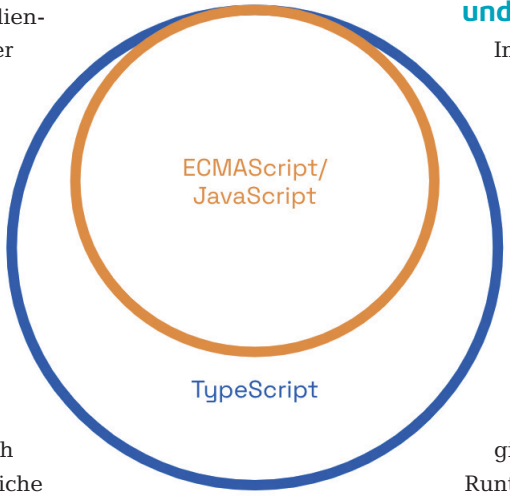
HELFER FÜR TYPESCRIPT

Tipps und Tools für die TypeScript-Praxis

TypeScript hat echte objektorientierte Programmierung in die Welt des Browsers gebracht. Die wenigen Hürden, die es mitbringt, lassen sich mit etwas Nacharbeit bewältigen.

Bei der Entwicklung von Webapplikationen gab es kaum eine technisch hervorragendere und abwechslungsreichere Zeit als die der letzten ungefähr fünf Jahre. Zahlreiche moderne Frontend-Frameworks wie Angular oder React entstanden, und auch die Umsetzung von Microservices mit Node.js und TypeScript hat große Fortschritte gemacht. Man kann also sagen, dass TypeScript die Umsetzung sowohl von serverseitigen als auch clientseitigen Anwendungen in diesem Webtechnologie-Stack deutlich professionalisiert und vereinfacht hat.

Die Historie von TypeScript beginnt Anfang der 2010er-Jahre, als es mit dem damaligen JavaScript noch nicht möglich war, saubere und objektorientierte Web-Frontends zu schreiben. Hier bediente man sich zu einem großen Teil der serverseitig gerenderten Frameworks und Sprachen, wie ASP.NET oder PHP. Doch die rasant aufkommende skalierbare Microservice- und Web-App-Welt kannte nur eine Richtung: Die Logik muss in das Frontend, und die Geschäftslogik wird als Service bereitgestellt. Hier kommt TypeScript ins Spiel – eine Open-Source-Sprache von Microsoft, die auf bestehendem JavaScript aufbaut. Dabei macht sie durch ihre statische Typisierung, fortschrittliche Funktionen und ihre Erweiterungen die Entwicklung von robusten und skalierbaren Anwendungen möglich.



Umfang von TypeScript und ECMAScript im Vergleich (Bild 1)

Dieser Beitrag nimmt Sie mit auf eine spannende Reise durch die Welt von TypeScript und zeigt Ihnen bewährte Best Practices sowie clevere Tipps und Tricks aus der Praxis, die Ihnen dabei helfen, das volle Potenzial dieser Sprache auszuschöpfen. TypeScript hat in den letzten Jahren eine exponentielle Zunahme der Beliebtheit erfahren und wird heute von vielen namhaften Unternehmen und Entwicklern weltweit eingesetzt. Die Sprache bietet eine Vielzahl von Vorteilen, darunter eine verbesserte Codequalität, frühzeitige Fehlererkennung (zum Beispiel mittels statischer Code-Analyse), bessere IDE-Unterstützung und eine erhöhte Produktivität im Entwicklungsprozess durch das objektorientierte Paradigma. Im Nachgang wird die von uns umgesetzte Anwendung über den sogenannten TypeScript-Transpiler (allgemein auch oft einfach Compiler genannt) in für einen Browser beziehungsweise die JavaScript-Runtime verständliches JavaScript übersetzt.

Genau an dieser Stelle werden wir aber auch mit den Problemen von JavaScript konfrontiert. TypeScript und JavaScript sind Sprachen – keine vollumfänglichen Frameworks. Der Funktionsumfang ist daher sehr limitiert und muss von der eingesetzten Runtime unterstützt werden.

Standardfunktionen in TypeScript und JavaScript

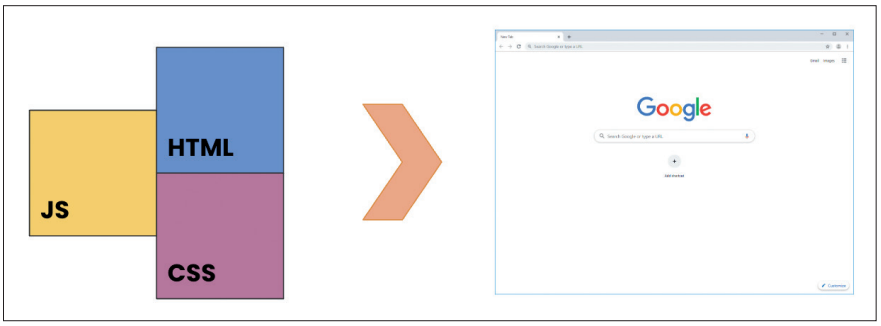
Im Gegensatz zu beispielsweise dem bekannten und stets beliebten .NET Framework, wo wir uns auf ein gewisses Grundpaket an Funktionen verlassen können, existiert dieses in der Welt von Java- und TypeScript nicht.

