

1. Atividades Práticas - Trabalhando com Diferentes Tipos de Inputs

Nesta atividade prática, vamos criar um exemplo simples passo a passo para ilustrar como podemos trabalhar com diferentes tipos de inputs de formulários em uma aplicação React.

1.1 Criando o projeto e iniciando o servidor local

Crie um projeto chamado **compras**, digitando o seguinte comando no terminal:

```
npm create vite@latest my-form-inputs -- --template react
```

Este processo de configuração inicial do projeto leva alguns segundos. Ao terminar, o Vite repassa instruções para que você termine de instalar as dependências do seu projeto. Desta forma, digite o seguinte comando para **entrar no diretório** recém criado do nosso projeto:

```
cd my-form-inputs
```

Em seguida, **instale as dependências** necessárias do projeto executando no terminal o seguinte comando:

```
npm install
```

Até aqui, você criou um projeto React usando o Vite e adicionou todas as dependências ao projeto. Você pode, agora, **abrir a aplicação no editor Visual Studio Code**. Para isso, dentro do diretório do projeto digite o seguinte comando no terminal:

```
code .
```

Após a execução deste comando, o Visual Studio Code deverá abrir com a pasta raiz do seu projeto sendo acessada. Em seguida, você inicializará um servidor local e executará o projeto em seu navegador. Digite a seguinte linha de comando no terminal:

```
npm run dev
```

Ao executar esse script, você iniciará um servidor local de desenvolvimento, executará o código do projeto, iniciará um observador que detecta alterações no código e abrirá o projeto em um navegador web. Ele irá rodar a aplicação em modo desenvolvimento em <http://localhost:5173/>.

1.2 Passo 1: Criando o Estado Inicial com useState

Primeiro, precisamos criar o estado do formulário. Usaremos o hook **useState** para armazenar os valores dos campos de input. Desta forma, altere o arquivo App.jsx da seguinte forma:

```
import { useState } from "react";

function App() {
  // Estado inicial para armazenar os valores dos campos do formulário
  const [formData, setFormData] = useState({
    nome: "",
    email: "",
    senha: "",
    experiencia: "",
    aceitaTermos: false,
    genero: "",
  });

  // Estado para armazenar os erros
  const [errors, setErrors] = useState({});

  return (
    <div>
      <h2>Formulário Controlado</h2>
      <form>{/* Campos do formulário */}</form>
    </div>
  );
}

export default App;
```

Este código utiliza o hook **useState** para gerenciar o estado do formulário. O estado inicial, armazenado em **formData**, contém campos como nome, email, senha, experiência, aceitação dos termos e gênero, todos iniciando com valores vazios ou padrões. Além disso, há um segundo estado, **errors**, que é usado para armazenar possíveis erros de validação. Este código estabelece a base para um formulário controlado, onde os valores e erros são gerenciados através do estado.

1.3 Passo 2: Adição dos campos do formulário

Neste passo, vamos adicionar os campos do nosso formulário. Desta forma, vamos começar adicionando os campos de entrada do usuário (**input**) no arquivo **App.jsx**:

```
import { useState } from "react";
function App() {
  ...
  return (
    <div>
      <h2>Formulário Controlado</h2>
      <form onSubmit={handleSubmit}>
        { /* Campo Nome */ }
        <div>
          <label>Nome:</label>
          <input type="text" name="nome" value={formData.nome}
onChange={handleChange} />
          {errors.nome && <p style={{ color: "red" }}>{errors.nome}</p>}
        </div>
        { /* Campo Email */ }
        <div>
          <label>Email:</label>
          <input type="email" name="email" value={formData.email}
onChange={handleChange} />
          {errors.email && <p style={{ color: "red" }}>{errors.email}</p>}
        </div>
        { /* Campo Senha */ }
        <div>
          <label>Senha:</label>
          <input type="password" name="senha" value={formData.senha}
onChange={handleChange} />
          {errors.senha && <p style={{ color: "red" }}>{errors.senha}</p>}
        </div>
      </form>
    </div>
  );
}
export default App;
```

Observe neste exemplo:

- Até o momento, o formulário possui três campos: nome, email e senha.

