Backend (Curio)

Conexão e sessão Curio, caso de uso e React Query.

- 05 — Curio: conexão e sessão
- 06 — Caso de uso
- 07 — React Query

05 — Curio: conexão e sessão

05 — Curio: conexão e sessão

> Como o front autentica, mantém e encerra a sessão com o backend Curio. Toda a integração com `@curio/client` está confinada em `src/lib/curio/` e `src/api/session.ts`. Nenhuma página importa `@curio/client` para configurar transporte.

Camadas

```
Página
  └─ useAuth()                src/context/AuthProvider.tsx    ← o que a página usa
└─ useAuthQuery()       src/hooks/useAuthQuery.ts       estado via React Query
└─ login/connect   src/api/session.ts              operações de sessão
└─ getSessionManager()  src/lib/curio/index.ts  transporte
                      └─ @curio/client
```

Componentes usam `useAuth()`. Nunca `useAuthQuery` diretamente, nunca `getSessionManager()`.

`src/lib/curio/`

Quatro arquivos.

`index.ts` — fábrica do transporte

```
// src/lib/curio/index.ts
import { V3RequestParser } from "@curio/client/parsers";import { Session } from "./Session";
import { SessionManager } from "./SessionManager";import { createV3Validator } from
"@curio/client/validators";
import { FetchLink } from "@curio/client/links";
export interface Config {  service: { url: string; server: string; system: string; port: string
};
  accessToken: string;
  resources: { get: string; put: string; viewer: string };
  logs: boolean;
}
const createSessionManager = (configuration: Config) => {  const link = new
FetchLink(configuration.service);  const requestParser = new
V3RequestParser(configuration.service);
  // pipeline: request -> parse -> json -> post -> valida  return new
SessionManager(configuration.service, (request) =>
    Promise.resolve(request)
      .then(requestParser.parse)
      .then(JSON.stringify)
      .then(link.parse)
      .then(link.post)
      .then((response) => response.json())
      .then(createV3Validator(request).validate)
  );
};
export const getSessionManager = async () => {
  const CONFIG_PATH = "./config.json";
  const cnfg = await (await fetch(CONFIG_PATH)).json();
  return createSessionManager(cnfg);
};
export type { Service as SV } from "./SessionManager";
```

```
export { Session, SessionManager };
export { isAuthError } from "./isAuthError";
export * from "./SessionManager";
```

> “ `getSessionManager()` **cria uma instância nova a cada chamada** — não é singleton, apesar do nome. Isso é aceitável porque só é chamado em `login`, `connect` e em requests anônimos. **Não chame dentro de componente ou hook de página.** Para falar com o backend a partir de uma tela, use a sessão do `useAuth()` — ver 06.

`Session.ts` — a sessão do app

```
// src/lib/curio/Session.ts
import { MainUseCase } from "@curio/client";
export class Session extends MainUseCase {
  module: number | undefined;
  storageToken: string | undefined;
}
```

Estende `MainUseCase` só para carregar `module` e `storageToken`. Se o projeto precisar de mais estado por sessão, é aqui.

`SessionManager.ts` — detecção de sessão morta

```
// src/lib/curio/SessionManager.tsimport { MainUseCase, RequestDriver, SecurityManager,
UseCaseMessageType } from "@curio/client";
import { Session } from "./Session";
import { isAuthError } from "./isAuthError";import { ABORT } from
"@curio/client/utils/constants";
```

```
export interface ConfigService {
  url: string;
  server: string;
  system: number;
  port: number;
  module: number;
  version: number;
}
// chave unica por app — nunca reaproveitar de outro projetoexport const STORAGEKEY =
"br.com.nomedoprojeto";
export class SessionManager extends SecurityManager {
  public session: MainUseCase | undefined;
  private _service: ConfigService;
  private _driver: RequestDriver;
  // eslint-disable-next-line @typescript-eslint/no-explicit-any -- @curio nao tipa o service
constructor(service: any, driver: RequestDriver) {
    super(service, driver);
    this._service = service;
    this._driver = driver;
  }
  public async openMainUseCase<U extends typeof MainUseCase>(username: string, password: string,
constructor?: U) {    this.session = await super.openMainUseCase(username, password,
constructor);
    this._attachListeners();
    return this.session! as InstanceType<U>;
  }
  public connectMainUseCase(mainUseCase: MainUseCase) {
    this.session = mainUseCase;
    this._attachListeners();
    return super.connectMainUseCase(mainUseCase);
  }
  // sessao anonima: pre-login (ex. obter versao)
  public anonymousSession() {
    return new Session(0, this._service, this._driver);
  }
  private _attachListeners() {    this.session!.addListener(ABORT, () =>
this._unauthenticate(false));    // eslint-disable-next-line @typescript-eslint/no-explicit-
```

```
any -- @curio nao tipa a mensagem    this.session!.addListener(UseCaseMessageType.RESPONSE,
(message: any) => {
      if (isAuthError(message.error)) this._unauthenticate(true);
    });
  }
  private _unauthenticate(expired: boolean) {
    this.session = undefined;
    if (expired) localStorage.removeItem(STORAGEKEY);
  }
}
```

