UI

Componentes, layout/menu, rotas, formulários e tema.

- 08 — Componentes
- 09 — Layout e menu lateral
- 10 — Rotas e proteção
- 11 — Formulários e validação
- 12 — Tema e estilo

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

Comp	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

Comp	Para que serve
`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

Comp	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)

Comp	Para que e serve
`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.

09 — Layout e menu lateral

09 — Layout e menu lateral

> O shell da aplicação: `AppBar`, `Drawer` com menu em árvore, área de conteúdo. `menuTree.ts` é a fonte única do menu — nenhum item é escrito em JSX. A navegação é **híbrida**: cada folha do menu abre por **rota** ou por **aba**, e o shell suporta as duas ao mesmo tempo.

O shell

```
<Main>                        ← elemento da rota protegida
  <TabsProvider>
    <MainContent>
      <AppBar>                título + sair      <Drawer>                menu a partir de
menuTree
      <Box component="main">        <TabBar />            abas abertas (some se não
houver)        <Box>                 ← <Outlet/>: rota atual, visível quando activeTabId ===
null
        {tabs.map(...)}       ← painéis de aba, todos montados, só o ativo visível
```

`Main` é elemento da rota protegida em `AppRouter` (10). Por isso `TabsProvider` pode usar `useNavigate`: já está dentro do router.

Os dois modos de navegação

Modo	O que acontece ao clicar	Estado da tela	URL muda?
`"route"`	`navigate("/" + path)` e zera a aba ativa	Perdido ao sair	Sim
`"tab"`	`openTab(node)` — painel novo fica montado	**Preservado** enquanto aberta	Não

Escolha por tela:

- `"route"` para tela de entrada e telas que se quer linkáveis/favoritáveis. Dashboard sempre.
- `"tab"` para telas de trabalho: cadastro, consulta, movimentação — onde o usuário alterna entre várias sem perder o que preencheu.

Por que abas preservam estado

Todos os painéis de aba ficam montados; só o ativo é visível (`display: none` nos demais). Um formulário meio preenchido continua lá ao voltar. Isso tem duas consequências que **não** são opcionais:

1. O `UseCaseManager` da página usa `autoClose={false}` — o caso de uso permanece aberto no

servidor enquanto a aba existir (06).

2. Cleanup de `useEffect` **não** dispara ao fechar a aba, porque o componente não desmonta por conta disso. Fechar a aba precisa ser explícito — ver `useTabCloseCallback`.

menuTree.ts

```
// src/routes/menuTree.ts
import React from "react";
import type { Session } from "@/lib/curio";
import { PATHS } from "./paths";
/** Como a folha do menu abre: navegando pela rota, ou numa aba. */export type MenuOpenMode =
"route" | "tab";
/**
 * Modo usado quando o no nao declara `mode`.
 * Troque para "route" se o produto nao usar abas.
 */
export const DEFAULT_OPEN_MODE: MenuOpenMode = "tab";
export interface MenuNode {
  label: string;
  icon?: React.ComponentType;  /** Só em folha (sem children) — caminho de rota sem barra
inicial */
  path?: string;
  /** Só em nó pai (sem path) — grupo expansível */
  children?: MenuNode[];
  /** Sobrescreve DEFAULT_OPEN_MODE nesta folha */
  mode?: MenuOpenMode;
  /**
   * Só em folha — executa em vez de abrir tela (relatório, disparo pontual).   * Tem
precedência sobre `mode`. Ver "Itens de menu que executam ação".
   */
  action?: (session: Session | undefined) => Promise<void>;
}
export const menuTree: MenuNode[] = [
  {
    label: "Dashboard",
```

```
    path: PATHS.DASHBOARD,
    mode: "route" // tela inicial vive na URL, nao numa aba
  },
  {
    label: "Cadastro",
    children: [      { label: "Fornecedor", path: PATHS.CADASTRO.FORNECEDOR }, // usa o
default
      { label: "Relatório mensal", path: PATHS.RELATORIOS.MENSAL, mode: "route" }
    ]
  }
];
```

Um projeto que não quer abas troca `DEFAULT_OPEN_MODE` para `"route"` e não declara `mode` em lugar nenhum. `TabsProvider`, `TabBar` e `TabPageRenderer` continuam no código, inertes — `TabBar` retorna `null` sem abas. Nada a remover.

Invariantes

Regra	Por quê
Nó tem `path` **ou** `children`, nunca ambos	`MenuTreeItem` decide por `if (node.children)`; pai nunca navega
`action` dispensa `path`	Nó de ação não navega — não precisa de rota

Regra	Por quê
`mode` só em folha	Nó pai não abre nada, só expande
`path` sempre de `PATHS`, nunca literal	Renomear a rota em um lugar só
`label` único entre irmãos	É usado como `key` do React
Toda folha existe também em `routes.ts`	O `TabPageRenderer` resolve o `path` na mesma tabela de rotas

A última é a que mais quebra: uma folha em `menuTree` sem entrada em `routes.ts` abre uma aba com "Rota não encontrada".

