DOCUMENTAÇÃO TÉCNICA COMPLETA

Migração e Reestruturação da Plataforma de Gestão de Contactos

Referência técnica de ponta a ponta: arquitetura, schema, API, autenticação, sincronização e configuração de ambientes.

React 19 · React Router v7 · Node.js / Express · MySQL (mysql2) · ExcelJS · Tiptap · WordPress REST API · Render

---

0 Visão Geral

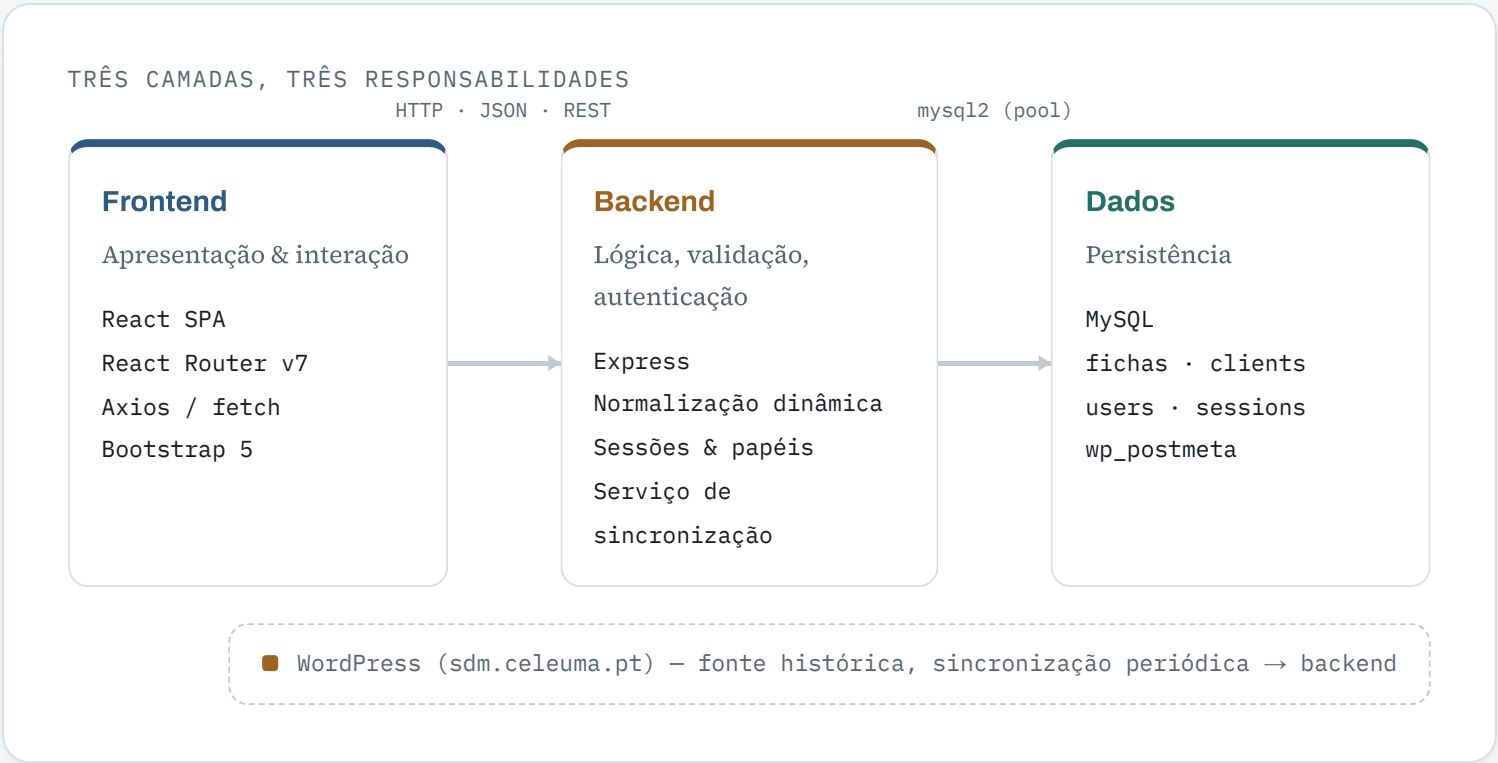
A aplicação substitui a antiga plataforma de gestão de contactos da Celeuma — construída sobre WordPress, com lógica de negócio espalhada por plugins e temas PHP — por um sistema em três camadas bem separadas: uma interface React, uma API REST em Node.js e uma base de dados MySQL própria.

O WordPress mantém-se como fonte de dados históricos, sincronizado periodicamente para a base de dados local por um serviço dedicado (secção 6). A aplicação nova gere cerca de **11 500 registos operacionais** — fichas comerciais e clientes — mais páginas de conteúdo editorial, utilizadores e sessões, distribuídos por seis tabelas (secção 2).

Este documento assume acesso ao código-fonte (`backend/server.js` , `backend/sync-service.js` , `frontend/src/`) e serve como referência para quem for dar continuidade ao projeto.

1 Arquitetura

As três camadas correm como serviços independentes no Render — o frontend como site estático, o backend como serviço web — e comunicam exclusivamente por HTTP/JSON.



O acesso entre camadas é restrito por **CORS** (variável `FRONTEND_URLS`, lida em `server.js` e usada como lista branca de origens) e por **sessão** (token gerado no login, ver secção 5). O backend corre num único processo Node — não há filas, workers separados nem cache — o que é suficiente para o volume atual mas é o primeiro limite a rever se o número de utilizadores concorrentes crescer significativamente.

2 Modelo de Dados

Seis tabelas MySQL sustentam a aplicação. `fichas`, `clients` e `users` carregam nomenclatura herdada do WordPress — mais colunas do que o `CREATE TABLE` de arranque presente no código, que serve apenas para inicializar uma base vazia. `sessions` e `app_passwords` são internas à aplicação e o `CREATE TABLE` do código é, nesse caso, exato.

O campo `legacy_id` é o elo entre gerações de dados: liga cada registo local ao post original do WordPress, permitindo deduplicar com segurança e voltar a sincronizar sem criar cópias (`dedupe.js`, índice único por `legacy_id`).

fichas

COLUNA	TIPO	NOTAS
`id`	INT AI PK	chave local
`legacy_id`	INT UNIQUE	post ID original no WordPress
`client_legacy_id`	INT	quase sempre NULL — a relação real vive em `wp_podsrel`
`title`, `post_status`, `post_visibility`, `post_date`, `author`	TEXT / VARCHAR / DATETIME	identificação e classificação
`tipo_contacto`, `pessoa_contacto`, `contacto`, `data_contacto`, `inicio_contacto`, `fim_contacto`, `motivo_resumo_contacto`	vários	registo do contacto comercial
`contacto_efetuado`, `follow_up`, `novo_contacto`	TINYINT(1)	sinalizadores
`tipo_proximo_contacto`, `data_proximo_contacto`	VARCHAR / DATE	agendamento
`data_apresentacao_proposta`, `estado_proposta`, `descritivo_proposta`, `servicos_proposta`, `valor_total_proposta`	vários	proposta comercial
`possibilidade_negocio`, `motivo_possibilidade_negocio`	VARCHAR / TEXT	probabilidade de fecho
`servicos_adjudicados`, `valor_total_adjudicado`	TEXT / DECIMAL(12,2)	proposta adjudicada
`descritivo_fatura`, `valor_fatura`, `data_fatura`, `data_prevista_recebimento`, `data_ultimo_contacto_financeiro`	vários	faturação
`relatorio_errado`, `porque_relatorio_errado`	TINYINT(1) / TEXT	auditoria — assinala registos a rever
`assunto_tratado`, `anexos`	TINYINT(1) / TEXT	estado do processo, URLs de anexos
`created_at`, `updated_at`	TIMESTAMP	auditoria de escrita local

clients

COLUNA	NOTAS
`id`, `legacy_id`	chave local e post ID original
`denominacao_fiscal`, `nif`	identificação fiscal
`contacto_empresa`, `pessoa_contacto_nome`, `pessoa_contacto_cargo`, `pessoa_contacto_telefone_email`	contacto
`morada`, `localidade`	localização — `localidade` foi acrescentada nesta fase do projeto
`comercial_id`, `author`	gestor de conta responsável
`estado`, `visibilidade`, `senha_visibilidade`, `publicado_em`	estado editorial (publicado / rascunho / lixo)
`observacoes`	TEXT, agora editável com o RichTextEditor (secção 3)
`created_at`, `updated_at`	auditoria

users, sessions, app_passwords

TABELA	COLUNAS PRINCIPAIS	PROPÓSITO
`users`	`id`, `name`, `email`, `role`, `password_hash`, `nome`, `apelido`, `alcunha`, `nome_mostrado`, `bio`, `site_url`, `registado`, `imagem_url`, `preferences`	contas — `role` só assume `admin` / `contributor` / `editor`
`sessions`	`id`, `user_id`, `token` (único), `user_agent`, `ip`, `created_at`, `last_active`	uma linha por sessão ativa
`app_passwords`	`id`, `user_id`, `name`, `password_hash`, `created_at`, `last_used`	tokens para integrações externas

wp_postmeta

Tabela nativa do WordPress, migrada por inteiro. Guarda campos que a sincronização REST nunca traz para `fichas` / `clients` — nomeadamente tudo o que é proposta, adjudicação e faturação — sob o modelo chave-valor `(meta_id, post_id, meta_key, meta_value)`. O

backend junta-a às tabelas principais por `COALESCE`, dando sempre prioridade à coluna local quando esta já tem valor.

3 Frontend

`App.js` é o ponto central: define as rotas com `BrowserRouter`, aplica o tema de cor ativo e regista os atalhos de teclado opcionais. Cada módulo vive na sua própria página em `pages/`; os componentes de layout partilhados (barra lateral, barra superior, editor de texto rico) vivem em `components/`.

Rotas

ROTA	COMPONENTE	ACESSO
`/`	Home	autenticado
`/fichas`, `/fichas/nova`, `/fichas/:id/editar`	Fichas, FichaFormPage	autenticado, visibilidade filtrada por papel
`/clientes`, `/clientes/novo`, `/clientes/:id/editar`	Clientes, NovoCliente, EditarCliente	autenticado
`/paginas`, `/paginas/novo`, `/paginas/:id/editar`	Paginas, NovaPagina, EditarPagina	autenticado
`/relatorios`	Relatorios (query `?tipo=`)	autenticado; aba "Clientes" só admin
`/admin/utilizadores`	UsersAdmin	só admin — redireciona para `/` caso contrário
`/perfil`	Perfil	autenticado
`/login`	Login	público

Resolução do URL da API

`api.js` resolve o backend de duas formas: as páginas de dados usam `axios` com um único `baseURL` calculado uma vez ao carregar a app; o *login* usa `apiFetch`, que tenta uma lista de candidatos em sequência até um responder.

```
export const API_BASE_URL = normalizeBaseUrl(
  process.env.REACT_APP_API_BASE_URL ||
  process.env.REACT_APP_API_URL ||
  inferDefaultBaseUrl()   // heurística por nome de host — ver secção 9
);
```

O valor mais fiável é a variável de ambiente explícita no build; a heurística de adivinhar o nome do backend a partir do nome do frontend só funciona se os dois serviços seguirem uma convenção específica (`fichas-frontend` → `fichas-backend`) — ver problema conhecido na secção 9.

Componentes partilhados

- **RichTextEditor** (Tiptap) — usado em fichas, páginas e observações de clientes; alterna entre modo Visual e HTML, com barra de formatação (negrito, itálico, listas, alinhamento, links) e é desativável por preferência do utilizador (`disableVisualEditor`).
- **Sidebar / Topbar** — adaptam-se ao papel do utilizador autenticado (admin vê "Utilizadores" e o relatório "Clientes"; os restantes não).
- **Sistema de temas** — 10 esquemas de cor predefinidos mais um construtor de temas próprios, todos expressos como variáveis CSS (`--theme-*`) aplicadas em `App.js` e persistidas em `localStorage` .

Relatórios (client-side)

Os cinco relatórios — Propostas, Propostas Adjudicadas, Contactos a Efetuar, Gestores, Ficha do Cliente — são calculados inteiramente no browser a partir dos dados brutos de `GET /api/fichas` , `GET /api/clientes` e `GET /api/comerciais` . A exportação para Excel usa `ExcelJS` diretamente no cliente, com bordas pretas finas em todas as células com dados, cabeçalhos a cores e larguras de coluna ajustadas por relatório.

4 Backend & API

Um único ficheiro `server.js` concentra rotas, normalização e autenticação; `sync-service.js` isola a lógica de sincronização. Todo o pedido segue o mesmo caminho: CORS → parsing JSON → rota → normalização → base de dados → resposta.

Mapeamento dinâmico de colunas

`getFichaColumnValueMap()` resolve a heterogeneidade de nomes de campo entre a aplicação nova e os dados migrados (o título de uma ficha pode chegar como `title`, `titulo` ou `post_title`):

```
const getFichaColumnValueMap = (body) => {
  const src = body || {};
  const title = asText(src.title || src.titulo || src.post_title);
  return {
    legacy_id: asNullable(src.legacy_id),
    title,
    post_status: normalizeStatus(src.post_status || src.estado),
    post_visibility: normalizeVisibility(src.post_visibility || src.visibilidade),
    author: asText(src.author || src.autor || src.gestor || src.comercial_id),
    data_contacto: asNullable(src.data_contacto),
    contacto_efetuado: normalizeFichaBoolean(src.contacto_efetuado),
    // ... mais de 30 campos, cada um com o seu alias e normalização
  };
};
```

É a mesma função, com o mesmo mapa de alias, usada tanto em `POST /api/fichas` como em `PUT /api/fichas/:id` — só as colunas que a tabela realmente tem (confirmadas via `SHOW COLUMNS` em cada pedido) são gravadas, o que torna o endpoint tolerante a diferenças de schema entre ambientes.

Datas à prova de corrupção

Todo o valor de data vindo do WordPress passa por `safeDateSql()`, que só aceita anos entre 1990 e 2100:

```
const safeDateSql = (metaValueExpr) =>
  `(CASE WHEN STR_TO_DATE(${metaValueExpr}, '%Y%m%d')
     BETWEEN '1990-01-01' AND '2100-12-31'
   THEN STR_TO_DATE(${metaValueExpr}, '%Y%m%d')
   ELSE NULL END)`;
```

Qualquer serial corrompido (o clássico "1899/1900" de uma data zero mal interpretada) vira `NULL` ainda dentro da query SQL — nunca chega ao frontend nem a uma exportação Excel. O mesmo critério de ano plausível é replicado em JavaScript no frontend (`isPlausibleYear`, `relatorios.js`), como segunda barreira.

Catálogo de endpoints

FICHAS		
GET	/api/fichas	listagem, resolvida com wp_postmeta
GET	/api/fichas/propostas-meta	campos de proposta/faturação, só existem em wp_postmeta
GET	/api/fichas/:id	detalhe
POST	/api/fichas	criar
PUT	/api/fichas/:id	atualizar (inclui mover para o lixo)
DELETE	/api/fichas/:id	eliminação definitiva

CLIENTES		
GET	/api/comerciais	lista de gestores para selects, com fallback "legacy"
GET	/api/clientes	listagem
GET	/api/clientes/:id	detalhe
POST	/api/clientes	criar
PUT	/api/clientes/:id	atualizar
DELETE	/api/clientes/:id	eliminação definitiva

PÁGINAS		
GET	/api/paginas	listagem — inclui publicadas, rascunho e lixo
GET	/api/paginas/:id	detalhe
POST	/api/paginas	criar
PUT	/api/paginas/:id	atualizar
DELETE	/api/paginas/:id	eliminação definitiva

PERFIL, SESSÕES E SEGURANÇA		
POST	/api/login	autenticação, cria sessão
GET	/api/perfil	dados do utilizador atual

PUT	/api/perfil	atualizar perfil e preferências
POST	/api/perfil/password	alterar palavra-passe
GET	/api/sessions	listar sessões ativas
DELETE	/api/sessions/:id	terminar sessão
GET	/api/sessions/validate	validar token em uso (polling do frontend)
POST	/api/app-passwords	gerar senha de aplicação
DELETE	/api/app-passwords/:id	revogar

ADMINISTRAÇÃO		
GET	/api/admin/users	listar utilizadores
PUT	/api/admin/users/:id/role	alterar papel — ver secção 5
POST	/api/admin/impersonate	simular sessão — ver secção 5

SINCRONIZAÇÃO		
GET	/api/sync/status	estado do serviço — ver secção 6
POST	/api/sync/webhook	notificação do WordPress, validada por WEBHOOK_SECRET

5 Autenticação e Autorização

Palavras-passe

As contas criadas na aplicação nova usam hash SHA-256 simples (sem sal). As contas migradas do WordPress mantêm o hash *phpass* original (`$P$` / `$H$` , MD5 iterado) — `verifyStoredPassword()` reconhece o formato pelo prefixo e escolhe o algoritmo correto, para não obrigar a um reset de password em massa na migração.

Sessão do utilizador comum

`POST /api/login` valida a password e cria uma linha em `sessions` com um token aleatório de 32 bytes. O frontend guarda esse token em `sessionStorage` e o `App.js` valida-o periodicamente (a cada 30s) contra `GET /api/sessions/validate`, forçando logout se a sessão tiver sido terminada noutro sítio.

Operações de administrador

O papel (`role`) de cada utilizador só assume três valores: `admin`, `contributor` ou `editor` — reforçado tanto pelo `ENUM` do `CREATE TABLE` de arranque como por `normalizeUserRole()`. As rotas mais sensíveis, no entanto, não usam o token de sessão para autorizar — usam um mecanismo próprio:

```
const assertAdminByLogin = (actingLogin, cb) => {
  resolveUserByLogin(actingLogin, (err, user) => {
    if (err) return cb(err);
    if (!user) return cb(new Error('Utilizador não encontrado'));
    if (!isAdminRole(user.role)) return cb(new Error('Acesso negado'));
    return cb(null, user);
  });
};
```

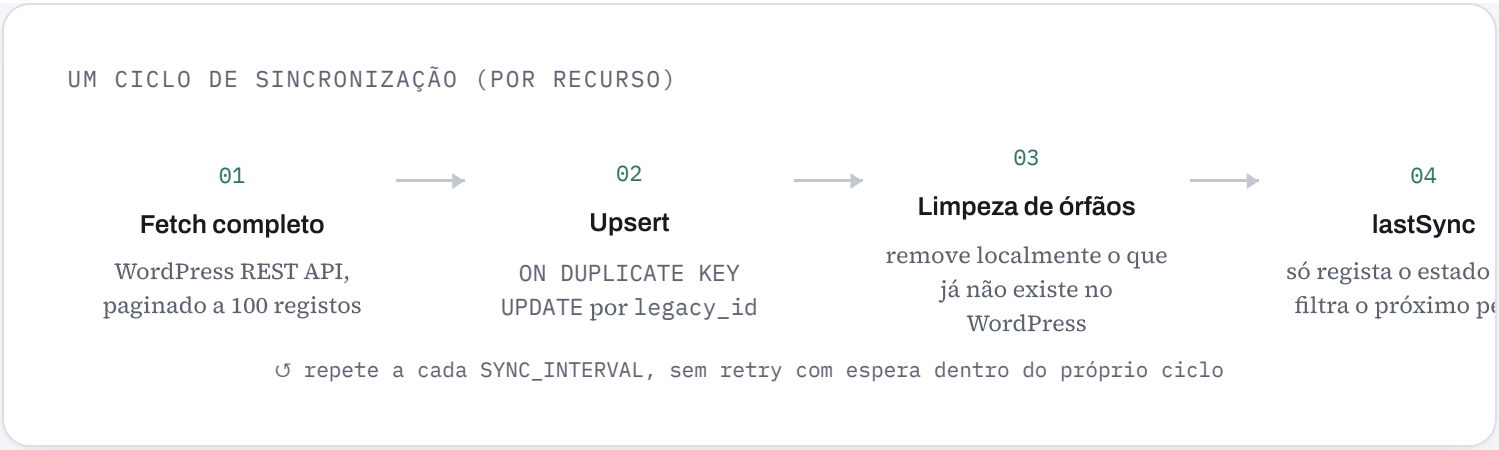
`PUT /api/admin/users/:id/role` e `POST /api/admin/impersonate` recebem um `actingLogin` no corpo do pedido, resolvem esse utilizador na base de dados e verificam o papel — sem validar que esse pedido vem mesmo de uma sessão autenticada. Ver secção 9 para o impacto disto.

Senhas de aplicação

Uma via de autenticação alternativa para integrações externas: cada utilizador pode gerar múltiplas senhas nomeadas (`app_passwords`), independentes da password principal e revogáveis individualmente sem afetar o resto da conta.

6 Sincronização com o WordPress

O `SyncService` (`sync-service.js`) corre um ciclo a cada `SYNC_INTERVAL` (1 minuto por omissão) para fichas, clientes e páginas, arrancado no boot do servidor apenas se `WP_API_URL` estiver definida.



Apesar do nome, `lastSync` não torna a sincronização incremental — cada ciclo volta a pedir a coleção inteira. Uma falha numa chamada individual (rede, autenticação, limite de pedidos) fica registada em `lastError` e não interrompe o ciclo; a tentativa seguinte só acontece no próximo intervalo agendado, sem backoff dentro do mesmo ciclo.

GET /api/sync/status

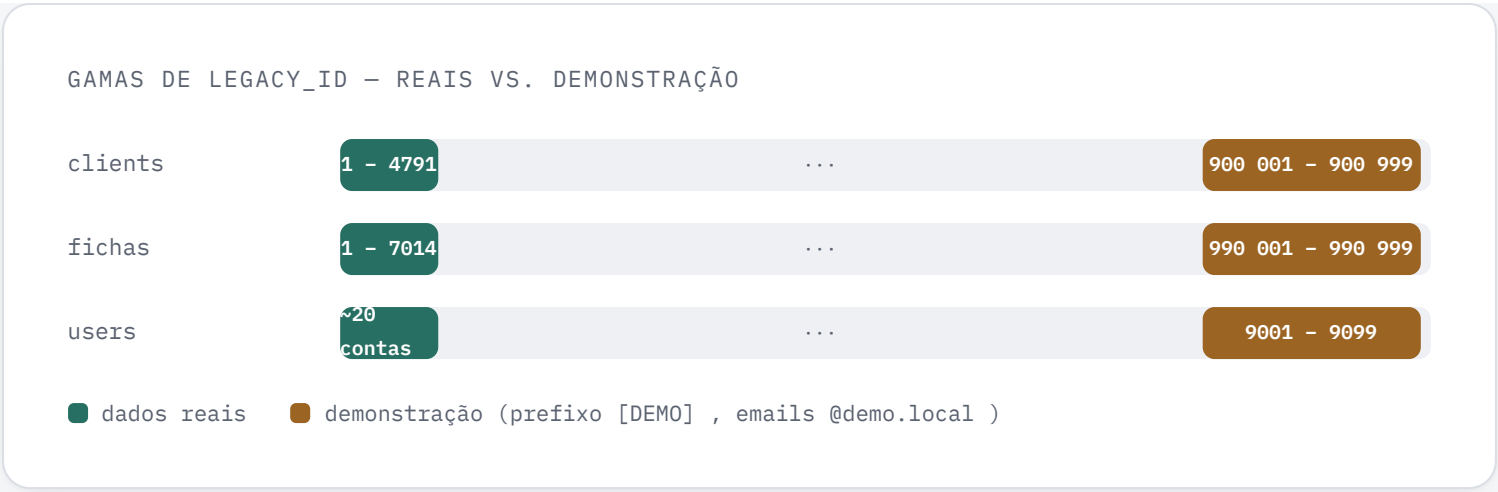
Expõe todo o estado interno do serviço sem precisar de aceder aos logs do servidor:

```
{
  "isRunning": true,
  "lastSync": { "pages": "...", "clients": "...", "fichas": "..." },
  "lastSyncComplete": { "pages": true, "clients": false, "fichas": false },
  "lastError": { "endpoint": "...", "status": 400, "body": { ... } },
  "lastFetchStats": { "clients": { "publish": 4791 }, "fichas": { "publish": 7014 } },
  "wpUrl": "https://sdm.celeuma.pt/wp-json"
}
```

`lastFetchStats` é particularmente útil para diagnóstico: distingue "o WordPress não tem registos pendentes/lixo agora" de "estamos a falhar a contabilizá-los" — sem isto, os dois cenários pareceriam idênticos de fora.

7 Ambientes e Dados de Demonstração

Para gravar vídeos ou demonstrar a aplicação sem expor dados reais de clientes, `backend/scripts/mock_data.sql` semeia registos claramente falsos, em gamas de `legacy_id` que nunca colidem com dados reais.



ISTO NÃO CHEGA SOZINHO

Ter dados de demonstração na base não isola nada por si só: o serviço de sincronização (secção 6) corre independentemente do que já lá está, e volta a importar dados reais do WordPress ao fim de um ciclo. O isolamento real, durante uma gravação, exige também remover `WP_API_URL` do ambiente — sem ela, `syncService.start()` nunca chega a ser chamado.

Configuração por ambiente

AMBIENTE	DB HOST	WP_API_URL	FRONTEND URL
Desenvolvimento	`localhost`	—	`localhost:3000`
Staging / demo	instância MySQL gerida externamente	desligada durante gravações	`sdm-estagio-m1yn.c`
Produção	instância MySQL gerida externamente	`https://sdm.celeuma.pt/wp-json`	subdomínio de celeu definir)

8 Deployment e Configuração

Frontend e backend são dois serviços Render separados — o frontend como site estático (build do CRA, com `public/_redirects` a reescrever qualquer rota para `index.html`, para que o F5 num URL como `/fichas/nova` não dê 404), o backend como serviço web Node. Deploy automático a cada `git push`, via `render.yaml`.

Variáveis de ambiente — backend

VARIÁVEL	PROPÓSITO
`DB_HOST`, `DB_USER`, `DB_PASS` (ou `DB_PASSWORD`), `DB_NAME`	ligação MySQL
`FRONTEND_URLS`	lista branca de CORS, separada por vírgulas
`WP_API_URL`, `WP_API_USER`, `WP_API_PASS`	acesso à REST API do WordPress — presença de `WP_API_URL` liga a sincronização automática
`SYNC_INTERVAL`	milissegundos entre ciclos (60000 = 1 min)
`SYNC_SECRET`, `WEBHOOK_SECRET`	tokens para endpoints de sincronização manual e webhook

Variáveis de ambiente — frontend

VARIÁVEL	PROPÓSITO
`REACT_APP_API_BASE_URL`	URL do backend — só entra no build, exige um novo deploy para ter efeito. Ver problema conhecido, secção 9.

9 Problemas Conhecidos

Registados durante o desenvolvimento — alguns já corrigidos, outros ainda em aberto.

Heurística de URL do backend falha silenciosamente EM ABERTO

`frontend/src/api.js — inferDefaultBaseUrl()`

Só reconhece a convenção de nomes `fichas-frontend` → `fichas-backend`. Nomes de serviço reais que não sigam esse padrão (ex: `sdm-estagio-m1yn` / `sdm-estagio`) fazem o frontend apontar para si próprio — o pedido "sucede" com uma resposta vazia em vez de falhar de forma visível, o que se manifestou como "Servidor de autenticação indisponível" no login.

Mitigação: definir `REACT_APP_API_BASE_URL` explicitamente no build do frontend evita depender da heurística.

Endpoints de administração sem validação de token de sessão EM ABERTO

backend/server.js — assertAdminByLogin()

`PUT /api/admin/users/:id/role` e `POST /api/admin/impersonate` autorizam-se apenas pelo `login` enviado no corpo do pedido e pelo papel desse utilizador na base de dados — não pelo token de sessão de quem está autenticado no browser. Ver secção 5.

Listagem de Páginas com fallback para dados de Fichas MITIGADO

backend/server.js — GET /api/paginas

Se a query a `wp_posts` falhar, o endpoint recorre a um fallback que devolve dados de `fichas` disfarçados de páginas (tipo de contacto no lugar do título). Visível num vídeo de demonstração gravado antes de a causa da falha da query principal ter sido corrigida.

Datas corrompidas ("1899/1900") CORRIGIDO

backend/server.js — safeDateSql() · frontend/src/pages/relatorios.js — isPlausibleYear()

Seriais de data inválidos vindos do WordPress apareciam ocasionalmente como 1899/1900. Reforçado em duas camadas: a query SQL só aceita anos entre 1990–2100, e o mesmo critério é replicado em JavaScript no frontend.

Sincronização não é incremental, apesar do nome lastSync DOCUMENTADO

backend/sync-service.js

Cada ciclo faz sempre um fetch completo — `lastSync` serve só para reporte de estado. Não é um bug per se, mas é fácil de ler mal o código e assumir o contrário; ver secção 6.