# React

- ***Biblioteca JavaScript***: Ferramenta para construir a interface do usuário (UI) de aplicações web.
- ***Foco em componentização***: UIs complexas são divididas em componentes menores, reutilizáveis e independentes.

# React

- ***Gerenciamento de estado***: Permite que os componentes reajam a mudanças nos dados e atualizem a UI de forma eficiente.
- ***Popularidade***: Uma das bibliotecas mais populares para desenvolvimento frontend, utilizada por empresas como Facebook, Instagram, Netflix e Airbnb.

# Vantagens para o Desenvolvimento Web

- ***Criação de UIs complexas***: React simplifica a construção de interfaces de usuário interativas e com grande quantidade de componentes.
- ***Virtual DOM***: O Virtual DOM (Document Object Model) do React otimiza as atualizações da interface, melhorando a performance da aplicação.

# Vantagens para o Desenvolvimento Web

- ***Comunidade ativa***: A grande comunidade React oferece suporte, bibliotecas, ferramentas e recursos para auxiliar no desenvolvimento.
- ***Documentação completa***: A documentação oficial do React é extensa e detalhada, facilitando o aprendizado e a resolução de problemas.

# Componentes

## Blocos de construção da interface

- ***Pilares da UI***: Componentes são peças independentes e reutilizáveis que formam a interface do usuário.
- ***Organização***: Permitem dividir a UI em partes menores e gerenciáveis, facilitando o desenvolvimento e manutenção.
- ***Reutilização***: Componentes podem ser usados em diferentes partes da aplicação, evitando repetição de código.

# Componentes

## Blocos de construção da interface

- ***Props (propriedades)***: Dados que podem ser passados para um componente para customizar seu comportamento.
- ***Estado***: Dados internos do componente que podem mudar ao longo do tempo, afetando a renderização.

# Exemplo

```
function Botao(props) {
  return <button onClick={props.onClick}>{props.texto}</button>;
}
```

# Estado em React

- ***Dados dinâmicos***: O estado representa informações que podem variar ao longo do tempo dentro de um componente.
- ***Influência na UI***: Mudanças no estado disparam a atualização da interface do usuário, mantendo-a sincronizada com os dados.
- ***Gerenciamento***: React oferece mecanismos para definir, atualizar e acessar o estado de forma eficiente.

# Estado em React

- ***Hooks (useState)***: Em componentes funcionais, o hook useState é utilizado para gerenciar o estado.
- ***this.state e this.setState***: Em componentes de classe, o estado é acessado através de this.state e atualizado com this.setState.

# Exemplo

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

function Contador() {
  const [contador, setContador] = useState(0);

  return (
    <div>
      <p>Contador: {contador}</p>
      <button onClick={() => setContador(contador + 1)}>Incrementar</button>
    </div>
  );
}
```

# JSX (JavaScript XML)

- ***Sintaxe***: JSX é uma extensão da sintaxe do JavaScript que permite escrever código que se assemelha a HTML dentro de componentes React.
- ***Propósito***: Facilita a criação e leitura de componentes, tornando o código mais intuitivo e visual.
- ***Transpilação***: O JSX não é interpretado diretamente pelo navegador. Ele é convertido em JavaScript puro através de ferramentas como o Babel.

# JSX (JavaScript XML)

- ***Vantagens***
  - Permite estruturar a UI de forma declarativa, similar ao HTML.
  - Facilita a inclusão de lógica JavaScript dentro da estrutura da UI.
  - Melhora a legibilidade do código, tornando-o mais fácil de entender e manter.

# Exemplo

