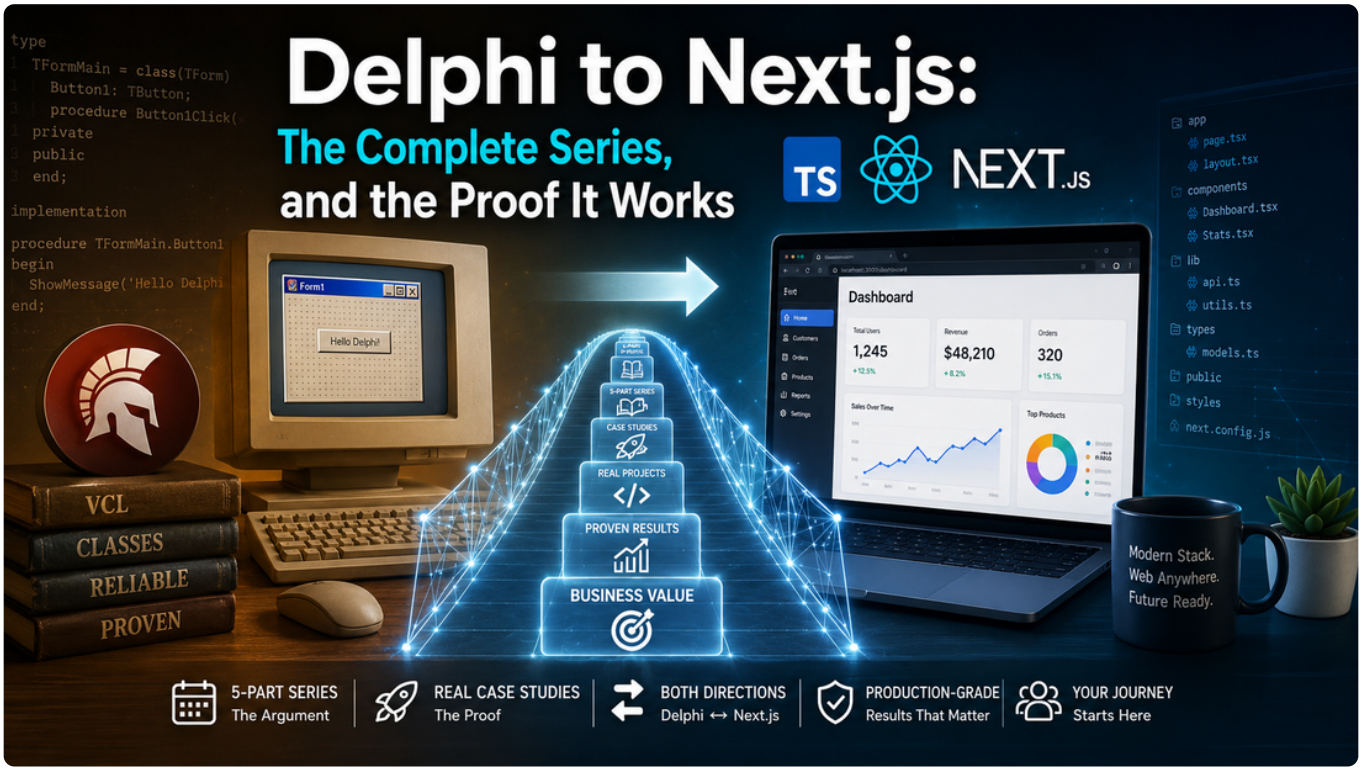
NEWS · JULY 8, 2026

De Delphi a Next.js: la serie completa, y la prueba de que funciona

By Dr. Holger Flick



A finales de 2025 publiqué una serie de cinco partes en la que sostenía que un desarrollador de Delphi — alguien con décadas de memoria muscular en Object Pascal — podía aprender TypeScript, React y Next.js más rápido de lo que temía, y que hacerlo era una decisión de negocio sensata, no una traición a una herramienta querida. La serie acabó convirtiéndose en el contenido más leído que he publicado jamás en este blog.

