Engineer

TypeScript

JavaScript mit vor der Ausführung geprüften Typen

Was es ist

TypeScript ist eine Programmiersprache von Microsoft, die JavaScript um Typannotationen und einen Compiler namens `tsc` ergänzt, der sie prüft. Der Name wird zu TS abgekürzt; die Dateien enden auf `.ts` oder, wenn sie JSX-Markup enthalten, auf `.tsx` . JavaScript-Syntax ist gültiges TypeScript, weshalb die Sprache als Obermenge von JavaScript bezeichnet wird. Der Compiler ist quelloffen und steht unter der Apache License 2.0.

Die Typen bestehen nur für die Dauer der Prüfung. Gewöhnliche Typannotationen werden entfernt, wenn TypeScript JavaScript ausgibt; sie validieren nichts zur Laufzeit und erfordern keine Laufzeitbibliothek zur Typprüfung. Typannotationen ändern nicht, wie sich der JavaScript-Code bei der Ausführung verhält. Dieser eine Umstand erklärt Nutzen und Grenzen zugleich: Die Typprüfung beurteilt die im Quelltext beschriebenen Typen; Daten, die während der Ausführung eintreffen, validiert sie nicht.

Das Typsystem ist strukturell. Zwei Typen sind kompatibel, wenn ihre Strukturen zusammenpassen, unabhängig von ihren Namen. Das passt zu JavaScripts üblicher Praxis, einfache Objekte weiterzugeben; C# und Java vergleichen dagegen deklarierte Namen. Viele Typen werden abgeleitet, weshalb Annotationen vor allem an Funktionssignaturen und exportierten Werten stehen. An einigen Stellen ist das System zudem bewusst nicht typsicher. `any` schaltet die Prüfung für einen Wert ab, und eine Typzusicherung mit `as` weist den Compiler an, sich auf die Typangabe der programmierenden Person zu verlassen. Beide verschwinden wie die übrigen Typangaben aus der Ausgabe.

Für die Ausführung gibt es drei gängige Wege. `tsc` prüft ein Projekt und gibt JavaScript aus. Bundler wie esbuild entfernen die Typen ungeprüft und überlassen die Prüfung einem gesonderten Lauf von `tsc --noEmit` . Node.js führt

seit v23.6.0 und v22.18.0 `.ts` -Dateien direkt aus, indem es die Typsyntax durch Leerraum ersetzt, ebenfalls ohne Typprüfung, und akzeptiert nur Syntax, die sich so entfernen lässt. TypeScript 7.0, angekündigt im Juli 2026, ist eine native Go-Version des Compilers, die nach Angaben des Teams vollständige Builds typischerweise 8- bis 12-mal schneller erledigt.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist TypeScript?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Oberflächen, die schnell laden, auf dem Smartphone funktionieren und sich in allen Browsern Ihrer Kunden gleich verhalten.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. Full-Stack-GIS-Kartenentwicklung geleitet — Full-Stack-Kartenentwicklung von der Datenbank bis zum Browser

Full-Stack-GIS-Kartenentwicklung geleitet Kontakt aufnehmen →

Warum es wichtig ist

Ein Team muss entscheiden, ob sich ein Build-Schritt und eine Typprüfung lohnen, weil sie Fehler vor der Ausführung finden. In einer Codebasis, die jede Woche von mehreren Personen geändert wird, ist das meist der Fall. Beim Umbenennen eines Feldes werden alle Verwendungsstellen sichtbar; der Editor vervollständigt und erklärt Code anhand der Typen, und die Signatur eines Moduls wird zum Vertrag, den aufrufender Code nicht unbemerkt verletzen kann. Dem stehen eine Prüfung bei jedem Build, eine Konfigurationsdatei mit vielen Optionen und Typdefinitionen gegenüber, die mit den beschriebenen Bibliotheken Schritt halten müssen.

