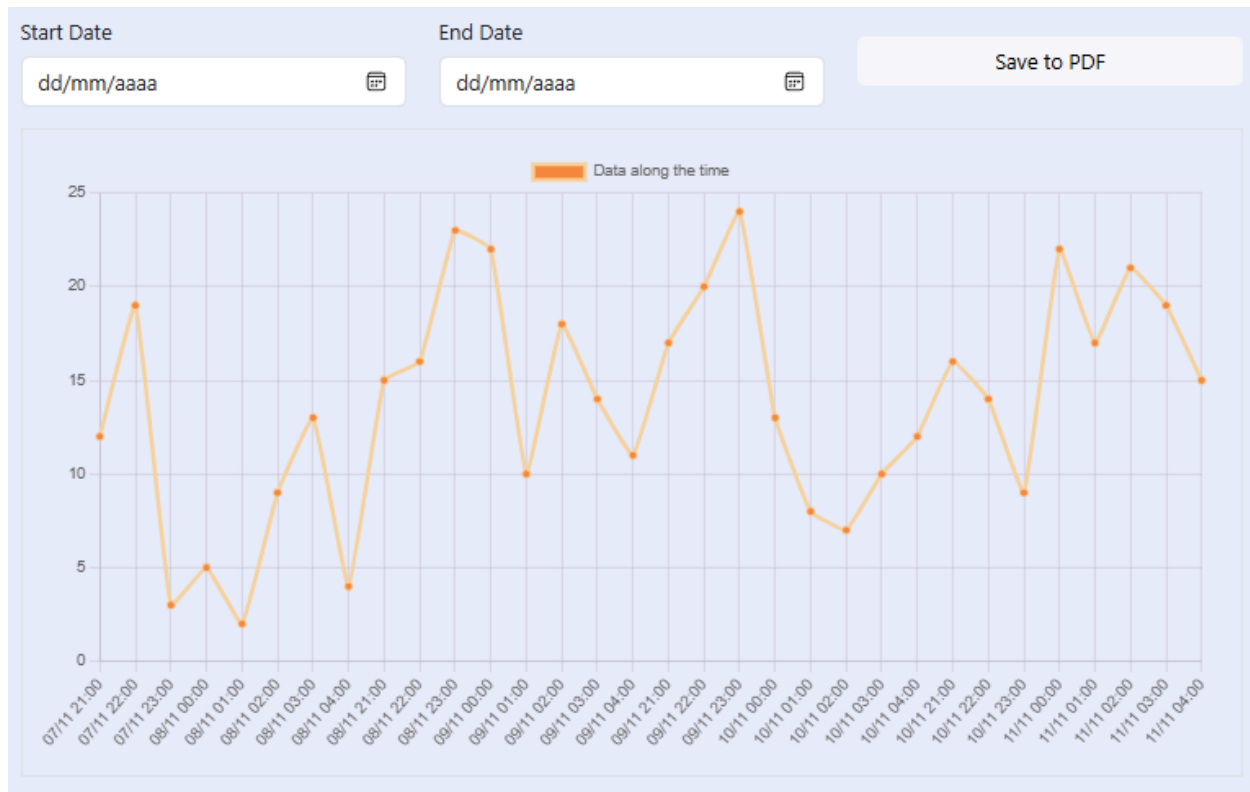
    </header>

  );

}

export default Header;
```

1.12 LineGraph (UI + Self-sufficient + NEED CONFIGURE HOOK)




O componente **LineGraph** é um gráfico de linhas interativo que exibe dados ao longo do tempo, com opções para filtragem por data e exportação do gráfico em PDF. Ele utiliza a biblioteca **react-chartjs-2** para renderizar o gráfico e os hooks **useGraph** e **useTranslation** para buscar os dados e fornecer suporte à tradução.

Dependências

- **react-chartjs-2** e **chart.js**: Para renderização e controle do gráfico.
- **html2canvas** e **jsPDF**: Para funcionalidade de exportação do gráfico em PDF.

■ Componentes personalizados

■ Hooks personalizados

- **moment**: Biblioteca para formatação de datas (opcional, mas recomendada para melhor formatação de data no gráfico).
- **react-i18next**: Para tradução dos textos e legendas no gráfico.
-  **useGraph (obrigatório)**: Hook customizado para buscar os dados de exibição do gráfico.

Props

Este componente não recebe props diretamente. Ele utiliza dados de estado e hooks para configurar a exibição do gráfico.

Estado Local

- **startDate (string)**: Define a data de início do filtro de exibição de dados no gráfico.
- **endDate (string)**: Define a data de fim do filtro de exibição de dados no gráfico.

Funcionalidade

1. Renderização do Gráfico:

- Utiliza dados buscados via **useGraph** para renderizar um gráfico de linha.
- Os dados são atualizados automaticamente conforme o filtro de datas é alterado.

2. Filtragem por Data:

- Os inputs de data permitem ao usuário definir um intervalo específico de datas para visualização.
- O gráfico exibe apenas os dados que se enquadram nesse intervalo.

3. Exportação em PDF:

- A função `exportToPDF` permite ao usuário exportar a visualização atual do gráfico em um arquivo PDF.
- Usa `html2canvas` para capturar o gráfico como imagem e `jsPDF` para gerar o PDF.

Exemplo de Uso

Inclua o componente `LineGraph` em qualquer seção da aplicação onde deseja exibir dados históricos em um gráfico de linhas..

```
import LineGraph from './components/LineGraph';

function Dashboard() {

  return (

    <div>

      <h1>Relatório de Dados</h1>


      <LineGraph />


      { /* Outros componentes do dashboard */ }

    </div>

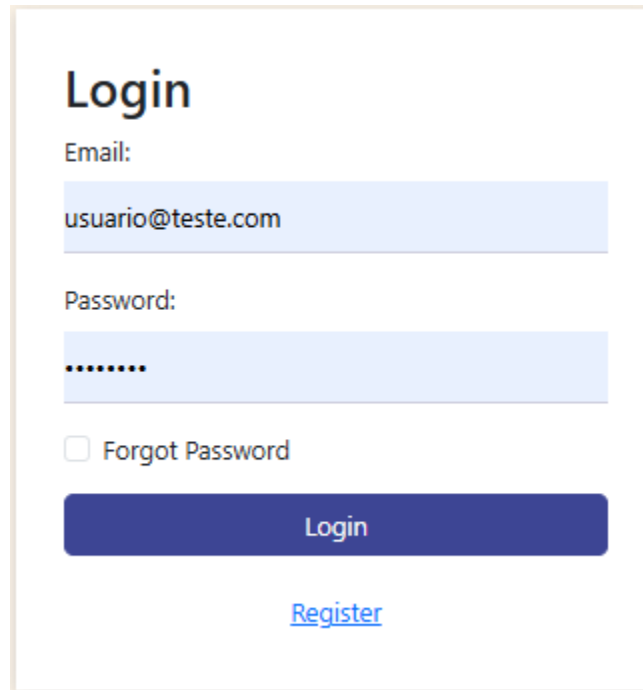
  );

}
```

 Componentes personalizados

 Hooks personalizados

1.13 LoginForm (UI + NEED CONFIGURE HOOK)


A mockup of a login form titled "Login". It features two input fields: "Email:" with the placeholder text "usuario@teste.com" and "Password:" with masked characters ".....". Below the password field is a checkbox labeled "Forgot Password". At the bottom, there is a dark blue "Login" button and a blue "Register" link.

Descrição

O componente **LoginForm** é um formulário de login que permite ao usuário inserir suas credenciais, enviar uma requisição de autenticação e, em caso de falha, exibir uma mensagem de erro. Além disso, ele inclui uma funcionalidade de recuperação de senha por meio de um modal que guia o usuário pelo processo de redefinição de senha.

Dependências

- **react-bootstrap**: Para os componentes de formulário e botão.
- **react-i18next**: Para tradução de textos dentro do formulário.
- **react-router-dom**: Para navegação à página de registro.

 Componentes personalizados

 Hooks personalizados

- **useForgotPassword**: Hook customizado para gerenciamento da recuperação de senha.
- **ForgotPasswordModal**: Modal para recuperação de senha.
- **theme**: Objeto de tema para aplicação de cores customizadas.

Props


- **email (string)**: Estado do e-mail inserido pelo usuário.
- **setEmail (function)**: Função para atualizar o e-mail no estado do componente pai.
- **password (string)**: Estado da senha inserida pelo usuário.
- **setPassword (function)**: Função para atualizar a senha no estado do componente pai.
- **handleSubmit (function)**: Função para submeter o formulário de login.
- **loading (boolean)**: Indica se a requisição de login está em andamento.
- **error (string)**: Mensagem de erro, exibida quando ocorre uma falha no login.

Estado Local

- **showForgotModal (boolean)**: Controla a visibilidade do modal de recuperação de senha.

Hooks Usados

- **useTranslation (obrigatório)**: Hook do react-i18next para traduzir os textos e placeholders.
- **useNavigate (obrigatório)**: Hook para redirecionar à página de registro.
- **useForgotPassword (opcional)**: Hook para funcionalidade de recuperação de senha (se habilitado).

 Componentes personalizados

 Hooks personalizados

Funcionalidade

1. Autenticação:

- Ao enviar o formulário, `handleSubmit` é chamado para autenticar o usuário.
- Caso `loading` esteja `true`, o botão de login exibe o texto de carregamento e é desabilitado.

2. Recuperação de Senha:

- O checkbox "Esqueci minha senha" abre o modal `ForgotPasswordModal`.
- Este modal guia o usuário pela recuperação de senha usando o hook `useForgotPassword` para enviar e validar o token de redefinição.

3. Redirecionamento para Registro:


- O botão "Registrar" redireciona para a página de registro usando o hook `useNavigate`.

Exemplo de Uso

Inclua o componente `LoginForm` em uma página de login para oferecer ao usuário uma interface para autenticação e recuperação de senha.

