**Engineer**

# TypeScript

JavaScript con tipos comprobados antes de ejecutarse

## Qué es

TypeScript es un lenguaje de programación de Microsoft que añade anotaciones de tipos a JavaScript, junto con un compilador, `tsc`, que las comprueba. Su nombre suele abreviarse como TS y sus archivos terminan en `.ts`, o en `.tsx` cuando contienen marcado JSX. La sintaxis de JavaScript es válida en TypeScript; por eso se dice que este lenguaje es un superconjunto de JavaScript. El compilador es de código abierto y se distribuye bajo la licencia Apache 2.0.

Los tipos solo duran lo que dura la comprobación. Las anotaciones de tipos ordinarias se eliminan cuando TypeScript emite JavaScript; no validan los datos durante la ejecución ni requieren una biblioteca que compruebe los tipos al ejecutar el programa. Las anotaciones no cambian el comportamiento del código JavaScript durante su ejecución. Este hecho explica tanto su utilidad como sus límites: el comprobador razona sobre los tipos descritos en el código fuente; no valida los datos que llegan mientras el programa se ejecuta.

El sistema de tipos es estructural. Dos tipos son compatibles cuando sus estructuras coinciden, sea cual sea su nombre. Esto encaja con la costumbre de JavaScript de pasar objetos simples de un lugar a otro; C# y Java, en cambio, comparan los nombres declarados. Gran parte de los tipos se infiere, de modo que las anotaciones aparecen sobre todo en las firmas de las funciones y en los valores exportados. Además, por diseño, el sistema no garantiza la seguridad de tipos en algunos casos. `any` desactiva la comprobación de un valor y una aserción con `as` indica al compilador que acepte el tipo que afirma quien programa. Ambos desaparecen del resultado junto con las demás anotaciones.

Hay tres formas habituales de ejecutarlo. `tsc` comprueba un proyecto y emite JavaScript. Empaquetadores como esbuild eliminan los tipos sin comprobarlos y dejan esa tarea para una ejecución independiente de `tsc --noEmit`. Desde las

versiones v23.6.0 y v22.18.0, Node.js ejecuta directamente archivos `.ts` sustituyendo la sintaxis de tipos por espacios en blanco, también sin comprobar los tipos, y solo admite sintaxis que pueda eliminarse de esa manera. TypeScript 7.0, anunciado en julio de 2026, es una versión nativa del compilador escrita en Go que, según su equipo, suele ser entre 8 y 12 veces más rápida en compilaciones completas.

Utilice el concepto en una decisión

Aplique «¿Qué es TypeScript?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. Interfaces que cargan rápido, funcionan en el teléfono y se comportan igual en los navegadores que utilizan sus clientes.

**Explore nuestros servicios** Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Dirección del desarrollo integral de mapas GIS — cartografía integral, de la base de datos al navegador

**Dirección del desarrollo integral de mapas GIS** Póngase en contacto →

## Por qué importa

La cuestión para un equipo es si una fase de compilación y un comprobador justifican su coste al detectar errores antes de ejecutar el código. En una base de código que varias personas modifican cada semana, suele compensar. Al cambiar el nombre de un campo aparecen todos los lugares que lo usaban; el editor completa y explica el código a partir de los tipos, y la firma de un módulo se convierte en un contrato que sus consumidores no pueden incumplir inadvertidamente. El coste es una comprobación en cada compilación, un archivo de configuración con muchas opciones y unas definiciones de tipos que deben mantenerse al día con las bibliotecas que describen.

El rigor de las comprobaciones es el ajuste más importante. La opción `strict` activa una familia de comprobaciones, entre ellas `strictNullChecks` y `noImplicitAny`; el manual describe el resultado como un regulador gradual, más que como un interruptor. Desde TypeScript 6.0 está activada de forma predeterminada, por lo que un proyecto nuevo empieza con comprobaciones estrictas. Uno anterior puede seguir teniéndola configurada como false y activarla después exige revisar todos los archivos afectados. Dos comprobaciones útiles quedan fuera de esa familia y deben activarse por separado: `noUncheckedIndexedAccess` y `exactOptionalPropertyTypes`.

La eliminación de los tipos marca el límite. Un programa que lee JSON de una solicitud de red, un archivo, una cola o un formulario solo cuenta con la palabra de quien programa acerca de la estructura de esos datos: cuando llegan, los tipos ya han desaparecido. Por tanto, los límites del programa necesitan validación durante la ejecución, idealmente generada a partir de la misma definición que los tipos para que ambas no puedan discrepar. TypeScript mantiene la coherencia interna del programa y deja sus límites a cargo del código.

El comprobador también aporta información a otras herramientas. Un analizador como typescript-eslint puede pedir al compilador información de tipos para detectar, por ejemplo, una promesa que nadie espera o un valor de tipo `any` que se propaga por un módulo. Así consigue un equipo que las infracciones de sus propias convenciones se notifiquen como errores, en lugar de quedar en comentarios de revisión.

## Cómo se aplica en la práctica

En los proyectos que muestra este sitio, la fiabilidad de TypeScript se sostiene con código generado y herramientas, en lugar de depender de convenciones. Tres páginas muestran cómo.

