08 — Componentes

08 — Componentes

> A regra dos três arquivos, o catálogo de `common/` e quando criar em vez de reaproveitar. Componente **não chama caso de uso**. Recebe dados e callbacks por props. Catálogo verificado: os componentes abaixo foram portados e testados de verdade (`tsc` / `lint` / `build`), não apenas lidos de uma implementação de referência — onde algo mudou na travessia, está marcado.

A regra dos três arquivos — só quando há conteúdo real

Todo componente mora em sua própria pasta:

```
src/components/common/FornecedorCard/
├── FornecedorCard.tsx          o componente├── FornecedorCard.styles.ts    objetos sx / styled
— SÓ se houver estilo real
└── index.ts                    re-export
```

```
// src/components/common/FornecedorCard/index.tsexport { default } from "./FornecedorCard";
export type { FornecedorCardProps } from "./FornecedorCard";
```

Depois **atualize o barrel da pasta pai** no mesmo commit:

```
// src/components/common/index.ts
```

```
export { default as FornecedorCard } from "./FornecedorCard";
```

Esquecer o barrel é a falha mais comum — leva a importes inconsistentes (uns via `@components`, outros via caminho completo). Ver 03.

Não crie um `.styles.ts` vazio "por consistência". É comum achar, em código de referência, componentes com um arquivo de estilo que só contém um comentário e uma função retornando `{}` — não porte esse boilerplate. Se o componente usa só `sx` inline pontual ou nenhum estilo próprio, ele tem **dois** arquivos (`.tsx` + `index.ts`), não três. O terceiro arquivo nasce quando há uma constante `SxProps<Theme>` real para exportar — ver 12.

Por que arquivo de estilo separado (quando existe)

Mantém o `.tsx` legível. Um componente com 200 linhas de JSX e 120 de `sx` inline é ilegível em revisão.

Onde colocar

```
O componente é usado por quantas telas?├─ 1  → src/pages/<Tela>/  (componente local, sem promover)
└─ 2+ ─ é agnóstico de domínio?
        ├─ sim → src/components/common/
        └─ não → src/components/<Nome>/
```

Compõe o shell da aplicação (barra, menu, abas, moldura de página)? → `src/components/layout/`.

Não promova por antecipação. Um componente só sai de `pages/` quando a segunda tela precisar dele. Abstrair cedo demais produz props que ninguém usa.

Catálogo de `common/`

Componentes já portados e verificados. Antes de criar qualquer coisa, verifique se um destes resolve.

Entrada de dados

Componente	Para que serve
`FormField`	`TextField` integrado ao react-hook-form, com toggle de senha
`SelectInput`	Select controlado; recebe `SelectOption[]`
`DatePickerInput`	Data via `<input type="date">` — ver gap abaixo
`CheckboxGroup`	Grupo de checkboxes; recebe `CheckboxOption[]`
`RadioButtonGroup`	Grupo de radios; recebe `RadioOption[]`
`MaskedInput`	Base de máscara — `CnpjInput` / `PhoneInput` / `CurrencyInput` se apoiam aqui
`CnpjInput`	CNPJ com máscara e validação (`utils/validators/Cnpj.ts`)
`PhoneInput`	Telefone com máscara
`CurrencyInput`	Moeda; exporta `parseCurrencyValue` para converter de volta a número

Exibição

Componente	Para que serve
`DataTable`	Tabela com paginação, seleção de linhas e menu de contexto por linha (`ContextAction[]`)
`ResultadosCard`	Card com título para agrupar resultado de busca
`FiltroBusca`	Bloco de filtro padrão de tela de busca
`TransferListCard`	Lista de transferência (mover itens entre dois lados)
`TransferActions`	Botões de ação do `TransferListCard`
`LoadingButton`	Botão com spinner durante ação assíncrona
`ActionModal`	Diálogo de confirmação com ação primária/secundária e estado de loading
`WelcomeCard`	Card de boas-vindas simples (título + descrição)

`PageContainer` (título, breadcrumbs, ações, `maxWidth`) mora em `src/components/layout/`, não em `common/`

— é moldura de página, não peça de formulário/exibição.

> **`DataTable` é a fusão de duas referências.** É comum encontrar, em bases de código que evoluíram organicamente, uma versão simples e uma versão "avançada" quase idênticas de um mesmo componente de tabela (uma com seleção e menu de contexto, outra sem). Neste padrão, existe **um só** `DataTable` — o superconjunto, com `selectable` / `contextActions` / `getRowId` opcionais. Quem não precisa dessas props simplesmente não as passa; não há razão para manter os dois.