```
import LoginForm from './components/LoginForm';
```

```
function LoginPage() {  
  
  const [email, setEmail] = useState('');  
  
  const [password, setPassword] = useState('');  
  
  const [loading, setLoading] = useState(false);  
  
}
```

 Componentes personalizados

 Hooks personalizados

```
const [error, setError] = useState('');

const handleLogin = (e) => {

    e.preventDefault();

    // Lógica de autenticação

};

return (

    <div>

        <LoginForm

            email={email}

            setEmail={setEmail}

            password={password}

            setPassword={setPassword}

            handleSubmit={handleLogin}

            loading={loading}


            error={error}


        />

    </div>

);

}
```

 Componentes personalizados

 Hooks personalizados

1.14 ModalComponent (UI)



Descrição


O `ModalComponent` é um componente de interface de usuário que exibe um modal (caixa de diálogo) utilizando o `react-bootstrap`. Ele permite que o modal seja exibido ou ocultado com base em uma prop booleana e inclui um botão para fechar ou salvar as mudanças. Este modal é uma solução genérica para exibir informações ou ações em uma sobreposição na página atual.

Dependências

- **react-bootstrap**: Biblioteca que fornece os componentes `Modal` e `Button`.
- **theme**: Objeto de tema customizado, utilizado para definir a cor do botão no rodapé do modal.

Props

- **showModal (boolean, obrigatório)**: Controla a visibilidade do modal.
 - `true`: O modal é exibido.
 - `false`: O modal é oculto.

 Componentes personalizados

 Hooks personalizados

- **handleClose (function, obrigatório):** Função chamada ao clicar no botão de fechar ou salvar mudanças. Usada para manipular o fechamento do modal no componente pai.

Estado Local

Este componente não possui estado local, pois a visibilidade e o fechamento são controlados pelas props.

Funcionalidade

1. Exibição do Modal:

- Quando `showModal` é `true`, o modal é exibido na tela com um título, conteúdo e rodapé com o botão "Salvar mudanças".

2. Ação do Botão:

- O botão "Salvar mudanças" utiliza `theme.primary` para definir sua cor e chama a função `handleClose` ao ser clicado, permitindo ao componente pai gerenciar o fechamento do modal.

Exemplo de Uso


Inclua o `ModalComponent` em um componente pai e controle sua visibilidade com o estado local desse componente pai.

```
import React, { useState } from 'react';

import ModalComponent from './components/ModalComponent';

function ExampleParentComponent() {

    const [showModal, setShowModal] = useState(false);
```

 Componentes personalizados

 Hooks personalizados

```
const toggleModal = () => setShowModal(!showModal);

return (

  <div>

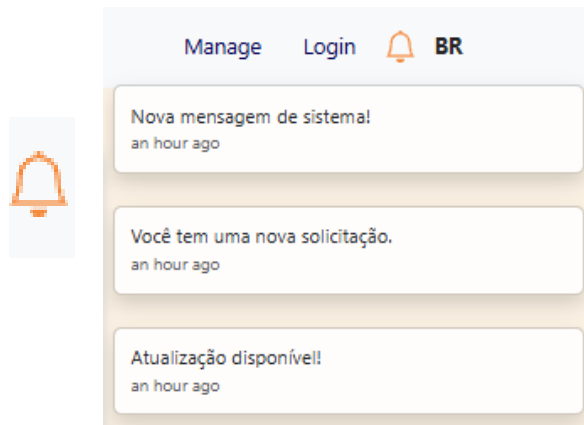
    <button onClick={toggleModal}>Abrir Modal</button>

    <ModalComponent showModal={showModal}
handleClose={toggleModal} />

  </div>

);
}
```

1.15 NotificationToast (UI)




O componente `NotificationToast` exibe uma lista de notificações como uma série de "toasts" (notificações flutuantes). Cada notificação possui uma mensagem e um timestamp formatado para indicar o tempo desde o seu recebimento. O ícone de notificação (sino) muda de cor se houver novas notificações, e o usuário pode abrir ou fechar o `ToastContainer` ao clicar no ícone.

Dependências

- **react-bootstrap**: Fornece os componentes `Toast`, `ToastContainer`, e `Button` para construir a interface da notificação.
- **react-icons**: Usado para incluir o ícone de sino (`BsBell`), que representa visualmente as notificações.
- **moment**: Biblioteca para manipulação e formatação de datas, exibindo o tempo decorrido desde o recebimento da notificação.

Props

 Componentes personalizados

 Hooks personalizados

- **notifications (array, obrigatório)**: Lista de objetos de notificação, cada um com as seguintes propriedades:
 - **id** (string ou number): Identificador único para cada notificação.
 - **message** (string): Texto da notificação a ser exibido.
 - **timestamp** (string ou Date): Data e hora da notificação para cálculo do tempo decorrido.

Estado Local

- **showToast (boolean)**: Controla a visibilidade do `ToastContainer`.
 - **true**: O `ToastContainer` com as notificações é exibido.
 - **false**: O `ToastContainer` está oculto.

Funcionalidade

1. Alternância de Visibilidade:

- O clique no botão com o ícone de sino chama a função `toggleToast`, que alterna a visibilidade do `ToastContainer`.


2. Ícone com Indicador de Notificação:

- O ícone `BsBell` muda de cor para laranja (`#f58a3d`) se houver notificações, e para cinza (`#222222`) se não houver.

3. Exibição do Tempo Decorrido:

- Cada notificação exibe o tempo decorrido desde o `timestamp` com a ajuda de `moment`, no formato "há x minutos/horas".

Exemplo de Uso

 Componentes personalizados

 Hooks personalizados

Inclua o `NotificationToast` em um componente que gerencie uma lista de notificações e passe essa lista como prop para exibir as notificações ao usuário.

```
import React, { useState } from 'react';

import NotificationToast from '../components/NotificationToast';

function Header() {

    const [notifications, setNotifications] = useState([

        { id: 1, message: 'mensagem 1', timestamp: new Date() },

        { id: 2, message: 'mensagem 2', timestamp: new Date() },

    ]);

    return (

        <header>

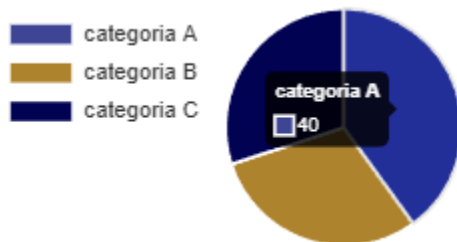
            <NotificationToast notifications={notifications} />

        </header>

    );

}
```

1.16 PieChart (UI)



O componente `PieChart` exibe um gráfico de pizza usando a biblioteca `Chart.js` e o componente `Pie` do `react-chartjs-2`.

Dependências

- **react-chartjs-2**: Fornece o componente `Pie`, que facilita a integração com a biblioteca `Chart.js` para gerar gráficos.
- **chart.js**: Biblioteca para criação de gráficos dinâmicos, utilizada para renderizar o gráfico de pizza.

Props

- **data (array, obrigatório)**: Lista de objetos, onde cada objeto contém as seguintes propriedades:
 - **label (string)**: Nome ou descrição do segmento no gráfico.
 - **value (number)**: Valor numérico que determina o tamanho do segmento no gráfico de pizza.

Funcionalidade

1. Renderização do Gráfico:

- O gráfico é gerado usando os dados passados pela prop `data`. Cada item da lista de dados contribui com uma fatia do gráfico de pizza, onde o valor de `value`

determina o tamanho da fatia e `label` é exibido na legenda.

2. Personalização de Cores:

- As cores das fatias são definidas manualmente no array `backgroundColor` com uma lista de cores específicas (`#3d4594`, `#ae832d`, `#020252`).

Exemplo de Uso

Inclua o `PieChart` passando uma lista de dados, onde cada item tem um `label` e um `value`.

```
import React from 'react';

import PieChart from '../components/PieChart';

const Dashboard = () => {

  const data = [

    { label: 'Categoria A', value: 30 },

    { label: 'Categoria B', value: 50 },

    { label: 'Categoria C', value: 20 },

  ];

  return (

    <div>
```




```
        <h3>Gráfico de Pizza</h3>


        <PieChart data={data} />

    </div>

    );

};
```

 Componentes personalizados

 Hooks personalizados

1.17 ProtectedRoute (Just Logic)

O componente `ProtectedRoute` é responsável por proteger rotas na aplicação, garantindo que somente usuários autenticados possam acessá-las. Caso o usuário não esteja autenticado, ele é redirecionado automaticamente para a página de login. Ele utiliza o contexto de autenticação (`AuthContext`) para verificar o estado de autenticação do usuário e, com base nisso, decide se o conteúdo da rota pode ser exibido ou se deve ocorrer o redirecionamento.

Dependências

- `react-router-dom`: Utilizado para manipulação de rotas e redirecionamento condicional.
- **`AuthContext`: Contexto de autenticação que gerencia o estado de login do usuário.**

Props

- `children` (`ReactNode`, obrigatório): O conteúdo que será renderizado se o usuário estiver autenticado, normalmente a página ou componente protegido.

Funcionalidade

1. Verificação de Autenticação:
 - O hook `useAuth` é utilizado para acessar o estado de autenticação (`isAuthenticated`) do contexto `AuthContext`.
2. Redirecionamento Condicional:

- Se `isAuthenticated` for verdadeiro, o conteúdo dos `children` passados como prop é renderizado, permitindo o acesso à rota protegida.
- Se `isAuthenticated` for falso, o usuário é redirecionado para a página de login (`/login`) usando o componente `Navigate` do `react-router-dom`.

Exemplo de Uso

No App.js, ou o arquivo principal importe este componente e proteja as views de usuários não autenticados