MenuTreeItem

```
// src/components/layout/Main/MenuTreeItem.tsx
import React, { useState } from "react";import { useLocation, useNavigate } from "react-router-dom";import { Collapse, List, ListItemButton, ListItemIcon, ListItemText } from "@mui/material";import { ExpandLess, ExpandMore } from "@mui/icons-material";import { DEFAULT_OPEN_MODE, type
```

```
MenuNode } from "@/routes";
import type { Session } from "@/lib/curio";import { useAuth } from "@context/AuthProvider";
import { useTabs } from "@context/TabsContext";import { useHookMutation } from
"@hooks/useHookMutation";
const noopAction = async () => undefined;
// renderizacao recursiva do menuTree.// folha: executa action, ou abre por rota (<Outlet/>) ou
por aba — conforme node.mode.const MenuTreeItem: React.FC<{ node: MenuNode; depth: number }> =
({ node, depth }) => {
  const navigate = useNavigate();
  const location = useLocation();
  const { session } = useAuth();
  const { openTab, setActiveTab, activeTabId } = useTabs();  const [open, setOpen] =
useState(false);
  // hook antes do early return de nó pai — ordem de hooks nao pode variar  const { mutateAsync:
executeAction, isPending } = useHookMutation<void, Session | undefined>(    node.action ??
noopAction
  );
  const Icon = node.icon;
  const pl = (depth + 1) * 2; // indenta por nivel
  if (node.children) {
    return (
      <>
        <ListItemButton sx={{ pl }} onClick={() => setOpen((prev) => !prev)}>          {Icon &&
(
            <ListItemIcon>
              <Icon />
            </ListItemIcon>
          )}
          <ListItemText primary={node.label} />          {open ? <ExpandLess /> : <ExpandMore
/>}
        </ListItemButton>
        <Collapse in={open} unmountOnExit>
          <List disablePadding>
            {node.children.map((child) => (              <MenuTreeItem key={child.label}
node={child} depth={depth + 1} />
            ))}
          </List>
```

```
        </Collapse>
      </>
    );
  }
  const mode = node.mode ?? DEFAULT_OPEN_MODE;
  const handleClick = async () => {
    // action tem precedencia: executa e nao navega
    if (node.action) {
      await executeAction(session);
      return;
    }
    if (mode === "tab") {
      openTab(node);
      return;
    }    // rota: precisa zerar a aba ativa, senao o painel dela continua por cima do
<Outlet/>
    setActiveTab(null);
    navigate(`/${node.path}`);
  };
  // so destaca a rota atual quando nenhuma aba esta ativa  const isSelected = !node.action &&
mode === "route" && activeTabId === null && location.pathname === `/${node.path}`;
  return (    <ListItemButton sx={{ pl }} selected={isSelected} disabled={isPending}
onClick={handleClick}>
      {Icon && (
        <ListItemIcon>
          <Icon />
        </ListItemIcon>
      )}
      <ListItemText primary={node.label} />
    </ListItemButton>
  );
};
export default MenuTreeItem;
```

O `setActiveTab(null)` ao navegar por rota não é detalhe. Sem ele, o painel da aba ativa continua visível e o `<Outlet/>` fica escondido atrás — o clique parece não fazer nada.

TabsContext

Guarda as abas, a ativa, e o registro de callbacks de fechamento.

Meml	Para que serve
`tabs` / `activeTabId`	Estado das abas
`openTab(node)`	Abre o nó numa aba nova; ignora nó sem `path`
`closeTab(id)`	Dispara o callback registrado, remove a aba, escolhe a próxima ativa
`setActiveTab(id | null)`	`null` devolve a tela ao `<Outlet/>`

Meml	Para que serve
`warningOpen` / `dismissWarning`	Aviso de limite de abas atingido
`registerCloseCallback` / `unregis`	Base do `useTabCloseCallback`

Decisões embutidas:

- **`MAX_TABS = 8`.** Passar disso vira aviso, não aba. Cada aba mantém um caso de uso aberto no servidor — o limite protege o backend, não só a interface.
- **Persistência em `localStorage`** (`"<projeto>:tabs"`). As abas sobrevivem ao refresh; o **estado interno das telas não** — elas remontam vazias. Só a lista de abas é restaurada.
- **Rótulo duplicado vira `(2)`.** Duas abas da mesma tela se distinguem.
- **Fechar a última aba navega para o dashboard**, evitando tela em branco.

TabPageRenderer

Resolve o `path` da aba na **mesma tabela `routes.ts`** que o router usa — não há registro paralelo de telas.

```
// src/components/layout/TabPageRenderer/TabPageRenderer.tsxconst TabPageRenderer: React.FC<{ path: string }> = ({ path }) => {  const route = routes.find((r) => r.path === path);  if (!route) return <Box sx={{ p: 3 }}>Rota não encontrada: {path}</Box>;  const PageComponent = route.element;  const Layout = route.layout as React.FC<{ children?: React.ReactNode }> | undefined;  const page = (    <Suspense fallback={loadingFallback}>      <PageComponent />    </Suspense>
```

```
  );
  // layout de rota recebe a pagina como children (fora de aba ele usa <Outlet/>)  const content
= Layout ? <Layout>{page}</Layout> : page;
  return <TabErrorBoundary>{content}</TabErrorBoundary>;
};
```

Dois cuidados:

- `Suspense` **por aba.** As páginas são `lazy()`; sem isso, abrir uma aba suspenderia o shell inteiro.
- `ErrorBoundary` **por aba.** Erro numa aba não pode derrubar as outras. O boundary oferece "Tentar novamente" em vez de tela branca.

> **Layout em aba recebe `children`, não `<Outlet/>`.** Se você usa `layout` em `routes.ts` (10), o componente de layout precisa renderizar `children` **e** funcionar com `<Outlet/>` fora de aba. O mais simples é aceitar `children` opcional e cair para `<Outlet/>` quando ele não vier.