`STORAGEKEY` identifica a sessão no storage. **Troque por um valor próprio do projeto novo.**

> **Cuidado com storage inconsistente.** Se o token é escrito em `sessionStorage` mas a limpeza remove de `localStorage` (ou vice-versa), a limpeza não limpa nada — são storages diferentes. **Escolha um e use o mesmo nos dois lugares.** O snippet acima e o de `session.ts` abaixo usam `sessionStorage` de forma consistente — token de sessão não deve sobreviver ao fechamento da aba.

`isAuthError.ts` — o que conta como falha de autenticação

```
// src/lib/curio/isAuthError.ts
import { DestinataryNotFoundError } from "@curio/client/errors";
// code -1 = sessao inexistente no servidorexport const isAuthError = (error: unknown): boolean
=>  error instanceof DestinataryNotFoundError || String((error as { code?: unknown } | null |
undefined)?.code) === "-1";
```

Ponto único de decisão. Se o backend introduzir outro código de sessão inválida, muda só aqui.

src/api/session.ts

```
// src/api/session.tsimport { ConnectionError, DestinataryNotFoundError } from
"@curio/client/errors";
import { Session, getSessionManager } from "../lib/curio";import { STORAGEKEY } from
"../lib/curio/SessionManager";
export interface LoginParams {
  email: string;
  password?: string;
}
export async function login(params: LoginParams) {
  const { email, password } = params;
  try {
    const sessionManager = await getSessionManager();    const session = await
sessionManager.openMainUseCase(email, password ?? "", Session);
    session.module = 0;
    session.storageToken = STORAGEKEY;    sessionStorage.setItem(STORAGEKEY,
session.token.toString());
    return session;
  } catch (error) {
    return error as Error;
  }
}
// reconecta a partir do token guardado (refresh de pagina)export async function connect(token:
string) {
  try {
    const sessionManager = await getSessionManager();    sessionManager.connectMainUseCase(new
Session(token, sessionManager.service, sessionManager.driver));    await
sessionManager.session!.timeoutCheck();    const session = sessionManager.session! as
Session;
    session.module = 0;
    session.storageToken = STORAGEKEY;    sessionStorage.setItem(STORAGEKEY,
session.token.toString());
    return session;
```

```
  } catch (error) {
    if (error instanceof DestinataryNotFoundError) return error;    if (error instanceof
ConnectionError) return error;
    return error as Error;
  }
}
export async function logoutSession(session?: Session) {
  if (session) session.abort();
  sessionStorage.removeItem(STORAGEKEY);
}
```

> `login` e `connect` **retornam o erro em vez de lançar.** Quem chama precisa testar `result instanceof Error`. É contraintuitivo e fácil de errar — se o projeto novo puder, prefira deixar lançar e tratar no hook.
>
> **Cuidado com `localStorage.clear()` no logout** — ele apaga preferências de UI não relacionadas à sessão. O snippet acima remove só a chave da sessão.

Requests anônimos (pré-login)

Para chamar o backend antes de autenticar — por exemplo, exibir a versão do servidor na tela de login:

```
// src/api/session.ts
export const VERSION = { useCase: "3916", getVersion: "RM_OBTER_VERSAO" };
export async function obterVersaoRequest() {
  const sessionManager = await getSessionManager();  const anonymousSession =
sessionManager.anonymousSession();  const uc = await
anonymousSession?.openUseCase(VERSION.useCase);  const response = await
uc?.sendRequest(VERSION.getVersion);  await uc?.abort(); // sempre fechar — sessao anonima nao
tem dono
```

```
  return response.Versao._ as string;
}
```

