Odisee Gent

Verslag Typescript

Persoonlijk ontwikkelingsplan

Raf Vanpuyvelde
10-8-2020

Contents

Wat is Typescript

De officiële documentatie luidt als volgt:

"Typescript is a typed superset of Javascript that compiles to plain javascript." [1]

- typescriptlang.org

Om deze stelling beter te begrijpen kan deze best opgedeeld worden in twee delen. Een 'typed superset' en 'compiles to plain javascript". Met typed superset wordt er bedoeld dat de standaard functionaliteiten van javascript overgenomen en versterkt worden. In dit geval is TS een syntactische superset van JS. Compiles to plain javascript wijst dan weer op de Typescript compiler zelf (tsc) Typescript gaat vertalen in vanilla JS.

Wat zijn de voordelen

[2], [3], [4], [5]

- Static typing (live type checking)
- Type definitions
- Hogere readability door gebruik van types
- Goed voor grote applicaties die werken met javascript
    - Goede tooling => code intelligentie -> hogere productiviteit
    - Moderne features van js gebruiken op oude browser door transpolation (superset van js)
    - Zeer schaalbaar
- Werken met types zorgt ervoor dat fouten preventief gevonden kunnen worden.
- Tooling zorgt voor betere refactoring
- Sneller werken door autocompletion (imports, ...)
- Mogelijk om geleidelijk aan typescript te implementeren

Wat zijn de nadelen

[2], [3], [4], [5]

- Initieel meer werk
- Leercurve

De onderdelen

Basic Types

Javascript heeft een aantal datatypes waaronder (maar niet beperkt tot) numbers, string, objects. Deze types zijn dynamisch. Met andere woorden een variabele kan (her)gebruikt worden om meerdere datatypes te bevatten.

Typescript stelt uiteraard dezelfde datatypes beschikbaar als Javascript (aangezien Typescript een superset is van Javascript). Het verschil tussen de twee zit hem in de mogelijkheid om variabelen sterk te typeren binnen Typescript.

Sterk getypeerde variabelen binnen typescript

```
TS: let isDone: boolean = false;
```

```
JS: let isDone = false;
```

Buiten de standaard beschikbare datatypes beschikt TS ook over tuples en enum's. Het any type kan gebruikt worden voor variabelen waarvan het type (nog) niet gekend is.

Tuples

```
// Declare a tuple type
let someTuple: [string, number];
// Initialize it
someTuple = ["test", 10]; // OK
someTuple = [10, "test"]; // Error
```

Enums

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

Type assertions zijn een functionaliteit binnen TS die overeenkomen met type casts uit bijvoorbeeld C#.

Type assertions

```
let someValue: any = "this is a string";
let strLength: number = (<string>someValue).length;
// Of met andere syntax
let strLength: number = (someValue as string).length;
```

Variable Declarations

Aangezien het keyword 'var' kan leiden tot verwarring, door onverwacht gedrag binnen een bepaalde scope van de code. Maakt men binnen TS gebruik van het keyword **'let'**. De functionaliteit van dit keyword is uiteraard dezelfde als in javascript. Indien een variabele niet meer mag aangepast worden na de initialisatie ervan dient men **'const'** te gebruiken. (Voor properties die niet veranderd mogen worden gebruikt men **'readonly'**).

Interfaces

Een van de kern functionaliteiten van TS zijn interfaces. Een interface vormt als het ware een contract tussen de interface zelf en de implementatie ervan. Dit wil zeggen dat de vorm van de interface gerespecteerd moet worden. Een implementatie van de interface moet dus de properties van die

interface bevatten. Een uitzondering hierop zijn de optionele properties, deze kunnen we herkennen door een '?'.

Interface met optionele properties

```
interface SquareConfig {
  color?: string; // '?' Wijst op optionele property
  width?: number;
}
function createSquare(config: SquareConfig): {color: string; area: number} {
  implementatie van SquareConfig
}
let mySquare = createSquare({color: "black"}); // Width is optioneel
```

Interfaces kunnen niet alleen objecten omschrijven maar ook functies. Ook indexeerbare types zoals arrays kunnen gebruik maken van interfaces.

