TYPESCRIPT: ANY – EIN TYP FÜR GEWISSE FÄLLE

Sanfte Migration

Umstellung auf TypeScript? Mit goloTS lernen Sie die Eigenheiten der Sprache kennen.

Wer sich erstmals mit TypeScript befasst, bekommt den Eindruck, dass eigentlich alles ganz einfach sei. Die gängigen Anleitungen beschreiben drei wesentliche Schritte, um ein Projekt von JavaScript auf TypeScript umzustellen. Als Erstes gilt es, TypeScript mithilfe von *npm* zu installieren:

```
$ npm install typescript --save-dev
```

Als Zweites ist eine Konfigurationsdatei namens *tsconfig.json* für den Compiler anzulegen, was im Zweifelsfall sogar automatisiert erfolgen kann:

```
$ npx tsc --init
```

Als Drittes sind die bestehenden JavaScript-Dateien umzubenennen. Aus der Dateiendung *.js* muss die Erweiterung *.ts* werden. Da TypeScript abwärtskompatibel zu JavaScript ist, lässt sich das auf dem Weg angelegte Projekt bereits anstandslos kompilieren. Nun kann man beginnen, die Welt von TypeScript zu erkunden, und sich daran machen, Code um TypeScript-spezifische Konstrukte zu erweitern.

Dieses Vorgehen funktioniert tatsächlich: Die Migration lässt sich nach und nach, Schritt für Schritt durchführen und erfordert keinen wochen- oder gar monatelangen Aufwand, während sich das Projekt in einem nicht lauffähigen Zustand zwischen zwei Sprachen befindet. So praktisch das auf den ersten Blick scheint, hat es doch auch einige Nachteile.

Nach und nach – oder alles auf einen Schlag?

Der größte Vorteil, wenn man ein Projekt nach und nach umstellt, besteht darin, dass sich die Migration im Vorbeigehen erledigen lässt. Der größte Nachteil daran ist, dass die Migration auf dem Weg sehr lange dauert und man unbequeme Änderungen unter Umständen auf sehr lange Zeit vor sich herschiebt. Denn auch wenn TypeScript abwärtskompatibel zu JavaScript ist, gibt es doch Dinge, die man in TypeScript anders schreiben würde – entweder weil die Sprache teilweise andere Konstrukte favorisiert oder weil das statische Typsystem zwar prinzipiell einen Mehrwert bietet, dafür aber gewisse Umbauten erforderlich sind.

Die Alternative besteht also in einer Migration auf einen Schlag. Diese ist zwar aufwendig und arbeitsintensiv, und sie zieht auch mit hoher Wahrscheinlichkeit eine Übergangszeit nach sich, in der das Projekt nicht lauffähig ist. Doch nach deren Durchführung hat man einen sauberen Stand, der die Konstrukte von TypeScript ideal aunutzen kann und nicht noch dauerhaft Altlasten mit sich zieht. Natürlich hängt die Entscheidung für den einen oder anderen Weg von der individuellen Situation ab, doch ist der harte Bruch häufig nicht so schlimm, wie man zunächst befürchtet.

Alles ganz strikt

Der einfachste Weg, einen harten Bruch zu bewirken, besteht darin, die Compiler-Option *--strict* zu aktivieren. Das lässt sich entweder als Parameter auf der Kommandozeile übergeben oder als Wert in der Datei *tsconfig.json* setzen:

```
{ "compilerOptions": { "strict": true } }
```

Dieser strikte Modus hat nichts (oder nur sehr bedingt) mit dem sogenannten „Strict Mode“ aus JavaScript zu tun. Zwar wird dieser Modus durch den *--strict*-Parameter auch aktiviert, er greift aber noch viel weiter: Abgesehen von einigen strengeren Typprüfungen macht sich insbesondere bemerkbar, dass die Werte *null* und *undefined* nicht mehr zulässig sind – es sei denn, die Werte wurden für eine Variable oder eine Eigenschaft explizit erlaubt.

Unabhängig davon, wie man das im konkreten Fall löst, kommt hier nun in Tutorials meist der Typ *any* ins Spiel. Dieser Typ steht zusätzlich zu den aus JavaScript bereits bekannten Typen wie *number* und *string* zur Verfügung und bedeutet so viel wie „irgendetwas“. Mit anderen Worten: Immer dann, wenn man sich in TypeScript nicht für einen Typ entscheiden kann – oder will –, ist *any* die Hintertür.

Der any-Typ – ja oder nein?

Die folgende Funktion soll helfen, das zu veranschaulichen:

```
const add =
  function (left, right) { return left + right; };
```

Ohne Kontext lässt sich nicht sagen, welchen Zweck diese Funktion hat. Es könnte sein, dass sie dazu dient, zwei Zahlen zu addieren. Sie könnte aber auch zwei Strings konkatenieren. Verwendet man den *--strict*-Parameter nicht, nimmt TypeScript hier entsprechend für *left* und *right* den Typ *any* an, womit die Funktion für Zahlen wie auch für Zeichenketten funktioniert. Diese Annahme wird als „implicit any“ bezeichnet.

Aktiviert man allerdings den *--strict*-Parameter, ist „implicit any“ verboten. Das heißt, TypeScript zwingt den Entwick-

ler nun dazu, zu spezifizieren, welche Typen erlaubt sind, wodurch die Funktion klarer definiert wird und TypeScript besser prüfen kann, ob Aufrufe der Funktion legitim sind oder nicht. Das statische Typsystem trägt hier also dazu bei, Unklarheiten zu vermeiden und Mehrdeutigkeit zu verringern:

```
const add = function (left: number, right: number):
  number { return left + right; };
```

Nun ist es bei dieser Funktion vermutlich verhältnismäßig einfach, sich festzulegen, da es unwahrscheinlich ist, dass die Funktion in der gegebenen Form tatsächlich für beides gleichermaßen verwendet wird.

