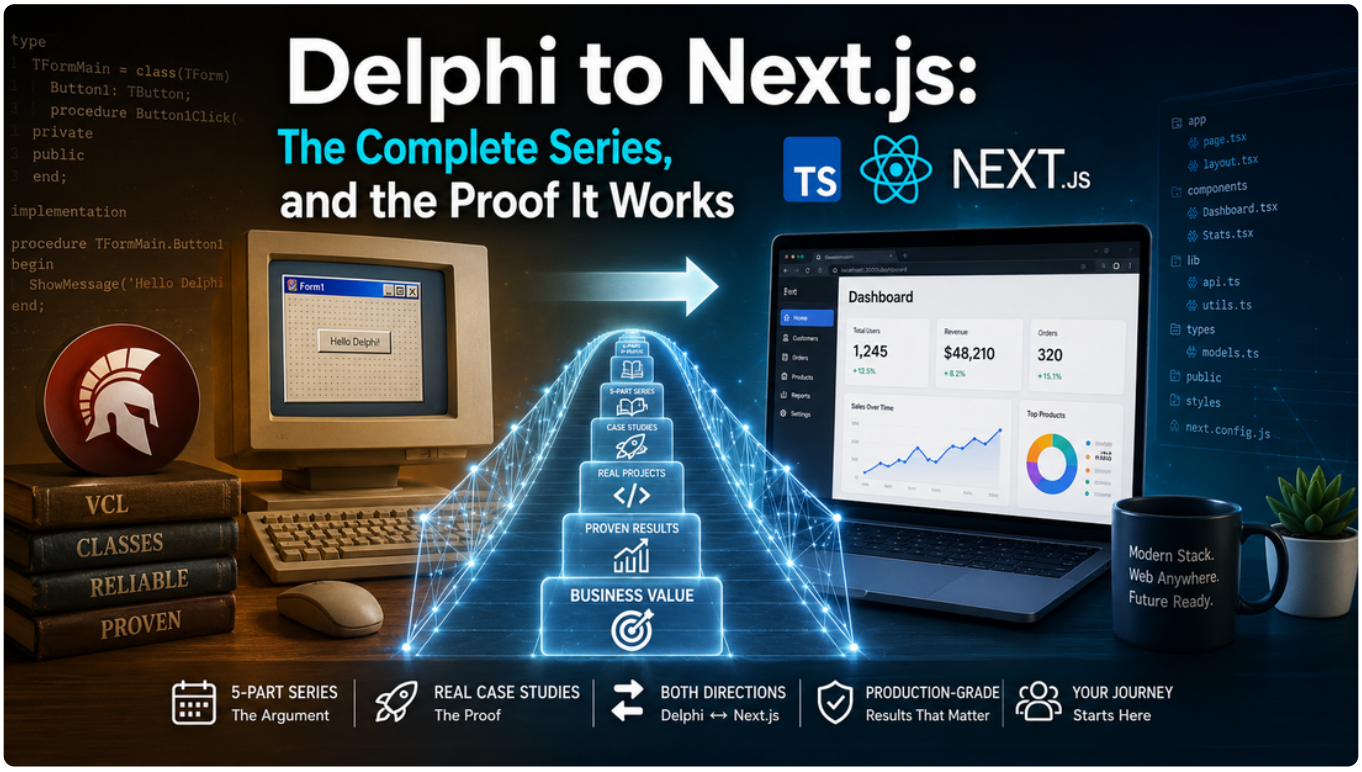
NEWS · JULY 8, 2026

De Delphi para Next.js: a série completa — e a prova de que funciona

By Dr. Holger Flick



No final de 2025 publiquei uma série de cinco partes defendendo que um desenvolvedor Delphi — alguém com décadas de memória muscular em Object Pascal — poderia aprender TypeScript, React e Next.js mais rápido do que temia, e que fazer isso era uma decisão de negócio sensata, não uma traição a uma ferramenta querida. A série acabou se tornando o conteúdo mais lido que já publiquei neste blog.