Fechando o caso de uso da aba

Como o componente não desmonta ao fechar a aba, o `close()` do caso de uso precisa ser explícito:

```
// src/pages/Fornecedor/Cadastro/IncluirFornecedorPage.tsximport { useUseCaseControls,
useTabCloseCallback } from "@hooks/index";
const IncluirFornecedorContent: React.FC = () => {  const { open, close, status } =
useUseCaseControls();
  useTabCloseCallback(close); // encerra o caso de uso quando a aba fechar
  // ...
};
```

`useTabCloseCallback` é no-op quando a página veio pelo `<Outlet/>` (sem aba) — a mesma página serve aos dois modos sem `if`.

Esquecer isso vaza caso de uso no servidor: o usuário fecha a aba, o front esquece dela, e o backend segue com a sessão do caso de uso aberta até expirar.

Para a página se fechar sozinha (botão "Sair"):

```
const { closeTab } = useTabs();
const tabId = useCurrentTabId();
const handleSair = () => {
  if (tabId) closeTab(tabId);
};
```

Registrar uma tela nova

Quatro passos, mesmo commit:

1. `paths.ts` — constante do caminho
2. `routes.ts` — rota com `lazy()` e `guard`
3. `menuTree.ts` — nó apontando para a constante, com `mode` se diferir do default
4. **Permissão** — se o projeto tiver controle de acesso

Pular o 3 dá rota acessível só por URL (às vezes é o que se quer). Pular o 2 dá item de menu que abre 404 no modo rota, ou "Rota não encontrada" no modo aba.

Ícones

```
import { Business } from "@mui/icons-material";
{ label: "Fornecedor", path: PATHS.CADASTRO.FORNECEDOR, icon: Business }
```

Passe o **componente**, não o elemento (`Business`, não `<Business />`). Se adotar ícones, use em todos os itens de primeiro nível — meio caminho fica pior que nenhum.

Menu, rota e permissão

Reg	Arq	Responde a
Cam	`paths.ts`	Qual é a URL
Rota	`routes.ts`	Que componente, sob qual guard
Menu	`menuTree.ts`	Onde aparece e como abre
Permissão	backend	Quem pode ver/usar

O menu não filtra por permissão. Todos os nós aparecem para qualquer usuário autenticado e o backend recusa a operação. Para esconder itens, acrescente `permissao?: string` ao `MenuNode` e filtre na renderização — é trabalho novo, não vem pronto.

Itens de menu que executam ação

Uma folha pode **executar uma função** em vez de abrir tela — relatório que só gera um PDF, disparo pontual sem interface própria.

```
// src/routes/menuTree.ts
{
  label: "Relatório de caixas abertas",
  action: handleAbrirRelatorioCaixas
}
```

```
// src/pages/Caixa/service/handlers.tsconst CAIXAS_ABERTAS = { USE_CASE_ID: "2702", RM:
"RM_OBTEM_CAIXAS_ABERTAS" };
export const handleAbrirRelatorioCaixas = async (session: Session | undefined) => {  if
(!session) return;
  const uc = await session.openUseCase(CAIXAS_ABERTAS.USE_CASE_ID);
  try {
    const result: ObtemRelatorioResponse = await uc.sendRequest(CAIXAS_ABERTAS.RM);    await
handleOpenReport(result.URIRelatorio._);
  } finally {
    uc.abort(); // sempre fecha — nao ha tela dona deste caso de uso
  }
};
```

O `MenuTreeItem` executa via `useHookMutation` (21):

```
const noopAction = async () => undefined;
// hook antes do early return de no pai — ordem de hooks nao pode variarconst { mutateAsync:
executeAction, isPending } = useHookMutation<void, Session | undefined>(  node.action ??
noopAction
);
const handleClick = async () => {
  // action tem precedencia: executa e nao navega
  if (node.action) {
    await executeAction(session);
    return;
  }
  // ... modo tab / route
};
```

Quatro pontos que não são opcionais:

- **O hook fica antes do `if (node.children)`.** Hook depois de early return muda a ordem entre renders e o React quebra. Daí o `noopAction` para nós que não têm `action`.
- **`action` tem precedência sobre `mode`.** Um nó com `action` não navega nem abre aba.
- **`isPending` desabilita o item** enquanto executa, evitando disparo duplo.
- **Erro vira notificação** pelo `mutationCache` global (07) — o handler não precisa de `try/catch` para exibir mensagem, só do `finally` que fecha o caso de uso.

> **Tipagem.** `action` é `(session: Session | undefined) => Promise<void>`. Declarar como `(params: unknown) => Promise<void>` exige um cast em cada nó do menu — não replique esse padrão.

Verificação

- [ ] Item `mode: "route"` navega e a URL muda
- [ ] Item `mode: "tab"` abre aba e a URL **não** muda
- [ ] Com aba ativa, clicar num item de rota esconde o painel e mostra o `<Outlet/>`
- [ ] Preencher um campo numa aba, trocar de aba e voltar — o valor continua lá
- [ ] Fechar aba dispara o `close()` do caso de uso (verifique no Network)
- [ ] Fechar a última aba volta ao dashboard
- [ ] Abrir 9 abas mostra o aviso de limite na nona
- [ ] F5 restaura a lista de abas
- [ ] Abrir duas vezes a mesma tela gera "Título" e "Título (2)"

Uma implementação que só usa abas (com o dashboard tratado à parte por um `if` sobre string literal) é um ponto de partida comum; o modo híbrido com `MenuOpenMode` é a generalização adotada aqui.