> **Componente domínio-neutro fora de `common/`.** É comum achar um componente estruturalmente genérico (ex.: um modal de confirmação) vivendo fora de `common/` só porque nasceu para resolver um caso específico primeiro. Pela própria regra deste guia ("Onde colocar"), ele pertence a `common/` — é onde está aqui. Ver anti-padrões em 13.

`DatePickerInput` não usa o `@mui/x-date-pickers`

O projeto depende de `@mui/x-date-pickers` e configura `LocalizationProvider` + `AdapterDateFns` em `main.tsx` (03) — mas **nenhum componente do catálogo usa o `<DatePicker>` real da biblioteca.** `DatePickerInput` é só um `<input type="date">` (ou `datetime-local` / `time`) por dentro de um `TextField`, com máscara e validação escritas à mão.

Isso funciona, mas significa que a dependência `@mui/x-date-pickers` está instalada e configurada **sem nenhum uso de referência**. Se a primeira tela real precisar de um seletor de calendário visual (não só um input nativo do browser), será a primeira vez que alguém usa o `<DatePicker>` da biblioteca neste projeto — não existe exemplo para copiar. Escreva-o consultando a documentação do `@mui/x-date-pickers` diretamente, e considere promovê-lo a um novo componente de `common/` depois.

Assinaturas de referência

`FormField` — o mais usado:

```
interface FormFieldProps<T extends FieldValues> extends Omit<TextFieldProps, "name" | "error" |
"helperText"> {
  name: FieldPath<T>;
  control: Control<T>;
  label: string;
  rules?: ControllerProps<T>["rules"];
  showPasswordToggle?: boolean;
}
```

Estende `TextFieldProps`, então aceita `size`, `fullWidth`, `disabled` etc. sem redeclarar. `error` e `helperText` são removidos porque vêm do `fieldState` do react-hook-form.

`DataTable`:

```
export interface Column<T = unknown> {
  id: string;
  label: string;
  minWidth?: number;
  align?: "right" | "left" | "center";
  format?: (value: unknown, row: T) => React.ReactNode;
  sortable?: boolean;
}
export interface ContextAction<T = unknown> {
  id: string;
  label: string;
  icon?: React.ReactNode;
  onClick: (row: T, index: number) => void;
  disabled?: (row: T) => boolean;
  divider?: boolean;
}
interface DataTableProps<T = unknown> {
  columns: Column<T>[];
  data: T[];
  loading?: boolean;
  error?: string | null;
  page?: number;
  rowsPerPage?: number;
```

```
  totalCount?: number;
  onPageChange?: (page: number) => void;
  onRowsPerPageChange?: (rowsPerPage: number) => void;  onRowClick?: (row: T, index: number) =>
void;
  emptyMessage?: string;
  rowsPerPageOptions?: number[];
  stickyHeader?: boolean;
  maxHeight?: number | string;  // selecao e menu de contexto — opcionais, ignore se nao
precisar
  selectable?: boolean;
  selectedRows?: T[];
  onSelectionChange?: (selectedRows: T[]) => void;
  getRowId?: (row: T) => string | number;
  contextActions?: ContextAction<T>[];
  showContextMenu?: boolean;
}
```

`loading`, `error` e `emptyMessage` são tratados **dentro** do componente. Não envolva `DataTable` em condicional de loading — passe a flag. Sem `selectable`/`contextActions`, a tabela se comporta como a versão simples — as props extras custam zero quando omitidas.

Para os demais, leia a interface no próprio arquivo. Não presuma props a partir do nome.

