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

Opção	Valor	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)
`retry`	`false`	Retry sobre caso de uso com estado no servidor pode duplicar efeito
`refetchOnWindowFocus`	`false`	Refetch ao alternar janela é ruído em app interno
`refetchOnReconnect`	`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.

Situação	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 dado 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

Tipo	Padrão	Exemplo
Query de coleção	`use{Entidade}s`	`useFornecedores`
Query de item	`use{Entidade}`	`useFornecedor`
Mutation de caso de uso	`use{Verbo}{Entidade}`	`useSalvaFornecedor`
Mutation genérica	`use{Verbo}{Entidade}Mutation`	`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.

Revision #2
Created Thu, Aug 20, 2026 4:52 PM by Geraldo Barbosa
Updated Tue, Aug 25, 2026 4:52 PM by Geraldo Barbosa