Interface voor een functie

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
  implementatie van SearchFunc
}
```

Het grote voordeel van interfaces, is dat deze net zoals in talen zoals C# of Java kunnen gebruikt worden in combinatie met klassen.

Interface voor een klasse

```
interface ClockInterface {
  currentTime: Date;
}
class Clock implements ClockInterface {
  currentTime: Date;
  constructor(h: number, m: number) { }
}
```

Nog een voordeel aan interfaces is het feit dat deze geëxtend kunnen worden door andere interfaces. Zo kan men de functionaliteit van een interface vergroten zonder de originele interface aan te passen.

Interface extenden door een andere interface

```
interface Shape {
  color: string;
}
interface Square extends Shape {
  sideLength: number;
}
let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

Classes

Javascript maakt pas sinds ECMAScript 6 gebruik van klasses. Typescript zorgt ervoor dat programma's geschreven in oudere versies ook al deze klasses kunnen gebruiken. Deze klasses zijn te vergelijken met die uit talen zoals C# of Java. Klasses bestaan altijd uit drie delen: properties, een constructor en methods.

Ook in TS is het mogelijk klasses te laten erven van andere klasses. Dit kan men bekomen met het keyword 'extends'. Net zoals in andere talen kunnen de members van een klasse private, public of protected zijn. Een protected member heeft ongeveer hetzelfde gedrag als een private member. Het enige verschil is dat een protected member wel kan geraadpleegd worden vanuit een extended klasse.

Basis klasse syntax in Typescript

```
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return "Hello, " + this.greeting;
  }
}
let greeter = new Greeter("world");
```

De klasses binnen TS maken ook gebruik van getters en setters (accessors). Deze kunnen net zoals in bijvoorbeeld een taal zoals C# gebruikt worden, om te interageren met members van een klasse. De syntax van deze getters en setters is dezelfde als in Javascript.

Accessors in Typescript

```
class Employee {
  private _fullName: string;
  get fullName(): string {
    return this._fullName;
  }
  set fullName(newName: string) {
    this._fullName = newName;
  }
}
let employee = new Employee();
employee.fullName = "Bob Smith";
```

Ook het gebruik van abstracte klasses is toegestaan in Typescript. Dit is standaard niet het geval in JS. Een abstracte klasse zorgt ervoor dat net zoals een interface er duidelijke afspraken zijn omtrent de signatuur van de implementatie. Het verschil zit hem in de mogelijkheid, om ook eventuele methodes te beschrijven binnen de abstracte klasse. Methods die niet bestaan in de abstracte klasse, maar wel in de niet abstracte klasse worden als fout beschouwd.

Generics

Binnen Javascript worden generics niet gebruikt, Typescript voorziet deze functionaliteit echter wel. Het doel van generics, is om componenten te maken die in te zetten zijn voor een meer dan één type.

Het voordeel van generics ten opzichte van het 'any' type waarover TS beschikt is dat het oorspronkelijke type niet verloren gaat.

Generics

```
function identity<T>(arg: T): T {
  return arg; // Arg heeft als type T, type info gaat niet verloren
}
```

Ook is het mogelijk de beschikbare types van een generic te limiteren, door de generic een interface te laten implementeren die properties van het gewenste type beschrijft.

Generics met interface

```
interface GenericIdentityFn<T> {
  (arg: T): T;
}
function identity<T>(arg: T): T {
  return arg;
}
let myIdentity: GenericIdentityFn<number> = identity;
```

Enums

Typescript maakt gebruik van enums om een set van veelgebruikte constanten te definiëren. Deze functionaliteit zit standaard niet in Javascript. Deze enums kunnen zowel numerisch zijn en/of strings bevatten. TS splits de members van een enum op in twee verschillende kampen: namelijk de constante en computed members. Om constante member te zijn moet deze aan een aantal voorwaarden voldoen.