- Cada campo é vinculado a uma propriedade específica do estado **formData**, e a função **handleChange** atualiza o estado quando os campos são modificados.
- Se houver erros de validação para um campo, como nome, email ou senha, uma mensagem de erro em vermelho será exibida abaixo do respectivo campo.
- O formulário também possui um manipulador de envio chamado **handleSubmit**, que será executado ao enviar o formulário.

Vamos continuar adicionando os campos no formulário. Porém, agora, vamos adicionar o campo de seleção (**select**) no arquivo **App.jsx** (após os campos de **input**):

```
...
{/* Campo de Seleção (Select) */}
<div>
  <label>Experiência:</label>
  <select
    name="experiencia"
    value={formData.experiencia}
    onChange={handleChange}
  >
    <option value="">Selecione seu nível de experiência</option>
    <option value="Iniciante">Iniciante</option>
    <option value="Intermediário">Intermediário</option>
    <option value="Avançado">Avançado</option>
  </select>
  {errors.experiencia && (
    <p style={{ color: "red" }}>{errors.experiencia}</p>
  )}
</div>
...
```

Observe neste código:

- O campo **select** permite que o usuário escolha uma opção. O valor selecionado é atualizado no estado **formData.experiencia**.
- Se o usuário não selecionar uma opção, uma mensagem de erro é exibida.

Em seguida, vamos adicionar os botões de opção (**radio buttons**) no arquivo **App.jsx** (após o campo de **select**):

```
...  
{/* Botões de Rádio (Radio Buttons) */}  
<div>  
  <p>Gênero:</p>  
  <label>  
    <input type="radio" name="genero" value="Masculino"  
checked={formData.genero === "Masculino"} onChange={handleChange}  
    />  
    Masculino  
  </label>  
  <label>  
    <input type="radio" name="genero" value="Feminino"  
checked={formData.genero === "Feminino"} onChange={handleChange}  
    />  
    Feminino  
  </label>  
  <label>  
    <input type="radio" name="genero" value="Outro"  
checked={formData.genero === "Outro"} onChange={handleChange}  
    />  
    Outro  
  </label>  
  {errors.genero && <p style={{ color: "red" }}>{errors.genero}</p>}  
</div>  
...
```

Observe neste código:

- Os **radio buttons** permitem que o usuário selecione uma única opção entre várias. O estado **formData.genero** é atualizado com o valor selecionado.
- Se nenhum gênero for selecionado, uma mensagem de erro será exibida.

Agora, vamos adicionar o campo de caixa de verificação (**checkbox**) no arquivo **App.jsx** (após os campos de **radio buttons**):

```
...
{/* Caixa de Verificação (Checkbox) */}
<div>
  <label>
    <input type="checkbox" name="aceitaTermos" checked={formData.aceitaTermos}
onChange={handleChange} />
    Aceito os termos e condições
  </label>
  {errors.aceitaTermos && (
    <p style={{ color: "red" }}>{errors.aceitaTermos}</p>
  )}
</div>
...
```

Observe neste código:

- O **checkbox** armazena um valor booleano (**true** ou **false**). Ele é usado para verificar se o usuário aceita os termos e condições.
- Se o **checkbox** não estiver marcado, exibimos uma mensagem de erro.

Finalmente, ao final destes campos, após o campo de caixa de verificação (**checkbox**), vamos adicionar o botão para submeter o formulário:

```
...
{/* Botão para submeter o formulário */}
<button type="submit">Enviar</button>
...
```

Observe que o clicar no botão "Enviar", a função **handleSubmit** é chamada (<form onSubmit={handleSubmit}>).

1.4 Passo 3: Lógica para lidar com as mudanças nos inputs

Agora, vamos adicionar ao nosso componente **App** a lógica para lidar com as mudanças nos inputs deste formulário. Neste contexto, após a definição dos nossos estados (**formData** e **errors**), vamos adicionar a função manipuladora de eventos chamada **handleChange**.

Para isso, altere o arquivo **App.jsx** da seguinte forma:

```
import { useState } from "react";
function App() {
  ...
  // Função para lidar com as mudanças nos inputs
  const handleChange = (event) => {
    const { name, value, type, checked } = event.target;

    // Se o tipo for "checkbox", usamos 'checked' para o valor
    const valor = type === 'checkbox' ? checked : value;

    setFormData((prevData) => ({
      ...prevData,
      [name]: valor, // Atualiza o campo correto no estado
    }));
  };
  ...
}
```

Observe neste exemplo:

- **Manipulação do estado:** Ele atualiza dinamicamente o estado (**formData**) de um input de formulário, dependendo do tipo do campo.
- **Desestruturação de event.target:** Extrai as propriedades **name**, **value**, **type** e **checked** do elemento que disparou o evento (o **input**).
- **Checkbox:** Se o campo for um **checkbox**, ele usa **checked** para determinar se está marcado; caso contrário, usa **value** (o valor digitado ou selecionado).
- **Atualização do estado:** A função **setFormData** atualiza o estado do formulário, mantendo os dados anteriores (**...prevData**) e alterando apenas o campo correspondente (**[name]: valor**).

1.5 Passo 4: Lógica de Validação do Formulário

Agora, vamos adicionar ao nosso componente **App** a lógica para realizar a validação dos dados vindos deste formulário. Neste contexto, após a definição da função manipuladora de eventos chamada **handleChange**, vamos adicionar a função chamada **validateForm**.

Para isso, altere o arquivo **App.jsx** da seguinte forma:

```
...
// Função para validar o formulário
const validateForm = () => {
  const newErrors = {};
  if (!formData.nome) {
    newErrors.nome = "O campo Nome é obrigatório";
  }
  if (!formData.email) {
    newErrors.email = "O campo Email é obrigatório";
  } else if (!/\S+@\S+\.\S+/.test(formData.email)) {
    newErrors.email = "Por favor, insira um email válido";
  }
  if (!formData.senha) {
    newErrors.senha = "O campo Senha é obrigatório";
  } else if (formData.senha.length < 6) {
    newErrors.senha = "A senha deve ter pelo menos 6 caracteres";
  }
  if (!formData.experiencia) {
    newErrors.experiencia = "Por favor, selecione seu nível de experiência";
  }
  if (!formData.genero) {
    newErrors.genero = "Por favor, selecione seu gênero";
  }
  if (!formData.aceitaTermos) {
    newErrors.aceitaTermos = "Você deve aceitar os termos e condições";
  }
  return newErrors;
};
...
```

Observe neste código:

- A função **validateForm** realiza a validação dos campos do formulário, verificando se todos os campos obrigatórios foram preenchidos corretamente.
- Ela cria um objeto **newErrors** para armazenar as mensagens de erro.
- Se o campo **"nome"** estiver vazio, adiciona a mensagem "O campo Nome é obrigatório".
- Para o **email**, verifica se está vazio ou se tem um formato inválido e adiciona a mensagem correspondente.
- A **senha** precisa ter ao menos 6 caracteres, caso contrário, adiciona um erro.

- Também verifica se o campo "**experiência**" e "**gênero**" estão preenchidos, e se os termos foram aceitos.
- A função retorna o objeto com os erros encontrados.

1.6 Passo 5: Lógica de Submissão do Formulário

Para finalizar, vamos adicionar a lógica de submissão dos dados do formulário. Desta forma, vamos adicionar a função **handleSubmit** após a definição da função **validateForm**. Altere o arquivo **App.jsx** da seguinte forma:

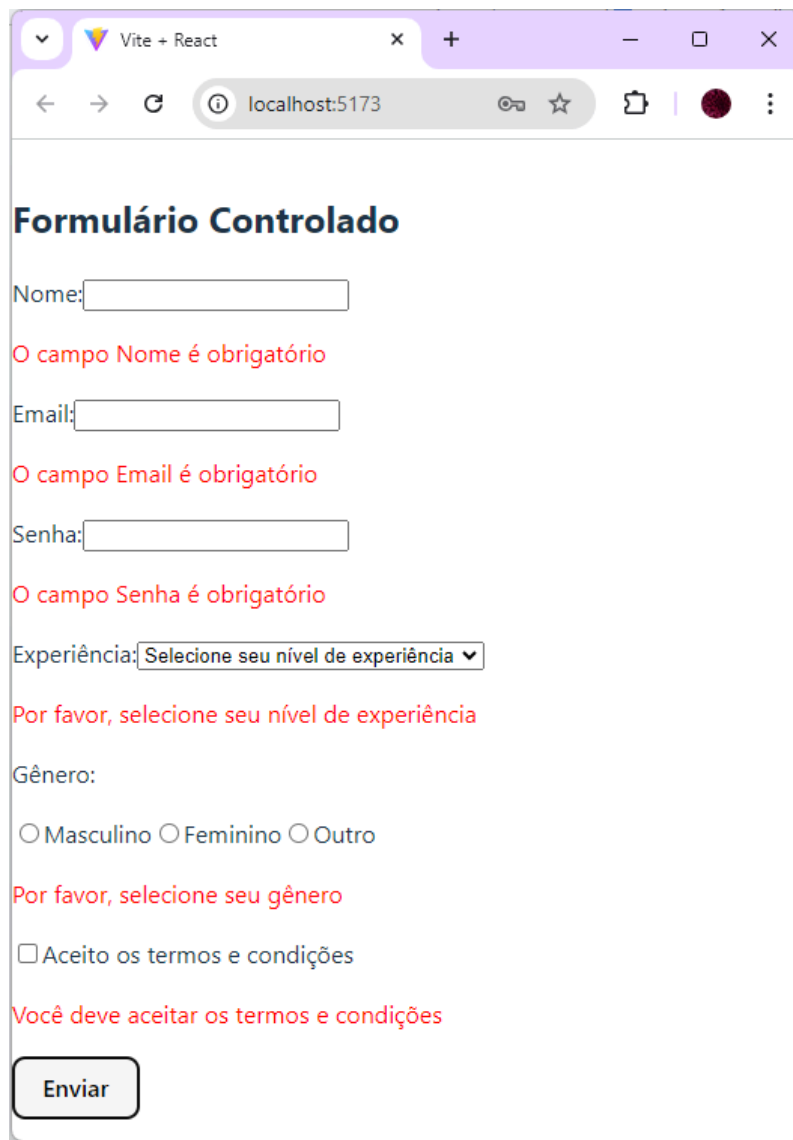
```
...
// Função para lidar com a submissão do formulário
const handleSubmit = (event) => {
  event.preventDefault(); // Evita o comportamento de recarregar a página
  // Valida o formulário antes de submeter
  const validationErrors = validateForm();
  setErrors(validationErrors);
  if (Object.keys(validationErrors).length === 0) {
    // Se não houver erros, exibe os dados
    alert(`
      Nome: ${formData.nome}\n
      Email: ${formData.email}\n
      Senha: ${formData.senha}\n
      Experiência: ${formData.experiencia}\n
      Gênero: ${formData.genero}\n
      Aceita os Termos: ${formData.aceitaTermos ? "Sim" : "Não"}
    `);
    setFormData({
      nome: "",
      email: "",
      senha: "",
      experiencia: "",
      aceitaTermos: false,
      genero: "",
    }); // Limpa o formulário após o envio
  }
};
...
```

Observe neste código que ao clicar no botão "Enviar", a função **handleSubmit** é chamada. Ela impede o comportamento padrão do formulário (recarregar a página) e valida os

campos. Se a validação passar, o formulário exibe os dados e limpa os campos. Caso contrário, exibe as mensagens de erro.

1.4 Passo 3: Testar a aplicação

Agora, você pode rodar o projeto para verificar se as validações estão funcionando corretamente:



The screenshot shows a web browser window with the title 'Vite + React' and the address bar displaying 'localhost:5173'. The page content is a form titled 'Formulário Controlado'. The form contains the following fields and validation messages:

- Nome:** A text input field. Below it, a red error message reads: 'O campo Nome é obrigatório'.
- Email:** A text input field. Below it, a red error message reads: 'O campo Email é obrigatório'.
- Senha:** A text input field. Below it, a red error message reads: 'O campo Senha é obrigatório'.
- Experiência:** A dropdown menu with the placeholder text 'Selecione seu nível de experiência'. Below it, a red error message reads: 'Por favor, selecione seu nível de experiência'.
- Gênero:** Three radio buttons labeled 'Masculino', 'Feminino', and 'Outro'. Below them, a red error message reads: 'Por favor, selecione seu gênero'.
- Aceito os termos e condições:** A checkbox. Below it, a red error message reads: 'Você deve aceitar os termos e condições'.
- Enviar:** A button at the bottom of the form.

Figura 1: Exemplo de validação de formulário