AuthProvider

```
// src/context/AuthProvider.tsx (trecho essencial)export const AuthProvider: React.FC<{
children: React.ReactNode }> = ({ children }) => {
  const { notification } = useNotification();
  const navigate = useNavigate();  const { session, isLoading, isAuth, login: authLogin, logout:
authLogout, refetchConnection } = useAuthQuery();
  // reconecta no boot se ha token guardado
  useEffect(() => {
    if (sessionStorage.getItem(STORAGEKEY)) refetchConnection();
  }, []);
  const logout = useCallback(() => {
    authLogout();
    navigate("/", { replace: true });
  }, [authLogout, navigate]);
  // fonte unica de logout: reage a notificacao de erro de auth.  // ref evita disparar 2x —
logout muda de identidade a cada render.  const handledNotificationRef = useRef<NotificationType
| null>(null);
  useEffect(() => {
    if (!notification) return;    const expiredSession = notification.message === "Sessão
expirada!";
    if (!notification.isAuthError && !expiredSession) return;    if
(handledNotificationRef.current === notification) return;    handledNotificationRef.current =
notification;
    logout();
  }, [notification, logout]);
  const login = async (params: LoginCredentials) => {
    await authLogin(params);
    navigate("/dashboard");
  };
  return (    <AuthContext.Provider value={{ isAuth, session, isLoading, login, logout
```

```
  }}>{children}</AuthContext.Provider>
    );
  };
```

Ciclo de vida

Event	O que acontece
Boot com token	`refetchConnection()` → `connect(token)` → sessão restaurada
Login	`login()` → token no `sessionStorage` → navega para `/dashboard`
Request com sessão morta	`isAuthError` → notificação com `isAuthError: true` → `AuthProvider` desloga
`ABORT` do servidor	Listener no `SessionManager` limpa a sessão

Event	O que acontece
Logout manual	`session.abort()` → limpa storage → navega para `/`

Detecção de expiração por string

```
const expiredSession = notification.message === "Sessão expirada!";
```

Comparar mensagem literal é frágil: muda a tradução no backend, o auto-logout para de funcionar sem erro visível. **Preferível** é o backend sinalizar com um código que `isAuthError` reconheça. Se precisar manter, isole a string numa constante e documente a dependência.

06 — Caso de uso

06 — Caso de uso

> Como uma tela fala com o backend Curio. Padrão vigente: `UseCaseManager` envolve a página, `useUseCaseControls` abre/fecha, `useCurioMutation` envia requests. **Não use `session.openUseCase()` direto numa página.** Esse é o caminho de baixo nível.

O modelo mental

O Curio não é REST. Não há endpoint por recurso. Há:

1. Um **caso de uso**, identificado por um id numérico (`"2544"`). Ele tem estado no servidor: você o abre, interage, e fecha.
2. **Requests nomeados** dentro dele (`"RM_INCLUI_OBJETO"`), que recebem e devolvem objetos.

Um caso de uso é aproximadamente "uma tela do sistema" do lado do servidor. Por isso o casamento natural é **um caso de uso por página**.

```
Página (UseCaseManager useCaseId="2544")
  ├─ useUseCaseControls()   → open() / close() / status  └─
useCurioMutation("RM_INCLUI_OBJETO")  → mutate() / mutateAsync()
```

Cadeia de chamada

Toda chamada ao backend segue a mesma cadeia de três camadas, sempre nesta ordem:

```
Componente (.tsx)  →  hook específico da feature (service/hooks.ts)  →  useCurioMutation
```

- **Componente** — chama o hook da feature, nunca `useCurioMutation` direto.
- **Hook específico da feature** — uma função de uma linha por request, tipada, nomeada pelo verbo do RM. É a única coisa que sabe qual string vai para o backend.
- **`useCurioMutation`** — genérico, não sabe nada sobre Fornecedor. Ver 21.

```
// service/hooks.ts — o hook específico é a ÚNICA camada que conhece o nome do RMexport const
useSalvaFornecedor = () => useCurioMutation<void, SalvaFornecedorRequest>(FORNECEDOR_RMS.SALVAR);
```

```
// Page.tsx — o componente so conhece o hook, nunca a string do RMconst { mutateAsync: salvar,
isPending } = useSalvaFornecedor();
```

Nunca pule uma camada. Chamar `useCurioMutation("RM_SALVA_OBJETO")` direto no componente espalha o nome do request por várias telas — renomear o RM no backend exige grep no projeto inteiro em vez de editar uma linha.