- String of numerische literal
- Referentie naar eerder gedeclareerde enum member
- Enum expressie binnen quotes
- Expressie die +, -, ~ operators bevat
- Binaire operators (+, -, *, /, %, <<, >>, >>>, &, |, ^) gebruikt in een constante enum expressie.

In elk ander geval is een enum waarde computed.

Enum binnen typescript

```
enum Direction {
  Up = 1, // Startwaarde = 1
  Down,
  Left,
  Right,
}
```

Constante vs Computed enum members

```
enum FileAccess {
  // constant members
  None,
  Read  = 1 << 1,
  Write   = 1 << 2,
  ReadWrite  = Read | Write,
```

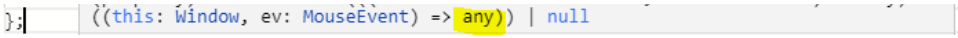
```
  // computed member
  G = "123".length
}
```

Type Inference

Type inference is een begrip dat zowel binnen JS als TS gebruikt wordt. Dit slaat simpelweg op de manier dat variabelen hun type wordt afgeleid. We kunnen deze opsplitsen in twee soorten.

1) Best common type: Het algoritme bepaalt het meest toepasselijke type aan de hand van de andere types binnen de context.
    a. `let x = [0, 1, null]; // Int komt meer voor`
2) Contextual type: Het type van een expressie hangt af van zijn locatie.

Contextual type TS




```
// MouseEvent heeft impliciet het 'any' type door de context van 'onmousedown'.
window.onmousedown = function(mouseEvent: any) {
  console.log(mouseEvent.button);
};
```

Type Compatibility

Typescript maakt gebruikt van 'nominal typing' ipv 'structural typing' [7] zoals bv Java of C#. Deze term bepaalt hoe een taal beslist dat twee types dezelfde zijn. Wanneer men bijvoorbeeld een klasse Foo heeft, met een bepaalde signatuur (dezelfde members) die dezelfde is als de klasse Bar, dan kan men in een structureel getypeerd systeem zeggen dat deze twee types dezelfde zijn. In een nomimaal getypeerd systeem zouden de klasses Foo en Bar niet dezelfde zijn ondanks dezelfde signatuur.

Nominal typing vs Structural typing

```
class Foo { method(input: string) { /* ... */ } }
class Bar { method(input: string) { /* ... */ } }
let foo: Foo = new Bar(); // Werkt door nominal typing
```

Aangezien functies binnen TS ook sterk getypeerd zijn, moeten deze ook vergeleken kunnen worden. Bij functies is dit iets complexer. Elke parameter van de secundaire functie b (met a in a = b als primaire functie) moet beschikbaar zijn en hetzelfde type hebben als de overeenkomstige parameter uit functie a. Hetzelfde geldt voor de parameter(s) van het return type van de functie.

Type compatibility functies

```
let x = (a: number) => 0;
```

```
let y = (b: number, s: string) => 0;

y = x; // OK

x = y; // Error, door ontbrekende parameter

let x = () => ({name: "Alice"});

let y = () => ({name: "Alice", location: "Seattle"});

x = y; // OK

y = x; // Error, x returned geen location property
```

Bij het vergelijken van twee klasses wordt er enkel gekeken naar de members van deze klasses. Een uitzondering hierop, is de constructor en eventueel statische members van deze klasses.

Advanced Types

Intersection types

Eén van de geavanceerdere types binnen Typescript zijn 'intersection'[8] types. Deze zijn een samenvoeging van meerdere types. Deze samenvoeging is vervolgens een type op zich die alle members van de oorspronkelijke types bevat. Deze types worden hoofdzakelijk gebruikt voor mixins en andere speciale gevallen.