```
function MeuComponente() {
  return (
    <div>
      <h1>Olá, mundo!</h1>
      <p>Este é um exemplo de JSX.</p>
    </div>
  );
}
```

# Create React App

- **Ferramenta oficial:** Criada pela equipe do React para simplificar a criação de novos projetos.
- **Ambiente de desenvolvimento pronto:** Configura automaticamente as ferramentas necessárias para começar a desenvolver em React.

# Create React App

- **Zero configuração:** Elimina a necessidade de configurar manualmente ferramentas como Webpack e Babel.
- **Facilita o desenvolvimento:** Inclui recursos como live reload, que atualiza automaticamente o navegador ao salvar alterações no código.
- **Ótimo para iniciantes:** Permite que você se concentre em aprender React sem se preocupar com configurações complexas.

# Create React App - Como usar

1. **Instalar:** `npm install -g create-react-app`
2. **Criar um novo projeto:** `npx create-react-app meu-app`
3. **Iniciar o servidor de desenvolvimento:** `cd meu-app` e `npm start`

# Estrutura de um Projeto React

- **Pasta `public`:** Contém arquivos estáticos como `index.html` (ponto de entrada da aplicação), imagens e outros recursos.
- **Pasta `src`:** Armazena o código fonte da aplicação.
  - `index.js`: Arquivo principal que renderiza o componente raiz da aplicação.
  - `App.js`: Componente raiz, geralmente responsável por definir a estrutura geral da UI.
  - Outras pastas e arquivos: Componentes, estilos, utilitários, etc.

# Organização de Componentes

- **Componentes reutilizáveis:** Agrupe componentes relacionados em pastas separadas.
- **Estilos:** Utilize CSS Modules ou styled-components para organizar os estilos de cada componente.
- **Nomenclatura:** Utilize nomes descritivos para componentes e arquivos.

# Exemplo

```
meu-app/
├── public/
│   ├── index.html
│   └── ...
└── src/
    ├── App.js
    ├── index.js
    ├── components/
    │   ├── Botao.js
    │   └── ...
    └── styles/
        ├── App.css
        └── ...
```

# Hello World em React

```
import React from 'react';

function App() {
  return (
    <div>
      <h1>Olá, mundo!</h1>
    </div>
  );
}

export default App;
```

# Hello World: Explicação

1. **Importar React:** `import React from 'react';`
   - Essencial para usar a biblioteca React.
2. **Criar o componente `App`:** `function App() { ... }`
   - Função que retorna o que será renderizado na tela (JSX).

# Hello World: Explicação

3. **Retornar JSX:** `return ( ... );`
   - JSX é uma sintaxe que mistura HTML com JavaScript.
   - No exemplo, retornamos um elemento `div` com um título `h1`.
4. **Exportar o componente:** `export default App;`
   - Permite que o componente seja usado em outros arquivos.

# Tipos de Componentes React: Funcionais vs. Classes

# Componentes Funcionais

- **Simples e concisos:** Funções JavaScript que retornam JSX.
- **Fácil leitura e manutenção:** Código mais limpo e direto.
- **Ideal para a maioria dos casos:** Componentes de apresentação, sem estado complexo.
- **Hooks:** Utilizam hooks (ex: `useState`, `useEffect`) para gerenciar estado e outros recursos.

# Exemplo

```
function BemVindo(props) {
  return <h1>Olá, {props.nome}!</h1>;
}
```

# Componentes de Classe

- **Mais recursos:** Herdam de `React.Component`.
- **Ciclo de vida:** Métodos para controlar diferentes fases do componente (montagem, atualização, desmontagem).
- **Estado:** Gerenciam estado interno com `this.state` e `this.setState`.
- **Menos comum:** Recomenda-se usar componentes funcionais com hooks sempre que possível.

# Exemplo

```
class BemVindo extends React.Component {
  render() {
    return <h1>Olá, {this.props.nome}!</h1>;
  }
}
```

# Escolhendo o Tipo de Componente

- **Componentes funcionais:**
  - Preferência para a maioria dos casos.
  - Mais simples, legíveis e fáceis de testar.
- **Componentes de classe:**
  - Quando você precisa de recursos avançados, como ciclo de vida ou gerenciamento de estado complexo.

# Props em React: Personalizando Componentes

# Props (Propriedades)

- **Dados de entrada:** Valores passados de um componente pai para um componente filho.
- **Customização:** Permitem tornar os componentes mais flexíveis e reutilizáveis.
- **Acesso:** Acessadas dentro do componente filho como `props.nomeDaProp`.
- **Imutabilidade:** Props são somente leitura, não podem ser modificadas diretamente pelo componente filho.

# Exemplo

```
function Saudacao(props) {
  return <p>Olá, {props.nome}!</p>;
}

// Uso:
<Saudacao nome="Maria" />
```

# Estado em React: Tornando a Interface Dinâmica

# Estado

- **Dados internos:** Informações privadas do componente que podem mudar ao longo do tempo.
- **Atualização da UI:** Mudanças no estado disparam uma nova renderização do componente, atualizando a interface.
- **Gerenciamento:**
  - **Componentes funcionais:** Use o hook `useState`.
  - **Componentes de classe:** Use `this.state` e `this.setState`.

# Exemplo