Convenção `_RMS`: use case e requests num só objeto

Cada feature declara **um objeto de constantes** com o id do caso de uso e o nome de todos os seus requests. Um arquivo, uma fonte da verdade.

```
// src/pages/Cadastro/Fornecedor/service/constants.ts
export const FORNECEDOR_RMS = {
  USE_CASE: "4821",
  OBTEM_DADOS: "RM_OBTEM_DADOS_FORNECEDOR",
  SALVAR: "RM_SALVAR_DADOS_FORNECEDOR",
  REMOVER: "RM_REMOVER_FORNECEDOR"
};
```

Regras:

- **Nome do objeto**: `{ENTIDADE}_RMS`, maiúsculo, no plural de "requests do módulo" — não é o plural da entidade.
- `USE_CASE` é sempre a primeira chave. É o único valor que o `UseCaseManager` da página consome.
- Os demais valores são os nomes exatos dos requests, em `SCREAMING_SNAKE_CASE` com prefixo `RM_` no **valor** (a chave do objeto pode ser mais curta e legível — `SALVAR`, não `SALVAR_DADOS_FORNECEDOR`).
- **Nenhuma string de RM ou de id de caso de uso literal fora deste arquivo.** Nem em `hooks.ts`, nem na página.

`hooks.ts` importa a constante em vez de repetir a string:

```
// service/hooks.ts
import { useCurioMutation } from "@/hooks";
import { FORNECEDOR_RMS } from "./constants";
import type {
```

```
  SalvaFornecedorRequest,
  BuscaFornecedoresRequest,
  BuscaFornecedoresResponse,
  FornecedorInicialResponse
} from "./interfaces";
export const useIncluiFornecedor = () => useCurioMutation<FornecedorInicialResponse,
void>(FORNECEDOR_RMS.OBTEM_DADOS);
export const useBuscaFornecedores = () =>  useCurioMutation<BuscaFornecedoresResponse,
BuscaFornecedoresRequest>(FORNECEDOR_RMS.SALVAR);
export const useRemoveFornecedor = () => useCurioMutation<unknown, { Fornecedor: string
}>(FORNECEDOR_RMS.REMOVER);
```

E a página importa a mesma constante para o `useCaseId` do `UseCaseManager` — ver "A página" abaixo. **Um único arquivo muda quando o backend renumerar o caso de uso ou renomear um request.**

Anatomia de uma tela

Uma tela que fala com o backend tem quatro arquivos:

```
src/pages/Fornecedor/Cadastro/
├── IncluirFornecedorPage.tsx    componente + UseCaseManager├── schemas.ts
validação zod
└── service/    ├── constants.ts               FORNECEDOR_RMS — useCaseId + nomes dos
requests
    ├── interfaces.ts              tipos de request/response    └── hooks.ts
um hook por request
```

`service/interfaces.ts`

Tipa o que entra e sai do backend. Convenção `_Prefixo` — ver 13.

```
```

```
// src/pages/Fornecedor/Cadastro/service/interfaces.ts
export interface FornecedorXML {
  _OID: string;
  _Nome: string;
  _CNPJ: string;
  _Ativo: boolean;
  Endereco?: EnderecoXML;
}
export interface EnderecoXML {
  _OID: string;
  _Logradouro: string;
  _Cidade: string;
}
// resposta de abertura: backend devolve o objeto recem-criadoexport interface
FornecedorInicialResponse {
  Fornecedor: FornecedorXML;
}
export interface SalvaFornecedorRequest {
  Fornecedor: {
    _OID: string;
    _Nome: string;
    _CNPJ: string;
    Endereco?: { _OID: string };
  };
}
export interface BuscaFornecedoresRequest {
  OBJECTID: { _Nome: string; _Ativo: boolean };
}
export interface BuscaFornecedoresResponse {
  Response: FornecedorXML[];
}
```

service/hooks.ts

Um hook por request. Uma linha cada.

```
```

```
// src/pages/Fornecedor/Cadastro/service/hooks.tsimport { useCurioMutation } from
"@/hooks/useCurioMutation";
import { FORNECEDOR_RMS } from "./constants";
import {
  BuscaFornecedoresRequest,
  BuscaFornecedoresResponse,
  FornecedorInicialResponse,
  SalvaFornecedorRequest
} from "./interfaces";
export const useIncluiFornecedor = () => useCurioMutation<FornecedorInicialResponse,
void>(FORNECEDOR_RMS.OBTEM_DADOS);
export const useBuscaFornecedores = () =>  useCurioMutation<BuscaFornecedoresResponse,
BuscaFornecedoresRequest>(FORNECEDOR_RMS.SALVAR);
export const useSalvaFornecedor = () => useCurioMutation<void,
SalvaFornecedorRequest>(FORNECEDOR_RMS.SALVAR);
```

Assinatura: `useCurioMutation<TResposta, TParametros>(nomeDoRequest)`. Use `void` quando não há parâmetros ou não há resposta útil.