Intersection types

```
interface IStudent {
    id: string;
    age: number;
}
interface IWorker {
    companyId: string;
}
type A = IStudent & IWorker;
let x: A;
x.age = 5;
x.companyId = 'CID5241';
x.id = 'ID3241';
```

Union types

Een 'union' type lijkt op een intersection type, deze zijn echter niet hetzelfde. Een union type beschrijft een waarde, die een type kan zijn uit een lijst van voor gedefinieerde types. Dit zorgt ervoor dat men

het ‘any’ type niet moet gebruiken en dus enkel fouten kan krijgen tijdens compilatie en niet run-time. Nog een verschil ten opzichte van intersection types, is dat de beschikbare members van het union type enkel de overeenkomstige members van de oorspronkelijke types zijn (i.p.v. alle members).

Union types

```
interface Bird {
  fly();
  layEggs();
}
interface Fish {
  swim();
  layEggs();
}
function getSmallPet(): Fish | Bird {}
let pet = getSmallPet();
pet.layEggs(); // Ok
pet.swim();  // Error, Enkel ‘layEggs’ is gezamenlijk beschikbaar
```

Type guards

Een typeguard is een expressie binnen TS die een runtime check uitvoert om een type binnen een bepaalde scope te garanderen. De twee meest voorkomende type guards zijn ‘typeof’ en ‘instanceof’.

Type guards

```
function isFish(pet: Fish | Bird): pet is Fish {
  return (<Fish>pet).swim !== undefined;
}
if (isFish(pet))
  pet.swim();
else
  pet.fly();
```

Symbols

Een symbol binnen Typescript is een primitief type, deze worden gebruikt als unieke keys. Hierdoor kunnen properties die geset worden door een symbol niet overschreven worden aangezien het symbol altijd uniek is.

Iterators and Generators

Er kan over een object geïtereerd worden als dat object de 'Symbol.iterator' property implementeerd. Deze property maakt gebruik van het Symbol type aangezien elke index binnen een itereerbaar object uniek moet zijn. TS beschikt over een aantal ingebouwde types die reeds beschikken over deze implementatie (Array, Map, Set, String, Int32Array, Uint32Array).

Typescript beschikt over twee speciale for-statements, namelijk de 'for..of' en de 'for..in' statement. De eerste itereert over de keys van de gegeven lijst, de andere over de values.

For.. of vs For.. in statements

```
let pets = new Set(["Cat", "Dog", "Hamster"]);

pets["species"] = "mammals";

for (let pet in pets) {

   console.log(pet); // "species"

}

for (let pet of pets) {

  console.log(pet); // "Cat", "Dog", "Hamster"

}
```

Namespaces & Modules

Typescript maakt net zoals javascript gebruik van modules. Deze zorgen ervoor dat onze applicatie overzichtelijk blijft door de afscheiding van code. Binnen TS kan men zowel met een import als een referentie een module gebruiken. De import-statement is echter de betere optie.

Modules kunnen zowel geïmporteerd als geëxporteerd worden. Een geëxporteerde module kan gemakkelijk hernoemd worden, door de 'export … as …' syntax. Elke module kan optioneel maximaal één default export bevatten. Deze kunnen ook gewoon waarden bevatten.

Ambient modules zijn modules, die een verzameling van meerdere modules bevatten. Hierdoor bestaat de mogelijkheid enkel de verzameling de moeten refereren binnen een bestand. Ambient modules maken gebruik van de '.d.ts' extensie.

Net zoals JS maakt TS gebruik van namespaces, een namespace is een regio die de scope van onderdelen binnen die namespace afbakent. Dit om naam gerelateerde conflicten te vermijden.

Door gebruik te maken van reference tags binnen ts bestanden, kan men binnen typescript de output van meerdere files concateneren tot een gemeenschappelijke output file met de correcte referenties. Men kan natuurlijk ook nog steeds de gegenereerde JS-files apart laden binnen de applicatie.

Module Resolution

Om een module te kunnen importeren, moet de compiler weten waar deze zich bevindt. Een module kan relatief en niet relatief geïmporteerd worden. Een relatieve import begint met '/', './' of '../'. Een niet relatieve import niet.

Er bestaan twee systemen, om een module op te sporen binnen Typescript. Namelijk ‘Node’ en ‘Classic’ (standaard optie). Node is een nabootsing van de Node.js manier van module resolutie.

