Engineer

TypeScript

JavaScript avec types vérifiés avant l'exécution

Ce que c’est

TypeScript est un langage de programmation de Microsoft qui ajoute des annotations de type à JavaScript, avec un compilateur, `tsc`, qui les vérifie. On abrège son nom en TS, et ses fichiers se terminent par `.ts`, ou `.tsx` lorsqu’ils contiennent du balisage JSX. La syntaxe JavaScript est valide en TypeScript, ce qui lui vaut d’être appelé un surensemble de JavaScript. Le compilateur est open source sous licence Apache 2.0.

Les types ne durent que le temps de la vérification. Les annotations de type ordinaires sont effacées lorsque TypeScript produit le JavaScript ; elles ne valident pas les données à l’exécution et ne nécessitent pas de bibliothèque de vérification des types à l’exécution. Les annotations de type ne changent pas le comportement du code JavaScript exécuté. Ce fait explique à la fois leur intérêt et leurs limites : le vérificateur raisonne sur les types décrits dans le code source ; il ne valide pas les données qui arrivent pendant l’exécution du programme.

Le système de types est structurel. Deux types sont compatibles lorsque leurs structures concordent, quel que soit leur nom, ce qui convient à l’habitude de JavaScript de faire circuler des objets ordinaires ; C# et Java comparent plutôt les noms déclarés. Une grande partie des types est déduite, si bien que les annotations figurent surtout dans les signatures de fonction et les valeurs exportées. Le système renonce aussi délibérément à la sûreté des types dans quelques cas. `any` désactive la vérification d’une valeur, et une assertion écrite avec `as` demande au compilateur de croire le programmeur sur le type. Les deux disparaissent du résultat produit, comme le reste des annotations.

Il existe trois façons courantes de l’exécuter. `tsc` vérifie un projet et produit du JavaScript. Les outils d’assemblage de modules comme esbuild retirent les types sans les vérifier et confient cette vérification à une exécution séparée de `tsc --`

`noEmit` . Depuis les versions v23.6.0 et v22.18.0, Node.js exécute directement les fichiers `.ts` en remplaçant la syntaxe des types par des espaces, sans vérifier les types non plus, et n'accepte que la syntaxe qui peut être effacée ainsi. TypeScript 7.0, annoncé en juillet 2026, est une version native du compilateur écrite en Go ; l'équipe annonce des compilations complètes généralement 8 à 12 fois plus rapides.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que TypeScript ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Des interfaces rapides, utilisables sur téléphone et cohérentes dans tous les navigateurs de vos clients.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Développement cartographique GIS full-stack — cartographie full-stack, de la base au navigateur

Développement cartographique GIS full-stack Contactez-nous →

Pourquoi c'est important

L'équipe doit décider si une étape de compilation et un vérificateur justifient leur coût en détectant les erreurs avant l'exécution du code. Dans une base de code que plusieurs personnes modifient chaque semaine, c'est généralement le cas. Renommer un champ fait apparaître tous les endroits qui l'utilisent, l'éditeur complète et explique le code à partir des types, et la signature d'un module devient un contrat que ses appelants ne peuvent pas enfreindre discrètement. En contrepartie, chaque compilation comprend une vérification, le fichier de configuration propose de nombreuses options et les définitions de type doivent suivre l'évolution des bibliothèques qu'elles décrivent.

Le niveau de rigueur est le réglage le plus important. L'option `strict` active une famille de vérifications, dont `strictNullChecks` et `noImplicitAny` , et le manuel présente le résultat comme un réglage progressif plutôt qu'un interrupteur. Depuis TypeScript 6.0, cette option est activée par défaut : un nouveau projet commence donc en mode strict ; un ancien peut encore la définir à false, et l'activer ensuite impose de reprendre chaque fichier concerné. Deux vérifications utiles se trouvent en dehors de cette famille et doivent être activées séparément : `noUncheckedIndexedAccess` et `exactOptionalPropertyTypes` .

L'effacement fixe la limite. Un programme qui lit du JSON depuis une requête réseau, un fichier, une file d'attente ou un formulaire n'a que la parole du programmeur concernant la structure de ces données : les types ont disparu

lorsqu'elles arrivent. Les frontières du programme exigent donc une validation à l'exécution, idéalement générée à partir de la même définition que les types, pour que la vérification et le type ne puissent pas diverger. TypeScript assure la cohérence interne d'un programme et laisse à du code le soin de contrôler ses frontières.

Le vérificateur alimente aussi d'autres outils. Un linter comme typescript-eslint peut demander des informations de type au compilateur : une règle peut alors signaler une promesse que personne n'attend ou une valeur de type `any` qui se propage dans un module. Une équipe transforme ainsi ses conventions en erreurs plutôt qu'en commentaires de revue.

À quoi cela ressemble en pratique

Dans les projets présentés sur ce site, la fiabilité de TypeScript repose sur le code généré et les outils plutôt que sur de simples conventions. Trois pages montrent comment.