```
import React from 'react';

import { Route, Routes } from 'react-router-dom';

import ProtectedRoute from './components/ProtectedRoute';

import Dashboard from './pages/Dashboard';

import Login from './pages/Login';

function App() {

  return (


    <Routes>

      <Route path="/login" element={<Login />} />

      <Route

        path="/dashboard"


        element={
```


 Componentes personalizados

 Hooks personalizados

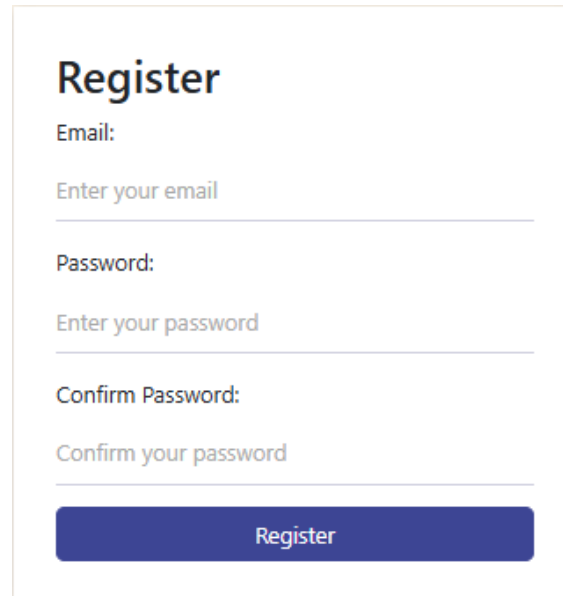
```
        <ProtectedRoute>
            <Dashboard />
        </ProtectedRoute>
    }
    />
</Routes>
);
}

export default App;
```

 Componentes personalizados

 Hooks personalizados

1.18 RegisterForm (UI)




Descrição

O componente `RegisterForm` é um formulário de registro de usuário que permite a criação de uma conta. Ele inclui campos para o email, senha e confirmação de senha. O componente também lida com mensagens de erro e um estado de carregamento, exibindo o progresso enquanto o formulário está sendo submetido. A tradução de textos é feita usando o hook `useTranslation` da biblioteca `react-i18next`.

Dependências

- **react-bootstrap**: Para os componentes de UI como `Form`, `Button`, e `Container`.
- **react-i18next**: Para gerenciar traduções e internacionalização.
- **theme**: Estilos de tema personalizados aplicados ao botão.

 Componentes personalizados

 Hooks personalizados

Props

- **email (string, obrigatório):** O estado que contém o email inserido pelo usuário.
- **setEmail (function, obrigatório):** Função que atualiza o estado do email no componente pai.
- **password (string, obrigatório):** O estado que contém a senha inserida pelo usuário.
- **setPassword (function, obrigatório):** Função que atualiza o estado da senha no componente pai.
- **confirmPassword (string, obrigatório):** O estado que contém a confirmação da senha inserida pelo usuário.
- **setConfirmPassword (function, obrigatório):** Função que atualiza o estado da confirmação de senha no componente pai.
- **handleSubmit (function, obrigatório):** Função chamada quando o formulário é submetido.
- **loading (boolean, obrigatório):** Indica se a requisição de registro está em andamento. Desabilita o botão de envio enquanto estiver `true`.
- **error (string, opcional):** Mensagem de erro a ser exibida se houver problemas ao submeter o formulário.

Funcionalidade

1. Campos do Formulário:

- O formulário inclui três campos obrigatórios: `email`, `senha`, e `confirmação de senha`. Todos os campos são controlados por estados no componente pai.

2. Envio do Formulário:

- Quando o formulário é submetido, a função `handleSubmit` é chamada para processar a submissão.

3. Exibição de Mensagens de Erro:

- Caso ocorra um erro no processo de registro, a mensagem de erro é exibida abaixo do formulário.

4. Desabilitação do Botão de Envio:

- O botão de envio fica desabilitado enquanto a variável `loading` for `true`.

5. Tradução:

- O componente usa `useTranslation` para adaptar o texto do formulário de acordo com o idioma selecionado na aplicação.

Exemplo de Uso

```
import React, { useState } from 'react';

import RegisterForm from './components/RegisterForm';

const RegisterPage = () => {

  const [email, setEmail] = useState('');

  const [password, setPassword] = useState('');


  const [confirmPassword, setConfirmPassword] = useState('');

  const [loading, setLoading] = useState(false);

  const [error, setError] = useState('');


  const handleSubmit = (e) => {

    e.preventDefault();
```

 Componentes personalizados

 Hooks personalizados

```
// Lógica de submissão do formulário

};

return (

<RegisterForm

    email={email}

    setEmail={setEmail}

    password={password}

    setPassword={setPassword}

    confirmPassword={confirmPassword}

    setConfirmPassword={setConfirmPassword}

    handleSubmit={handleSubmit}

    loading={loading}


    error={error}

/>

);

};

export default RegisterPage;
```

 Componentes personalizados

 Hooks personalizados

1.19 Scores (UI)

Bateria 10 %	Concentração 5 ppb	Temperatura 25 °C	Umidade 80 %
-----------------	-----------------------	----------------------	-----------------

O componente **Scores** exibe uma lista de pontuações ou métricas, representando dados de diferentes categorias. Para cada item, o componente mostra o rótulo, o valor, a unidade de medida e um status visual que indica se o valor está no seu estado máximo, mínimo ou dentro do intervalo esperado.

Dependências

- **React**: Biblioteca principal para a construção do componente.
- **./Scores.css**: Arquivo de estilos personalizado para o componente.

Props

- **scores (array de objetos, obrigatório)**: Uma lista de objetos contendo as pontuações ou métricas a serem exibidas. Cada objeto deve conter:
 - **label (string)**: O nome ou título da pontuação.
 - **value (number)**: O valor da pontuação.
 - **unit (string)**: A unidade de medida associada à pontuação.
 - **status (string)**: O estado da pontuação, que pode ser um dos seguintes:
 - **'max'**: A pontuação atingiu o valor máximo (indicado por uma seta para cima).

- `'min'`: A pontuação atingiu o valor mínimo (indicado por uma seta para baixo).
- `'ok'`: A pontuação está dentro do intervalo esperado (indicado por um check).

Funcionalidade

- **Renderização da Lista de Pontuações:** O componente mapeia o array `scores` e exibe cada item com o rótulo, valor e unidade de medida.
- **Indicação de Status:** Dependendo do valor de `status` de cada item:
 - Se for `'max'`, uma seta para cima (▲) é exibida.
 - Se for `'min'`, uma seta para baixo (▼) é exibida.
 - Se for `'ok'`, um ícone de check (✓) é exibido.

Exemplo de Uso

```
import React from 'react';

import Scores from './components/Scores';

const Dashboard = () => {


  const scores = [

    { label: 'Performance', value: 85, unit: '%', status: 'ok'
  },

    { label: 'Uptime', value: 99.9, unit: '%', status: 'max' },


    { label: 'Errors', value: 5, unit: 'count', status: 'min' }


  ];
```

 Componentes personalizados

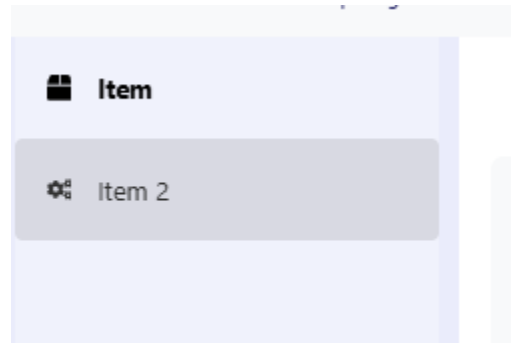
 Hooks personalizados

```
    return (  
      <div>  
        <h2>Dashboard</h2>  
        <Scores scores={scores} />  
      </div>  
    );  
};  
  
export default Dashboard;
```

 Componentes personalizados

 Hooks personalizados

1.20 SidebarMenu (UI)



O componente `SidebarMenu` é uma barra lateral que exibe uma lista de itens. Quando um item é selecionado, o componente notifica o componente pai por meio de uma função de callback.


Dependências

- **React**: Biblioteca principal para a construção do componente.
- **react-icons**: Biblioteca para ícones. No caso, são utilizados os ícones `FaBox` e `FaCogs`.
- **./SidebarMenu.css**: Arquivo de estilos personalizado para o componente, que define a aparência dos itens do menu.

Props

- **onSelectItemType (function, obrigatório)**: Função de callback chamada quando um item do menu é selecionado. Ela recebe o tipo do item selecionado como argumento.

Estado Local

 Componentes personalizados

 Hooks personalizados

- **selectedItem (string):** Armazena o tipo de item atualmente selecionado. Esse estado é utilizado para determinar qual item do menu deve ser destacado visualmente.

Funcionalidade

- **Seleção de Item:** O componente possui dois itens ("item" e "item 2") que podem ser selecionados. Quando um item é selecionado, o estado `selectedItem` é atualizado para refletir o tipo do item selecionado.
- **Notificação ao Componente Pai:** O componente chama a função `onSelectItemType` passada como prop para notificar o componente pai sobre a seleção do item. Isso permite que o pai execute ações com base na seleção.
- **Estilo de Seleção:** Quando um item é selecionado, uma classe CSS adicional (`selected`) é aplicada ao botão correspondente, alterando sua aparência visual.
- **Tamanho Controlado pelo Componente Pai:** O tamanho do `SidebarMenu` (como largura ou altura) deve ser definido pelo componente pai. O `SidebarMenu` em si não tem controle sobre seu tamanho, sendo adaptável ao espaço fornecido.


Exemplo de Uso

```
import React, { useState } from 'react';

import SidebarMenu from './components/SidebarMenu';


const Dashboard = () => {


  const [selectedItem, setSelectedItem] = useState(null);
```

 Componentes personalizados

 Hooks personalizados

```
const handleItemSelect = (itemType) => {  
    setSelectedItem(itemType); // Atualiza o item selecionado  
};  
  
return (  
    <div className="dashboard" style={{ display: 'flex' }}>  
        <SidebarMenu onSelectItemType={handleItemSelect} />  
        <div>  
            <h3>Item Selecionado: {selectedItem}</h3>  
        </div>  
    </div>  
    );  
};  
  
export default Dashboard;
```

 Componentes personalizados

 Hooks personalizados

1.21 Theme (UI + Self-sufficient)

O objeto `theme` contém uma série de cores personalizadas que são utilizadas para estilizar os componentes da aplicação.

Propriedades do Tema

- **primary (string)**: Cor principal utilizada para destaque. Exemplo de uso: botões de ação principal ou links importantes.
 - Valor: `#3d4594` (Azul escuro)
- **secondary (string)**: Cor secundária vibrante, usada para botões de ação ou alertas.
 - Valor: `#f58a3d` (Laranja vibrante)
- **terciary (string)**: Cor terciária, útil para detalhes ou contrastes sutis.
 - Valor: `#ae832d` (Marrom claro)
- **accent (string)**: Cor de destaque mais suave, ideal para links ou elementos chamativos.
 - Valor: `#6c7ae0` (Azul acinzentado)
- **background (string)**: Cor de fundo padrão para a interface.
 - Valor: `#f4f4f9` (Fundo claro e neutro)
- **backgroundWhite (string)**: Cor de fundo extra-clara, geralmente utilizada em elementos como cards ou áreas que precisam de um contraste forte com o fundo geral.
 - Valor: `#ffffff` (Branco)
- **backgroundDark (string)**: Cor de fundo escura, especialmente útil para modo noturno ou quando a interface exige um fundo mais profundo.
 - Valor: `#2d2d3a` (Escuro)
- **backgroundGray (string)**: Cor de fundo intermediária, ideal para áreas secundárias ou para suavizar a interface.


- Valor: `#cbcbcb` (Cinza claro)
- **backgroundBlue (string)**: Um tom suave de azul, adequado para destacar áreas menos prioritárias ou como fundo de componentes secundários.
 - Valor: `#eaecfb` (Azul claro)
- **textPrimary (string)**: Cor do texto principal, ideal para a maior parte do conteúdo textual.
 - Valor: `#222222` (Cinza escuro)
- **textSecondary (string)**: Cor para texto secundário, usada para subtítulos, descrições e informações menos relevantes.
 - Valor: `#595959` (Cinza médio)
- **textTerciary (string)**: Cor de texto terciária, geralmente utilizada para textos mais suaves e pouco importantes.
 - Valor: `#d1d1e0` (Cinza claro)
- **border (string)**: Cor para bordas e divisores. Ideal para separar visualmente os elementos de forma discreta.
 - Valor: `#d1d1e0` (Cinza claro)
- **highlight (string)**: Cor de destaque, normalmente usada para efeitos de hover ou alertas.
 - Valor: `#f9d29d` (Amarelo suave)
- **darkblue (string)**: Um tom muito escuro de azul, ideal para detalhes ou elementos que exigem uma aparência sólida e robusta.
 - Valor: `#020252` (Azul muito escuro)

Exemplo de Uso