En una plataforma, el cliente TypeScript del frontend nunca se escribe a mano. Se genera a partir de la propia descripción de la API, que Huma emite a partir de los tipos de Go; estos, a su vez, siguen el esquema de la base de datos y sus funciones. En la fase de implementación documentada, el cliente sumaba 44.076 líneas y los 61 manejadores simulados generados junto con él permitían ejecutar las pruebas del frontend contra el contrato real, en lugar de contra datos de prueba escritos a mano. Cada paso tiene su propia comprobación de protección. Una comprobación del contrato en CI compara lo que envía el frontend con lo que espera la API y hace fallar la compilación si difieren, apoyada por una sonda del esquema que inspecciona las estructuras reales. Abarca los campos opcionales del cuerpo, donde es fácil confundir un campo ausente con null, y las enumeraciones en parámetros de consulta. Un cambio de campo en un lado pasa así a hacer fallar una compilación minutos después del cambio, en vez de aparecer como un 422 en el navegador una vez publicada la incompatibilidad. A cambio, un cambio del contrato solo llega a los tipos del cliente al ejecutar de nuevo toda la cadena, y regenerarlos resulta laborioso.

Otra página establece el mismo listón para seis lenguajes, entre ellos TypeScript. Todo diagnóstico que comunica ESLint es un error: no hay un nivel de advertencias en el que refugiarse y se prohíben las supresiones dentro del código, por lo que cada hallazgo debe corregirse en vez de silenciarse. Los hooks previos al commit y al push ejecutan los analizadores y las pruebas; los límites de longitud de funciones y archivos mantienen legibles los módulos.

Los analizadores incluidos con un lenguaje no cubren todas las reglas propias de un proyecto. Por eso, quince reglas de ESLint personalizadas imponían los patrones de TypeScript de aquel repositorio en la fase de implementación documentada. Formaban parte de un motor de comprobaciones con 274 scripts, de los cuales 268 estaban registrados en los hooks de commit. Las comprobaciones contaban con 146 pruebas propias, porque una comprobación averiada lo da todo por válido sin que nadie lo advierta. Una regla trata de las pruebas, no de los tipos: un commit cuyo prefijo sea `fix:` debe incluir una prueba que falle sin la corrección. El tiempo de revisión dejó de dedicarse a la mecánica para centrarse en el diseño, a cambio de que los commits se volvieran lentos.

## Dónde surgen los problemas

La mayoría de los fallos de TypeScript nacen de confiar en los tipos más allá de lo que puede ver el compilador.

**Un campo de tipo cadena resulta ser undefined durante la ejecución.** Los datos llegaron de fuera del programa y se les asignó un tipo en lugar de comprobarlos. La biblioteca del DOM asigna a `res.json()` el tipo de promesa de `any`, de modo que el compilador acepta cualquier uso del resultado. Una aserción hace explícita esa confianza sin aportar seguridad:

```
const res = await fetch("/api/user");
const user = (await res.json()) as User; // trusted, never checked
user.email.toLowerCase(); // TypeError if email is missing
```

Trate los datos externos como `unknown`, valídelos en el límite con un esquema de validación durante la ejecución y genere el tipo a partir de ese esquema para mantenerlos sincronizados. Una regla de análisis que use información de tipos podrá detectar cualquier `any` que se cuele.

**El código con errores de tipos llega igualmente a producción.** `tsc` sigue escribiendo JavaScript cuando comunica errores porque `noEmitOnError` tiene false como valor predeterminado. Los empaquetadores y Node.js eliminan los tipos sin comprobarlos en absoluto. Ejecute `tsc --noEmit` como paso obligatorio en CI y en el hook previo al commit y haga fallar la compilación si su código de salida indica un error.

**Una consulta por índice devuelve undefined aunque el tipo lo descartaba.** La lectura de un elemento de una matriz o de una clave mediante una firma de índice se tipa como si el valor siempre existiera, incluso fuera de los límites de la matriz. Active `noUncheckedIndexedAccess` : añade `undefined` a esas lecturas y obliga al código a contemplarlo. Combínelo con `exactOptionalPropertyTypes` , que distingue una propiedad ausente de otra cuyo valor sea `undefined` .

**Un archivo que compila con tsc falla al ejecutarse en Node.js.** La eliminación de tipos solo retira sintaxis que carece de significado durante la ejecución. Las enumeraciones y los espacios de nombres que contienen valores detienen Node.js con `ERR_UNSUPPORTED_TYPESCRIPT_SYNTAX` ; un tipo importado sin la palabra clave `type` se interpreta como una importación de valor que falla durante la ejecución. Active `erasableSyntaxOnly` y `verbatimModuleSyntax` para que el compilador informe de ambos problemas antes que Node.js, escriba `import type` para los tipos y sustituya las enumeraciones por objetos literales marcados con `as const` .

## Lecturas adicionales

- Documentación de TypeScript — el manual, las páginas de referencia y las notas de versiones, mantenidos por el equipo de TypeScript.
- Conceptos básicos — el capítulo del manual sobre comprobación estática, emisión con errores, eliminación de tipos y rigor de las comprobaciones.
- Referencia de TSConfig: strict — cada opción del compilador y su valor predeterminado, incluidas las comprobaciones que activa strict.
- Compatibilidad de tipos — el subtipado estructural y los puntos en los que, por diseño, el sistema no garantiza la seguridad de tipos.
- Node.js: módulos: TypeScript — la eliminación de tipos, la sintaxis que rechaza y los ajustes del compilador que recomienda Node.js.
- Análisis con información de tipos — cómo typescript-eslint da a las reglas de análisis acceso al comprobador de tipos.

Glosario

Elija según sus restricciones reales

Describa el flujo de trabajo, las interfaces existentes y los usuarios que necesitan el cambio. Podemos definir una entrega comprobable y un traspaso que permita a su equipo mantener lo entregado.

**Vea cuánto cuesta** Póngase en contacto →

# Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es TypeScript?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

**Explore nuestros servicios**

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

**Creado con esmero y cariño por D.**

**CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N**

**Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.**

**Impreso desde https://engineer.company/es/glossary/typescript/**