A página

```
// src/pages/Fornecedor/Cadastro/IncluirFornecedorPage.tsximport React, { useEffect, useRef }
from "react";
import { Box, Button } from "@mui/material";
import { UseCaseManager } from "@curio/client/react";
import { useAuth } from "@/context/AuthProvider";import { useUseCaseControls, useValidatedForm }
from "@/hooks";
import { fornecedorSchema, type FornecedorFormData } from "./schemas";import { FORNECEDOR_RMS }
from "./service/constants";import { useIncluiFornecedor, useSalvaFornecedor } from
"./service/hooks";
const IncluirFornecedorContent: React.FC = () => {  const { open, status } =
useUseCaseControls();
  const { data: fornecedorData, mutate: incluirFornecedor, isPending: isIncluindo } =
useIncluiFornecedor();  const { mutateAsync: salvarAsync, isPending: isSalvando } =
```

```
useSalvaFornecedor();
  // refs evitam reabrir/reinicializar em re-render
  const hasAttemptedOpenRef = useRef(false);
  const hasInitializedRef = useRef(false);
  useEffect(() => {
    if (status === "idle" && !hasAttemptedOpenRef.current) {      hasAttemptedOpenRef.current =
true;
      open();
    } else if (status === "open" && !hasInitializedRef.current) {      hasInitializedRef.current
= true;
      incluirFornecedor();
    }
  }, [status, open, incluirFornecedor]);
  const form = useValidatedForm({
    schema: fornecedorSchema,
    defaultValues: { _Nome: "", _CNPJ: "" }
  });
  const handleSalvar = async (data: FornecedorFormData) => {
    await salvarAsync({
      Fornecedor: {
        _OID: String(fornecedorData?.Fornecedor._OID ?? ""),
        _Nome: data._Nome,
        _CNPJ: data._CNPJ
      },
      msgSucesso: "Fornecedor salvo com sucesso!"
    });
  };
  return (
    <Box>      <Button onClick={form.handleSubmit(handleSalvar)} disabled={isIncluindo ||
isSalvando}>
        Salvar
      </Button>
      {/* campos — ver 11 */}
    </Box>
  );
};
```

```
const IncluirFornecedorPage: React.FC = () => {
  const { session } = useAuth();
  return (    <UseCaseManager session={session} useCaseId={FORNECEDOR_RMS.USE_CASE}
autoClose={false} openOnMount={false}>
      <IncluirFornecedorContent />
    </UseCaseManager>
  );
};
export default IncluirFornecedorPage;
```

Por que dois componentes

`UseCaseManager` é um provider. Os hooks `useUseCaseControls` e `useCurioMutation` consomem o contexto dele, então **precisam estar em um componente filho**. O componente externo só faz o wiring; o conteúdo real vive no `Content`.

Isso não é opcional. Chamar `useCurioMutation` no mesmo componente que renderiza `UseCaseManager` falha em runtime.

Props do `UseCaseManager`

Pro	Val usu	Efeito
sess	useAuth()	Sessão autenticada
useC	FORNECEDOR_RMS.USE_CASE	Id do caso de uso no servidor

Pro	Val usu	Efeito
auto	false	`true` fecha ao desmontar. Use `false` com navegação por abas
open	false	`false` dá controle explícito da abertura via `open()`

Com `openOnMount={false}`, **a página é responsável por chamar `open()`** — daí o `useEffect` com o `hasAttemptedOpenRef`.

useUseCaseControls

```
// src/hooks/useUseCaseControls.tsexport const useUseCaseControls = (options:
UseUseCaseControlsOptions = {}) => {
  const { enableLogs = false } = options;  const { open, close, status, error,
triggersFromCurrentState } = useUseCaseManager();  const { setNotification } =
useNotification();
  // open() do curio lanca; aqui vira notificacao
  const openSafe = useCallback(
    async (openOptions?: OpenOptions) => {
      try {
        await open(openOptions);
      } catch (err) {        const message = err instanceof Error ? err.message : "Erro ao abrir
o caso de uso";        setNotification({ message, type: "error", isAuthError: isAuthError(err)
});
```

```
      }
    },
    [open, setNotification]
  );
  // ...  return { open: openSafe, close, status, error, logCurrentStateTriggers:
logCurrentState };
};
```

`status` assume `"idle"` → `"open"`. A máquina de estados típica da página é exatamente o `useEffect` mostrado acima: abrir quando `idle`, inicializar quando `open`.

Descobrir os requests disponíveis

Passe `enableLogs: true` para logar, em desenvolvimento, os triggers que o caso de uso expõe no estado atual:

```
const { open, status } = useUseCaseControls({ enableLogs: true });
```

Útil quando você não sabe o nome exato do request. **Remova antes de commitar.**

`useCurioMutation`

Envolve `useMutation` do React Query sobre o `sendRequest` do caso de uso.

```
const { data, mutate, mutateAsync, isPending, isError } = useCurioMutation<TData,
TParams>("RM_NOME");
```

Mensagens de sucesso e erro

`msgSucesso`, `msgErro` e `msgErroFallback` são passados junto dos parâmetros e **removidos antes de chegar ao backend**:

```
```

```
salvar({
  Fornecedor: { _OID: "123", _Nome: "ACME" },
  msgSucesso: "Fornecedor salvo com sucesso!",
  msgErro: "Não foi possível salvar."
});
```

`msgSucesso` dispara notificação verde no sucesso. Para erro, a mensagem exibida segue uma ordem de prioridade, decidida uma única vez no `mutationCache.onError` global (07) — não no hook:

1. **`msgErro`**, se informado — **sempre prevalece**, mesmo que o backend tenha devolvido mensagem própria. É o dev pedindo explicitamente para sobrescrever.
2. Mensagem do backend (`error.message`, ou `error.detail` se for `ValidationError`), se houver.
3. **`msgErroFallback`**, se informado — só é usado quando o backend não devolveu mensagem alguma.
4. Fallback genérico interno do hook ("Ocorreu um erro ao processar a requisição.").

```
salvar({
  Fornecedor: { _OID: "123", _Nome: "ACME" },  msgErroFallback: "Não foi possível salvar o fornecedor."  // se o backend devolver mensagem, ela aparece; msgErroFallback só entra se ele nao devolver nada
});
```

Omitir os três não silencia o erro — o `queryClient` global sempre notifica, no mínimo com o fallback genérico.

> **Não trate erro dentro de um hook específico da feature.** A prioridade acima é resolvida **uma vez**, no `mutationCache.onError` global. Um `onError` local em `useCurioMutation` duplicaria a notificação — foi exatamente esse bug que motivou consolidar a lógica num só lugar.

Callbacks

`onSuccess`, `onError`, `onSettled` e `onMutate` vão no **mesmo objeto** dos parâmetros, não num segundo argumento:

```
salvar({
  Fornecedor: { _OID: "123", _Nome: "ACME" },
  onSuccess: () => setModalAberto(true),
  onError: (error) => console.error(error)
});
```

Difere do React Query puro. `useCurioMutation` separa os callbacks dos parâmetros internamente.

Buscas genéricas: `useSearcher`

Para telas de busca sobre entidades que o backend expõe via operações genéricas (`120` = buscar, `134` = obter contexto), sem caso de uso dedicado:

```
const { getcontext, search } = useSearcher<FiltrosBusca, ResultadoBusca>("465");
const contexto = await getcontext.mutateAsync();const resultado = await search.mutateAsync({
_Nome: "ACME", _Ativo: true });
```

Usa a sessão do `useAuth()` diretamente, **sem** `UseCaseManager`. Use quando a tela é só filtro + lista. Se houver estado no servidor (incluir, alterar, salvar), use `UseCaseManager`.

Contrato com o backend

O que o front precisa saber — o resto é responsabilidade de quem escreve o caso de uso no servidor.

Aspe	Contrato

Identificação	Id numérico como string (`"4821"`)
Requests	Nome em `SCREAMING_SNAKE_CASE`, prefixo `RM_` (`RM_SALVA_OBJETO`)
Parâmetros	Objeto aninhado espelhando a entidade (`{ Fornecedor: { _OID, _Nome } }`)
Filtros	Frequentemente sob a chave `OBJECTID`
Resposta	Objeto com a entidade na raiz (`{ Fornecedor: {...} }`) ou `{ Response: [...] }`
Primitivos	Prefixo `_` (`_Nome`, `_OID`)

Objetos	Sem prefixo (`Endereco`, `Documentos`)
Datas	String ISO com tempo: `"2026-08-18T00:00:00.0"`
Erro	`Error` lançado pelo transporte; `ValidationError` traz `detail: string[]`
Sessão morta	`DestinataryNotFoundError` ou `code === "-1"` — ver 05

Não invente nomes de request. Eles vêm do caso de uso do servidor. Confirme com quem o implementou ou use `enableLogs`.

Erros comuns

Sinto	Causa
Hook lança "fora do contexto"	`useCurioMutation` no mesmo componente que renderiza o `UseCaseManager`

Sinto	Causa
Request roda antes do caso de uso abrir	Faltou aguardar `status === "open"`
Caso de uso reabre a cada render	Faltou o `useRef` de guarda
`msgSucesso` chega ao backend	Versão do `useCurioMutation` sem `delete params.msgSucesso`
Data rejeitada pelo backend	Faltou o sufixo `T00:00:00.0`

07 — React Query

07 — React Query

> Cache, tratamento global de erro e convenção de query keys. Versão **5.x** (01) — `cacheTime` chama-se `gcTime` desde a v5. Regra central: **nenhuma página trata erro de request para exibir mensagem — e nenhum hook específico trata, também.** A prioridade da mensagem é resolvida uma única vez, no `mutationCache.onError` global.

O `QueryClient`

```
// src/api/queryClient.tsimport { MutationCache, QueryCache, QueryClient } from "@tanstack/react-query";
import { ValidationError } from "@curio/client/errors";import { isAuthError } from "@/lib/curio";
import { NotificationType } from "@context/NotificationProvider";
// mensagem do backend, se houver — sem fallback, quem chama decide o que fazer na ausenciaconst getErrorMessage = (error: unknown): string | undefined => {  // ValidationError do curio traz lista de problemas
  if (error instanceof ValidationError) return error.detail.join("\n");  return error instanceof Error && error.message ? error.message : undefined;
};
interface MutationVarsWithMessages {
```

```
  msgErro?: string;
  msgErroFallback?: string;
}
export const createQueryClient = (setNotification: (notification: NotificationType) => void) =>
new QueryClient({
    defaultOptions: {
      queries: {
        staleTime: 5 * 60 * 1000,        gcTime: 10 * 60 * 1000, // v5: cacheTime foi renomeado
para gcTime
        retry: false,
        refetchOnWindowFocus: false,
        refetchOnReconnect: true
      },
      mutations: {
        retry: false,
        gcTime: 3 * 60 * 1000
      }
    },
    queryCache: new QueryCache({
      onError: (error) => {        const message = getErrorMessage(error) ?? "Ocorreu um erro ao
buscar os dados.";        setNotification({ message, type: "error", isAuthError:
isAuthError(error) });
      }
    }),
    mutationCache: new MutationCache({      // fonte unica da mensagem de erro de mutation —
useCurioMutation nao trata isso no      // proprio onError, senao a notificacao dispara duas
vezes (aqui + lá).
      onError: (error, variables) => {        const vars = variables as MutationVarsWithMessages
| undefined;        // msgErro: o dev pediu pra sobrescrever qualquer mensagem do backend —
sempre prevalece.        // sem msgErro: mensagem do backend; sem essa, msgErroFallback; sem
essa, fallback interno.
        const message =          vars?.msgErro ?? getErrorMessage(error) ??
vars?.msgErroFallback ?? "Ocorreu um erro ao processar a requisição.";        setNotification({
message, type: "error", isAuthError: isAuthError(error) });
      }
    })
```

```
  });
export default createQueryClient;
```

Por que é uma fábrica e não uma constante

O `QueryClient` precisa do `setNotification`, que só existe dentro do `NotificationProvider`. Daí o `createQueryClient(setNotification)` chamado em `main.tsx` dentro de `useState(() => ...)` — ver 03.

`useState` com inicializador de função, não `useMemo`: garante instância única mesmo sob StrictMode.

Defaults e o porquê

Op	Val	Razão
`staleTime`	5 min	Dados corporativos mudam devagar; evita refetch a cada navegação
`gcTime`	10 min	Mantém dados ao voltar para uma tela (era `cacheTime` até a v4)

Op	Val	Razão
`retry`	`false`	Retry sobre caso de uso com estado no servidor pode duplicar efeito
`refe`	`false`	Refetch ao alternar janela é ruído em app interno
`refe`	`true`	Voltar de queda de rede deve revalidar

`retry: false` **é a decisão mais importante.** Não mude sem entender que um `RM_SALVA_OBJETO` repetido pode gravar duas vezes.