```
import theme from './Theme/theme';

const Button = () => {

  return (
```

 Componentes personalizados

 Hooks personalizados


```
<button style={{ backgroundColor: theme.primary, color:
theme.textPrimary }}>
```

```
  Clique Aqui
```

```
</button>
```

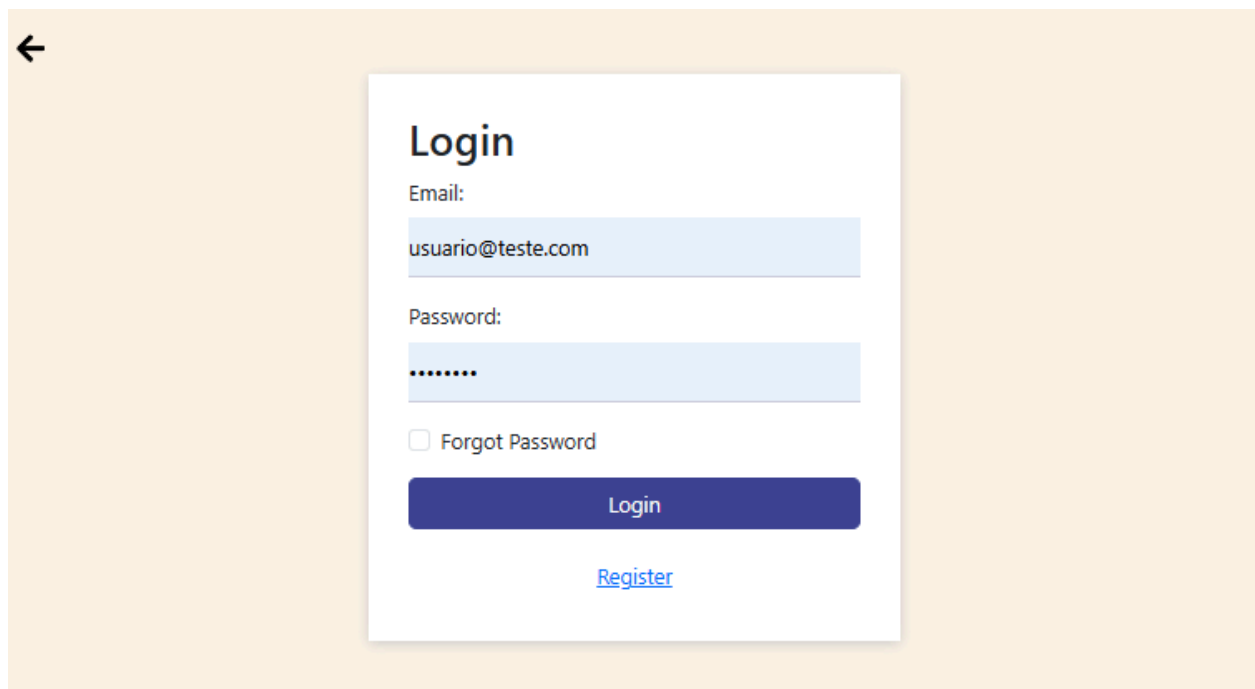
```
);
```

```
};
```

2. Views


2.1 LoginPage (NEED CONFIGURE HOOK)

*A página somente simula o login, para ser funcional substitua os dados mockados de "hooks/useAuthLogin" para sua API, lembre-se de seguir o mesmo response body (user, token, role etc ...)



A **LoginPage** é uma view que contém a interface para o login de usuários, combinando um formulário de autenticação e a lógica associada. Esta página é acessada pela rota `/login`. Após um login bem-sucedido, o usuário é redirecionado para a página principal do aplicativo.

Dependências:

 Componentes personalizados

 Hooks personalizados

- **React** (`useState`): Para gerenciar o estado local da página (email e senha).
- **React Router DOM** (`useNavigate`): Para redirecionar o usuário após o login bem-sucedido.
- **AuthContext** (`useAuth`): Para acessar a função `handleLogin` e os estados relacionados ao login (como `loading` e `error`).
- **LoginForm**: Componente que renderiza o formulário de login e gerencia a entrada do usuário.
- **BackButton**: Componente que permite ao usuário voltar para a página anterior.

Função `handleSubmit`:

- Responsável por capturar o envio do formulário de login. Realiza a chamada à função `handleLogin` do `AuthContext` para autenticar o usuário. Se o login for bem-sucedido, o usuário é redirecionado para a página principal.

Estado Local:

- `email`: Armazena o email digitado pelo usuário.
- `password`: Armazena a senha digitada pelo usuário.

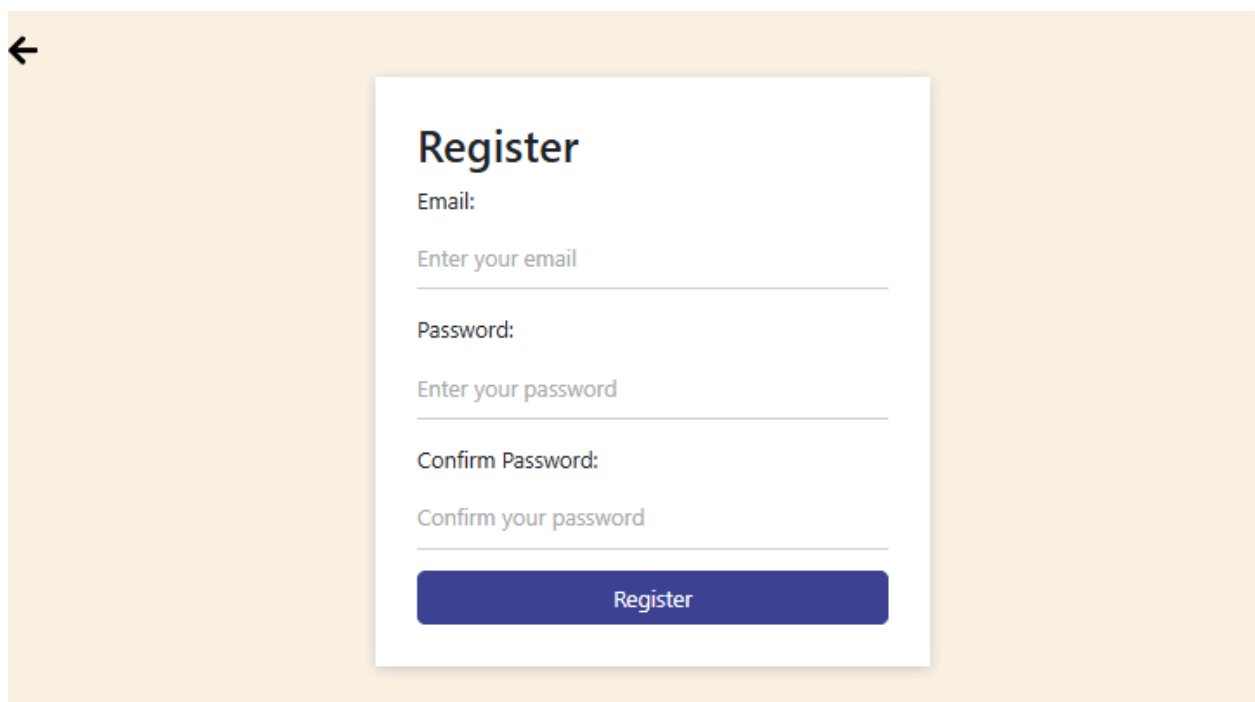
Fluxo:

1. O usuário preenche o email e a senha.
2. Ao submeter o formulário, o `handleSubmit` é acionado, autenticando o usuário.
3. Se o login for bem-sucedido, o usuário é redirecionado para a página inicial.

2.2 RegisterPage (NEED UPDATE CONTEXT + HOOK)

*A página somente simula o registro, para ser funcional adicione um método para registro no "context/AuthContext" e também no "hooks/useAuthLogin (API)" .


Ou se preferir crie outro hook como useAuthRegister (API)





Descrição:

A **RegisterPage** é uma *view* responsável pela interface de registro de usuários. Ela utiliza um formulário de registro, lida com a validação das informações e simula a criação de um novo usuário. Após o registro bem-sucedido, o usuário é redirecionado para a página principal.

Dependências:

 Componentes personalizados

 Hooks personalizados

- **React** (`useState`): Para gerenciar o estado local da página (dados do formulário, mensagens de sucesso/erro, etc.).
- **React Router DOM** (`useNavigate`): Para redirecionar o usuário após o registro bem-sucedido.
- **React Bootstrap** (`Alert`): Para exibir mensagens de erro ou sucesso.
-  **RegisterForm**: Componente que renderiza o formulário de registro e recebe os dados do usuário.
-  **BackButton**: Componente que permite voltar para a página anterior.

Função `handleRegister` :


(substitua o conteúdo daqui pelo método que você criar no `context`)

- Responsável por capturar o envio do formulário de registro. Valida se a senha e a confirmação da senha são iguais e simula uma chamada de API para criar o usuário. Se bem-sucedido, exibe uma mensagem de sucesso e redireciona o usuário para a página inicial.

Estado Local:

- `email`: Armazena o email digitado pelo usuário.
- `password`: Armazena a senha digitada pelo usuário.
- `confirmPassword`: Armazena a confirmação da senha.
- `loading`: Indica se o processo de registro está em andamento.
- `error`: Armazena a mensagem de erro, caso ocorra.
- `successMessage`: Armazena a mensagem de sucesso, caso o registro seja bem-sucedido.

Fluxo:

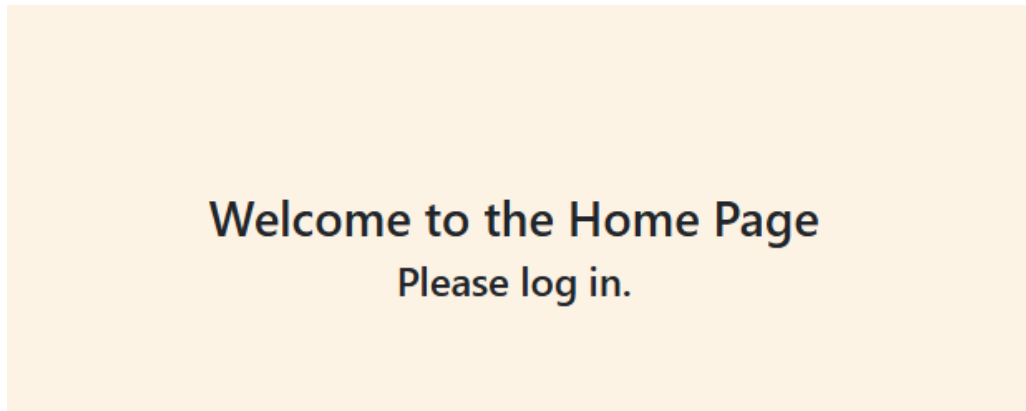
 Componentes personalizados

 Hooks personalizados

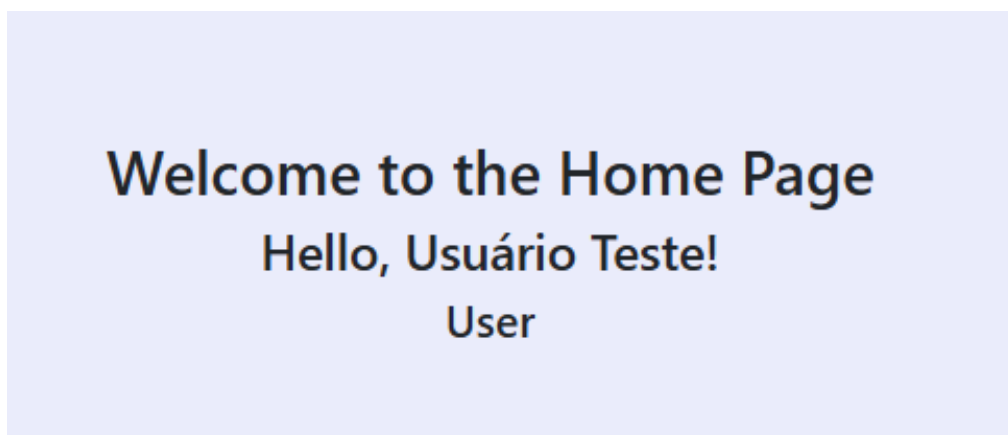
1. O usuário preenche os dados do formulário de registro (email, senha e confirmação).
2. Ao submeter o formulário, o `handleRegister` é acionado.
3. A validação das senhas é realizada, e a simulação de um registro na API é feita.
4. Se o registro for bem-sucedido, o usuário é redirecionado para a página principal.
5. Em caso de erro (como senhas não coincidentes), uma mensagem de erro é exibida.

2.3 HomePage

Com usuário não autenticado




Com usuário autenticado



A **HomePage** é a página principal após o login, responsável por exibir uma mensagem de boas-vindas personalizada ao usuário, juntamente com seu nome e cargo/role (se autenticado). Caso o usuário não esteja autenticado, é exibida uma mensagem solicitando o login.

Dependências:

 Componentes personalizados

 Hooks personalizados

- React: Para a criação do componente.
- **AuthContext (useAuth): Para acessar os dados do usuário autenticado (nome e cargo).**
- React-i18next (useTranslation): Para realizar a tradução dinâmica de textos na interface, com base na configuração de idioma atual.

Função:

- Exibe uma saudação personalizada com o nome do usuário e seu cargo, caso esteja autenticado.
- Se o usuário não estiver autenticado, exibe uma mensagem solicitando o login.

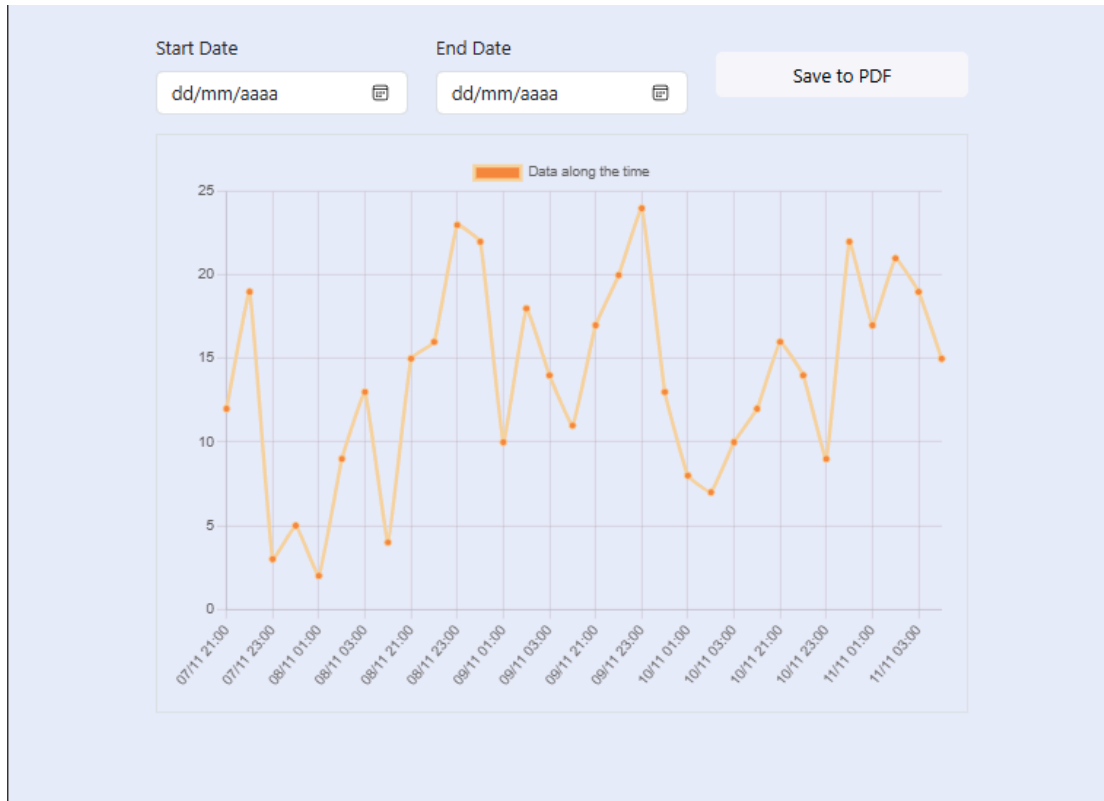
Estado:

- Não há estado local. A página depende do estado de autenticação global (contexto **AuthContext**).

Fluxo:

1. A página exibe uma saudação de boas-vindas traduzida, de acordo com o idioma selecionado.
2. Se o usuário estiver autenticado, é exibido o nome do usuário e seu cargo (também traduzido).
3. Se o usuário não estiver autenticado, é exibida uma mensagem solicitando que ele faça login.

2.4 GraphPage



A **GraphPage** é a página que exibe um gráfico de linha. Ela utiliza o componente **LineGraph** para renderizar um gráfico interativo ou estático, dependendo da implementação.

Dependências:

- React: Para a criação do componente.
- **LineGraph**: Componente de gráfico de linha.

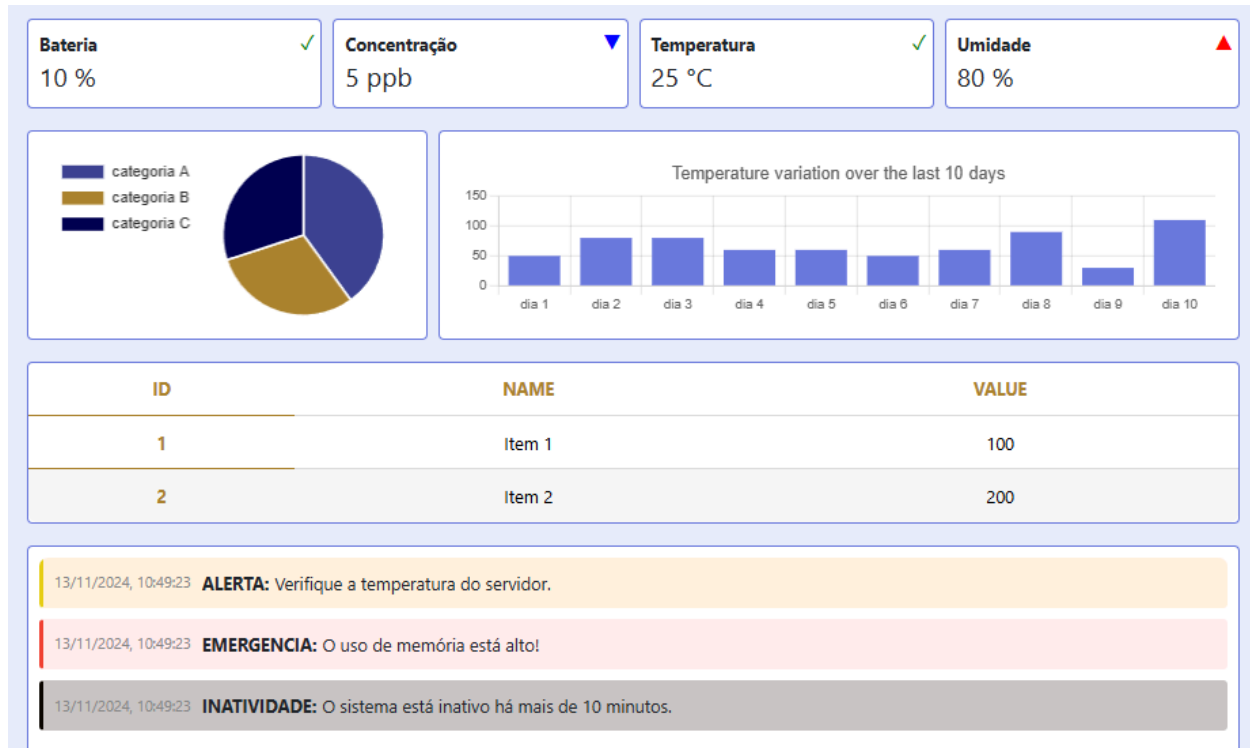
Função:

- Exibe um gráfico de linha no conteúdo da página, utilizando o componente **LineGraph**.

Componentes personalizados

Hooks personalizados

2.5 DashboardPage




A **DashboardPage** é uma página que exibe o componente **Dashboard**. Esta página é um contêiner simples, onde o componente **Dashboard** é renderizado.

Dependências:

- **React**: Para a criação do componente funcional.
- **Dashboard**: Componente que exibe o painel de controle, podendo incluir gráficos, tabelas e informações resumidas.

Função:

 Componentes personalizados

 Hooks personalizados

- Exibe o painel de controle (dashboard), sem qualquer estado local ou lógica adicional.

Fluxo:

1. A página renderiza o componente `Dashboard` dentro de um contêiner flexível.
2. O contêiner ocupa 100% da altura disponível na tela devido ao uso da classe `flex-grow-1`.

2.6 CRUDPage (NEED CONFIGURE HOOK)

Telas grandes

Item

Item 2

Item

Name

Enter item name

✓ [Create new item](#)

ID	Name	Actions
1	Item 1	<div><div></div><div></div></div>
2	Item 2	<div><div></div><div></div></div>
3	Item 3	<div><div></div><div></div></div>

Telas menores

Item


Item 2


Name

Enter item name

✓ [Create new item](#)







ID	Name	Actions
1	Item 1	<div><div></div><div></div></div>
2	Item 2	<div><div></div><div></div></div>
3	Item 3	<div><div></div><div></div></div>

 Componentes personalizados


 Hooks personalizados

A `CRUDView` é uma página que oferece uma interface para a manipulação de itens (CRUD). Ela permite `criar, ler, atualizar, excluir` e ativar dois tipos de itens, denominados `"item"` e `"item2"`. O componente inclui um `sidebar` para navegação entre os tipos de itens e a exibição das respectivas tabelas e formulários.

Dependências:

- React: Para a criação do componente funcional e gerenciamento de estado.
-  `useCRUDData`: Hook que gerencia a lógica de CRUD para os diferentes tipos de item (item e item2)