Sur une plateforme, le client TypeScript du frontend n'est jamais écrit à la main. Il est généré à partir de la description de l'API elle-même, que Huma produit à partir des types Go, eux-mêmes issus du schéma de la base de données et de ses fonctions. Au stade de mise en œuvre documenté, le client comptait 44 076 lignes, et 61 gestionnaires simulés générés à ses côtés permettaient aux tests du frontend de s'appuyer sur le véritable contrat plutôt que sur un jeu de données de test écrit à la main. Chaque maillon a son propre garde-fou. Une vérification du contrat dans la CI compare ce qu'envoie le frontend à ce qu'attend l'API et fait échouer la compilation en cas de divergence, avec une sonde du schéma sous-jacent qui examine les structures réelles. Elle couvre les champs facultatifs du corps, où l'absence d'une valeur et null se confondent facilement, ainsi que les énumérations des paramètres de requête. Un champ qui change d'un seul côté fait désormais échouer une compilation quelques minutes après la modification, au lieu de produire une erreur 422 dans le navigateur après la livraison de l'incohérence. En contrepartie, une modification du contrat n'atteint les types du client qu'en relançant la chaîne, et la régénération est fastidieuse.

Une autre page fixe une même exigence pour six langages, dont TypeScript. Chaque diagnostic signalé par ESLint est une erreur : il n'existe aucun niveau d'avertissement où se réfugier, et les suppressions locales de diagnostic sont

interdites : un problème doit donc être corrigé plutôt que masqué. Les hooks de pré-commit et de pré-push exécutent les linters et les tests, et les limites de longueur des fonctions et des fichiers gardent les modules lisibles.

Les linters fournis avec un langage s'arrêtent avant les règles propres à un projet : quinze règles ESLint spécifiques maintenaient donc les conventions TypeScript de ce dépôt au stade de mise en œuvre documenté. Elles appartenaient à un moteur de garde-fous de 274 scripts de vérification, dont 268 enregistrés dans les hooks de commit, et ces garde-fous disposaient de 146 tests qui leur étaient propres : un garde-fou défaillant laisse tout passer sans que personne le remarque. Une règle concerne les tests plutôt que les types : un commit préfixé par `fix:` doit inclure un test qui échoue sans la correction. Le temps de revue s'est déplacé de la mécanique vers la conception ; en contrepartie, les commits sont devenus lents.

Où les problèmes surviennent

La plupart des problèmes rencontrés avec TypeScript viennent d'une confiance accordée aux types au-delà de ce que le compilateur peut vérifier.

Un champ déclaré comme chaîne se révèle undefined à l'exécution. Les données viennent de l'extérieur du programme et ont reçu un type au lieu d'être vérifiées. La bibliothèque de types du DOM déclare `res.json()` comme une promesse de `any` : le compilateur accepte donc tout ce que le code fait du résultat, et une assertion rend cette confiance explicite sans la rendre sûre :

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

Traitez les données externes comme `unknown`, validez-les à la frontière avec un schéma utilisé à l'exécution et générez le type à partir de ce schéma pour que les deux restent synchronisés. Une règle de lint fondée sur les types peut ensuite signaler tout `any` qui échappe à ce contrôle.

Du code contenant des erreurs de type arrive tout de même en production. `tsc` écrit encore du JavaScript lorsqu'il signale des erreurs, car `noEmitOnError` vaut false par défaut, tandis que les outils d'assemblage de modules et Node.js

retirent les types sans aucune vérification. Imposez l'exécution de `tsc --noEmit` dans la CI et le hook de pré-commit, et faites échouer la compilation selon son code de sortie.

Un accès par indice renvoie undefined alors que le type l'excluait. La lecture d'un élément de tableau ou d'une clé à travers une signature d'index est typée comme toujours présente, même au-delà de la fin du tableau. Activez `noUncheckedIndexedAccess`, qui ajoute `undefined` à ces lectures et oblige le code à le traiter. Associez-lui `exactOptionalPropertyTypes`, qui distingue une propriété absente d'une propriété définie à `undefined`.

Un fichier qui compile avec tsc échoue lorsque Node.js l'exécute. Le retrait des types n'enlève que la syntaxe dépourvue de signification à l'exécution. Les énumérations et les espaces de noms contenant des valeurs arrêtent Node.js avec `ERR_UNSUPPORTED_TYPESCRIPT_SYNTAX`, et un type importé sans le mot-clé `type` est traité comme une importation de valeur qui échoue à l'exécution. Activez `erasableSyntaxOnly` et `verbatimModuleSyntax` pour que le compilateur signale les deux problèmes avant Node.js, écrivez `import type` pour les types et remplacez les énumérations par des objets littéraux marqués `as const`.

Pour aller plus loin

- Documentation TypeScript — le manuel, les pages de référence et les notes de version, maintenus par l'équipe TypeScript.
- The Basics — le chapitre du manuel sur la vérification statique, la génération malgré les erreurs, les types effacés et la rigueur.
- Référence TSConfig : strict — chaque option du compilateur avec sa valeur par défaut, dont les vérifications activées par strict.
- Compatibilité des types — le sous-typage structurel et les cas où le système renonce délibérément à la sûreté des types.
- Node.js : Modules : TypeScript — le retrait des types, la syntaxe refusée et les réglages du compilateur recommandés par Node.js.
- Lint fondé sur les informations de type — comment typescript-eslint donne aux règles de lint accès au vérificateur de types.



Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que TypeScript ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

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

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

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

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/typescript/