Am wichtigsten ist die Einstellung der Strenge. Das Flag `strict` aktiviert eine Gruppe von Prüfungen, darunter `strictNullChecks` und `noImplicitAny`; das Handbuch beschreibt das Ergebnis eher als Drehregler denn als Schalter. Seit TypeScript 6.0 ist das Flag standardmäßig aktiviert, sodass ein neues Projekt mit strengen Prüfungen beginnt. In einem älteren Projekt kann es weiterhin auf false stehen; es später einzuschalten bedeutet, jede betroffene Datei durchzuarbeiten. Zwei nützliche Prüfungen gehören nicht zu dieser Gruppe und müssen gesondert aktiviert werden: `noUncheckedIndexedAccess` und `exactOptionalPropertyTypes`.

Das Entfernen der Typen setzt die Grenze. Liest ein Programm JSON aus einer Netzwerkanfrage, einer Datei, einer Warteschlange oder einem Formular, beruht die angenommene Datenstruktur allein auf der Aussage der programmierenden Person, denn beim Eintreffen der Daten sind die Typen bereits weg. An den Grenzen ist deshalb eine Validierung zur Laufzeit nötig, idealerweise aus derselben Definition erzeugt wie die Typen, damit Prüfung und Typ nicht voneinander abweichen können. TypeScript sorgt im Inneren eines Programms für Konsistenz und überlässt die Ränder dem Code.

Die Typprüfung versorgt auch andere Werkzeuge. Ein Linter wie typescript-eslint kann Typinformationen beim Compiler abfragen, sodass eine Regel ein Promise meldet, auf das niemand wartet, oder einen Wert vom Typ `any`, der sich durch ein Modul zieht. So führt ein Verstoß gegen die eigenen Konventionen zu einer Fehlermeldung statt zu einem Review-Kommentar.

Wie es in der Praxis aussieht

In den auf dieser Website gezeigten Projekten wird die Verlässlichkeit von TypeScript durch generierten Code und Werkzeuge gesichert, statt durch Konventionen. Drei Seiten zeigen, wie das geschieht.

Auf einer Plattform wird der TypeScript-Client des Frontends nie von Hand geschrieben. Er wird aus der Beschreibung der API selbst erzeugt, die Huma aus den Go-Typen ausgibt; diese folgen wiederum dem Datenbankschema und seinen Funktionen. Zum dokumentierten Implementierungsstand umfasste der Client 44.076 Zeilen. 61 gleichzeitig generierte Mock-Handler ließen die Frontend-Tests gegen den tatsächlichen Vertrag laufen statt gegen eine handgeschriebene Fixture. Jeder Übergang hat seine eigene Prüfung. Eine Vertragsprüfung in CI vergleicht, was das Frontend sendet, mit dem, was die API erwartet, und lässt den Build bei Abweichungen fehlschlagen. Darunter untersucht eine Schemaprüfung die tatsächlichen Strukturen. Erfasst werden optionale Body-Felder, bei denen fehlend und null leicht verwechselt werden, und Enums in Abfrageparametern. Ändert sich ein Feld nur auf einer Seite, scheitert nun wenige Minuten nach der Änderung ein Build, statt dass nach der Auslieferung der Abweichung ein 422 im Browser erscheint. Im Gegenzug erreicht eine Vertragsänderung die Typen des Clients nur, wenn die Kette erneut durchlaufen wird, und die erneute Generierung ist lästig.

Eine weitere Seite legt für sechs Sprachen, darunter TypeScript, denselben Maßstab an. Jede von ESLint gemeldete Diagnose ist ein Fehler. Es gibt keine Warnstufe, in der sich etwas verstecken ließe, und Inline-Unterdrückungen sind verboten: Ein Befund muss behoben statt stummgeschaltet werden. Pre-Commit- und Pre-Push-Hooks führen Linter und Tests aus; Längengrenzen für Funktionen und Dateien halten Module lesbar.