```
function Contador() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrementar</button>
    </div>
  );
}
```

# Eventos em React: Interagindo com a Interface

# Eventos

- **Ações do usuário:** Cliques, submissões de formulários, teclas pressionadas, etc.
- **Gatilhos:** Eventos disparam funções no seu código React.

# Eventos

- **Manipulação:** React permite que você responda a esses eventos e atualize a interface do usuário de acordo.
- **Sintaxe:**
  - Atributos de evento: `onClick`, `onSubmit`, `onChange`, etc.
  - Funções de manipulação: Definidas como valores dos atributos de evento.

# Exemplo

```
function MeuBotao() {
  function handleClick() {
    alert('Botão clicado!');
  }

  return <button onClick={handleClick}>Clique aqui</button>;
}
```

# Eventos Sintéticos

- **Camada de abstração:** React cria seus próprios eventos, chamados de eventos sintéticos, que envolvem os eventos nativos do navegador.
- **Compatibilidade:** Garante que o código funcione da mesma forma em diferentes navegadores.
- **Normalização:** Simplifica o acesso às informações do evento, como as coordenadas do mouse.
- **Prevenção de comportamento padrão:** Use `e.preventDefault()` para evitar ações padrão do navegador (ex: enviar um formulário).

# Exemplo

```
function MeuFormulario() {
  function handleSubmit(event) {
    event.preventDefault(); // Impede o envio padrão do formulário
    // Lógica para processar os dados do formulário
  }

  return (
    <form onSubmit={handleSubmit}>
      {/* ... campos do formulário */}
    </form>
  );
}
```

# Formulários em React: Controlando a Entrada do Usuário

# Componentes Controlados

- **Estado como fonte da verdade:** O valor do campo do formulário é armazenado no estado do componente React.
- **Sincronização:** O estado é atualizado sempre que o usuário digita algo no campo.
- **Vantagens:**
  - Fácil acesso aos dados do formulário.
  - Validação em tempo real.
  - Manipulação flexível dos dados.

# Exemplo

```
function MeuInput() {
  const [valor, setValor] = useState('');

  return (
    <input
      type="text"
      value={valor}
      onChange={(e) => setValor(e.target.value)}
    />
  );
}
```

# Validação e Submissão

- **Regras:** Defina regras para garantir que os dados inseridos sejam válidos (ex: formato de e-mail, campos obrigatórios).
- **Feedback:** Exiba mensagens de erro para o usuário caso os dados sejam inválidos.
- **Submissão:** Impeça o envio do formulário se houver erros de validação.

# Exemplo

```
function MeuFormulario() {
  // ... lógica de validação

  return (
    <form onSubmit={handleSubmit}>
      {/* ... campos do formulário */}
      <button type="submit" disabled={!formularioValido}>Enviar</button>
    </form>
  );
}
```

# Ciclo de Vida dos Componentes React

# O que é o Ciclo de Vida?

- **Analogia:** Assim como os seres vivos, componentes React têm um ciclo de vida com diferentes fases.
- **Fases:**
  - **Montagem:** O componente é criado e inserido no DOM.
  - **Atualização:** O componente reage a mudanças em props ou estado.
  - **Desmontagem:** O componente é removido do DOM.

# O que é o Ciclo de Vida?

- **Métodos de Ciclo de Vida:** Funções especiais que permitem executar código em momentos específicos do ciclo de vida.

# Métodos de Ciclo de Vida

# Montagem

- `constructor()` : Inicializa o estado do componente.
- `static getDerivedStateFromProps()` : Calcula o estado inicial com base em props.
- `render()` : Descreve o que o componente deve exibir na tela (JSX).
- `componentDidMount()` : Executa após o componente ser inserido no DOM (bom para buscar dados da API).

# Atualização

- `static getDerivedStateFromProps()` : Atualiza o estado com base em novas props.
- `shouldComponentUpdate()` : Decide se o componente deve ser atualizado (otimização de performance).
- `render()` : Atualiza a UI com base no novo estado ou props.
- `getSnapshotBeforeUpdate()` : Captura informações do DOM antes da atualização.
- `componentDidUpdate()` : Executa após a atualização do componente.

# Desmontagem

- `componentWillUnmount()` : Executa antes do componente ser removido do DOM (bom para limpar timers, cancelar requisições, etc.).

# Exemplo