10 — Rotas e proteção

10 — Rotas e proteção

> Três arquivos de dados (`paths.ts`, `routes.ts`, `menuTree.ts`) e um de comportamento (`AppRouter.tsx`). Toda rota é declarada em tabela, nunca em JSX espalhado. Todo componente de página é carregado com `lazy()`.

Separação dados / comportamento

Pas	Cor	Pode ter JSX?
`src/routes`	Dados (constantes)	Não
`src/router`	Montagem do router	Sim

Isso permite consumir a tabela de rotas para outros fins (menu, breadcrumb, verificação de permissão) sem arrastar o React Router junto.

`paths.ts`

```
// src/routes/paths.ts
export const PATHS = {
  LOGIN: "login",
  DASHBOARD: "dashboard",
  CADASTRO: {
    FORNECEDOR: "cadastro/fornecedor",
    TIPO_FORNECEDOR: {
      INCLUIR: "cadastro/tipo-fornecedor/incluir",      ALTERAR: "cadastro/tipo-
fornecedor/alterar",
      EXCLUIR: "cadastro/tipo-fornecedor/excluir"
    }
  },
  RELATORIOS: {
    MENSAL: "relatorios/mensal",
    ANUAL: "relatorios/anual"
  }
} as const;
```

Regras:

- **Sem barra inicial.** `AppRouter` concatena (`path={`/${path}`}`). Uma barra a mais gera `//rota`.
- `as const` no final — dá tipos literais e impede mutação acidental.
- Chaves em `SCREAMING_SNAKE_CASE`; valores em `kebab-case`.
- Aninhamento espelha a hierarquia de URL, e normalmente a do menu.
- Português nos segmentos de URL (é sistema interno em pt-BR). **Escolha um idioma e mantenha** — ver os anti-padrões em 13.

routes.ts

```
// src/routes/routes.ts
import { lazy } from "react";
import { PATHS } from "./paths";
import { RelatoriosLayout } from "@pages/Relatorios";
export type RouteGuard = "public" | "protected" | "auth-only";
```

```
export interface AppRoute {
  path: string;
  element: React.LazyExoticComponent<React.ComponentType>;
  guard: RouteGuard;
  /** Layout intermediário opcional entre o shell e a página */
  layout?: React.ComponentType;
}
export const routes: AppRoute[] = [
  {
    path: PATHS.LOGIN,
    element: lazy(() => import("@pages/Login")),
    guard: "auth-only"
  },
  {
    path: PATHS.DASHBOARD,
    element: lazy(() => import("@pages/Dashboard")),
    guard: "protected"
  },
  {
    path: PATHS.CADASTRO.FORNECEDOR,    element: lazy(() =>
import("@pages/Cadastro/Fornecedor")),
    guard: "protected"
  },
  {
    path: PATHS.RELATORIOS.MENSAL,    element: lazy(() =>
import("@pages/Relatorios/Mensal")),
    guard: "protected",
    layout: RelatoriosLayout
  }
];
export const NotFoundPage = lazy(() => import("@pages/NotFound"));
```

Guards

Gua	Cor	Uso	tamento

`auth-only`	Só para **não** autenticados. Autenticado é redirecionado ao dashboard	Login
`public`	Acessível sempre, sem ajuda shell	Termos, Ajuda
`protected`	Exige sessão; renderiza dentro do shell	Todo o resto (`Main`)

Não existe default: `guard` é obrigatório. Isso é proposital — esquecer torna a rota pública por acidente, e o compilador impede.

Lazy sempre

```
element: lazy(() => import("@pages/Cadastro/Fornecedor"));
```

Toda página é lazy. Sem exceção — até a de login. Cada tela vira um chunk próprio; o bundle inicial não cresce com o sistema. Por isso `pages/<Tela>/index.ts` com `export default` é obrigatório.

Layouts intermediários

`layout` insere um componente entre o shell e a página, para grupos de telas que compartilham sub-navegação (abas internas, cabeçalho de seção). O `AppRouter` agrupa rotas por layout automaticamente. Omita quando não houver.

AppRouter.tsx

```
// src/router/AppRouter.tsx
import React, { Suspense } from "react";import { Routes, Route, Navigate } from "react-router-dom";
import { useAuth } from "@context/AuthProvider";import ProtectedRoute from "@components/ProtectedRoute";
import LoadingScreen from "@components/LoadingScreen";import Main from "@components/layout/Main";
import { routes, NotFoundPage, type AppRoute } from "@/routes";
// agrupa por layout pra montar uma <Route> pai por layoutfunction groupByLayout(routeList: AppRoute[]) {  return routeList.reduce<Map<React.ComponentType | null, AppRoute[]>>((map, route) => {
    const key = route.layout ?? null;
    if (!map.has(key)) map.set(key, []);
    map.get(key)!.push(route);
    return map;
  }, new Map());
}
const AppRouter: React.FC = () => {
  const { isAuth } = useAuth();
  const authOnlyRoutes = routes.filter((r) => r.guard === "auth-only");  const publicRoutes = routes.filter((r) => r.guard === "public");  const protectedRoutes = routes.filter((r) => r.guard === "protected");
  const protectedByLayout = groupByLayout(protectedRoutes);
  return (
    <Suspense fallback={<LoadingScreen />}>
      <Routes>        <Route path="/" element={<Navigate to={isAuth ? "/dashboard" : "/login"} replace />} />
        {authOnlyRoutes.map(({ path, element: Element }) => (          <Route key={path} path={`/${path}`} element={isAuth ? <Navigate to="/dashboard" replace /> : <Element />}
```

```
/>
        ))}
        {publicRoutes.map(({ path, element: Element }) => (          <Route key={path}
path={`/${path}`} element={<Element />} />
        ))}
        <Route
          element={
            <ProtectedRoute>
              <Main />
            </ProtectedRoute>
          }>          {[...protectedByLayout.entries()].map(([Layout, layoutRoutes]) =>
{
            const children = layoutRoutes.map(({ path, element: Element }) => (
<Route key={path} path={`/${path}`} element={<Element />} />
            ));
            if (!Layout) return <React.Fragment
key="__root">{children}</React.Fragment>;
            return (              <Route key={Layout.displayName ?? Layout.name}
element={<Layout />}>
                {children}
              </Route>
            );
          })}
        </Route>
        <Route path="*" element={<NotFoundPage />} />
      </Routes>
    </Suspense>
  );
};
export default AppRouter;
```