Als Standard wird die ECMAScript-Spezifikation herangezogen (Bild 1). Dies ist die Vorlage für die Umsetzung aktueller JavaScript-Implementierungen. Doch es gibt auch hier APIs, die nicht von allen Runtimes in gleicher Weise umgesetzt werden (können) und auch mit der Unterscheidung zu tun haben, welchen Anwendungszweck die jeweilige Applikation bedient. Eine JavaScript/TypeScript-Anwendung in einem Webbrowser hat ein grundsätzlich anderes Umgebungsumfeld als zum Beispiel eine Node.js-Applikation. Der Browser agiert wesentlich restriktiver; die Webseite, die als Kontext für unsere Applikation dient, ist ein gesondert geschützter und abgesicherter Bereich, in dem durch die Nutzerinteraktion ganz andere Funktionen bereitgestellt werden als in einer Backend-Umgebung (Bild 2).

Aus diesem Grund folgt die Empfehlung: Es prüfe, wer sich an Runtimes bindet. Es gilt genau darauf zu achten, wo der Code verwendet werden soll und welche APIs eingesetzt ▶

werden. Sollte eine Anwendungskomponente auf mehreren Runtimes oder gar zwischen Front- und Backend gemeinsam genutzt werden, so gilt es zu identifizieren, ob diese Funktionalität überall gegeben ist (Bild 3).

Aus der Erfahrung kann man sagen, dass gerade die Hardware neuer Schnittstellen sowie plattformspezifische Kommunikationswege, zum Beispiel Datenübertragung mit Bluetooth, Zugriff auf das Dateisystem et cetera, große Probleme bereiten, wenn solche Komponenten vom Backend auf das Frontend übernommen werden möchten.



Kontext einer browserbasierten Applikation: HTML, CSS und JavaScript (Bild 2)

Best Practices für die Typisierung in TypeScript

Wer mit TypeScript beginnt, sieht in erster Linie den objektorientierten Aufbau und die Typisierung als Fokuspunkte dieser Sprache. Die Verwendung von Typen ermöglicht es dem TypeScript-Compiler, frühzeitig potenzielle Fehler zu erkennen und eine klare Dokumentation des Codes zu erstellen. In diesem Abschnitt werden wir bewährte Best Practices für die effektive Typisierung und Deklarationen in TypeScript behandeln.

Eine der grundlegenden Best Practices in TypeScript ist die explizite Angabe von Typen für Variablen und Funktionen. Indem wir die Typen deklarieren, geben wir dem Compiler klare Anweisungen, welche Art von Daten erwartet wird. Dies erleichtert das Verständnis des Codes und minimiert Fehler, die durch unbeabsichtigte Datentyp-Konvertierungen entstehen können. Beispiele für häufig verwendete Grundtypen in TypeScript sind *number*, *string*, *boolean*, *array* und *object* (*function* wird in diesem Fall herausgenommen).

```
// Beispiel für die Verwendung von Typen
function addNumbers(a: number, b: number): number {
  return a + b;
}

let myString: string = "Hallo, TypeScript!";
let myArray: number[] = [1, 2, 3, 4, 5];
```

Daneben bietet die Sprache auch eine Typinferenz, bei der der Compiler den Typ automatisch ableitet, wenn keine explizite Typdeklaration vorliegt. Dadurch können wir den Code kürzer und lesbarer halten, ohne dabei auf die Vorteile der Typisierung zu verzichten.

Allerdings sollte man sich bewusst sein, dass eine explizite Typisierung in bestimmten Situationen sinnvoll ist, um das Verständnis zu verbessern und potenzielle Fehler zu minimieren.

```
// Beispiel für Typinferenz
let myNumber = 42;
  // TypeScript erkennt den Typ als "number" an.
```

Diese Inferenz gilt jedoch nicht nur für Variablen, sondern ebenso für Aufrufparameter oder Funktionsrückgabe-Typen.

Zu bedenken ist allerdings, dass in diesem Fall eine kleine Änderung, wie sie zum Beispiel durch den Aufruf einer Funktion und eine bewusste oder unbewusste Änderung des Rückgabewerts zustande kommen kann, das Verhalten sämtlicher Aufrufe drastisch beeinflussen kann. Gerade im Hinblick auf asynchrone Vorgänge ist dies aus der Erfahrung heraus ein Stolperstein gewesen.