Erro é global

O `onError` do `QueryCache` / `MutationCache` captura **toda** falha e converte em notificação — inclusive as de `useCurioMutation` e `useHookMutation`. Isso vale tanto para a página quanto para o hook específico da feature: **nenhum dos dois deve ter seu próprio `onError` de notificação.** Um `onError` local que também chama `setNotification` dispara duas notificações para o mesmo erro — uma do global, outra do local — porque o `mutationCache.onError` roda para toda mutation, sempre. Consequência prática:

```
// ERRADO — duplica a mensagem: a global ja apareceu
const handleSalvar = async () => {
  try {
    await salvarAsync(payload);
  } catch (error) {
    setNotification({ type: "error", message: "Erro ao salvar" });
  }
};
// CERTO — deixa o erro subir; global notifica
const handleSalvar = async () => {
  await salvarAsync(payload);
  setModalSucesso(true);
};
// CERTO — try/catch so pra controlar fluxo, sem notificar
const handleSalvar = async () => {
  try {
    await salvarAsync(payload);
    setModalSucesso(true);
  } catch {
    // notificacao ja veio do global; aqui so nao abre o modal
  }
};
```

Use `try/catch` **para controlar fluxo**, nunca para exibir mensagem de erro de request.

Para customizar a mensagem, use `msgErro` (sobrescreve sempre) ou `msgErroFallback` (só quando o backend não devolve mensagem) no `useCurioMutation` (06) em vez de capturar.

Query keys

Array, do mais genérico ao mais específico:

```
["fornecedores"]                                  // dominio inteiro["fornecedores",
"lista"]                        // uma colecao["fornecedores", "lista", { ativo: true }]
```

```
// colecao com filtro
["fornecedores", "detalhe", oid]                // um item
```

Regras:

- Primeiro elemento é o domínio, sempre string.
- Segundo é a operação (`"lista"`, `"detalhe"`).
- Parâmetros que afetam o resultado entram na key. Se não entrarem, o cache serve dado errado.
- Nunca interpole (`` `fornecedores-${oid}` ``) — impede invalidação por prefixo.

Centralize por domínio:

```
// src/pages/Fornecedor/service/queryKeys.ts
export const fornecedorKeys = {
  all: ["fornecedores"] as const,
  listas: () => [...fornecedorKeys.all, "lista"] as const,  lista: (filtros: FiltrosFornecedor)
=> [...fornecedorKeys.listas(), filtros] as const,  detalhe: (oid: string) =>
[...fornecedorKeys.all, "detalhe", oid] as const
};
```

Invalidar tudo do domínio: `queryClient.invalidateQueries({ queryKey: fornecedorKeys.all })`.

Query vs mutation

O Curio inverte a intuição de REST.

Situ	Use	Por quê

Request dentro de um `UseCaseManager`	**mutation**	Tem efeito no estado do caso de uso, mesmo "só lendo"
Busca disparada por botão	**mutation**	Ação do usuário, não ao que se auto-revalida
Dado que carrega sozinho e revalida	query	Ex.: lista do dashboard
Reconexão de sessão	query	Ver `useAuthQuery`

Por isso `useCurioMutation` e `useSearcher` usam `useMutation`, não `useQuery`. Não é engano.

Invalidação

```
const queryClient = useQueryClient();
const handleSalvar = async (data: FornecedorFormData) => {  await salvarAsync({ Fornecedor: {
...data }, msgSucesso: "Salvo!" });  await queryClient.invalidateQueries({ queryKey:
fornecedorKeys.listas() });
};
```

Em telas com `UseCaseManager`, o padrão mais comum é **refazer o request no próprio caso de uso** (`buscarFornecedores()` de novo) em vez de invalidar cache — o estado vive no servidor, não no cache.

Devtools

`<ReactQueryDevtools initialIsOpen={false} />` está em `main.tsx`. É removido automaticamente do bundle de produção pela própria biblioteca. Não condicione a `import.meta.env.DEV` manualmente.

Nomenclatura

Tip	Pad	Exemplo
Query de coleção	`use{`	`useFornecedores`
Query de item	`use{`	`useFornecedor`
Mutation de caso de uso	`use{`	`useSalvaFornecedor`
Mutation genérica	`use{`	`useCreateFornecedorMutation`

Hooks de caso de uso usam o verbo em português, espelhando o nome do request do backend (`RM_SALVA_OBJETO` → `useSalvaFornecedor`). Isso torna rastreável qual hook corresponde a qual request.