Pontos que não são acidentais:

- **Um `<Suspense>` na raiz** cobre todos os `lazy()`. Não envolva página por página.
- **Rotas protegidas ficam sob uma única `<Route>` pai** com `ProtectedRoute` + `Main` — o shell não remonta ao navegar entre telas protegidas.

- `path="*"` **por último** captura o 404.
- `/` **redireciona** conforme `isAuth`.

ProtectedRoute

```
// src/components/ProtectedRoute/ProtectedRoute.tsx
import React from "react";
import { Navigate, useLocation } from "react-router-dom";import { useAuth } from
"@context/AuthProvider";
import LoadingScreen from "@components/LoadingScreen";
const ProtectedRoute: React.FC<{ children: React.ReactNode }> = ({ children }) => {  const {
isAuth, isLoading } = useAuth();
  const location = useLocation();
  // enquanto reconecta por token, nao decidir ainda
  if (isLoading) return <LoadingScreen />;  if (!isAuth) return <Navigate to="/login" state={{
from: location }} replace />;
  return <>{children}</>;
};
export default ProtectedRoute;
```

O `isLoading` é essencial. Sem ele, um refresh de página redireciona para o login antes da reconexão por token terminar. Foi por isso que `App.tsx` também exibe `LoadingScreen` enquanto `isLoading` — ver 05.

`state={{ from: location }}` preserva o destino para redirecionar depois do login.

Adicionar uma rota

```
// 1. src/routes/paths.ts
CADASTRO: {
  FORNECEDOR: "cadastro/fornecedor",
  CLIENTE: "cadastro/cliente"          // novo
```

```
}
// 2. src/routes/routes.ts
{
  path: PATHS.CADASTRO.CLIENTE,
  element: lazy(() => import("@pages/Cadastro/Cliente")),
  guard: "protected"
}
// 3. src/routes/menuTree.ts
{ label: "Cliente", path: PATHS.CADASTRO.CLIENTE }
```

E crie `src/pages/Cadastro/Cliente/index.ts` com `export default`, senão o `lazy()` falha em runtime sem erro de compilação.

Rotas com parâmetro

O padrão declarativo suporta parâmetro na string:

```
{
  path: "cadastro/fornecedor/:oid",  element: lazy(() =>
import("@pages/Cadastro/Fornecedor/Detalhe")),
  guard: "protected"
}
```

Na página, `useParams()`. **Não coloque rota parametrizada no `menuTree`** — não há valor de parâmetro para navegar a partir do menu.

> Com navegação por abas, rotas parametrizadas costumam ser dispensáveis: o contexto passa pelo `TabsContext` em vez de pela URL. Se o projeto novo dispensar abas, rotas parametrizadas voltam a ser o caminho natural.

11 — Formulários e validação

11 — Formulários e validação

> react-hook-form + zod, sempre via `useValidatedForm`. Schema é a fonte única do formato: o tipo do formulário é **inferido** dele, nunca escrito à mão. Campos do formulário usam os mesmos nomes `_Prefixo` do backend.

`useValidatedForm`

```
// src/hooks/useValidatedForm.tsimport { useForm, UseFormProps, UseFormReturn } from "react-hook-form";
import { zodResolver } from "@hookform/resolvers/zod";
import { z } from "zod";
interface UseValidatedFormProps<T extends z.ZodType> extends Omit<UseFormProps<z.infer<T>>, "resolver"> {
  schema: T;
}
export const useValidatedForm = <T extends z.ZodType>({
  schema,
  ...formProps
}: UseValidatedFormProps<T>): UseFormReturn<z.infer<T>> =>
  useForm<z.infer<T>>({
    resolver: zodResolver(schema),
```

```
    mode: "onChange", // valida em tempo real
    ...formProps
  });
```

Use sempre este hook, nunca `useForm` direto. Ele garante o resolver e o `mode: "onChange"` uniformes.