Vollständig typisierte Funktionen bieten eine klare und eindeutige Dokumentation über die erwarteten Eingabe- und Rückgabetypen. Dies erleichtert anderen Personen die Verwendung und den Aufruf der Funktion. Gleichzeitig ermöglicht die Typisierung des Rückgabewerts dem Compiler, potenzielle Fehler zu erkennen, wenn eine Funktion versehentlich einen anderen Datentyp zurückgibt als erwartet.

```
// Beispiel für typisierte Funktion und Rückgabewert
function greet(name: string): string {
  return `Hallo, ${name}!`;
}
```

Als Empfehlung kann man jedoch mitgeben, einen möglichst einheitlichen Stil innerhalb eines Entwicklungsteams zu etablieren. Beide Varianten haben hier ihre jeweiligen Vor- und Nachteile, und auch zum Thema Lesbarkeit sind Entwickler hier unterschiedlicher Auffassung. Ist es lesbarer, einen kürzeren Code zu haben, oder ist es lesbarer, einen längeren Code zu haben, der mir jedoch die Information des Typs direkt aufzeigt? Keineswegs sollten jedoch Aufrufparameter und/oder Eigenschaften eines Objekts ohne explizite Angabe von Typen verwendet werden. Gerade bei Aufrufparametern wird ab dem Zeitpunkt von dem Typ *any* ausgegangen. Der Typ *any* beschreibt, wie der Name es vermuten lässt, sämtliche Typen. Ab diesem Zeitpunkt hat man die „typisierte Welt" verlassen und wandelt auf den alten JavaScript-Pfaden.

Interfaces und Generics richtig einsetzen

Interfaces und Generics sind zwei leistungsstarke Konzepte in TypeScript, die die Flexibilität und Wiederverwendbarkeit des Codes deutlich erhöhen.

In diesem Kapitel werden wir uns ein wenig mit der Verwendung von Schnittstellen und Klassen befassen und dabei die Unterschiede von Schnittstellen in TypeScript im Vergleich zu C# erläutern. Ein grundsätzlicher Unterschied zu Interfaces und Generics in anderen Programmiersprachen ist,

dass man im Hinterkopf behalten muss, dass letzten Endes der geschriebene Code als JavaScript in der Runtime (zum Beispiel im Browser) läuft. Auch wenn JavaScript und TypeScript sich in den letzten Jahren immer mehr aneinander angleichen, fehlen grundsätzliche statische Konzepte wie Interfaces und Generics.

Dies führt dazu, dass Interfaces zwar als Typ definiert werden können, sich jedoch nur zur Entwicklungszeit und Compile-Zeit verwenden lassen. Das Gleiche gilt hier auch für Generics, was zum Beispiel ein späteres Identifizieren eines Typs wesentlich schwieriger macht. Dies ist häufig in Auflistungen gegeben, wo unter Umständen identifiziert werden muss, von welchem generischen Typ die aktuelle Objektinstanz abhängt, um die Geschäftslogik anzupassen. Dieses Konzept existiert demnach so in TypeScript und JavaScript nicht (Bild 4). Wie damit aber umgegangen werden kann, wird im späteren Verlauf des Artikels erläutert.

Schnittstellen und Klassen sind zentrale Bausteine, um objektorientierte Prinzipien in TypeScript anzuwenden. Sie ermöglichen uns, die Struktur von Objekten und Klassen zu definieren und somit die Typisierung und Wartbarkeit des Codes zu verbessern. Hierbei ähnelt die Syntax extrem der von C#. Dieses Konstrukt kann dann als Vertrag dienen, um in bekannter Weise sicherzustellen, dass Objekte bestimmte Eigenschaften und Methoden haben. Die Verwendung von Schnittstellen erleichtert auch die Zusammenarbeit in Teams, da sie eine klare und eindeutige Dokumentation über die erwarteten Strukturen bieten.

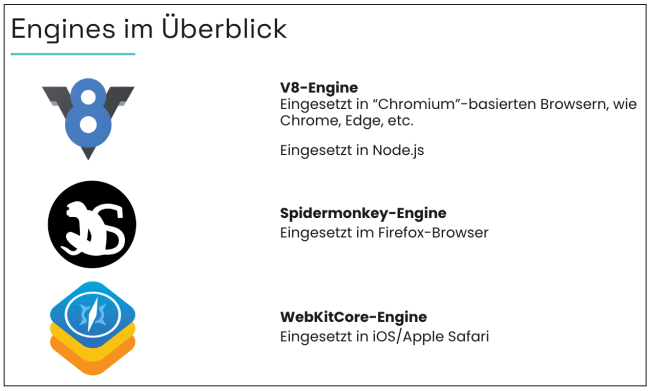
```
// Beispiel einer Schnittstelle in TypeScript
interface PersonInterface {
  name: string;
  age: number;
  greet(): string;
}

class Student implements PersonInterface {
  constructor(public name: string, public age: number) {
}

  greet() {
    return `Hallo, ich bin ${this.name} und ${this.age}
      Jahre alt.`;
  }
}
```

Im Vergleich zu C# gibt es in TypeScript einige Unterschiede bei der Verwendung von Interfaces. In TypeScript sind Schnittstellen wesentlich offener und ermöglichen die flexible Erweiterung von Typen, während C#-Interfaces normalerweise geschlossen und nicht erweiterbar sind. TypeScript-Schnittstellen können auch optionale Eigenschaften enthalten, die in C# nicht direkt unterstützt werden. Diese Flexibilität bietet uns mehr Freiheit beim Entwurf und der Verwendung von Schnittstellen.