 - ***Atualize para sua API, é possível usar este hook para operar mais de um elemento/endpoint.**
-  `SidebarMenu`: Componente que renderiza um menu lateral para seleção do tipo de item.
-  `ItemTable*`: Componente que exibe os dados dos itens em formato de tabela.
-  `ItemForm*`: Componente para criação e edição de itens.
-  `Item2Table*`: Componente que exibe os dados de "item2" em formato de tabela.
-  `Item2Form*`: Componente para criação e edição de "item2".
- React-Bootstrap: Para layout e UI, como `Container`, `Row`, `Col`, `Tabs`, e `Tab`.

***Atualize estes itens para se encaixar no seu banco de dados ou para sua model/tabela, {nome-da-model}Form e {nome-da-model}Table, exemplo UsersForm e UsersTable.**

 Componentes personalizados

 Hooks personalizados

Estado:

- `itemType`: Controla o tipo de item atualmente selecionado ("item" ou "item2").
- `data`: Dados dos itens obtidos por meio do hook `useCRUDData`.
- `itemToEdit`: Armazena o item que está sendo editado.
- `isSidebarVisible`: Controla a visibilidade do sidebar com base no tamanho da tela.

Funções:

- `handleCreateOrUpdate`: Função responsável por criar ou atualizar um item.
- `handleEdit`: Função chamada ao editar um item, passando o item para a edição.
- `handleDelete`: Função responsável pela exclusão de um item.
- `handleCancelEdit`: Cancela a edição do item.
- `handleActivate`: Ativa um item (caso de "item2").
- `handleItemTypeChange`: Altera o tipo de item a ser exibido (de "item" para "item2" ou vice-versa).

Responsabilidade:

- A `CRUDView` combina os componentes de formulário e tabela, permitindo realizar operações CRUD para dois tipos de dados distintos.
- Exibe os dados em uma tabela e fornece formulários para criar ou editar itens.

- O sidebar alterna sua visibilidade com base no tamanho da tela (oculta em telas pequenas).
- Alterna entre os tipos de item e exibe diferentes formulários e tabelas conforme o tipo selecionado.

Fluxo:

- 1.0 `SidebarMenu` permite que o usuário escolha o tipo de item a ser gerenciado.
- 2.0 tipo de item selecionado determina qual tabela e formulário serão exibidos.
- 3.0 conteúdo da página é exibido de forma responsiva, com um sidebar em telas maiores e com abas quando o sidebar é oculto.
- 4.0 estado `itemToEdit` é usado para gerenciar a edição de um item existente.
- 5.0 hook `useCRUDData` gerencia as operações de CRUD para o tipo de item selecionado.

3. Hooks

3.1 useAuthLogin

O hook `useAuthLogin` gerencia a autenticação do usuário, realizando o login através de uma API externa (ou simulação de autenticação). Ele mantém os estados de carregamento (`loading`) e de erro (`error`) para controlar o fluxo da requisição de login e exibir mensagens apropriadas para o usuário.

Dependências:

- **React**: Para o uso do `useState` e gerenciamento do estado interno.
- **Fetch**: Embora comentado, é a ferramenta que seria usada para a requisição à API externa (caso a implementação real fosse ativada).

Estado:

- **loading**: Indica se o processo de autenticação está em andamento.
- **error**: Armazena mensagens de erro retornadas durante o processo de login.

Funções:

- **login**: Função assíncrona responsável por realizar a autenticação. Ela simula a validação de credenciais ou se conecta com uma API externa.
 - **Parâmetros**:
 - `email`: O endereço de e-mail do usuário.
 - `password`: A senha do usuário.
 - **Retorna**:

- Um objeto contendo o token e os dados do usuário, se a autenticação for bem-sucedida.
- **Lança:**
 - Um erro caso as credenciais estejam incorretas ou ocorra outro problema durante o processo de autenticação.

Fluxo:

1. O hook inicia com a configuração dos estados `loading` (para controle de processo) e `error` (para capturar erros).
2. Quando a função `login` é chamada, o estado `loading` é ativado e o estado `error` é resetado.
3. O processo de autenticação simula a verificação das credenciais. Se as credenciais forem válidas, ele retorna um objeto com um token e os dados do usuário (simulados no momento).
4. Caso as credenciais sejam inválidas, a função lança um erro que é tratado no bloco `catch`.
5. A função `login` retorna o token e os dados do usuário quando o login é bem-sucedido ou lança um erro em caso de falha.

!!!!Observação!!!!:

O código atualmente simula o login com credenciais fixas (`usuario@teste.com` e `senha123`). A implementação comentada com `fetch` pode ser usada para realizar a autenticação real com uma API externa. Para usar a integração com uma API, o trecho com `fetch` precisa ser descomentado e configurado com a URL da API apropriada.

3.2 useCRUDData

O hook `useCRUDData` foi projetado para realizar operações CRUD (Criar, Ler, Atualizar, Deletar) em dados genéricos, permitindo manipular diferentes tipos de "itens" ou "modelos" de dados. Ele permite que você use um único hook para gerenciar dados de diferentes "tabelas" ou "entidades" no seu sistema. Pode ser usado tanto com dados locais simulados quanto com integração a uma API externa.

Dependências:

- **React**: Para o uso de `useState` e `useEffect` para gerenciamento de estados e controle de efeitos colaterais.
- **Fetch (comentado)**: Para realizar requisições à API externa. O código comentado pode ser ativado para integrar com a API real.


Parâmetros:

- **itemType** (opcional, padrão: `'item'`): Tipo do item que será manipulado (ex: `'item'`, `'tipo2'`, `'sensor1'`, `'user'`). Este valor influencia os dados carregados inicialmente e as operações CRUD subsequentes.

Estado:

- **data**: Contém os dados dos itens que estão sendo gerenciados. Inicialmente, são simulados com base no `itemType` e, em um cenário real, podem ser carregados de uma API externa.

Funções:

 Componentes personalizados

 Hooks personalizados

- **createItem**: Cria um novo item localmente e o adiciona ao estado `data`.
 - **Parâmetros**:
 - `item`: Objeto com os dados do item a ser criado.
 - **Efeito**: Adiciona o item ao estado `data` e, opcionalmente, envia o item para a API externa.
- **updateItem**: Atualiza um item existente no estado `data`.
 - **Parâmetros**:
 - `updatedItem`: O item com os dados atualizados.
 - **Efeito**: Atualiza o item no estado `data` e, opcionalmente, envia a atualização para a API externa.
- **deleteItem**: Remove um item do estado `data` com base no seu ID.
 - **Parâmetros**:
 - `id`: O ID do item a ser removido.
 - **Efeito**: Remove o item do estado `data` e, opcionalmente, exclui o item da API externa.

Fluxo:

1. **useEffect**: O hook carrega dados simulados no estado `data` com base no tipo de item (`itemType`). Isso permite que o hook se adapte a diferentes tipos de dados e manipule os itens corretamente.
2. **Operações CRUD**:
 - **Criar**: Ao chamar `createItem`, um novo item é criado localmente e adicionado ao estado `data`. O código para integrar com uma API externa está comentado, mas pode ser facilmente ativado.
 - **Atualizar**: `updateItem` atualiza um item específico no estado `data`. O código de integração com a API está disponível para sincronizar as atualizações.

- **Deletar:** `deleteItem` remove um item do estado `data` e também pode ser configurado para excluir o item da API externa.

!!!!Observação!!!!:

Atualmente, os dados são simulados no hook. Para integrar com uma API real, basta descomentar o código `fetch` presente nas funções `createItem`, `updateItem` e `deleteItem`. A URL da API deve seguir o padrão de `host:port/{itemType}`, onde `{itemType}` é o tipo de item sendo manipulado.

3.3 useDashboardData

O hook `useDashboardData` é responsável por gerenciar os dados necessários para preencher um dashboard. Ele faz chamadas para diferentes funções que simulam a obtenção de dados de uma API (como pontuações, gráficos, tabela e alertas) e gerencia o estado de carregamento dos dados. O hook retorna os dados do dashboard organizados em um único objeto e também indica se os dados ainda estão sendo carregados.

Dependências:

- **React**: Para o uso de `useState` e `useEffect` para gerenciamento de estados e controle de efeitos colaterais.