```
class MyComponent extends React.Component {
  componentDidMount() {
    // Buscar dados da API
  }

  componentWillUnmount() {
    // Cancelar requisição em andamento
  }

  render() {
    // ...
  }
}
```

# Hooks em React

# O que são Hooks?

- **Funções especiais:** Adicionam recursos poderosos a componentes funcionais do React.
- **Estado e efeitos colaterais:** Permitem gerenciar estado, realizar chamadas à API, manipular o DOM e muito mais.

# O que são Hooks?

- **Alternativa às classes:** Tornam desnecessário usar componentes de classe na maioria dos casos.
- **Reutilização de lógica:** Facilitam a extração e compartilhamento de lógica entre componentes.
- **Código mais limpo:** Componentes funcionais com hooks são mais concisos e fáceis de ler.

# useState

- **Declaração de estado:** `const [estado, setEstado] = useState(valorInicial);`
- **`estado`:** Variável que armazena o valor atual do estado.
- **`setEstado`:** Função para atualizar o estado e acionar uma nova renderização do componente.
- **Valor inicial:** Pode ser qualquer tipo de dado (número, string, objeto, etc.).

# Exemplo

```
function Contador() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={() => setCount(count + 1)}>Incrementar</button>
    </div>
  );
}
```

# useEffect

- **Execução após renderização:** Permite executar código após o componente ser renderizado na tela.
- **Efeitos colaterais:** Ideal para buscar dados de APIs, manipular o DOM, configurar subscriptions, etc.
- **Sintaxe:** `useEffect(funcao, dependencias);`
- **Dependências:** Um array opcional que controla quando o efeito deve ser reexecutado.

# Exemplo

```
function DadosDoUsuario() {
  const [usuario, setUsuario] = useState(null);

  useEffect(() => {
    fetch('/api/usuario')
      .then(res => res.json())
      .then(data => setUsuario(data));
  }, []); // Executa apenas uma vez após a montagem

  return (
    <div>
      {usuario ? <p>Olá, {usuario.nome}!</p> : <p>Carregando...</p>}
    </div>
  );
}
```

# Alpine.js

# O que é Alpine.js?

- **Framework leve:** Projetado para adicionar interatividade a páginas web sem a complexidade de frameworks maiores.
- **JavaScript declarativo:** Permite controlar o comportamento da interface diretamente no HTML, usando atributos especiais.

# O que é Alpine.js?

- **Fácil de aprender:** Sintaxe simples e intuitiva, ideal para iniciantes e projetos menores.
- **Integração com outras ferramentas:** Funciona bem com outras bibliotecas e frameworks, como React, Vue.js e Tailwind CSS.

# Principais recursos

- **Diretivas:** Atributos especiais que permitem controlar a exibição de elementos, iterar sobre dados, manipular classes CSS e muito mais.
- **Eventos:** Facilita a resposta a eventos do usuário, como cliques, submissões de formulário e alterações de input.

# Principais recursos

- **Gerenciamento de estado:** Permite armazenar dados no componente e atualizar a interface automaticamente quando esses dados mudam.
- **Componentes:** Crie componentes reutilizáveis para organizar e encapsular a lógica da sua interface.

# Exemplo

```
<div x-data="{ open: false }">
  <button @click="open = !open">Toggle</button>
  <div x-show="open">
    Conteúdo oculto
  </div>
</div>
```

# x-data

- **Inicialização:** Define o estado inicial do componente.
- **Escopo de dados:** Cria um objeto que armazena os dados do componente.
- **Exemplo:**

```
<div x-data="{ count: 0, message: 'Olá!' }">
  <button @click="count++">Incrementar</button>
  <p x-text="message"></p>
</div>
```

# x-bind

- **Vinculação de dados:** Conecta propriedades de elementos HTML a dados do componente.
- **Atalho:** Use `:` (dois pontos) para abreviar.
- **Exemplo:**

```
<input type="text" x-bind:value="message">
<input type="text" :value="message">
```

# x-show e x-if

- **Exibição condicional:** Controlam a visibilidade de elementos com base em condições.
- **x-show:** Alterna a propriedade `display` do elemento.
- **x-if:** Remove ou adiciona o elemento do DOM.
- **Exemplo:**

```
<div x-show="count > 5">
  Este texto aparece quando o contador é maior que 5.
</div>
```

# x-for

- **Repetição:** Itera sobre um array e renderiza elementos para cada item.
- **Sintaxe:** `x-for="(item, index) in items"`
- **Exemplo:**

```
<ul>
  <template x-for="item in items">
    <li x-text="item"></li>
  </template>
</ul>
```

# @click e outros eventos

- **Manipulação de eventos:** Responde a eventos do usuário, como cliques, submissões de formulário, etc.
- **Sintaxe:** `@evento="expressao"`
- **Exemplo:**

```
<button @click="count++">Incrementar</button>
```

# HTMX

# O que é HTMX?

- **Biblioteca JavaScript:** Estende o HTML para permitir interações ricas sem escrever muito JavaScript.
- **Ajax sem dor de cabeça:** Facilita o envio de requisições assíncronas para atualizar partes da página.
- **Atributos poderosos:** Use atributos `hx-` para definir o comportamento de elementos HTML.
- **Alternativa leve:** Ideal para projetos que precisam de interatividade simples e rápida.

# Exemplo

```
<button hx-post="/click" hx-swap="outerHTML">
  Clique aqui
</button>
```

# Atributos Principais

- `hx-get`, `hx-post`, etc.: Fazem requisições HTTP para URLs específicas.
- `hx-trigger`: Define eventos que disparam as requisições (ex: clique, mudança de valor).
- `hx-target`: Especifica qual elemento deve ser atualizado com a resposta da requisição.
- `hx-swap`: Controla como o conteúdo atualizado é inserido na página (ex: substituir o elemento, adicionar antes/depois).

# Exemplos

```
<div id="meu-conteudo" hx-get="/conteudo" hx-trigger="load">
  Carregando...
</div>
```

```
<form hx-post="/contato" hx-target="#mensagem" hx-swap="outerHTML">
  <input type="text" name="nome" placeholder="Nome" required>
  <input type="email" name="email" placeholder="Email" required>
  <textarea name="mensagem" placeholder="Mensagem" required></textarea>
  <button type="submit">Enviar</button>
</form>

<div id="mensagem"></div>
```

# Benefícios:

- **Menos JavaScript:** Reduz a quantidade de código JavaScript necessário.
- **Melhor performance:** Requisições parciais são mais rápidas que recarregar a página inteira.
- **Mais simples:** A sintaxe declarativa do HTMX é mais fácil de aprender e usar.
- **Compatibilidade:** Funciona em navegadores modernos sem precisar de polyfills.

# Casos de Uso

- **Atualização de conteúdo dinâmico:** Exibir dados de uma API, carregar mais resultados, etc.
- **Formulários:** Enviar formulários sem recarregar a página, validar campos em tempo real.
- **Navegação:** Criar experiências de navegação mais fluidas, sem transições bruscas.
- **Infinidade de possibilidades:** Use sua criatividade para criar interações personalizadas!

# Express.js

# O que é Express.js?

- **Framework minimalista:** Simplifica a criação de servidores web e APIs em Node.js.
- **Rotas e middlewares:** Define como a aplicação responde a diferentes requisições HTTP.
- **Flexibilidade:** Permite personalizar e estender a funcionalidade com middlewares e plugins.
- **Popularidade:** Um dos frameworks mais utilizados para desenvolvimento backend em Node.js.

# Exemplo

```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Olá, mundo!');
});

app.listen(3000, () => {
  console.log('Servidor rodando na porta 3000');
});
```

# Rotas

- **Endpoints:** Definem os caminhos que a aplicação pode receber requisições.
- **Métodos HTTP:** `app.get()`, `app.post()`, `app.put()`, `app.delete()`, etc.
- **Parâmetros:** Capturam valores dinâmicos na URL (ex: `/usuario/:id`).

# Exemplos

- Rota simples

```
app.get('/ola', (req, res) => {
  res.send('Olá, mundo!');
});
```

- Rota com parâmetro

```
app.get('/usuario/:id', (req, res) => {
  const id = req.params.id;
  res.send(`Dados do usuário ${id}`);
});
```

# Exemplos

- Rota com query string

```
app.get('/produtos', (req, res) => {
  const categoria = req.query.categoria;
  // Buscar produtos por categoria
  res.send(`Produtos da categoria ${categoria}`);
});
```

- Rota com post

```
app.post('/cadastro', (req, res) => {
  const dados = req.body;
  // Salvar dados no banco de dados
  res.send('Cadastro realizado com sucesso!');
});
```

# Exemplos

- Rota com agrupadas

```
const router = express.Router();

router.get('/', (req, res) => { /* ... */ });
router.get('/:id', (req, res) => { /* ... */ });
router.post('/', (req, res) => { /* ... */ });

app.use('/produtos', router);
```