Eine weitere Besonderheit stellt die Instanzierung von Objekten dar. Während es in C# grundsätzlich nicht möglich ist,



Aktuell verbreitete JavaScript-Runtimes (Bild 3)

ein Interface als Objektinstanz zu konstruieren, ist eine solche Verwendung in TypeScript durchaus vorgesehen. Dies hat aber weniger mit einer Flexibilität der Sprache zu tun; vielmehr geht es darum, dass in JavaScript Objekte mit beliebigen Eigenschaften direkt erstellt werden können. In diesem Fall sieht die Verwendung unserer Schnittstelle zum Beispiel folgendermaßen aus:

```
interface PersonInterface{
  name : string;
  age : number;
}

let newPerson = {
  name: "Max Mustermann",
  age: 32
} as PersonInterface;
```

Dies ergibt jedoch nur dann Sinn, wenn die Interfaces wirklich als reine DTOs ohne Logik verwendet werden. Eine solche Verwendung kann aber unter Umständen bei der Verwendung von externen APIs durchaus nützlich sein.

Im Gegensatz dazu bieten Generics eine Möglichkeit, parametrisierten Code zu schreiben, der mit verschiedenen Datentypen wiederverwendet werden kann. Mit Generics lassen sich Funktionen, Klassen oder Schnittstellen schreiben, die nicht auf einen bestimmten Datentyp beschränkt sind, sondern mit unterschiedlichen Datentypen arbeiten können.

Die Syntax für Generics verwendet den bereits aus C# bekannten *angle bracket*-Operator (<>), gefolgt von einem Buchstaben oder einem beliebigen Namen, der den generischen Typ repräsentiert. In der Praxis wird der generische Typ durch den tatsächlichen Datentyp ersetzt, wenn die Funktion oder Klasse aufgerufen oder instanziert wird.

```
// Beispiel einer generischen Funktion in TypeScript
function reverse<T>(array: T[]): T[] {
  return array.reverse();
}

let numbers: number[] = [1, 2, 3, 4, 5];
let reversedNumbers = reverse(numbers);
```

```
  // reversedNumbers hat den Typ number[]

let names: string[] = ["Alice", "Bob", "Charlie"];
let reversedNames = reverse(names);
  // reversedNames hat den Typ string[]
```

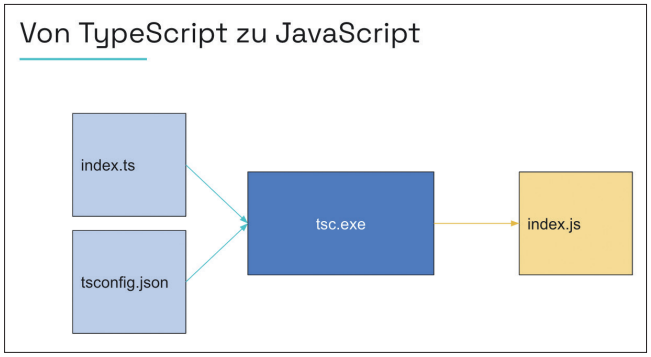
Generics haben vielfältige Vorteile und könnten dazu beitragen, den Code flexibel und wiederverwendbar zu gestalten. Aber auch sie unterliegen in diesem Fall den Eigenschaften von JavaScript und damit der Einschränkung zur Laufzeit, den generischen Typ nur unter größerem Aufwand feststellen zu können.

Anwendungsfälle für Generics ergeben sich oft bei der Arbeit mit Datenstrukturen wie Arrays, Listen, Stacks und Queues, aber auch bei der Verwendung von Containerklassen, die mit unterschiedlichen Datentypen arbeiten sollen. Ein klassisches Beispiel ist die *Array*-Klasse in TypeScript, die Generics verwendet, um Arrays mit verschiedenen Datentypen zu erstellen. Damit bietet dieses Konzept eine elegante und flexible Lösung, den Code zu optimieren, doppelte Implementierungen zu vermeiden und komplexe Anwendungen zu vereinfachen.