En aquel momento, la serie era un *argumento*. Se apoyaba en correspondencias de conceptos — interfaces de Delphi con interfaces de TypeScript, componentes VCL con componentes React, estructura de proyecto con estructura de framework — y en estrategias de migración que yo había usado en trabajos de consultoría. Lo que aún no podía incluir era una prueba pública y documentada.

A lo largo de 2026, eso cambió. Entre febrero y julio publiqué una serie de casos de estudio que sometieron las afirmaciones de la serie a la realidad de producción: una aplicación legacy en Delphi 7 reconstruida como una plataforma web completa en Next.js en un sprint de una semana, una aplicación multicapa en Delphi de seis años migrada a un stack moderno de Next.js en unas cuatro horas — y, en la dirección contraria, proyectos que demuestran que puedes *conservar* tu frontend en Delphi y aun así ganar superpoderes modernos.

Este artículo es la historia completa en un solo lugar: qué afirmaban las cinco partes, qué demostraron los casos de estudio y por dónde empezar si llegas ahora.

El argumento que planteó la serie

Las cinco partes fueron diseñadas como un único arco, del *porqué* al *qué* y al *cómo*, y cada una se sostiene por sí sola si solo necesitas esa pieza.

El arco de cinco partes: caso de negocio ’ lenguaje ’ modelo de UI ’ framework ’ estrategia

Fíjate en que solo la primera y la última caja están resaltadas: la serie empieza y termina con preguntas de *negocio*, y las tres partes técnicas del medio existen para servirlas. Esto es lo que afirmaba cada parte.

Parte 1 — La oportunidad de TypeScript

La Parte 1 defendía que expandirse hacia el stack web es una decisión de negocio, no una declaración de moda tecnológica. Los clientes esperan cada vez más aplicaciones que funcionen en cualquier dispositivo sin instalación, y toda la cadena de herramientas de destino — TypeScript, React, Next.js, editores, bases de datos — está disponible sin coste de licencias. El artículo era explícito en algo que sigo sosteniendo: **Delphi sigue siendo fenomenal** en lo que mejor hace, especialmente el escritorio de Windows y el software empresarial intensivo en bases de datos. El argumento nunca fue "abandona Delphi"; fue "añade la web a tu alcance".

Un hilo hizo que toda la serie resultara menos ajena: Anders Hejlsberg, autor original de Turbo Pascal y arquitecto jefe de Delphi, es el arquitecto principal de TypeScript. El lenguaje que ibas a aprender fue moldeado por la misma mente que moldeó el que ya conoces.

Parte 2 — TypeScript para desarrolladores Delphi

La Parte 2 recorría el lenguaje lado a lado con Object Pascal: tipado fuerte, interfaces, clases, genéricos, enumeraciones — concepto por concepto. La afirmación era que un desarrollador de Delphi lee TypeScript con una sensación de reconocimiento y no de extrañeza, y que el material genuinamente nuevo (union types, tipado estructural, funciones de primera clase) amplía lo que ya sabes en lugar de reemplazarlo.

Parte 3 — React y el modelo de componentes

La Parte 3 traducía React al modelo mental que todo desarrollador de Delphi ya tiene: componentes. Soltar un `TButton` en un formulario, establecer propiedades, manejar eventos — React es la misma idea composicional para la web, donde las props hacen el papel de las propiedades published y el estado hace el papel de los datos que gestiona tu formulario. El propósito del artículo era la evaluación, no el tutorial: entender el modelo lo bastante bien como para juzgar si la UI de tu aplicación se traduce.