`mode: "onChange"` valida a cada digitação. Para formulário grande e caro de validar, sobrescreva para `"onBlur"` — o spread permite.

schemas.ts

Um arquivo por tela, ao lado do componente.

```
// src/pages/Fornecedor/Cadastro/schemas.ts
import { z } from "zod";
export const fornecedorSchema = z.object({  _Nome: z.string().min(1, "Informe o nome").max(120,
"Máximo de 120 caracteres"),
  _CNPJ: z
    .string()
    .min(1, "Informe o CNPJ")
    .regex(/^\d{2}\.\d{3}\.\d{3}\/\d{4}-\d{2}$/, "CNPJ inválido"),  _Email: z.string().email("E-
mail inválido").optional().or(z.literal("")),
  _Ativo: z.boolean(),
  _DataCadastro: z.string().min(1, "Informe a data")
});
export type FornecedorFormData = z.infer<typeof fornecedorSchema>;
export const filtroFornecedorSchema = z.object({
  _Nome: z.string(),
  _Ativo: z.boolean()
});
export type FiltroFornecedorData = z.infer<typeof filtroFornecedorSchema>;
```

Regras:

- **Nunca escreva a interface do formulário à mão.** Sempre `z.infer<typeof schema>`. Escrever

as duas cria divergência silenciosa.

- Toda regra tem mensagem em português — ela vai direto para a tela.
- Nomes de campo espelham o backend (`_Nome`), evitando mapeamento no submit.
- Um schema por formulário. Tela com dois formulários independentes tem dois schemas.

Campo opcional que aceita vazio

```
_Email: z.string().email("E-mail inválido").optional().or(z.literal(""));
```

`.optional()` sozinho não basta: um `TextField` limpo devolve `""`, não `undefined`, e `""` falha na validação de e-mail.

O formulário

Duas formas, ambas válidas.

Com `FormField` (preferível)

```
import { FormField } from "@components/common";
<FormField name="_Nome" control={form.control} label="Nome" size="small" fullWidth />;
```

Encapsula o `Controller` e liga `error`/`helperText` ao `fieldState`. Use quando o campo é um `TextField` comum.

Com `Controller` (quando precisa do componente MUI cru)

```
<Controller
  name="_Nome"
```

```
  control={form.control}
  render={({ field, fieldState }) => (
    <TextField
      {...field}
      label="Nome"
      size="small"
      fullWidth
      error={!!fieldState.error}
      helperText={fieldState.error?.message}
    />
  )}
/>
```

Mais verboso, mas necessário quando o componente não é um `TextField` ou exige props que o `FormField` não repassa.

> Evite usar `Controller` direto quando `FormField` bastaria — **prefira `FormField`** e reserve `Controller` para os casos que realmente precisam do componente MUI cru.

Submit

```
const form = useValidatedForm({
  schema: fornecedorSchema,  defaultValues: { _Nome: "", _CNPJ: "", _Ativo: true, _DataCadastro:
dataHoje }
});
const handleSalvar = async (data: FornecedorFormData) => {
  await salvarAsync({
    Fornecedor: {
      _OID: String(fornecedorData?.Fornecedor._OID ?? ""),
      _Nome: data._Nome,
      _CNPJ: data._CNPJ
    },
```

```
    msgSucesso: "Fornecedor salvo com sucesso!"
  });
};
<Button onClick={form.handleSubmit(handleSalvar)}>Salvar</Button>;
```

`form.handleSubmit(fn)` só chama `fn` se a validação passar. **Não valide manualmente antes** — é o que o resolver faz.

`defaultValues` é obrigatório

Sempre forneça `defaultValues` com todos os campos. Sem ele, o campo nasce não-controlado e vira controlado na primeira digitação — o React emite warning e o `reset()` não limpa direito.

Reset

```
const handleNovoRegistro = () => {
  form.reset(); // volta aos defaultValues  filtroForm.reset({ _Nome: "", _Ativo: true }); //
valores explicitos
};
```

Inputs com máscara

`CnpjInput`, `PhoneInput`, `CurrencyInput` e `DatePickerInput` de `common/` — ver 08.

Moeda precisa de conversão no submit:

```
import { CurrencyInput, parseCurrencyValue } from "@components/common";
const handleSalvar = (data: FormData) => {  salvar({ Fornecedor: { _Limite:
parseCurrencyValue(data._Limite) } });
};
```

O campo guarda o texto formatado (`"1.234,56"`); o backend quer número. `parseCurrencyValue` faz a ponte. Esquecer isso envia string e o backend rejeita.

Datas

O backend Curio espera data com tempo:

```
const dataFormatada = `${data._DataExpiracao}T00:00:00.0`;
```

O campo do formulário guarda `"2026-08-18"`; o request precisa de `"2026-08-18T00:00:00.0"`. Faça a conversão **no handler**, não no schema — o schema descreve o formulário, não o payload.

Para gerar a data de hoje no formato do input:

```
const hoje = new Date();const dataHoje = `${hoje.getFullYear()}-${String(hoje.getMonth() +
1).padStart(2, "0")}-${String(hoje.getDate()).padStart(2, "0")}`;
```

Não use `toISOString().split("T")[0]` — converte para UTC e retorna o dia anterior à noite no fuso brasileiro.

Validação vinda do servidor

O backend valida de novo. Um `ValidationError` do Curio traz `detail: string[]`, já convertido em notificação pelo `queryClient` (07).

Não tente espelhar toda regra de servidor no zod. O zod cobre o que dá para verificar no cliente (obrigatório, formato, tamanho); regra de negócio é do servidor.