Funções auxiliares:

- **Modifique o fetch para usar uma Api**, não há problema caso não tenha dados de algum fetch, o componente **Dashboard** apenas oculta o item caso esteja vazio.

- **fetchScores**: Função simulada que retorna os dados de pontuação, como baterias, temperatura, etc.
- **fetchChartData**: Função simulada que retorna dados para gráficos, incluindo gráfico de pizza e gráfico de barras.
- **fetchTableData**: Função simulada que retorna os dados para uma tabela com ID, nome e valor.
- **fetchAlerts**: Função simulada que retorna alertas de sistema, como mensagens de erro ou status.

Estrutura dos Dados: O hook retorna os seguintes dados, organizados em um objeto:

- **Independente da Api procure manter esta estrutura de dados**

 Componentes personalizados

 Hooks personalizados


- **scores:** Um array de objetos contendo dados de pontuação (ex: Bateria, Temperatura, etc.). Cada objeto possui:
 - **label:** Nome do indicador (ex: Bateria).
 - **value:** O valor atual do indicador (ex: 10).
 - **unit:** A unidade do valor (ex: '%').
 - **status:** O status do indicador (ex: 'ok', 'min', 'max').
- **chartData:** Um objeto contendo dados para gráficos. Inclui:
 - **pie:** Dados para um gráfico de pizza, com rótulos e valores (ex: { label: 'categoria A', value: 40 }).
 - **bar:** Dados para um gráfico de barras, com rótulos e valores (ex: { label: 'dia 1', value: 50 }).
- **tableData:** Um array de objetos contendo dados de uma tabela (ex: { id: 1, name: 'Item 1', value: 100 }).
- **alerts:** Um array de objetos contendo alertas, com informações como:
 - **timestamp:** Data e hora do alerta.
 - **type:** O tipo de alerta (ex: 'alerta', 'emergencia', 'inatividade').
 - **message:** A mensagem do alerta.

Estado:

- **data:** Objeto que armazena os dados do dashboard, contendo **scores**, **chartData**, **tableData** e **alerts**.
- **loading:** Booleano que indica se os dados estão sendo carregados (inicialmente **true**).

Fluxo:

1. Carregamento de Dados:

 Componentes personalizados

 Hooks personalizados

- Quando o hook é chamado, o `useEffect` dispara automaticamente, iniciando o carregamento dos dados do dashboard.
- O `loadData` faz chamadas assíncronas para as funções `fetchScores`, `fetchChartData`, `fetchTableData` e `fetchAlerts`, que retornam os dados necessários.

2. Atualização de Estado:

- Assim que os dados são carregados, o estado `data` é atualizado com os dados recebidos e o estado `loading` é alterado para `false`.

3. Exceções:

- Se ocorrer um erro durante o carregamento dos dados, o erro é capturado e exibido no console. O estado `loading` é configurado como `false`, mesmo que a requisição falhe.

3.4 useForgotPassword

O hook `useForgotPassword` gerencia o processo de recuperação de senha em três etapas:


1. Envio de email para recuperação de senha.
2. Validação de um token enviado para o usuário.
3. Atualização da senha.


Estados do Hook

- **step**: A etapa atual do processo (1: Envio de email, 2: Validação do token, 3: Atualização da senha).
 - **username**: Email ou nome de usuário.
 - **setUsername**: Função para atualizar o `username`.
 - **token**: Token de recuperação.
 - **setToken**: Função para atualizar o `token`.
 - **newPassword**: Nova senha.
 - **setNewPassword**: Função para atualizar a `newPassword`.
 - **message**: Mensagem de sucesso ou erro.
 - **error**: Erro retornado durante o processo.
 - **loading**: Indicador de carregamento.
-

Funções do Hook (substitua por sua API)

- **handleForgotPassword**: Inicia o processo de recuperação de senha enviando um email com um token. Simula o envio com `setTimeout`.
- **handleValidateToken**: Valida o token enviado ao usuário. Simula a validação com `setTimeout`.
- **handleUpdatePassword**: Atualiza a senha do usuário. Simula a atualização com `setTimeout`.

 Componentes personalizados

 Hooks personalizados

3.5 useGraph

O hook `useGraph` gerencia a obtenção de dados para gráficos (com timestamps e valores) e lida com o estado de carregamento e erros.

Estados do Hook

- **data**: Objeto contendo:
 - **timestamps**: Array de timestamps (datas e horas) para os dados do gráfico.
 - **values**: Array de valores correspondentes a cada timestamp.
- **loading**: Indicador de carregamento (verdadeiro enquanto os dados estão sendo carregados).
- **error**: Erro retornado caso algo falhe ao buscar os dados.

Funções do Hook

- **fetchData**: Função assíncrona que simula a obtenção dos dados fixos para o gráfico (simulação de requisição com `setTimeout`). **Ela pode ser substituída por uma chamada real à API.**

3.6 useNotifications

O hook `useNotifications` gerencia a busca e o estado de notificações, incluindo os estados de carregamento e erro.

Estados do Hook

- **notifications**: Lista de notificações recebidas. Cada notificação possui:
 - **id**: Identificador único da notificação.
 - **timestamp**: Hora da notificação.
 - **message**: Mensagem da notificação.
 - **type**: Tipo da notificação (ex: `info`, `alert`, `update`).
- **loading**: Estado de carregamento (indica se os dados estão sendo buscados).
- **error**: Erro que ocorre caso a busca pelas notificações falhe.

Funções do Hook

- **fetchNotifications**: Função assíncrona que simula a obtenção de notificações com `setTimeout`. **Ela pode ser substituída por uma chamada real à API.**

3.7 useWindowSize

O hook `useWindowSize` fornece as dimensões atuais da janela do navegador (largura e altura) e atualiza esses valores sempre que a janela é redimensionada.

Estado do Hook

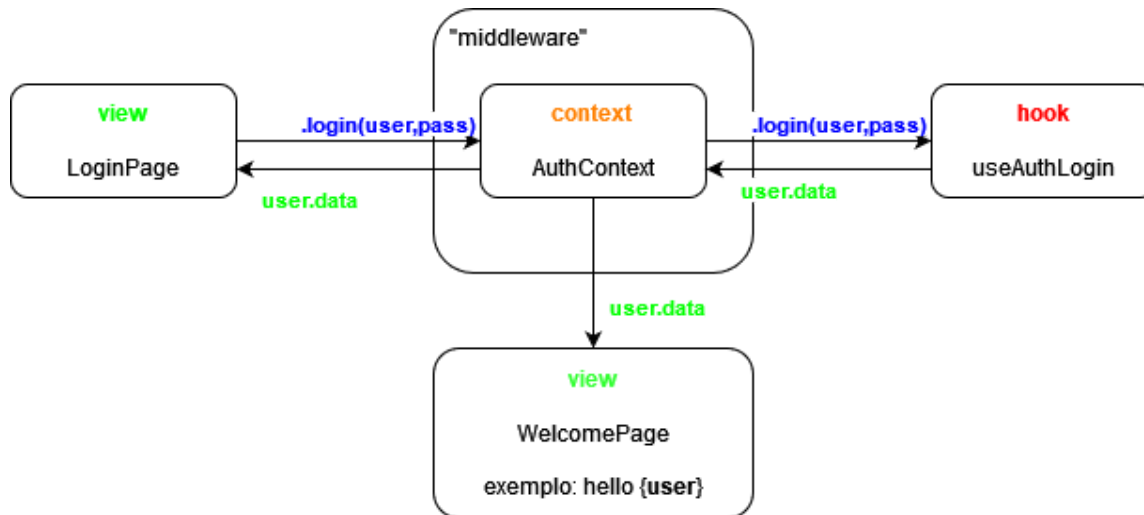
- **`windowSize`**: Objeto contendo as dimensões da janela:
 - **`width`**: Largura da janela (em pixels).
 - **`height`**: Altura da janela (em pixels).

Funcionamento

- O hook utiliza o `window.innerWidth` e `window.innerHeight` para capturar as dimensões iniciais da janela.
- Ele escuta o evento `resize` no `window` para atualizar as dimensões sempre que a janela for redimensionada.
- A função de limpeza (`return` no `useEffect`) remove o ouvinte de eventos quando o componente for desmontado, evitando vazamentos de memória.

4. Contexts ***Importante de para login e nível de acesso**

AuthContext



O **AuthContext** é um contexto React que gerencia o estado de autenticação do usuário e fornece métodos para fazer login e logout. Esse contexto **encapsula o hook** `useAuthLogin` e realiza **armazenamento de dados do usuário no localStorage** para persistência de sessão.

1. Contexto: AuthContext

- **Descrição:** `AuthContext` armazena e distribui o estado de autenticação do usuário, além de prover o background específico com base no status da autenticação.

2. Provider: AuthProvider

- **Descrição:** Componente Provider que envolve os componentes filhos e distribui as funções e estados de autenticação.
- **Propriedades:**
 - `children`: Componentes filhos que terão acesso ao contexto.

■ Componentes personalizados

■ Hooks personalizados

3. Valores Fornecidos pelo Contexto:

- **user**: Objeto com informações do usuário autenticado (ou `null` se não autenticado).
- **isAuthenticated**: Booleano que indica se o usuário está autenticado.
- **handleLogin**: **Função assíncrona para realizar login com a API** e armazenar dados no estado e `localStorage`.
 - **Parâmetros**:
 - `email` (string): Email do usuário.
 - `password` (string): Senha do usuário.
 - **Retorno**: Popula o `user` e `authToken` no `localStorage` e atualiza `isAuthenticated`.
- **handleLogout**: Função síncrona que encerra a sessão do usuário, limpa dados do `localStorage` e redefine `user` e `isAuthenticated`.
- **loading**: Estado de carregamento da requisição de login (importado de `useAuthLogin`).
- **error**: Estado de erro da requisição de login (importado de `useAuthLogin`).
- **bgColor**: **Cor de background, que muda dependendo do status de autenticação.**

4. Hook: `useAuth`

- **Descrição**: Hook personalizado que encapsula `useContext(AuthContext)` para facilitar o acesso ao contexto.
 - **Uso**:

```
const { user, isAuthenticated, handleLogin, handleLogout, loading, error, bgColor } = useAuth();
```
-

Fluxo de Autenticação

1. Login

- `handleLogin(email, password)`: Chama o `login` de `useAuthLogin`, armazena `user` e `token` no estado e `localStorage`, e define `isAuthenticated` como `true` se as credenciais forem válidas.

2. Logout

- `handleLogout()`: Reseta o estado de `user` e `isAuthenticated`, além de remover o `user` e `authToken` do `localStorage`.

3. Persistência de Sessão

- Ao carregar o aplicativo, o `useEffect` verifica `localStorage` para restaurar `user` e `authToken`, se existirem, e define `isAuthenticated`.

6. Services ***Importante de para tradução de textos**

1. Language

- a. Atualize os arquivos `en.json` e `pt.json`, com as novas palavras ou termos que devem ser traduzidos, lembrando que cada chave é única.
- b. É possível adicionar mais linguagens modificando o arquivo `language.js`, mas lembre-se de atualizar o componente `LanguageSwitcher`.


7. Sugestão de API's do Backend

O **Content-Type** esperado é:

- ``application/json``: Para todas as requisições e respostas que envolvem dados JSON.

-**Autenticação**: Para projetos mais robustos, algumas APIs podem exigir um token JWT (JSON Web Token) no cabeçalho, **não é obrigatório (é só um exemplo em API:NOTIFICAÇÕES)**, é só uma forma de segurança

*Para evitar grandes modificações nos hooks e componentes procure manter a estrutura das responses e parâmetros mais idênticas possíveis a estes exemplos.

 Componentes personalizados

 Hooks personalizados

1. API: Recuperação de Senha (Forgot Password)

****sugestão com base no hook `useForgotPassword`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: **POST**

Endpoint: **`/forgot-password`**

Parâmetros (Body):