Declaration Merging

Declaration merging is een principe waarbij twee declaraties met dezelfde naam worden samengevoegd tot een declaratie met diezelfde naam. Een declaratie binnen Typescript kan een namespace, type of value zijn. Klassen kunnen momenteel niet gemerged worden met andere klassen binnen Typescript. Hiervoor kunnen we mixins gebruiken.

Declaration Type	Namespace	Type	Value
Namespace	X		X
Class		X	X
Enum		X	X
Interface		X	
Type Alias		X	
Function			X
Variable			X

Figuur 1 Declaration merging [6]

Merging interfaces

Interfaces met dezelfde naam kunnen gemerged worden. De volgorde van de interface hangt af van een tweetal factoren. Functies binnen de interface, worden geordend van onder naar boven. De laatste interface heeft de hoogste prioriteit. In het geval dat het type een string literal is, veranderd de regel. String literals hebben namelijk voorrang. Deze worden vervolgens ook geordend van onder naar boven.

Merging interfaces Typescript

```
interface Box {
  height: number;
  width: number;
}
interface Box {
  scale: number;
}
let box: Box = {height: 5, width: 6, scale: 10};
```

Merging namespaces

Namespaces kunnen net zoals interfaces gemerged worden. Enkel geëxporteerde members van een namespace kunnen in de merge aangehaald worden. Namespaces kunnen mergen met andere namespaces maar ook met classes, functies en enums.

Merging namespaces Typescript

```
namespace Animals {
  export class Zebra { }
}
namespace Animals {
  export interface Legged { numberOfLegs: number; }
  export class Dog { }
}
// Resultaat
namespace Animals {
  export interface Legged { numberOfLegs: number; }
  export class Zebra { }
  export class Dog { }
}
```

JSX

Typescript heeft ook de mogelijkheid de XML-achtige syntax van JSX te gebruiken. Vooral de volgende onderdelen zijn belangrijk.

- Type asserties moeten de ‘as’ syntax gebruiken ipv de ‘<type>naam’ syntax.
- Type checking binnen JSX beschikt over twee soorten elementen
    - Intrinsic-elements (begint met lowercase letter)
        - Elementen eigen aan de dom (div, span, …) of eigen geregistreerde componenten.
    - Value-based elements (begint met uppercase letter)
        - Elementen die beschikbaar zijn in de scope.
- Om intrinsic-elementen te kunnen typechecken moeten deze geregistreerd zijn in de ‘JSX.IntrinsicElements’ interface.
-

Decorators

Een decorator[9] is een soort declaratie die kan toegevoegd worden aan een klasse declaratie, methodes, accessors, properties en parameters. Decorators worden aangeduid door een ‘@’ voor de naam van de expressie. Decorators worden best ondergebracht in een decorator factory, op deze manier is er volledige controle over de decorator en de mogelijkheid parameters mee te geven aan de factory.

Class decorators

Een klasse decorator wordt toegepast op de constructor van een klasse. Deze kan gebruikt worden om de klasse definitie aan te passen, te observeren of te vervangen.

Decorators in action

```
@Frozen // Decorator

class IceCream {}

function Frozen(constructor: Function) {

  Object.freeze(constructor);

  Object.freeze(constructor.prototype);

}

class FroYo extends IceCream {} // Error, constructor is frozen
```

Mixins

Een mixin binnen TS is een manier om klassen op te bouwen aan de hand van meerdere herbruikbare deelklassen. Mixins kunnen geïmplementeerd worden, net zoals interfaces, doormiddel van het 'implements' keyword. Hierdoor worden enkele de types van de deelklassen gebruikt en niet de implementatie die erachter zit. Om te voldoen aan de signatuur van de deelklassen, moeten er wel de nodige members (stand-in properties en types) worden toegevoegd aan de klasse.

Triple-Slash Directives

Deze bevatten een enkele XML-tag, deze geven informatie door aan de compiler. Triple-slash directives zijn enkel geldig aan het begin van een bestand.

