PW III

Slides I - Nexjs

Estruturando a Interface (Design)

O que é Componentização?

Conceito: Dividir interfaces complexas em partes menores, independentes e reutilizáveis.

Por que usar? Manutenção facilitada, reaproveitamento de código e consistência visual.

Contexto Next.js: Como o React no Next.js lida com essas peças (Server Components vs. Client Components na prática).

Introdução ao Atomic Design

Conceito: Metodologia criada por Brad Frost inspirada na química para construir *Design Systems*.

A Metáfora: Não desenhamos páginas inteiras de uma vez, construímos sistemas a partir de elementos básicos.

Fases do Atomic Design

Átomos: Botões, inputs, labels, ícones (Elementos HTML básicos estilizados).

Moléculas: Um campo de busca (Input + Botão de Busca + Label).

Organismos: Um cabeçalho completo (Logo + Menu de Navegação + Campo de Busca).

Templates: A estrutura da página sem os dados reais (Wireframes de alta fidelidade em código).

Páginas: O template preenchido com dados reais (onde o Next.js injeta o conteúdo vindo do SSR/CSR).

Button

```
// src/components/atoms/Button.jsx

export function Button({ children, variant = 'primary', ...props }) {

  const baseStyles = "px-4 py-2 rounded-md font-semibold transition-colors flex items-center justify-center gap-2";

  const variants = {

    primary: "bg-blue-600 text-white hover:bg-blue-700",

    secondary: "bg-gray-200 text-gray-800 hover:bg-gray-300",

  };

  return (

    <button className={`${baseStyles} ${variants[variant]}`} {...props}>

      {children}

    </button>

  );

}
```

Input

```
// src/components/atoms/Input.jsx

export function Input({ ...props }) {
  return (
    <input
      className="w-full px-3 py-2 border border-gray-300 rounded-md focus:outline-none focus:ring-2 focus:ring-blue-500 transition-all"
      {...props}
    />
  );
}
```

Label

```
// src/components/atoms/Label.jsx

export function Label({ text, ...props }) {
  return (
    <label
      className="block text-sm font-medium text-gray-700 mb-1"
      {...props}
    >
      {text}
    </label>
  );
}
```

O que são Props? (A Teoria)

Conceito: *Props* (abreviação de Propriedades) são a forma como passamos informações de um componente "Pai" para um componente "Filho".

A Analogia: Pense nas *props* como os parâmetros que você passa para uma função JavaScript tradicional. Ou, no mundo HTML, como os atributos de uma tag (ex: o `src` de uma `<img>`).

Regra de Ouro: Props são **somente leitura** (*read-only*). O componente filho que recebe a prop não pode alterá-la, ele só pode usá-la para se renderizar.

Como criamos o componente (O Filho):

```
// Recebendo as props 'titulo' e 'cor'
export function Titulo({ titulo, cor }) {
  return (
    <h1 style={{ color: cor }}>
      {titulo}
    </h1>
  );
}
```

Como usamos o componente (O Pai):

```
// Passando valores diferentes para o mesmo componente
export function Pagina() {
  return (
    <div>
      <Titulo titulo="Bem-vindo ao Sistema" cor="blue" />
      <Titulo titulo="Erro ao carregar" cor="red" />
    </div>
  );
}
```

O que é a prop children? (A Teoria)

Conceito: É uma *prop* especial e nativa do React. Ela representa todo o conteúdo que você coloca **entre** a tag de abertura e a tag de fechamento de um componente.

A Analogia: Pense no `children` como uma "caixa vazia" ou um "buraco" no seu componente. Você está dizendo: *"Eu vou desenhar a borda e o fundo aqui, mas quem usar esse componente decide o que vai dentro da caixa".*

Por que é essencial no Atomic Design? É assim que criamos *Templates* e *Organismos* que englobam outros componentes menores.

Como criamos o componente (O Filho):

```
// O botão não sabe o que vai ter dentro dele, ele apenas renderiza o 'children'
export function BotaoMagico({ children }) {
  return (
    <button className="bg-purple-500 p-4 rounded text-white shadow-lg">
      {children}
    </button>
  );
}
```

Como usamos o componente (O Pai):

```
export function Pagina() {
  return (
    <div>
      {/* Exemplo 1: Passando apenas texto */}
      <BotaoMagico>
        Clique Aqui
      </BotaoMagico>

      {/* Exemplo 2: Passando outros componentes e HTML (Composição!) */}
      <BotaoMagico>
        <SearchIcon />
        <span>Buscar Produtos</span>
      </BotaoMagico>
    </div>
  );
}
```

Estruturando a Lógica (Arquitetura)

O Problema: Código "espaguete" onde a chamada da API, a regra de negócio e o HTML estão todos misturados no mesmo arquivo.

A Solução: Separação de Responsabilidades (*Separation of Concerns*). O componente visual não precisa saber *como* os dados são salvos, apenas *como* exibi-los.

Camadas da Arquitetura Front-end

Camada de Apresentação (UI): Nossos componentes do Atomic Design.

Camada de Domínio/Negócio: Onde vivem os Casos de Uso.

Camada de Infraestrutura/Serviços: Chamadas de API (fetch), bibliotecas externas, acesso ao LocalStorage.

O que são Casos de Uso (Use Cases)?

Definição: Uma ação específica que o usuário pode realizar no sistema (ex: `RealizarLoginUseCase`, `AdicionarProdutoCarrinhoUseCase`).

A Regra de Ouro: Um Caso de Uso no front-end deve ser apenas JavaScript/TypeScript puro. Ele **não pode** importar nada do React ou do Next.js. Ele recebe dados, processa a regra e devolve um resultado.

Juntando Tudo (A Prática)

- 📁 `src/`
  - 📁 `components/` *(Onde vive o Atomic Design)*
    - 📁 `atoms/`
    - 📁 `molecules/`
    - 📁 `organisms/`
  - 📁 `useCases/` *(Nossa regra de negócio)*
    - 📄 `LoginUser.ts`
  - 📁 `services/` *(Infraestrutura)*
    - 📄 `api.ts`
  - 📁 `app/` *(Ou `pages/`, as rotas do Next.js)*
    - 📄 `page.tsx`