```
{  
  "email": "<EMAIL>"  
}
```

Resposta:

- **Status 200 (OK)**: Caso o email seja encontrado, um token será enviado para o email.

```
{  
  "message": "Token enviado para o email <EMAIL>"  
}
```

- **Status 404 (Not Found)**: Caso o email não seja encontrado no sistema.

```
{  
  "error": "Email não encontrado."  
}
```

2. API: Validação de Token

****sugestão com base no hook `useForgotPassword`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: **POST**

Endpoint: **/validate-token**

Parâmetros (Body):

```
{  
  "token": "<TOKEN>"  
}
```

Resposta:

- **Status 200 (OK):** Caso o token seja válido.

```
{  
  "message": "Token válido, insira a nova senha."  
}
```
- **Status 400 (Bad Request):** Caso o token seja inválido.

```
{  
  "error": "Token inválido."  
}
```

3. API: Atualização de Senha

****sugestão com base no hook `useForgotPassword`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: **POST**

Endpoint: **`/update-password`**

Parâmetros (Body):

```
{  
  "token": "<TOKEN>",  
  "newPassword": "<NEW_PASSWORD>"  
}
```

Resposta:

- **Status 200 (OK):** Caso a senha seja atualizada com sucesso.

```
{  
  "message": "Senha atualizada com sucesso."  
}
```
- **Status 400 (Bad Request):** Caso a senha não atenda aos requisitos.

```
{  
  "error": "A senha não atende aos requisitos."  
}
```

4. API: Notificações

****sugestão com base no hook `useNotification`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: **GET**

Endpoint: **`/notifications`**


Parâmetros (Query ou Headers):


- Authorization: ``Bearer <TOKEN>`` (Opcional, para autenticação baseada em token)

Resposta:

- Status 200 (OK): Lista de notificações.

```
[
  {
    "id": 1,
    "timestamp": <TIMESTAMP>,
    "message": "Nova mensagem de sistema!",
    "type": "info"
  },
  {
    "id": 2,
    "timestamp": <TIMESTAMP>,
    "message": "Você tem uma nova solicitação.",
    "type": "alert"
  },
  {
    "id": 3,
    "timestamp": <TIMESTAMP>,
    "message": "Atualização disponível!",
    "type": "update"
  }
]
```

 Componentes personalizados

 Hooks personalizados

```
    }  
  ]  
- Status 500 (Internal Server Error): Caso ocorra um erro ao  
  buscar as notificações.
```

```
{  
  "error": "Erro ao buscar notificações"  
}
```

5. API: Dados do Gráfico

****sugestão com base no hook `useGraph`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: **GET**

Endpoint: **/graph-data**

Parâmetros (Query ou Headers):

- startDate: Data de início (formato `YYYY-MM-DD`)
- endDate: Data de fim (formato `YYYY-MM-DD`)

Resposta:

- **Status 200 (OK):** Dados do gráfico (valores e timestamps).

```
{
  "timestamps": [
    "2024-11-08T00:00:00Z", "2024-11-08T01:00:00Z", ...
  ],
  "values": [
    12, 19, 3, 5, 2, 9, ...
  ]
}
```

- **Status 500 (Internal Server Error):** Caso ocorra um erro ao buscar os dados do gráfico.

```
{
  "error": "Erro ao buscar dados do gráfico"
}
```

6. API: Login

****sugestão com base no hook `useAuthLogin`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: `POST`

Endpoint: `/auth/login`

Parâmetros (Body):

- `email`: (string) Email do usuário.
- `password`: (string) Senha do usuário.

Resposta:

- **Status 200 (OK):** Autenticação bem-sucedida, retorna token e dados do usuário.

```
{  
  "token": "token_simulado_123",  
  "user": {  
    "email": "usuario@teste.com",  
    "name": "Usuário Teste",  
    "role": "user"  
  }  
}
```

- **Status 401 (Unauthorized):** Credenciais incorretas.

```
{  
  "error": "Credenciais incorretas"  
}
```

7. API: Get Item

****sugestão com base no hook `useCRUDData`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: **GET**

Endpoint: **`/crud/{itemType}`**

Parâmetros (Body):

`-itemType`: (string) Tipo de item a ser manipulado (ex: **`item`**, **`tipo2`**).


- Exemplo: **`{SERVERIP}/item`**


Resposta:

- **Status 200 (OK)**: Dados do item (ou lista de itens) solicitados.


```
[
  {
    "id": 1,
    "name": "Item 1"
  },
  {
    "id": 2,
    "name": "Item 2"
  }
]
```


- **Status 500 (Internal Server Error)**: Erro ao tentar recuperar os dados.

 Componentes personalizados

 Hooks personalizados


```
{  
  "error": "Erro ao buscar dados"  
}
```

 Componentes personalizados

 Hooks personalizados

8. API: Criar Item

****sugestão com base no hook `useCRUDData`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: `POST`

Endpoint: `/crud/{itemType}`

Parâmetros (Body): ***Muda conforme objeto/item/model/tabela**

- **Exemplo:**

- `name:` (string, opcional) Nome do item a ser criado.
- `status:` (string, opcional) Status do item (ex: Ativo, Inativo).

Resposta:

- **Status 201 (Created):** Item criado com sucesso.

- **Exemplo:**

```
{
  "id": 3,
  "name": "Novo Item",
  "status": "Ativo"
}
```

9. API: Atualizar Item

****sugestão com base no hook `useCRUDData`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: PUT

Endpoint: /crud/{itemType}/{itemId}

Parâmetros (Body): *Muda conforme objeto/item/model/tabela

- **Exemplo:**

- name: (string, opcional) Novo nome do item.
- status: (string, opcional) Novo status do item (ex: Ativo, Inativo).

Resposta:

- **Status 200 (OK):** Item atualizado com sucesso.
 - **Exemplo:**

```
{
  "id": 3,
  "name": "Item Atualizado",
  "status": "Ativo"
}
```

10. API: Deletar Item

****sugestão com base no hook `useCRUDData`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: **DELETE**

Endpoint: `/crud/{itemType}/{itemId}`

Parâmetros (Query ou Headers):

- `itemId: (int)` ID do item a ser deletado.

Resposta:

- **Status 200 (OK):** Item deletado com sucesso.

```
{  
  "message": "Item deletado com sucesso"  
}
```

11. API: Dados para Scores

****sugestão com base no hook `useDashboardData`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: `GET`

Endpoint: `/dashboard/scores`

Parâmetros:

Nenhum.

Resposta (OBS: Mantenha todas keys iguais, o valor da key "status" deve estar entre "ok", "min" ou "max"):

Status 200 (OK): Retorna os dados de pontuação com os campos `label`, `value`, `unit` e `status`.

```
[
  { "label": "Bateria", "value": 10, "unit": "%", "status":
    "ok" },
  { "label": "Concentração", "value": 5, "unit": "ppb",
    "status": "min" },
  { "label": "Temperatura", "value": 25, "unit": "°C",
    "status": "ok" },
  { "label": "Umidade", "value": 80, "unit": "%", "status":
    "max" }
]
```

12. API: Dados para Gráficos (Pie + Bar)

****sugestão com base no hook `useDashboardData`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: GET

Endpoint: /dashboard/chartdata


Parâmetros:


Nenhum.

Resposta (OBS: Mantenha todas keys iguais):

Status 200 (OK): Retorna os dados para gráficos do tipo pie e bar

```
{
  "pie": [
    { "label": "categoria A", "value": 40 },
    { "label": "categoria B", "value": 30 },
    { "label": "categoria C", "value": 30 }
  ],
  "bar": [
    { "label": "dia 1", "value": 50 },
    { "label": "dia 2", "value": 80 },
    { "label": "dia 10", "value": 110 }
  ]
}
```

 Componentes personalizados

 Hooks personalizados

13. API: Dados para Tabela

****sugestão com base no hook `useDashboardData`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: `GET`

Endpoint: `/dashboard/tableData`

Parâmetros:

Nenhum.

Resposta (OBS: Keys e valores são totalmente flexíveis):

Status 200 (OK):

```
[  
  { "id": 1, "name": "Item 1", "value": 100, "teste": 2 },  
  { "id": 2, "name": "Item 2", "value": 200, "teste": 2 }  
]
```

14. API: Dados para Alertas

****sugestão com base no hook `useDashboardData`, antes de criar a API verifique se é necessário este item no seu projeto****

Método: GET

Endpoint: `/dashboard/alerts`

Parâmetros:

Nenhum.

Resposta (OBS: Mantenha todas keys iguais, o valor da key "type" deve estar entre "alerta", "emergencia" ou "inatividade"):

Status 200 (OK):

```
[
  { "timestamp": "2024-11-19 10:00:00", "type": "alerta",
    "message": "Verifique o servidor." },
  { "timestamp": "2024-11-19 11:00:00", "type":
    "emergencia", "message": "O uso de memória está alto!" },
  { "timestamp": "2024-11-19 12:00:00", "type":
    "inatividade", "message": "O sistema está inativo há mais
    de 50 minutos." }
]
```