Vorausgehend haben wir also nun festgestellt, dass die Identifizierung von Objekttypen, sobald man mit einem generischen Typ oder einer Schnittstelle arbeitet, in der Laufzeit nicht mehr ganz so einfach ist. Durch die klare Instanzierung der Objekte und die klare Definition der Typen, wie wir es aus C# und .NET im Allgemeinen kennen, können wir an beliebiger Stelle auf den Objekttyp prüfen. Nehmen wir unser Beispiel des *PersonInterface* noch einmal und schauen wir uns an, was geschehen würde, wenn wir eine Funktion bauen, die ein *PersonInterface* übergeben bekommt.

```
interface Person {
  name: string;
  age: number;
  greet(): string;
}

class Student implements Person {
  constructor(public name: string, public age: number) {
  }
```



TypeScript wird für die Runtime in ausführbares JavaScript konvertiert (Bild 4)

```
  greet() {
    return `Hallo, ich bin ${this.name} und ${this.age}
      Jahre alt.`;
  }
}

class Teacher implements Person {
  constructor(public name: string, public age: number,
    public course : string) { }

  greet() {
    return `Hallo, ich bin ${this.name}, ${this.age}
      Jahre alt und unterrichte ${this.course}.`;
  }
}

function displayPerson(person : Person){

  //if person ist Student => Gehe zu URL "student/..."
  //if person ist Teacher => Gehe zu URL "teacher/..."

}

let studentObj = new Student("Max Mustermann", 21);
let teacherObj = new Teacher("Anna Müller", 34,
  "Mathematik");

displayPerson(studentObj);
```

Betrachten wir hier die Funktion *displayPerson*. In diesem Fall haben wir zwei Implementierungen von dem Interface *Person* und möchten – je nachdem, um welchen Typ es sich handelt – auf zum Beispiel eine andere Seite im Browser wechseln.

Nachdem wir zur Laufzeit jedoch ausschließlich ein Objekt mit den Eigenschaften *name*, *age* oder *course* haben, können wir an dieser Stelle keine Identifizierung durchführen. Hierfür gibt es nun drei Methoden, das Problem zu lösen.

Die erste Variante wäre das Prüfen auf eine bestimmte Eigenschaft, auf die wir uns verlassen können. Ist in diesem Beispiel etwa die Eigenschaft *course* vorhanden, können wir von einem Objekt des Typs *Teacher* ausgehen.

Die zweite Variante beschreibt einen Diskriminator. Hier würden wir eine gesonderte Eigenschaft, zum Beispiel *type* erstellen, die entweder mit *student* oder *teacher* befüllt wird und uns ein Alleinstellungsmerkmal gibt.

Die dritte Option wäre die „Zwiebel"-Variante. Hierbei erstellen wir schlicht ein neues Objekt nach folgendem Vorbild:

```
interface PersonAbstraction{
  teacher? : Teacher;
  student? : Student;
}
```

Hierbei werden die einzelnen Typen als optionale Eigenschaften angesehen und können auf deren Existenz geprüft werden. Das kann zum Beispiel so aussehen:

```
function displayPerson(person : PersonAbstraction)
{
  if(person.teacher)
  {
    //das Objekt ist ein Lehrer
  }
  else if(person.student)
  {
    //das Objekt ist ein Schüler
  }
}

let studentObj = new Student("Max Mustermann", 21);

displayPerson({ student: studentObj });
```

Eine etwas exotische Version, die jedoch gerade in der Kommunikation zwischen Web APIs eingesetzt werden kann, um vollständig typisiert zu arbeiten und sich nicht auf die Umsetzbarkeit eines Diskriminators stützen zu müssen.

Zusammenfassend bieten Interfaces und Generics in TypeScript umfangreiche Möglichkeiten, die Flexibilität, die Typsicherheit und die Wiederverwendbarkeit unseres Codes erheblich zu steigern.

Durch die Verwendung von Schnittstellen können wir benutzerdefinierte Typen definieren und die Struktur von Objekten klar dokumentieren. Generics ermöglichen uns die Erstellung von parametrisierten Funktionen und Klassen, die mit verschiedenen Datentypen arbeiten können, und führen zu einer effizienten und sauberen Codebasis.

Mehr Codequalität, einheitliche Formatierungen und statische Code-Analysen

Der Stil eines Codes und dessen Qualität sind entscheidend für die Lesbarkeit, die Wartbarkeit und die Konsistenz eines TypeScript-Projekts. Dabei helfen statische Analysetools wie die sogenannten Linter-Tools. Diese helfen uns, potenzielle Probleme im Code frühzeitig zu erkennen und sicherzustellen, dass der Code den definierten Standards entspricht. Darüber hinaus werden wir einen Blick auf die Codedokumentation mit JSDoc werfen, um eine umfassende und leicht verständliche Dokumentation für den Code zu erstellen.

Eine einheitliche Formatierung ist ein essenzieller Bestandteil jedes professionellen TypeScript-Projekts. Durch eine konsistente Formatierung wird der Code übersichtlicher und leichter lesbar, was die Zusammenarbeit im Entwicklerteam erleichtert. Coding-Standards helfen dabei, klare Regeln für die Namensgebung, Einrückung, Zeilenumbrüche und andere stilistische Aspekte festzulegen.

