NILS HARTMANN | https://nilshartmann.net

Let's type!

Ein praktische Einführung in TypeScript

Slides: http://bit.ly/voxxed-vienna-typescript

NILS HARTMANN

Programmierer aus Hamburg

Java
JavaScript, TypeScript, React

Trainings, Workshops

nils@nilshartmann.net

"TypeScript is probably the most important language right now (...)

TypeScript makes JavaScript twice as good, and that's a conservative estimate (...)

In terms of impact, TypeScript is the most important thing right now possibly."

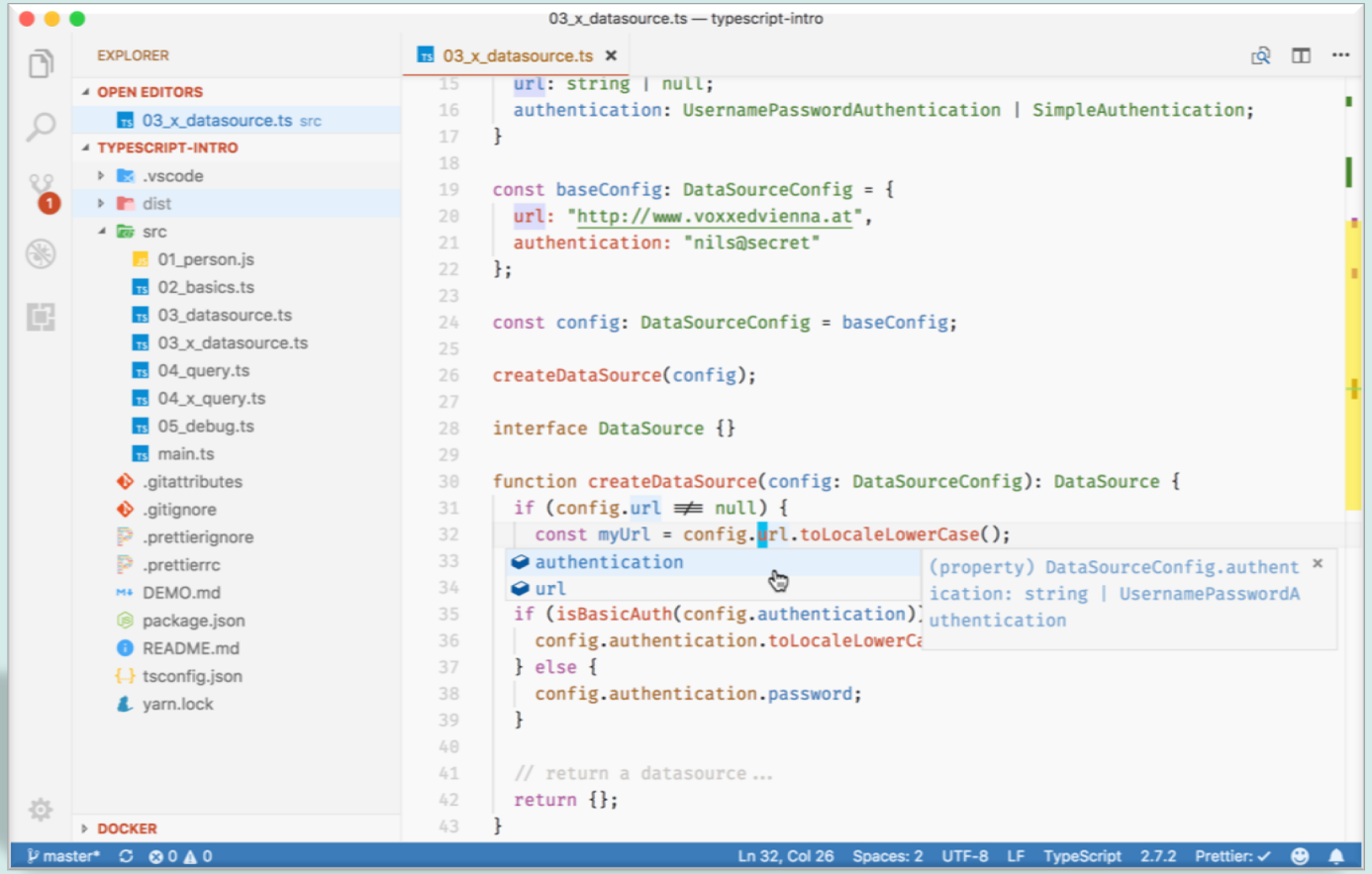
-- Rod Johnson (Creator of Spring Framework)
(https://thenewstack.io/spring-rod-johnson-enterprise-java/)

TypeScript

TYPESCRIPT AT A GLANCE

TypeScript: Obermenge von JavaScript

- Jeder gültige JavaScript Code ist gülter TypeScript Code (theoretisch...)
- Ergänzt JS um Typ-System, Sichtbarkeiten, Enums und Dekoratoren
- Compiler erzeugt aus TypeScript-Code JavaScript (ES3, ES5, ES6)-Code
- Entwickelt von Microsoft
  - http://www.typescriptlang.org/



Praktische Einführung!

https://github.com/nilshartmann/typescript-intro/

Syntax

TYPE ANNOTATIONS

Typen verwenden

Variablen

```
let foo: string; // built-in types, for example: string, number, boolean
```

TYPE ANNOTATIONS

Typen verwenden

Variablen

```
let foo: string; // built-in types, for example: string, number, boolean
```

Funktionen

```
function sayIt(what: string) {
   return `Saying: ${what}`;
}
```

TYPE ANNOTATIONS

Using Types

Variables

```
let foo: string; // built-in types, for example: string, number, boolean
```

Functions

```
function sayIt(what: string) {
   return `Saying: ${what}`;
}
```

Typ Angaben sind optional, Typ wird von TS abgeleitet

```
let result = 7; // abgeleiteter Typ: number
result = sayIt('Lars') // Fehler! (abgeleiteter Typ von sayIt: string)
```

TYPE ANNOTATIONS

any Type

```
let foo: any; // erlaubt alle Typen, kein Typ-Checking findet mehr statt

foo = "Klaus"; // OK
foo = 7; // OK
foo = null; // OK
```

TypeScript weist any implizit immer dann zu, wenn TS keinen Typ bestimmen kann:

```
function sayWhat(s) {
  // s ist any
}
```

Im "strict mode" weist TypeScript nie "any" zu, stattdessen gibt es einen Fehler (man kann aber selber any verwenden)

UNION TYPES

Ein *Union Type* zeigt an, das ein Wert *verschiedene Typen* haben

```
let foo: string | number;

foo = 7; // OK
foo = "Seven"; // auch OK

foo = false; // Fehler
```

NULL UND UNDEFINED

null und *undefined* sind eigene Typen in TypeScript

- Typen sind nicht nullable und können nicht undefined sein (mit "strictNullChecks")

```
let a: string = "Klaus";
a = null; // Error
```

Mit Union type können wir null zulassen:

```
let a: string | null = "Klaus;
a = null; // OK
```

Gleiches gilt für undefined:

```
let a: string | undefined;
```

Empfehlung: bei neuen Projekten "strictNullChecks" einschalten!

String Literal Type

- Mit dem "String Literal Type" kann definiert werden, welche Werte ein String annehmen kann

```
type Language = "Java" | "TypeScript";  // Java oder TypeScript

const java:Language = "Java"; // OK
const cpp:Language = "C++"; // FEHLER
```

Eigene Typen – Interfaces definieren "Struktur" eines Objektes

```
interface Person {              // alternativ: type
  firstName: string,
  lastName: string|null,        // nullable Type ("ein String oder null")
  age?: number                  // optional type (darf undefined sein)
}
```

Eigene Typen – Interfaces definieren "Struktur" eines Objektes

```
interface Person {            // alternativ: type
  firstName: string,
  lastName: string|null,      // nullable Type ("ein String oder null")
  age?: number                // optional type (darf undefined sein)
}

function sayHello(p: Person) {
  console.log(`Hello, ${p.lastName}`);
  p.lastName.toUpperCase(); // Fehler: "Object is possibly null"
}

sayHello({firstName: 'Klaus', lastName: null}); // OK
sayHello({firstName: 'Klaus', lastName: 777}); // Fehler: lastName not a string
sayHello({firstName: 'Klaus', lastName: 'Mueller', age: 32}); // OK
```

STRUKTURELLE IDENTITÄT

TypeScript arbeitet mit "Struktureller Identität" (*structural identity*)

```
interface Person {
  name: string
}

interface Animal {
  name: string
}

// Eine Person erzeugen…
const p:Person = { name: 'Klaus' };

// …person einem Animal zuweisen
const a:Animal = p; // OK, da Person and Animal dieselbe Struktur haben
                    // (wäre in Java/C# nicht erlaubt)
```

Class Syntax wie in ES6, aber mit Sichtbarkeiten

```
class Person {
  private name: string

  constructor(name: string) {
    this.name = name;
  }
}

const p = new Person("Klaus");
console.log(p.name); // FEHLER: "name" nicht sichtbar
```

Generics

```
interface Person { name: string };
interface Movie { title: string };

let persons:Array<Person> = [];
let movies:Array<Movie> = [];

persons.push({name: 'Klaus'});      // OK
movies.push({title: 'Batman'});     // OK
persons.push({title: 'Casablanca'}) // Fehler (Property 'title' not in Person)
```

TYPE CHECKING JAVASCRIPT CODE

Type Checking kann auch für JS Dateien eingeschaltet werden!

- Mit der *ts-check Direktive* am Anfang einer Datei
- "Typ-Informationen" können über JSDoc hinzugefügt werden

```
// @ts-check

/**
 * @param {string} name  The name
 * @param {number} age The age
 */
function newPerson(name, age) {
  name.toLowerCase(); // OK
  age.toLowerCase(); // ERROR Property 'toLowerCase' does not exist on type 'number'
}
```

Vielen Dank!

Slides: http://bit.ly/voxxed-vienna-typescript

Beispiel-Code: https://github.com/nilshartmann/typescript-intro/

Fragen?