Como escrever um componente

```
// src/components/common/FornecedorCard/FornecedorCard.tsx
import React from "react";
import { Card, CardContent, Typography } from "@mui/material";import { cardStyle } from
"./FornecedorCard.styles";
export interface FornecedorCardProps {
  nome: string;
  cnpj: string;
  ativo: boolean;
  onSelecionar?: (cnpj: string) => void;
}
```

```
const FornecedorCard: React.FC<FornecedorCardProps> = ({ nome, cnpj, ativo, onSelecionar }) =>
{
  const handleClick = () => onSelecionar?.(cnpj);
  return (
    <Card sx={cardStyle(ativo)} onClick={handleClick}>
      <CardContent>
        <Typography variant="h6">{nome}</Typography>        <Typography variant="body2"
color="text.secondary">
          {cnpj}
        </Typography>
      </CardContent>
    </Card>
  );
};
export default FornecedorCard;
```

```
// src/components/common/FornecedorCard/FornecedorCard.styles.tsimport { SxProps, Theme } from
"@mui/material";
// depende do estado "ativo" — por isso funcao, nao constanteexport const cardStyle = (ativo:
boolean): SxProps<Theme> => ({
  opacity: ativo ? 1 : 0.6,
  cursor: "pointer"
});
```

Regras aplicadas:

- Interface de props **exportada** — outras telas precisam dela para tipar wrappers.
- `React.FC<Props>` com props destruturadas na assinatura.
- Callback de prop nomeado `on{Ação}`; handler interno `handle{Ação}` — ver 13.
- `export default` do componente; o barrel dá o nome.
- Zero import de `@curio/client`, `useAuth`, `useCurioMutation`.
- Estilo em constante nomeada exportada, não objeto `styles.algo` — ver 12.

Componente não decide o próprio

posicionamento externo

```
// ERRADO — o card decide seu proprio tamanho de grid; quem usa nao tem escolhaconst WelcomeCard
= ({ title, description }) => (
  <Grid item xs={12} sm={6}>
    <Card>...</Card>
  </Grid>
);
// CERTO — o card so sabe renderizar a si mesmo; o layout e responsabilidade de quem chamaconst
WelcomeCard = ({ title, description }) => <Card>...</Card>;
// quem usa decide o grid:
<Grid item xs={12} sm={6}>
  <WelcomeCard title="Bem-vindo!" />
</Grid>;
```

Um componente reutilizável que embute sua própria posição num grid externo (`<Grid item xs={...}>`) só funciona no layout onde foi escrito primeiro. É um erro comum em cards de boas-vindas/destaque; a versão deste catálogo não faz isso.

Componente não chama backend

```
// ERRADO — componente comum acoplado a caso de uso
const FornecedorCard = ({ oid }) => {
  const { data } = useBuscaFornecedor();
  return <Card>{data?.Fornecedor._Nome}</Card>;
};
// CERTO — a pagina busca, o componente exibeconst FornecedorCard = ({ nome, cnpj }:
FornecedorCardProps) => <Card>{nome}</Card>;
```

Exceção: componentes de `layout/` podem consumir Context (`useAuth`, `useTabs`, `useNotification`). São o shell, não peças reutilizáveis.

Genéricos

Componentes que recebem coleção tipada usam genérico com default:

```
function DataTable<T = unknown>({ columns, data }: DataTableProps<T>) { ... }
```

Permite `<DataTable<FornecedorXML> columns={cols} data={fornecedores} />` com `format` tipado, e segue funcionando sem anotação.

O que não copiar de uma implementação existente

Ao investigar código de referência, é comum achar pastas em `components/` que **parecem** genéricas mas na verdade não pertencem a um catálogo de padrão:

Sinal	Por quê
Nome carrega uma entidade concreta de domínio (ex.: `Empresa/`, `FiltroXyzForm/`)	É um formulário/listagem específico daquele domínio, não um padrão
Estruturalmente genérico mas usado só numa tela específica (ex.: um card de dashboard)	O valor dele está no acoplamento àquela tela específica, não é reutilizável de verdade
Não é um componente de UI, e sim uma tela de demonstração/catálogo manual	Não pertence ao catálogo de componentes reutilizáveis

Se uma tela nova precisar de algo parecido, escreva-o de novo neste projeto — não é o mesmo esforço de copiar um componente genuinamente genérico como `DataTable`, porque o valor de um componente acoplado está justamente no acoplamento à tela original dele.

Antes de criar um componente

1. Existe em `common/`? Use.
2. Existe no MUI? Use o MUI direto — não envolva `Button` só para trocar a cor padrão (isso é tema, ver 12).

3. Só uma tela usa? Deixe em `pages/`.
4. Nenhuma das anteriores? Crie em `common/`, com os arquivos que realmente precisar e o barrel atualizado.

Revision #2

Created Thu, Aug 20, 2026 4:53 PM by Geraldo Barbosa

Updated Tue, Aug 25, 2026 4:53 PM by Geraldo Barbosa