Naquela época, a série era um *argumento*. Ela se apoiava em mapeamentos de conceitos — interfaces do Delphi para interfaces do TypeScript, componentes VCL para componentes React, estrutura de projeto para estrutura de framework — e em estratégias de migração que eu havia usado em trabalhos de consultoria. O que ela ainda não podia incluir era uma prova pública e documentada.

Ao longo de 2026, isso mudou. Entre fevereiro e julho publiquei uma sequência de estudos de caso que submeteram as afirmações da série à realidade de produção: uma aplicação legada em Delphi 7 reconstruída como uma plataforma web completa em Next.js em um sprint de uma semana, uma aplicação multicamadas em Delphi com seis anos de vida migrada para uma stack moderna em Next.js em cerca de quatro horas — e, na direção oposta, projetos que provam que você pode *manter* seu frontend Delphi e ainda assim ganhar superpoderes modernos.

Este post é a história completa em um só lugar: o que as cinco partes afirmaram, o que os estudos de caso provaram e por onde começar se você está chegando agora.

O argumento que a série apresentou

As cinco partes foram desenhadas como um único arco, do *porquê* passando pelo *o quê* até o *como*, e cada uma se sustenta sozinha se você precisar apenas daquela peça.



O arco de cinco partes: caso de negócio ’ linguagem ’ modelo de UI ’ framework ’ estratégia

Repare que apenas a primeira e a última caixas estão destacadas: a série começa e termina com questões de *negócio*, e as três partes técnicas no meio existem para servi-las. Eis o que cada parte afirmou.

Parte 1 — A oportunidade TypeScript

A Parte 1 defendeu que expandir para a stack web é uma decisão de negócio, não uma questão de moda tecnológica. Os clientes esperam cada vez mais aplicações que rodem em qualquer dispositivo sem instalação, e toda a cadeia de ferramentas de destino — TypeScript, React, Next.js, editores, bancos de dados — está disponível sem custo de licenciamento. O post foi explícito sobre algo que eu ainda sustento: **o Delphi continua fenomenal** naquilo que faz de melhor, especialmente desktop Windows e software corporativo com uso intensivo de banco de dados. O argumento nunca foi "abandone o Delphi"; foi "adicione a web ao seu alcance".

Um fio condutor fez toda a série parecer menos estranha: Anders Hejlsberg, autor original do Turbo Pascal e arquiteto-chefe do Delphi, é o arquiteto principal do TypeScript. A linguagem que você aprenderia foi moldada pela mesma mente que moldou a que você já conhece.

Parte 2 — TypeScript para desenvolvedores Delphi

A Parte 2 percorreu a linguagem lado a lado com o Object Pascal: tipagem forte, interfaces, classes, generics, enums — conceito por conceito. A afirmação era que um desenvolvedor Delphi lê TypeScript com uma sensação de reconhecimento, e não de estranhamento, e que o material genuinamente novo (union types, tipagem estrutural, funções de primeira classe) estende o que você sabe em vez de substituí-lo.

Parte 3 — React e o modelo de componentes

A Parte 3 traduziu o React para o modelo mental que todo desenvolvedor Delphi já tem: componentes. Solte um `TButton` em um formulário, defina propriedades, trate eventos — o React é a mesma ideia composicional para a web, com props fazendo o papel das propriedades published e o state fazendo o papel dos dados que seu formulário gerencia. O propósito do post era avaliação, não tutorial: entender o modelo bem o suficiente para julgar se a UI da sua aplicação se traduz.