Parte 4 — Estructura y arquitectura de Next.js

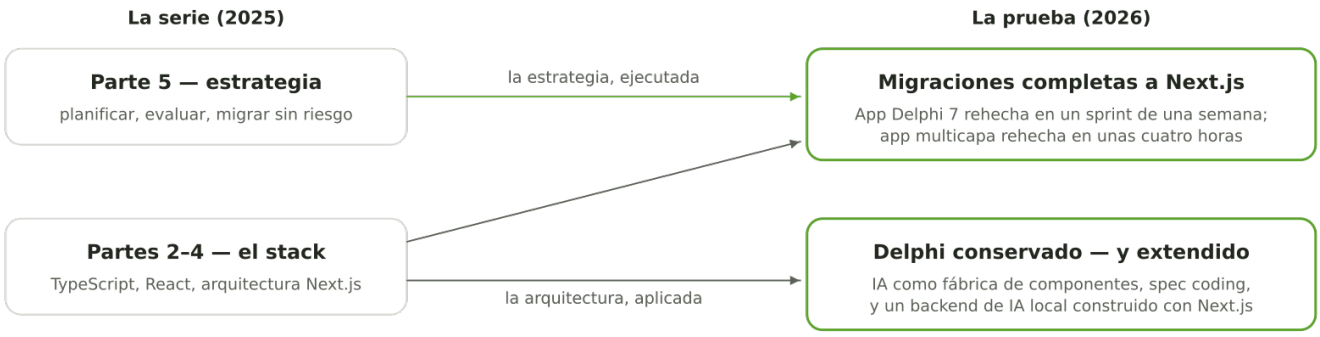
La Parte 4 presentaba Next.js como el equivalente de la estructura de proyecto de Delphi, su biblioteca de runtime y sus herramientas de despliegue, todo combinado: enrutamiento basado en archivos, rutas de API para la lógica de backend, acceso a bases de datos y la distinción servidor/cliente. La conclusión clave era arquitectónica: una aplicación Next.js puede ser el sistema *completo*, frontend y backend en un solo proyecto desplegable.

Parte 5 — Estrategia y planificación de la migración

La Parte 5 fue la parte escrita para dueños de negocio y no para compiladores: cuatro estrategias de migración (API wrapper, desarrollo en paralelo, módulo a módulo, solo funcionalidades nuevas), cómo evaluar una aplicación con honestidad, y por qué la "gran reescritura" es un mito que hay que evitar. Su mensaje central: la migración es gestión de riesgos, y la estrategia correcta depende de tu aplicación, tu equipo y tus clientes — no del entusiasmo de nadie por un framework.

Del argumento a la evidencia

En los meses posteriores a la serie, cinco casos de estudio publicados pusieron a prueba real sus afirmaciones centrales — y van en *ambas* direcciones: algunos reemplazan por completo un sistema Delphi con Next.js, otros conservan deliberadamente Delphi y lo extienden con capacidades modernas.



Cómo los casos de estudio de 2026 se corresponden con la serie de 2025

Lee el diagrama de izquierda a derecha: el pensamiento estratégico de la Parte 5 impulsó las migraciones completas, mientras que el conocimiento del stack de las Partes 2–4 es la materia prima de cada proyecto de la derecha — incluidos los que deliberadamente *conservan* Delphi.

El camino de ida: dos migraciones completas a Next.js

La afirmación más contundente de la serie era la de la Parte 5: que una aplicación Delphi real, con datos de verdad, puede pasar a la web sin una marcha fúnebre de varios años. Dos proyectos publicados la respaldan.

Adapt or Disappear: How AI Turned a 2-Year Project Into a 1-Week Sprint documenta la aplicación legacy de un cliente construida en **Delphi 7** — lanzada en 2002, sin Unicode, sin HiDPI — sobre Firebird 1.5, reconstruida como una plataforma full-stack en TypeScript sobre Next.js con Prisma y una UI responsiva apta para móviles. Autenticación, control de acceso basado en roles, datos relacionales complejos y un pipeline que migró datos reales de producción — unas 16.000 líneas de TypeScript escritas a medida, entregadas en un sprint de una semana, un multiplicador de productividad de 15–25× frente a las estimaciones estándar del sector que el artículo desglosa en detalle. Los clientes llevaban más de veinte años pidiendo esa modernización.