Erros comuns

Sinto	Causa
"changing an uncontrolled input"	Faltou o campo em `defaultValues`
`reset()` não limpa	Idem
Submit não dispara e nada aparece	Validação falhou num campo sem `FormField` / `helperText` visível
Backend rejeita a data	Faltou o sufixo `T00:00:00.0`
Backend recebe moeda como texto	Faltou `parseCurrencyValue`
Tipo do form diverge do schema	Interface escrita à mão em vez de `z.infer`

12 — Tema e estilo

12 — Tema e estilo

> Um `createTheme` central, locale pt-BR, e a regra de quando usar `sx` versus `styled`. Estilo repetido em duas telas vira tema; estilo repetido entre 2+ **componentes** vira `theme/commonStyles.ts`. Sem dark mode na referência — se o projeto precisar, é decisão do início.

O tema

```
// src/theme/index.ts
import { createTheme } from "@mui/material/styles";
import { ptBR } from "@mui/material/locale";
const palette = {  primary: { main: "#1976d2", light: "#42a5f5", dark: "#1565c0", contrastText:
"#ffffff" },  secondary: { main: "#dc004e", light: "#ff5983", dark: "#9a0036", contrastText:
"#ffffff" },  error: { main: "#f44336", light: "#e57373", dark: "#d32f2f", contrastText:
"#ffffff" },  warning: { main: "#ff9800", light: "#ffb74d", dark: "#f57c00", contrastText:
"#000000" },  info: { main: "#2196f3", light: "#64b5f6", dark: "#1976d2", contrastText:
"#ffffff" },  success: { main: "#4caf50", light: "#81c784", dark: "#388e3c", contrastText:
"#ffffff" }
};
export const theme = createTheme(
```

```
  {
    palette,
    typography: {
      fontFamily: '"Roboto", "Helvetica", "Arial", sans-serif'      // escala h1..overline
definida explicitamente — ver arquivo de referencia
    },
    shape: { borderRadius: 8 },
    spacing: 8,
    components: {
      MuiButton: {
        styleOverrides: {          root: { textTransform: "none", borderRadius: 8, fontWeight:
500, padding: "8px 16px" },
          contained: {
            boxShadow: "0 2px 4px rgba(0,0,0,0.1)",            "&:hover": { boxShadow: "0 4px
8px rgba(0,0,0,0.15)" }
          }
        }
      },
      MuiCard: {        styleOverrides: { root: { boxShadow: "0 2px 8px rgba(0,0,0,0.1)",
borderRadius: 12 } }
      },
      MuiTextField: {        styleOverrides: { root: { "& .MuiOutlinedInput-root": {
borderRadius: 8 } } }
      },
      MuiPaper: {
        // remove gradiente de elevacao do MUI        styleOverrides: { root: { backgroundImage:
"none" } }
      }
    }
  },
  ptBR // locale dos componentes MUI
);
export default theme;
```

O segundo argumento `ptBR` traduz textos internos do MUI (paginação, date pickers). É separado do `adapterLocale={ptBR}` de `date-fns` em `main.tsx` — **os dois são necessários** e vêm de pacotes

diferentes:

```
import { ptBR } from "@mui/material/locale"; // textos dos componentesimport { ptBR } from "date-fns/locale"; // formatação de datas
```

`sx` vs `styled` vs arquivo de estilo

Situa	Use
1–2 propriedades pontuais	`sx` inline
Bloco de estilo de um componente	Constante nomeada em `Componente.styles.ts`
Estilo que depende de prop/estado	Função em `.styles.ts` que recebe o valor
Componente novo com estilo próprio e complexo	`styled()` em `.styles.ts`
Estilo repetido em 2+ **componentes**	`src/theme/commonStyles.ts`

Situa	Use
Estilo repetido em 2+ **telas via MUI slot**	Tema (`components.styleOverrides`)

`sx` inline — pouco e óbvio

```
<Typography variant="subtitle2" sx={{ mr: 2 }}>
```

Aceitável até ~2 propriedades. Acima disso, vai para o arquivo de estilos.

`.styles.ts` — uma constante nomeada por estilo

Convenção do padrão: cada estilo é uma **constante exportada e nomeada**, não uma propriedade de um objeto `styles`. O nome descreve o que o estilo faz, sufixado com `Style`.

```
// src/pages/Cadastro/Fornecedor/FornecedorPage.styles.tsimport { SxProps, Theme } from
"@mui/material";
export const outerBoxStyle: SxProps<Theme> = {
  flexGrow: 1,
  p: 3,
  height: "calc(100vh - 64px)",
  overflow: "hidden"
};
export const listLoadingStyle: SxProps<Theme> = {
  display: "flex",
  justifyContent: "center",
  alignItems: "center",
  height: "100%"
```

```
};
export const unselectedItemStyle: SxProps<Theme> = {
  display: "flex",
  justifyContent: "center",
  alignItems: "center",
  height: "100%"
};
```

```
import { listLoadingStyle, outerBoxStyle, unselectedItemStyle } from "./FornecedorPage.styles";
<Box sx={outerBoxStyle}>
  <Box sx={listLoadingStyle}>...</Box>
</Box>;
```

Por que constantes nomeadas e não um objeto `styles = { root, header, ... }`:

- **O import fica explícito** — `import { outerBoxStyle }` mostra exatamente o que a tela usa; um objeto `styles` obriga a abrir o arquivo para saber quais chaves existem.
- **Renomear é seguro.** Renomear `styles.root` para outra coisa exige revisar todo uso de `styles.` no arquivo; renomear `outerBoxStyle` é um rename de símbolo, com suporte de qualquer editor.
- **Facilita promover para `commonStyles.ts`** — mover uma constante exportada para outro arquivo é um corte-e-cola; extrair uma chave de dentro de um objeto exige reescrever os dois lados.