Die mit einer Sprache gelieferten Linter reichen nicht bis zu den projektspezifischen Regeln. Deshalb sicherten zum dokumentierten Implementierungsstand fünfzehn eigene ESLint-Regeln die TypeScript-Muster dieses Repositorys. Sie gehörten zu einem Prüfsystem mit 274 Prüfskripten, von denen 268 in den Commit-Hooks registriert waren. Die Prüfungen hatten selbst 146 Tests, denn eine defekte Prüfung lässt alles unbemerkt durch. Eine Regel betrifft Tests statt Typen: Ein Commit mit dem Präfix `fix:` muss einen Test enthalten, der ohne die Korrektur fehlschlägt. Die Review-Zeit verlagerte sich von formalen Fragen auf Entwurfsentscheidungen; im Gegenzug wurde das Committen langsam.

Wo es schiefgeht

Die meisten TypeScript-Fehler entstehen, wenn den Typen über das hinaus vertraut wird, was der Compiler sehen kann.

Ein als String typisiertes Feld ist zur Laufzeit undefined. Die Daten kamen von außerhalb des Programms und erhielten einen Typ, statt geprüft zu werden. Die DOM-Bibliothek typisiert `res.json()` als Promise von `any`; deshalb akzeptiert der Compiler alles, was der Code mit dem Ergebnis anstellt. Eine Typzusicherung macht das Vertrauen ausdrücklich, aber nicht sicher:

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

Externe Daten sollten als `unknown` behandelt und an der Systemgrenze zur Laufzeit anhand eines Schemas validiert werden. Der Typ sollte aus diesem Schema erzeugt werden, damit beide zusammenpassen. Eine typgestützte Lint-Regel kann dann jedes `any` melden, das dennoch durchrutscht.

Code mit Typfehlern erreicht trotzdem die Produktion. `tsc` schreibt auch dann JavaScript, wenn es Fehler meldet, weil `noEmitOnError` standardmäßig false ist. Bundler und Node.js entfernen Typen, ohne sie überhaupt zu prüfen. `tsc --noEmit` sollte als verpflichtender Schritt in CI und im Pre-Commit-Hook laufen; ein Fehler-Exitcode muss den Build scheitern lassen.

Ein Indexzugriff liefert undefined, das der Typ ausgeschlossen hatte. Das Lesen eines Array-Elements oder eines Schlüssels über eine Indexsignatur wird so typisiert, als sei der Wert immer vorhanden, selbst jenseits des Array-Endes. Wird `noUncheckedIndexedAccess` aktiviert, fügt es solchen Zugriffen `undefined` hinzu und zwingt den Code, damit umzugehen. Ergänzend sollte `exactOptionalPropertyTypes` aktiviert werden; es unterscheidet eine fehlende Eigenschaft von einer auf `undefined` gesetzten.

Eine mit tsc kompilierbare Datei scheitert bei der Ausführung durch Node.js. Das Entfernen der Typen beseitigt nur Syntax ohne Laufzeitbedeutung. Enums und Namespaces, die Werte enthalten, lassen Node.js mit `ERR_UNSUPPORTED_TYPESCRIPT_SYNTAX` abbrechen. Ein ohne das Schlüsselwort `type` importierter Typ gilt als Wertimport und scheitert zur Laufzeit. Mit `erasableSyntaxOnly` und `verbatimModuleSyntax` meldet der Compiler beides vor Node.js. Für Typen sollte `import type` verwendet werden; Enums lassen sich durch mit `as const` markierte Objektliterale ersetzen.

Weiterführende Lektüre

- TypeScript Documentation — Handbuch, Referenzseiten und Versionshinweise, gepflegt vom TypeScript-Team.
- The Basics — das Handbuchkapitel über statische Prüfung, Ausgabe trotz Fehlern, entfernte Typen und Strenge.
- TSConfig Reference: strict — jede Compileroption mit ihrem Standardwert, einschließlich der durch strict aktivierten Prüfungen.
- Type Compatibility — strukturelle Subtypisierung und die Stellen, an denen das Typsystem bewusst nicht typsicher ist.
- Node.js: Modules: TypeScript — das Entfernen von Typen, die dabei abgewiesene Syntax und die von Node.js empfohlenen Compilereinstellungen.
- Linting with Type Information — wie typescript-eslint Lint-Regeln Zugang zur Typprüfung gibt.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist TypeScript?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/typescript/