Four Hours to Migrate a 6-Year-Old Delphi Multi-Tier App lleva el mismo resultado más lejos: una aplicación en producción — un frontend en TMS WEB Core, un backend REST en TMS XData y una base de datos Firebird con decenas de miles de filas de historial real de usuarios — reconstruida sobre Next.js 16, React 19, Prisma 7 y PostgreSQL. Nuevo esquema, nueva autenticación, gráficas recreadas, una herramienta de migración de datos en vivo, scripts de despliegue. Unas cuatro horas, de principio a fin.

Ambos artículos son deliberadamente honestos sobre por qué las cifras del titular *no* son la historia. La velocidad fue la recompensa de meses dedicados a aprender a manejar el desarrollo asistido por IA como una herramienta seria de ingeniería: convenciones de proyecto preparadas, acceso controlado a las fuentes legacy y revisión experta constante de todo lo que producía la IA. El bucle de detectar-y-corregir, no el cronómetro, es el verdadero hallazgo.

> **Dónde las cifras del titular no generalizan**
> No leas esos casos de estudio como "cualquier app Delphi se migra en una tarde". Cada una fue migrada por alguien que conocía a fondo tanto el sistema de origen como el stack de destino, tras meses de preparación deliberada de herramientas de IA y convenciones. Lo que demuestran es el *techo*: con los conceptos de la serie interiorizados y la asistencia moderna de la IA, la curva de costes que la Parte 5 describió en 2025 se ha desplazado drásticamente hacia abajo. Tu primera migración no te llevará cuatro horas — pero tampoco tiene ya por qué llevarte años.

El camino de vuelta: conservar Delphi y añadirle superpoderes

La otra promesa de la serie — la de los primerísimos párrafos de la Parte 1 — era que Delphi sigue siendo fenomenal y que nadie te pide abandonarlo. Tres proyectos publicados demuestran que la misma caja de herramientas moderna hace más fuerte tu trabajo *existente* en Delphi.

AI Won't Replace Delphi Developers. But... parte de un hilo de foro donde un desarrollador había pasado días buscando un componente para convertir bitmaps a escala de grises — y muestra a la IA produciendo una solución VCL limpia y sin dependencias en unos dos minutos. La lección generaliza: la IA convierte los problemas de "encontrar y evaluar una biblioteca de terceros" en problemas de "generar exactamente el código que necesito", enteramente dentro de Delphi.

Spec Coding with Delphi va un paso más allá: una herramienta nativa que escanea una base de datos mantenida por la comunidad con más de 12.000 juegos de PC, detecta instalaciones a través del registro de Windows y respalda los archivos de guardado con subida opcional a S3 — construida en una tarde con Claude Code y compilada en un único ejecutable nativo de 4 MB. La metodología que el artículo bautiza como *spec coding* — escribir especificaciones precisas, dejar que la IA implemente, revisar con rigor — es la misma disciplina que hay detrás de las grandes migraciones, aplicada a código Delphi completamente nuevo.

Give Your Delphi App a Brain — Without Sending a Single Byte to the Cloud cierra el círculo combinando ambos mundos: un frontend Delphi VCL curtido en batalla se queda exactamente donde está, y a su lado un pequeño servicio web en Next.js ejecuta un modelo de IA local — en el ejemplo desarrollado, convirtiendo una factura subida en registros de gastos estructurados y revisables. El lado Delphi apenas necesita más que una llamada REST, y el backend incluye gratis una UI web de administración. Esto es la arquitectura de la Parte 4 (Next.js como frontend y backend autocontenidos) y la estrategia de "solo funcionalidades nuevas" de la Parte 5 en un mismo artefacto.

Juntos, estos tres responden al miedo que originó la serie: añadir este stack a tu caja de herramientas no significa abandonar quince años de código Delphi que funciona. A veces significa extenderlo con capacidades que ninguna cantidad de parsing con `TStringList` va a entregar jamás.

