Clean Code no React

▼ Desacoplamento

Quando Desacoplar

- Quando há algo repetitivo.
- Quando algo pode ser isolado do seu contexto: quando uma parte do JavaScript está associada apenas a uma pequena parte da interface.

Dica para Não Desacoplar em Excesso

- Quando algo se repete pouco (e não prejudica a leitura do arquivo) e não tem uma lógica JavaScript associada a ele.

Serão Desacoplados no arquivo AppInitial.tsx

- **Footer**: possui a constante `currentYear` só para ele.
- **Header**.

▼ Componentes Puros

Problema

Componentes que dependem de seus componentes pais podem gerar problemas de performance e serem inutilizados em diferentes contextos.

Como Evitar

Evite levar lógica JavaScript para o componente que o torne dependente de outro.

Desacoplamento

Componentes devem ser capazes de serem executados em diferentes partes do código. Caso contrário, significa que ainda estão acoplados ao componente pai.

▼ Funções e Eventos

Nomeação de Funções e Eventos

Prefixo HANDLE

Criar função que é disparada por um evento do usuário (como um clique).
Exemplo:

```
function handleCreateNewTodo() {}
```

Prefixo ON

Criado quando um componente recebe uma função que é disparada a partir do evento de um usuário. **Exemplo:**

```
<Header onCreateNewTodo={handleCreateNewTodo} />
```

▼ Configuração vs Composição

Existem duas formas de personalizar um componente:

1. **Configuração:** Recebe parâmetros para definir condições.
2. **Padrão de Composição:** Usa vários componentes menores para criar um componente maior e reutilizável.

Configuração

Usada para criar possíveis variações de um mesmo componente. Afeta apenas a aparência.

Exemplo:

```
interface InputProps {
  label?: string;
}

export default function InputOld({ label }: InputProps) {
  return (
    <div>
```

```
      {label ? <label>{label}</label> : null}
      <input type="text" />
    </div>
  );
}
```

Composição

Permite criar componentes flexíveis e reutilizáveis, utilizando `children`.

Exemplo:

```
interface RootProps {
  children: React.ReactNode;
}

export function Root({ children }: RootProps) {
  return <div>{children}</div>;
}

export function FormField() {
  return <input type="text" />;
}

interface LabelProps extends React.LabelHTMLAttributes<HTMLLabelEl
ement> {}

export function Label(props: LabelProps) {
  return <label {...props} />;
}

// Instanciando:
<Input.Root>
  <Input.Label htmlFor="name" id="name-label" />
  <Input.FormField />
</Input.Root>;
```

▼ Organização de Condicionais

Evitar Criar Condicionais Dentro da Camada de HTML

Forma errada:

```
{
  todos.length === 0 && <p>Nenhum todo cadastrado</p>;
}
```

Melhor Forma: Criar uma Variável para a Condicional

```
const isTodoListEmpty = todos.length === 0;

{
  isTodoListEmpty && <p>Nenhum todo cadastrado</p>;
}
```

Isso torna o código mais legível e organizado.

▼ Conclusão

- **Desacoplamento** evita repetição e melhora a organização do código.
- **Componentes puros** são mais performáticos e reutilizáveis.
- **Padrão de nomeação** melhora a compreensão do código.
- **Composição** é uma abordagem mais modular do que simples configuração.
- **Separar lógica e condicional** da camada de HTML melhora a legibilidade.