De belangrijkste:

- /// <reference path="…" />
    - Duidt dependencies aan tussen bestanden
- /// <reference types="…" />
    - Duidt dependencies aan tussen packages
- /// <reference lib="…" />
    - Gebruikt om een 'lib' file te includeren
- /// <reference no-default-lib="true" />
    - Deze referentie duidt de huidige lib file als een default lib file

Type Checking JavaScript Files

Binnen typescript is het mogelijk, zowel TS als JS-bestanden te type checken. Wanneer types niet automatisch afgeleid kunnen worden uit de context, kan er gebruik gemaakt worden van JSDoc. Dit werkt ongeveer op dezelfde manier als type annotaties in typescript bestanden.

Utility Types

Utility types zijn veelvoorkomende types die Typescript beschikbaar stelt. Deze kunnen gebruikt worden om types te transformeren.

- Partial<T>
    - Maakt een type waarvan alle properties van het meegegeven type optioneel zijn.
- Readonly<T>
    - Maakt een type waarvan alle properties van het meegegeven type read-only zijn.
- Record<K,T>
    - Maakt een type met een set properties K van het type T. Kan gebruikt worden, om properties van een type te mappen op een ander type.
- Pick<T,K>
    - Maakt een type die van het type T een set properties K gebruikt.
- Omit<T,K>
    - Maakt een type met alle properties van T zonder property K
- Exclude<T,U>
    - Maakt een type van alle properties van T die niet kunnen toegewezen worden aan U
- Extract<T,U>
    - Omgekeerde van Exclude
- NonNullable<T>
    - Maakt een type waarbij 'null' en 'undefined' geen onderdeel van T mogen zijn.
- ReturnType<T>
    - Maakt een type waarvan het return type het return type van T is
- InstanceType<T>
    - Maakt een type van het type van de instantie T
- Required<T>
    - Maakt een type waarvan alle properties van het meegegeven type required zijn.
- ThisType<T>
    - Geeft geen type terug maar wordt gebruikt als aanduiding voor het contextuele 'this' type.

Bronnen

[1] "Typescript, JavaScript that scales" (-), typescriptlang.org, -, [online]. Beschikbaar: https://www.typescriptlang.org. [Geraadpleegd op 10/01/2020].

[2] Ekaterina Novoseltseva, " Top Typescript Advantages ", apiumhub.com, -, [online]. Beschikbaar: https://apiumhub.com/tech-blog-barcelona/top-typescript-advantages/. [Geraadpleegd op 10/01/2020].

[3] Microsoft Developer, "#FiveThings Why TypeScript is for You", youtube.com, -, [online]. Beschikbaar: https://www.youtube.com/watch?v=wYgSiFaYSSo. [Geraadpleegd op 10/01/2020].

[4] "Benefits of TypeScript" (-), ionicframework.com, -, [online]. Beschikbaar: https://ionicframework.com/docs/v3/developer-resources/typescript/. [Geraadpleegd op 10/01/2020].

[5] Ben Awad, “Typescript vs Javascript", youtube.com, -, [online]. Beschikbaar: https://www.youtube.com/watch?v=D6or2gdrHRE. [Geraadpleegd op 10/01/2020].

[6] "Typescript handbook " (-), typescriptlang.org, -, [online]. Beschikbaar: https://devdocs.io/typescript/handbook. [Geraadpleegd op 10/01/2020].

[7] "Nominal & Structural Typing Differences between nominal and structural typing" (-), flow.org, -, [online]. Beschikbaar: https://flow.org/en/docs/lang/nominal-structural/. [Geraadpleegd op 10/01/2020].

[8] Ibrahim Šuta, “Intersection types in TypeScript", codingblast.com, -, [online]. Beschikbaar: https://codingblast.com/typescript-intersection-types/. [Geraadpleegd op 10/01/2020].

[9] Jeff Delaney, “TypeScript Decorators by Example", fireship.io, -, [online]. Beschikbaar: https://fireship.io/lessons/ts-decorators-by-example/. [Geraadpleegd op 10/01/2020].