Um dies zu erreichen, empfehlen sich Tools wie ESLint und Prettier. Bei ESLint wird der Code vor dem Kompilieren/Transpilieren überprüft und auf Fehler geprüft. Dies können unter anderem fehlende Prüfungen auf *undefined*/*null* sein, falsch eingesetzte Typen (*any*) oder nicht genutzte Variablen, um nur einige zu nennen. Des Weiteren bietet ESLint auch ein Basispaket an Standardformatierungen, die eingehalten werden müssen. Diese können zum Beispiel die Verwendung von Tabs/Leerzeichen betreffen oder die Verwendung von einfachen oder doppelten Anführungszeichen.

Wer dann in der IDE ein Plug-in wie Prettier nutzt, kann auf die Funktionen und Regeln zur Entwicklungszeit zugreifen. Hierbei werden die ESLint-Regeln unterstützt, es können aber noch weitaus detailliertere Regeln in einer Prettier-Konfigurationsdatei im Projekt-Repository hinterlegt werden. Dies hat den Vorteil, dass man anfangs ins Detail gehen und die Regeln für jedes Teammitglied definieren kann. Gerade in der Zusammenarbeit mit mehreren Personen oder gar unterschiedlichen Teams, Standorten oder Unternehmen eignet sich das Vorgehen, um eine möglichst einheitliche Style-Linie durchzuhalten. Diese Regeln können dann automatisch beim Speichern einer Datei angewendet werden und ersparen sehr viel Korrekturarbeiten im Rahmen eines Code-Reviews.

Ob und wie stark man diese Regeln definieren möchte, ist eine sehr individuelle Frage. Grundsätzlich möchte ich jedoch empfehlen, mit Tools wie ESLint zumindest einfache Regeln wie Tabs/Spaces oder Anführungszeichen zu standardisieren.

Dokumentation und immer an andere Entwickler denken

Die Codedokumentation ist ein wesentlicher Bestandteil einer gut gepflegten Codebasis. Sie bietet eine umfassende Beschreibung der Funktionalität von Klassen, Funktionen und Schnittstellen. Damit kann Entwicklern in kurzen Stichpunkten eine Funktion erläutert und deren Verwendung beschrieben werden.

TypeScript verwendet für die Codedokumentation JSDoc, eine Konvention, die aus der JavaScript-Welt stammt und von TypeScript vollständig unterstützt wird. JSDoc ermöglicht es uns, spezielle Kommentare vor Funktionen, Klassen oder Schnittstellen zu setzen und wichtige Informationen zu dokumentieren, wie zum Beispiel Parameter, Rückgabetypen und eine kurze Beschreibung der Funktionalität. Diese Kommentare werden von den meisten modernen IDEs erkannt und können Entwicklern kontextbezogene Informationen direkt während der Entwicklung anzeigen.

```
/**
 * Berechne die Summe von zwei Zahlen.
 *
 * @param {number} a - 1. Zahl.
 * @param {number} b - 2. Zahl.
 * @returns {number} Die Summe der beiden Zahlen.
 */

function sumNumbers(a: number, b: number): number {
  return a + b;
}
```

Hier können, ähnlich wie bei anderen bekannten Codedokumentations-Formaten, in einem Block-Kommentar die Funktion und die Verwendung der einzelnen Parameter beschrieben werden. Im Fall von TypeScript empfiehlt es sich, den Typ (in diesem Fall *numbers*) ebenfalls anzugeben. Dies hat den Hintergrund, dass die Dokumentation der Funktion so auch ▶

aus der JavaScript-Welt heraus genutzt werden kann und sich nicht auf das Vorhandensein des typisierten Codes stützt.

Die Codedokumentation mit JSDoc erleichtert nicht nur die Verwendung des Codes durch andere Entwickler, sondern unterstützt auch die automatische Generierung von Dokumentationen mithilfe von Tools wie TypeDoc. Durch eine solche verständliche Dokumentation kann die Wartbarkeit und Erweiterbarkeit des Projekts erheblich verbessert werden.

Arbeiten mit Aufzählungen: Da gab es doch mal LINQ, oder?

„Ach – als .NET-Entwickler ist man schon verwöhnt", sage ich mir immer wieder, wenn mit TypeScript gearbeitet wird. Im Gegensatz zu .NET ist das Arbeiten mit Daten und insbesondere Aufzählungen dort ein zweischneidiges Schwert.

Die eine Seite ist, dass JavaScript/TypeScript aus der Flexibilität von Aufzählungen profitieren kann. Ein Array hat keine festgelegte Größe und kann, egal wann, manipuliert werden, während in .NET dafür mit Listenkonstrukten gearbeitet werden muss. Auf der anderen Seite gibt es in JavaScript nur sehr rudimentäre Filter- und Manipulationsmöglichkeiten einer Aufzählung. Diese reichen zwar für einen großen Anteil aus, doch Verkettungen oder das Heraussuchen eines Minimal- oder Maximalwerts sind ohne das bekannte LINQ nicht so einfach umzusetzen.

Für diese Problemstellung existiert die Bibliothek Lodash. Lodash ist eine beliebte JavaScript-Bibliothek, die die Entwicklung von JavaScript- und TypeScript-Anwendungen vereinfacht. Sie bietet eine Fülle von Funktionen, die in den Bereichen Array-Operationen, Objektmanipulation, Datenfilterung, String-Verarbeitung, mathematische Operationen und vielen mehr eingesetzt werden können. Eines der Hauptmerkmale von Lodash ist seine Fokussierung auf Performance und Effizienz, wodurch komplexe Datenverarbeitungsaufgaben optimiert werden.

Zwar liefert Lodash auch einige Standardfunktionen aus JavaScript mit, bildet diese aber alle unter einer gut strukturierten Bibliothek ab. Hier ein Beispiel, wie mit Arrays gearbeitet werden kann:

```
import * as _ from 'lodash';

const numbers: number[] = [1, 2, 3, 4, 5];

// Beispiel: Verwendung der "map"-Funktion, um die
// Inhalte des Arrays zu verdoppeln
const doubledNumbers = _.map(numbers, (num) => num * 2);
// Ergebnis: [2, 4, 6, 8, 10]

// Beispiel: Verwendung der "filter"-Funktion, um
// ungerade Zahlen zu entfernen
const evenNumbers = _.filter(numbers, (num) => num % 2
  === 0);
// Ergebnis: [2, 4]

// Beispiel: Verwendung der "sum"-Funktion, um die Summe
// der Zahlen zu berechnen
const sumOfNumbers = _.sum(numbers);
// Ergebnis: 15
```

Diese Funktionen erinnern an die *Select()*-, *Where()*- und *Sum()*-Funktionen, die wir für Auflistungen aus .NET kennen. Da Lodash eine üblicherweise reine JavaScript-Bibliothek ist, können wir hier auch aus den Vollen der Flexibilität untypisierter Objekte schöpfen und einzelne Eigenschaften entfernen oder mehrere Objekte kombinieren. Dies ist insbesondere bei der Arbeit mit Schnittstellen sinnvoll, wenn man Informationen nicht an den Client herausgeben möchte oder auf der anderen Seite keine neue Datenklasse für die Kombination zweier Informationen aufbauen möchte.

```
import * as _ from 'lodash';

const user = {
  name: 'Max Mustermann',
  age: 30,
  city: 'Augsburg'
};

// Beispiel: "omit"-Funktion, um das Alter aus dem
// Objekt zu entfernen
const userWithoutAge = _.omit(user, 'age');
// Ergebnis: { name: 'Alice', city: 'Berlin' }

// Beispiel: "merge"-Funktion, um zwei Objekte
// zusammenzuführen
const additionalInfo = {
  occupation: 'Lead-Developer',
  company: 'ACME Corp.'
};

const mergedUser = _.merge(user, additionalInfo);
// Ergebnis: { name: 'Max Mustermann', age: 30,
// city: 'Augsburg', occupation: 'Lead-Developer',
// company: 'ACME Corp.' }
```

Neben nützlichen Funktionen bei der Arbeit mit Arrays oder Objekten gibt es weitere Funktionen, um mit mathematischen Operatoren zu arbeiten oder eben auch mit Texten in Strings. Gerade das Formatieren von Texten zu verschiedenen „Cases" ist eine wunderbare Funktion, um zum Beispiel aus einem Namen eine ID zu generieren oder Anreden zu formatieren.

```
import * as _ from 'lodash';

// Beispiel: Verwendung der "words"-Funktion, um die
// Wörter im Text zu zählen
const sentence = 'Hallo, dies ist ein Beispieltext';
const wordCount = _.words(sentence).length;
// Ergebnis: 6

// Beispiel: Name für Anrede vorbereiten
const name = 'max mustermann';
```

```
const formattedName = _.upperFirst(name);
// Ergebnis: 'Max Mustermann'

// Beispiel: Firmenname für "ID" formatieren
const company = 'ACME-Corporation Limited';
const formattedId = _.camelCase(company);
// Ergebnis: 'AcmeCorporationLimited'
```

Die hier gezeigten Beispiele sind nur ein kleiner Ausschnitt der vielen Möglichkeiten, die Lodash für die effiziente Verarbeitung von Daten in TypeScript bietet. Die Bibliothek bringt eine umfangreiche Dokumentation mit und eine aktive Community, die Entwickler bei Fragen und Herausforderungen unterstützt. Mit dieser Vielzahl von Funktionen und Utility-Methoden kann Lodash die Arbeit mit Daten in TypeScript erheblich erleichtern. Von Array-Operationen über Objektmanipulation bis hin zur String-Verarbeitung bietet Lodash eine einheitliche Syntax und eine breite Palette von Funktionen, die eine effiziente Datenverarbeitung ermöglichen.