Parte 4 — Estrutura e arquitetura do Next.js

A Parte 4 posicionou o Next.js como o equivalente da estrutura de projeto, da biblioteca de runtime e das ferramentas de deploy do Delphi combinadas: roteamento baseado em arquivos, rotas de API para a lógica de

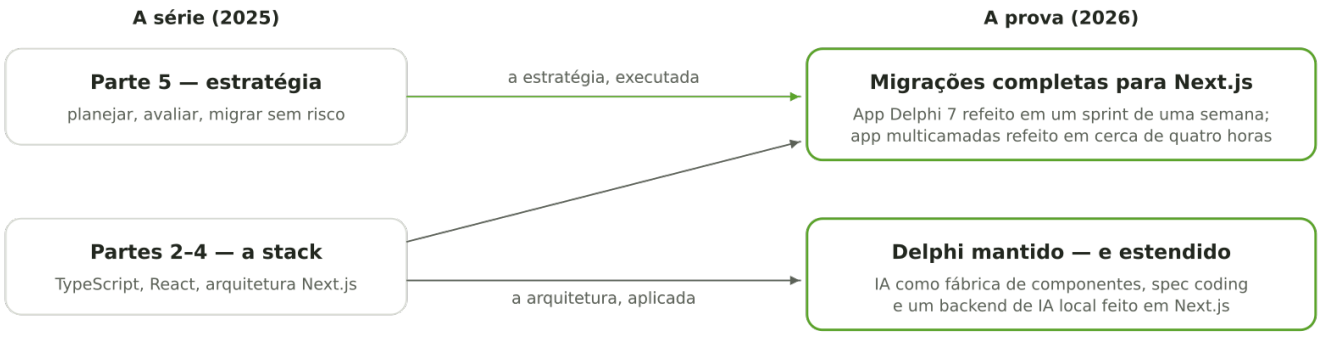
backend, acesso a banco de dados e a distinção entre servidor e cliente. A conclusão principal era arquitetural — uma aplicação Next.js pode ser o sistema *inteiro*, frontend e backend em um único projeto implantável.

Parte 5 — Estratégia e planejamento de migração

A Parte 5 foi a parte escrita para donos de negócio, não para compiladores: quatro estratégias de migração (API wrapper, desenvolvimento paralelo, módulo a módulo, apenas funcionalidades novas), como avaliar uma aplicação com honestidade e por que o "grande rewrite" é um mito a ser evitado. Sua mensagem central: migração é gestão de risco, e a estratégia certa depende da sua aplicação, da sua equipe e dos seus clientes — não do entusiasmo de alguém por um framework.

Do argumento à evidência

Nos meses seguintes à série, cinco estudos de caso publicados colocaram suas afirmações centrais à prova de verdade — e eles seguem em *ambas* as direções: alguns substituem um sistema Delphi por Next.js por completo, outros deliberadamente mantêm o Delphi no lugar e o estendem com capacidades modernas.



Como os estudos de caso de 2026 se conectam à série de 2025

Leia o diagrama da esquerda para a direita: o pensamento estratégico da Parte 5 conduziu as migrações completas, enquanto o conhecimento da stack das Partes 2–4 é a matéria-prima de todos os projetos à direita — inclusive daqueles que deliberadamente *mantêm* o Delphi.

O caminho adiante: duas migrações completas para Next.js

A afirmação mais pesada da série era a da Parte 5: que uma aplicação Delphi real, carregando dados de verdade, pode migrar para a web sem uma marcha da morte de vários anos. Dois projetos publicados a sustentam.

Adaptar-se ou desaparecer: como a IA transformou um projeto de 2 anos em um sprint de 1 semana documenta a aplicação legada de um cliente construída em **Delphi 7** — lançada em 2002, sem Unicode, sem HiDPI — rodando sobre Firebird 1.5, reconstruída como uma plataforma TypeScript full-stack em Next.js com Prisma e uma UI responsiva, pronta para dispositivos móveis. Autenticação, controle de acesso baseado em papéis, dados relacionais complexos e um pipeline migrando dados reais de produção — cerca de 16.000 linhas de TypeScript artesanal entregues em um sprint de uma semana, um multiplicador de produtividade de 15–25× em relação às estimativas padrão da indústria que o post analisa em detalhe. Os clientes vinham pedindo essa modernização havia mais de vinte anos.