Lo que 2026 añadió al argumento de 2025

Las correspondencias de conceptos de la serie han envejecido bien — TypeScript sigue leyéndose como un lenguaje familiar para un desarrollador de Delphi, y el modelo de componentes sigue traduciéndose. Mi opinión, y la defenderé: **TypeScript es el segundo lenguaje más natural que un desarrollador de Delphi puede aprender hoy.** La evidencia más sólida a favor sigue siendo el linaje — Hejlsberg fue el arquitecto de ambos — más todo lo que la Parte 2 demostró lado a lado.

> **La otra cara**
> Delimita esa opinión con honestidad: vale para el *lenguaje*, no para el *ecosistema*. El mundo de npm se mueve más rápido que el mundo de Delphi, las herramientas de build tienen más piezas móviles que un único `.dproj`, y el modelo de programación asíncrona es territorio genuinamente nuevo. El lenguaje se sentirá como en casa en cuestión de semanas; la fluidez con el ecosistema lleva meses. Los casos de estudio de arriba se hicieron *después* de ganarse esa fluidez, no antes.

Lo que la serie no podía anticipar del todo es cuánto cambiaría la economía el desarrollo asistido por IA. Los plazos de la Parte 5 asumían migración a mano; los resultados de una semana y de cuatro horas muestran lo que se vuelve posible cuando un desarrollador que entiende ambos stacks supervisa a una IA que hace la traducción mecánica. El criterio — el diseño del esquema, detectar salidas incorrectas o feas, saber qué aspecto tiene lo "correcto" — sigue siendo enteramente humano, y es exactamente el conocimiento que enseñan las cinco partes.

> **Alternativas**
> Todo lo que hay en el lado de destino de esta serie es open source y gratuito: Next.js, React y TypeScript se desarrollan públicamente en GitHub. Next.js tampoco es el único camino creíble — Nuxt (Vue) y SvelteKit resuelven el mismo problema de forma excelente, y los conceptos de las Partes 2–5 se transfieren a ellos casi sin cambios. Y si tu equipo quiere la web sin salir de Pascal, TMS WEB Core compila código Delphi a JavaScript y es una opción genuinamente productiva — el caso de estudio de las cuatro horas migró *desde* él no porque fallara, sino porque los objetivos de ese proyecto habían superado su encaje.

Por dónde empezar

Si llegas a esta serie ahora, hay un orden de lectura natural.

1. Parte 1 — La oportunidad de TypeScript para el caso de negocio y el planteamiento honesto de lo que Delphi sigue haciendo mejor.
2. Parte 2 — TypeScript, Parte 3 — React y Parte 4 — Next.js en orden, para el lenguaje, el modelo de UI y el framework — cada uno mapeado a conceptos de Delphi que ya dominas.
3. Parte 5 — Estrategia de migración antes de escribir una sola línea de un proyecto real, porque los fallos de estrategia salen más caros que los errores de sintaxis.
4. Después, la prueba, en ambas direcciones: la migración de Delphi 7 en una semana y la migración multicapa en cuatro horas para ver cómo es una mudanza completa — y la solución de escala de grises

en dos minutos, spec coding de una herramienta nativa en una tarde y el escáner de documentos con IA local para ver cómo es *no* mudarse y, en su lugar, extender.

En resumen

La serie de 2025 afirmaba que los conceptos duramente ganados de un desarrollador de Delphi se transfieren al stack web moderno, que la transición es una decisión de negocio manejable y no un salto de fe, y que Delphi y Next.js pueden coexistir en una misma estrategia de producto. Los casos de estudio de 2026 convirtieron cada una de esas afirmaciones de argumento en práctica documentada y reproducible — algunos migrando sistemas reales en producción, otros demostrando que no hacía falta migración alguna.

> La serie enseñó el mapa. Los casos de estudio recorrieron la carretera — en ambas direcciones.

Si estás sopesando esta decisión para tu propia aplicación, el marco de evaluación de la Parte 5 es el punto de partida — y si quieres un guía con experiencia para el viaje, hablemos.

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 —