Ist any gleich any?

Jedoch ist das nicht bei allen Funktionen so einfach möglich. Dafür gibt es im Wesentlichen zwei Gründe, die bei einem hinreichend komplexen Projekt gewöhnlich beide auftreten:

- Zum einen gibt es Funktionen, die eigentlich kein *any* akzeptieren sollten, bei denen das Angeben eines konkreten Typs aber so aufwendige Änderungen nach sich ziehen würde, dass man diesen Aufwand zumindest zu diesem Zeitpunkt während der Migration scheut. Hier wählt man *any* also unter Umständen, um den Compiler vorläufig zufriedenzustellen, und muss später noch aufräumen.
- Zum anderen gibt es Funktionen, die tatsächlich *any* als Parameter akzeptieren, wie etwa die von Haus aus verfügbare Funktion *console.log*, der man einen beliebigen Wert für die Ausgabe auf dem Bildschirm übergeben kann. Da diese Funktion ganz bewusst mit jedem beliebigen Typ funktionieren soll, wäre hier *any* tatsächlich die richtige Angabe.

Trägt man an all diesen Stellen *any* ein, löst man zwei aus semantischer Sicht verschiedene Probleme auf die gleiche Weise. Gelangt man nun früher oder später an den Punkt, an dem man die zunächst nur vorläufig eingetragenen *any* konkretisieren könnte, stellt sich die Frage, wie man alle betroffenen Stellen wiederfindet. Immerhin kann man schlecht nach *any* suchen, denn dann findet man auch all jene Funktionen, bei denen *any* explizit gewünscht ist.

Natürlich lässt sich das Problem so umgehen, dass man jedem *any* einen Kommentar hinzufügt, der beschreibt, ob diese Verwendung tatsächlich explizit gewünscht oder nur ein vorläufiger Workaround ist, doch ist das ziemlich aufwendig. Da zudem nicht alle Kommentare gleichermaßen formuliert werden, ist nun auch nicht mehr klar, wonach man überhaupt suchen muss – im Zweifelsfall geht man also doch wieder alle Stellen im Code durch, bei denen man ein *any* findet.

Ein Alias für any

Glücklicherweise gibt es einen einfachen Weg, wie man das Problem beseitigen kann. TypeScript kennt nämlich das *type*-Schlüsselwort, mit dem sich neue Typen definieren lassen. Im einfachsten Fall kann man *type* dazu verwenden, einem bestehenden Typ einen Aliasnamen zu geben (der, weil es ein benutzerdefinierter Typ ist, üblicherweise mit einem Großbuchstaben beginnt):

```
type Text = string;
```

Anschließend kann man *Text* als Typnamen verwenden, wo man eigentlich *string* hätte schreiben müssen. Das hat einen sehr praktischen Seiteneffekt. Auf dem Weg kann nämlich auch ein Aliasname für *any* vergeben werden:

```
type Todo = any;
```

Mit diesem Typ lässt sich nun wunderbar unterscheiden, ob man eine Variable oder einen Parameter tatsächlich als *any* typisieren will oder ob der Typ nur als vorläufiger Platzhalter dient. Da es sich bei *Todo* nun um einen eigenständigen Typ handelt, lässt sich danach auch leicht suchen. Denkbar wäre auch, mehrere solche Typen einzuführen, um verschiedene Arten von *any* zu klassifizieren.

Da dieser Typ nur temporär in dem Projekt enthalten sein wird, sollte man ihn nicht beim übrigen Quellcode ablegen. Bewährt hat sich beispielsweise, auf oberster Ebene des Projekts ein Verzeichnis namens *types* anzulegen, in dem man eine Datei *Todo.ts* mit folgendem Inhalt anlegt:

```
export type Todo = any;
```

Dieser Typ lässt sich dann überall dort, wo er benötigt wird, wie folgt importieren und verwenden, wobei natürlich der Pfad zum *types*-Verzeichnis anzupassen ist:

```
import { Todo } from '../types/Todo';
```

Ist die Migration abgeschlossen, wird der Typ nicht mehr länger benötigt und kann ohne Weiteres gelöscht werden.

Fazit

Wenn man ein Projekt von JavaScript auf TypeScript umstellt, gibt es zahlreiche Dinge, die man über das Typsystem wissen sollte. Selbst wer außer JavaScript auch C# kennt, wird in TypeScript einige Überraschungen erleben – positiver wie negativer Art. Diesen Eigenheiten wird sich die Kolumne goloTS in den zukünftigen Folgen nach und nach widmen.

Die Möglichkeit, den Typ *any* mit dem Alias *Todo* versehen zu können, ist gerade zu Beginn eine der wichtigsten Erkenntnisse, die man leider meist viel zu spät hat und die so auch in kaum einem Tutorial beschrieben wird. Daher ist das gerade am Anfang der wichtigste Tipp, um eine Migration auf Dauer weitaus einfacher und zielführender zu gestalten. ■

Golo Roden
ist Gründer und CTO der the native web GmbH. Er berät Unternehmen zu Technologien und Architekturen im Web- und Cloud-Umfeld wie TypeScript, Node.js, React, CQRS, Event-Sourcing und Domain-Driven Design (DDD).
www.thenativeweb.io

dnpCode A2005goloTS