Quatro horas para migrar um app Delphi multicamadas de 6 anos leva o mesmo resultado mais longe: uma aplicação em produção — um frontend TMS WEB Core, um backend REST TMS XData e um banco de dados Firebird com dezenas de milhares de linhas de histórico real de usuários — reconstruída sobre Next.js 16, React 19, Prisma 7 e PostgreSQL. Novo schema, nova autenticação, gráficos recriados, uma ferramenta de migração de dados ao vivo, scripts de deploy. Cerca de quatro horas, do início ao fim.

Ambos os posts são deliberadamente honestos sobre por que os números da manchete *não* são a história.

A velocidade foi o retorno de meses gastos aprendendo a conduzir o desenvolvimento assistido por IA como uma ferramenta séria de engenharia: convenções de projeto preparadas, acesso controlado aos fontes legados e revisão constante, por um especialista, de tudo o que a IA produzia. O ciclo de detectar-e-corrigir, não o cronômetro, é a verdadeira descoberta.

> **Onde os números da manchete não se generalizam**
> Não leia esses estudos de caso como "qualquer app Delphi migra em uma tarde". Cada um foi migrado por alguém que conhecia profundamente tanto o sistema de origem quanto a stack de destino, após meses de preparação deliberada de ferramentas de IA e convenções. O que eles provam é o *teto*: com os conceitos da série internalizados e assistência moderna de IA, a curva de custo que a Parte 5 descreveu em 2025 caiu dramaticamente. Sua primeira migração não vai levar quatro horas — mas também não precisa mais levar anos.

O caminho de volta: manter o Delphi e adicionar superpoderes

A outra promessa da série — aquela dos primeiríssimos parágrafos da Parte 1 — era que o Delphi continua fenomenal e que ninguém está pedindo que você o abandone. Três projetos publicados provam que a mesma caixa de ferramentas moderna torna seu trabalho Delphi *existente* mais forte.

A IA não vai substituir desenvolvedores Delphi. Mas... parte de uma thread de fórum em que um desenvolvedor havia passado dias caçando um componente de conversão de bitmap para escala de cinza — e mostra a IA produzindo uma solução VCL limpa e sem dependências em cerca de dois minutos. A lição se generaliza: a IA transforma problemas de "encontrar e avaliar uma biblioteca de terceiros" em problemas de "gerar exatamente o código de que preciso", inteiramente dentro do Delphi.

Spec Coding com Delphi dá um passo além: uma ferramenta nativa que varre um banco de dados curado pela comunidade com mais de 12.000 jogos de PC, detecta instalações via registro do Windows e faz backup de arquivos de save com upload opcional para S3 — construída em uma tarde com o Claude Code e compilada em um único executável nativo de 4 MB. A metodologia que o post batiza de *spec coding* — escrever especificações precisas, deixar a IA implementar, revisar com rigor — é a mesma disciplina por trás das grandes migrações, aplicada a código Delphi novo em folha.

Dê um cérebro ao seu app Delphi — sem enviar um único byte para a nuvem fecha o ciclo combinando os dois mundos: um frontend Delphi VCL testado em batalha fica exatamente onde está, e um pequeno serviço web em Next.js ao lado dele executa um modelo de IA local — no exemplo trabalhado, transformando uma nota fiscal enviada em registros de despesa estruturados e revisáveis. O lado Delphi precisa de pouco mais que uma chamada REST, e o backend entrega de brinde uma UI web de administração. Isso é a arquitetura da Parte 4 (Next.js como frontend e backend autocontidos) e a estratégia "apenas funcionalidades novas" da Parte 5 em um único artefato.

Juntos, esses três respondem ao medo que deu origem à série: adicionar essa stack à sua caixa de ferramentas não significa abandonar quinze anos de código Delphi que funciona. Às vezes significa estendê-lo com capacidades que nenhuma quantidade de parsing com `TStringList` jamais vai entregar.