Tipar como `SxProps<Theme>` dá autocomplete e valida os tokens.

Estilo dependente de estado — função nomeada

```
// src/components/layout/Main/Main.styles.ts
import { SxProps, Theme } from "@mui/material";
const DRAWER_WIDTH = 260;
```

```
export const mainRootStyle: SxProps<Theme> = {
  display: "flex",
  minHeight: "100vh"
};
// depende do estado do drawer — por isso funcao, nao constanteexport const drawerStyle = (open:
boolean): SxProps<Theme> => ({
  width: open ? DRAWER_WIDTH : 0,
  flexShrink: 0,
  "& .MuiDrawer-paper": { width: DRAWER_WIDTH, boxSizing: "border-box" }
});
```

```
<Drawer sx={drawerStyle(drawerOpen)} />
```

Mesma regra: nome exportado, não uma chave de objeto. A única diferença é que o valor é uma função.

`styled()`

```
import { styled } from "@mui/material/styles";
import { Box } from "@mui/material";
export const PainelDestacado = styled(Box)(({ theme }) => ({
  padding: theme.spacing(2),
  borderRadius: theme.shape.borderRadius,
  backgroundColor: theme.palette.grey[100]
}));
```

Use quando o resultado é um **componente** reutilizável, não um conjunto de props de estilo.

Sempre use tokens do tema

```
// ERRADO — valor solto
<Box sx={{ padding: "16px", color: "#1976d2", borderRadius: "8px" }} />
```

```
// CERTO — tokens
<Box sx={{ p: 2, color: "primary.main", borderRadius: 1 }} />
```

`spacing: 8` significa que `p: 2` = 16px. Mudar o espaçamento base reajusta o app inteiro; valores soltos não acompanham.

Atalhos comuns: `p` / `m` (padding/margin), `px` / `py`, `mt` / `mb` / `ml` / `mr`, `gap`.

Estilo repetido vira `commonStyles.ts` ou tema

Uma constante de `.styles.ts` nasce local ao componente. Quando a **segunda** tela precisar do mesmo estilo, promova-a — não copie a definição.

```
// src/theme/commonStyles.ts
import { SxProps, Theme } from "@mui/material";
// Estilos reaproveitados por mais de uma tela.// Regra: nasce em <Componente>.styles.ts; move pra ca quando a 2a tela precisar.
/** Centraliza o conteudo nos dois eixos ocupando toda a altura disponivel. */export const centeredFillStyle: SxProps<Theme> = {
  display: "flex",
  justifyContent: "center",
  alignItems: "center",
  height: "100%"
};
/** Area de conteudo de uma tela dentro do shell — desconta a AppBar. */export const outerBoxStyle: SxProps<Theme> = {
  flexGrow: 1,
  p: 3,
  height: "calc(100vh - 64px)",
  overflow: "hidden"
};
```

Quando o repetido é uma propriedade de um **slot do MUI** (todo `TextField` da aplicação, todo `Button`), o lugar certo é o tema, não `commonStyles.ts`:

```
// src/theme/index.ts
components: {
  MuiTextField: {    styleOverrides: { root: { "& .MuiOutlinedInput-root": { borderRadius: 8 } } }
}
  }
}
```

O mesmo vale para `defaultProps`:

```
components: {
  MuiTextField: { defaultProps: { size: "small", fullWidth: true } }
}
```

Isso elimina `size="small" fullWidth` repetido em cada campo. **Defina `defaultProps` desde o início** — evita repetir as mesmas props em toda tela.

Regra de decisão:

O que se repet	Vai para
Um layout específico (centra área de conteúdo)	`theme/commonStyles.ts`
Uma propriedade de todo compo MUI de um tipo	`theme/index.ts`

Tipografia

Use `variant`, não `fontSize`:

```
// ERRADO
<Typography sx={{ fontSize: "1.25rem", fontWeight: 400 }}>Título</Typography>
// CERTO
```

```
<Typography variant="h3">Título</Typography>
```

A escala está definida no tema. Se um tamanho não existe lá, ou você quer a variante errada, ou falta uma variante no tema.

Cores

Só da paleta:

```
<Typography color="text.secondary" /><Box sx={{ bgcolor: "background.paper", borderColor:
"divider" }} />
<Button color="primary" />
```

Nenhum hex fora de `src/theme/index.ts`. Se precisar de uma cor nova, adicione à paleta.

Dark mode

A referência não tem. Se o projeto novo precisar:

1. Extraia a paleta para `light` e `dark`.
2. Crie o tema com `mode` vindo de estado (`useMediaQuery("(prefers-color-scheme: dark)")` + preferência do usuário em `localStorage`).
3. Envolva com `useMemo` para não recriar a cada render.
4. **Auditar todo `sx`** — qualquer hex solto quebra no modo escuro.

Adotar depois é caro exatamente pelo passo 4. Decida no início.

Responsividade

Breakpoints do MUI dentro do `sx`:

```
<Box sx={{ display: { xs: "none", md: "block" }, p: { xs: 1, md: 3 } }} />
```

Se o projeto precisar de suporte a mobile, trate desde o começo — retrofitar layout responsivo é reescrever telas.

> **Este documento adota constantes nomeadas** como padrão — uma constante por estilo, em vez de um objeto `styles = { root, drawer, ... }` por arquivo — ver a comparação em "`.styles.ts` — uma constante nomeada por estilo" acima.