Die Sache mit der Zeit …

Zeit spielt im Leben eine große Rolle, und diese Rolle finden wir als Entwickler auch jeden Tag in unserer Arbeit wieder. Es gibt kaum eine Applikation, die nicht an der ein oder anderen Stelle mit dem Faktor Zeit umgehen muss, sei es für die Verwaltung von Terminen, das Berechnen von Zeitspannen oder das Formatieren von Zeitangaben für die Benutzeroberfläche. Auch hier bieten .NET und viele andere Programmiersprachen vollständig integrierte Libraries, die die Arbeit mit Datums- und Zeitangaben sehr komfortabel machen. Einfach mal eine Zeitspanne zwischen zwei Zeitpunkten berechnen und als Minuten zurückgeben? In vielen Sprachen ein typischer Einzeiler.

In JavaScript sieht das anders aus. Zwar gibt es den Datentyp *Date*, der einen Zeitpunkt definiert, jedoch existieren nahezu keine praktisch nutzbaren Möglichkeiten, mit einem Datum komfortabel zu rechnen (außer natürlich, man möchte immer mit Millisekunden arbeiten) oder dieses zu manipulieren. Eine Lösung namens Moment.js ist eine leistungsstarke JavaScript-Bibliothek, die speziell für die Arbeit mit Datum und Zeit entwickelt wurde. Der folgende Überblick soll uns mit Moment.js und seinen Funktionen vertraut machen, die uns dabei unterstützen, komplexe Aufgaben rund um Datum und Zeit in TypeScript zu bewältigen.

Moment.js ist intuitiv zu verwenden und bietet eine große Anzahl von Funktionen für die Manipulation, Formatierung und Validierung von Datum und Zeit. Moment.js unterstützt gängige, internationale Datumsformate und Zeitzonen, was es zu einer zuverlässigen Wahl für die Arbeit mit Zeitzonen und internationalen Datumsformaten macht.

Gerade das Parsen und Formatieren von Datumsangaben ist für die Umsetzung von Webapplikationen ein enormer Vorteil. Hier empfehle ich, ebenfalls als Regel innerhalb des Teams zu definieren, dass sämtliche Datumsinformationen, die umgewandelt werden müssen, mit den Werkzeugen von Moment.js verarbeitet werden müssen. Dadurch werden typische Probleme wie etwa die unbeabsichtigte Darstellung einer UTC-Zeit als lokale Zeit und dergleichen verhindert.

```
import moment from 'moment';

// Beispiel: Parsen eines Datums- und Zeit-Strings
const dateTime = '2023-01-01 12:30:00';
const parsedDateTime = moment(dateTimeStr);

console.log(parsedDateTime); // Ausgabe:
2023-01-01T12:30:00.000Z

// Beispiel: Formatierung des Datums und der Zeit im
// gewünschten Format
const formattedDateTime =
  parsedDateTime.format('DD.MM.YYYY HH:mm');
console.log(formattedDateTime);
  // Ausgabe: 01.01.2023 12:30
```

Moment.js ist eine ausgezeichnete Wahl für die Arbeit mit Datum und Zeit. Die Bibliothek ist dabei ein wertvolles Werkzeug für die Verwaltung von Terminen, die Berechnung von Zeitspannen oder das Formatieren von Datum und Uhrzeit für die Ausgabe.

Was gibt es sonst noch so?

In der Praxis der letzten Jahre mit TypeScript konnte ich feststellen: Es gibt fast nichts, was es nicht gibt. Durch die enorm große Entwickler-Community in Sachen JavaScript findet man zahlreiche Bibliotheken für alle möglichen Einsatzzwecke. Sei es die Generierung von eindeutigen GUIDs nach verschiedenen Regeln (Bibliothek: uuid), die Kommunikation mit Web APIs (Bibliothek: Axios), Kommunikation zwischen mehreren Tabs der gleichen Domain (Bibliothek: krasimir/lsbridge) oder das Arbeiten mit dem lokalen Browser-Storage (Bibliothek: store.js).

Fazit

Die Welt von JavaScript und TypeScript ist auf der einen Seite enorm flexibel und bietet zahlreiche Lösungsansätze. Auf der anderen Seite ist eine wesentlich höhere Anlaufhürde zu nehmen. Gerade wer aus vollständig integrierten Frameworks wie eben .NET kommt, wird in den ersten Wochen immer mit fehlenden APIs und Funktionen kämpfen.

Aus diesem Grund ist es wichtig, vor dem ersten produktiven Projekt eine eigene Zusammenstellung an Bibliotheken und Grundsätzen zu deklarieren, die als Vorlage dienen können. ■

Patrick Schnell
ist Geschäftsführer von schnell.digital, das auf Entwicklung und Architektur von Digitalisierungsprojekten und auf Beratung spezialisiert ist. Dabei kommen Technologien wie .NET, Angular, TypeScript und Node.js zum Einsatz.
https://schnell.digital

dnpCode A2402TypeScript