O que 2026 acrescentou ao argumento de 2025

Os mapeamentos de conceitos da série envelheceram bem — o TypeScript ainda se lê como uma linguagem familiar para um desenvolvedor Delphi, e o modelo de componentes ainda se traduz. Minha opinião, e eu a defendo: **TypeScript é a segunda linguagem mais natural que um desenvolvedor Delphi pode aprender hoje.** A evidência mais forte continua sendo a linhagem — Hejlsberg arquitetou ambas — além de tudo o que a Parte 2 demonstrou lado a lado.

> **O outro lado**
> Delimite essa opinião com honestidade: ela vale para a *linguagem*, não para o *ecossistema*. O mundo do npm se move mais rápido que o mundo Delphi, o ferramental de build tem mais peças móveis que um único `.dproj`, e o modelo de programação assíncrona é território genuinamente novo. A linguagem vai parecer familiar em semanas; a fluência no ecossistema leva meses. Os estudos de caso acima foram feitos *depois* que essa fluência foi conquistada, não antes.

O que a série não podia ter antecipado por completo é o quanto o desenvolvimento assistido por IA mudaria a economia da coisa. Os cronogramas da Parte 5 pressupunham migração manual; os resultados de uma semana e de quatro horas mostram o que se torna possível quando um desenvolvedor que entende as duas stacks supervisiona uma IA que faz a tradução mecânica. O julgamento — desenho de schema, capturar saídas erradas ou feias, saber como é o "correto" — continua sendo inteiramente humano, e é exatamente o conhecimento que as cinco partes ensinam.

> **Alternativas**
> Tudo no lado de destino desta série é open source e gratuito: Next.js, React e TypeScript são desenvolvidos publicamente no GitHub. O Next.js também não é o único caminho crível — Nuxt (Vue) e SvelteKit resolvem o mesmo problema com excelência, e os conceitos das Partes 2–5 se transferem para eles quase sem mudanças. E se a sua equipe quer a web permanecendo em Pascal, o TMS WEB Core compila código Delphi para JavaScript e é uma opção genuinamente produtiva — o estudo de caso de quatro horas migrou *a partir* dele não porque ele falhou, mas porque os objetivos daquele projeto haviam superado o seu encaixe.

Por onde começar

Se você está chegando à série agora, existe uma ordem natural de leitura.

1. Parte 1 — A oportunidade TypeScript para o caso de negócio e o enquadramento honesto do que o Delphi ainda faz de melhor.
2. Parte 2 — TypeScript, Parte 3 — React e Parte 4 — Next.js em sequência, para a linguagem, o modelo de UI e o framework — cada um mapeado para conceitos Delphi que você já domina.
3. Parte 5 — Estratégia de migração antes de escrever uma única linha de um projeto real, porque falhas de estratégia custam mais caro que erros de sintaxe.
4. Depois, a prova, nas duas direções: a migração do Delphi 7 em uma semana e a migração multicamadas em quatro horas para ver como é uma mudança completa — e a solução de escala de cinza em dois

minutos, spec coding de uma ferramenta nativa em uma tarde e o scanner de documentos com IA local para ver como é *não* migrar e, em vez disso, estender.

Conclusão

A série de 2025 afirmou que os conceitos duramente conquistados de um desenvolvedor Delphi se transferem para a stack web moderna, que a transição é uma decisão de negócio administrável em vez de um salto de fé, e que Delphi e Next.js podem coexistir em uma única estratégia de produto. Os estudos de caso de 2026 transformaram cada uma dessas afirmações de argumento em prática documentada e reproduzível — alguns migrando sistemas reais de produção, outros provando que nenhuma migração era necessária.

> A série ensinou o mapa. Os estudos de caso percorreram a estrada — nas duas direções.

Se você está pesando essa decisão para a sua própria aplicação, o framework de avaliação da Parte 5 é o ponto de partida — e se quiser um guia experiente para a viagem, vamos conversar.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —