DIE PHILOSOPHIE VON NODE.JS

Anders denken

Warum Sie Node.js ausprobieren sollten – ein Vergleich mit .NET. Vorsicht, spitze Feder!

Der Wechsel von einer technologischen Plattform zur anderen ist zwiespältig. Da er zumeist auch mit einem Wechsel der primär verwendeten Programmiersprache einhergeht, ist in der Regel zumindest die Syntax neu. Trotzdem stellt das in den wenigsten Fällen ein Problem dar, da sich viele Sprachen syntaktisch sehr ähnlich sind, insbesondere dann, wenn man lediglich innerhalb der Familie wechselt, beispielsweise von Java zu C#.

Dramatische Wechsel, die das grundlegende Konzept der Syntax austauschen, sind selten. Dennoch können sie vorkommen.

Das gilt zum Beispiel immer dann, wenn jemand zu Lisp wechselt, das gänzlich anders funktioniert als nahezu jede andere Programmiersprache (siehe [1]).

Beruhigend für die meisten Entwickler ist bei einem Technologiewechsel allerdings, dass die grundlegenden Konzepte stets gleich bleiben.

Wer weiß, wie die HTTP-Anfragen in .NET verarbeitet werden, findet ein ähnliches Vorgehen auch in Java wieder – oder gar in Lisp.

Das liegt in der Natur des zu lösenden Problems, in dem Fall dem Bearbeiten von HTTP-Anfragen. Dinge mögen auf verschiedenen Plattformen unterschiedlich benannt sein, aber konzeptionell ähneln sie sich häufig stark.

Hinzu kommt das Beschäftigen mit neuen Bibliotheken, die an und für sich die gleichen Konzepte anders strukturieren. Man muss einfach lernen, sich wieder zurechtzufinden. Bis man weiß, welche Funktion oder welche Klasse sich in welchem Paket verbirgt, vergeht eine Weile, doch ist auch das keine wirkliche Herausforderung.

Nimmt man all das zusammen, klingt der Wechsel von einer Technologie zu einer anderen gar nicht mehr so abschreckend, sondern durchaus machbar. Zwar mag man eine gewisse Zeit dafür benötigen, aber auch das scheint überschaubar zu sein.

Unterschiedliche Philosophien

Leider vergessen die meisten Entwickler einen gravierenden Punkt, wenn nicht gar den gravierendsten Punkt schlechthin: Jede Entwicklungsplattform folgt einer anderen Philosophie. Diese Philosophie ist dafür verantwortlich, welche Muster auf der jeweiligen Plattform zählen, was als guter oder schlechter Code gilt und wie man Dinge üblicherweise löst.

Häufig bemerken erfahrene Entwickler das fehlende Hintergrundwissen rasch. War es beispielsweise in C und C++ auf Grund der manuellen Speicherverwaltung üblich, Variablen zu Beginn einer Funktion zu deklarieren, ist das unter .NET dank der automatischen Garbage-Collection nicht länger nötig [2].

Ehemalige C- oder C++-Entwickler konnte man unter C# lange Zeit hervorragend unter anderem an dieser Eigenheit erkennen. Technisch gesehen ist es völlig irrelevant, aber es gibt bestimmte Gründe, warum Dinge eher auf die eine oder die andere Art gemacht werden. Wer von einer anderen Plattform her kommt, überträgt diese Gründe zunächst und fällt damit mit der Denkart im neuen System auf.

Wer von der Desktop- oder der klassischen Webentwicklung unter C# her kommt, bringt ebenfalls solches Hintergrundwissen mit. Unter C# ist beispielsweise lange Zeit die Verwendung von SOAP äußerst gängig gewesen.

Je stärker eine Lösung auf die Bedürfnisse von Unternehmen zugeschnitten werden musste, desto stärker war die Rolle von SOAP. Der Grund dafür liegt auf der Hand: C# verfügt über ein statisches Typsystem, und SOAP bietet ein ebensolches, mit Kontrakten und allem, was sonst noch dazugehört. Das kommt C#-Entwicklern entgegen.

In der modernen Webentwicklung hingegen macht man sich als SOAP-Fan mit hoher Wahrscheinlichkeit lächerlich. Hier ist REST das Mittel der Wahl, das auf eine statische Typisierung und strenge Kontrakte explizit verzichtet.

Das flexible JSON-Format tritt an die Stelle von XML, die diversen WS-Star-Protokolle sind ersatzlos gestrichen, und als kleinster gemeinsamer Nenner gilt, Daten über HTTP zu übertragen.

C# gegen den Rest der Welt

Das schreckt C#-Entwickler zunächst ab, da es nicht der gewohnten Denkart entspricht und daher diverse Vorteile nicht bieten kann. Diese Vorteile sind allerdings nicht objektiv als solche zu erkennen, sondern existieren nur in ihrem jeweiligen Kontext.

Was man häufig nicht weiß, ist, ob die gleichen Probleme auf einer anderen Plattform überhaupt existieren, und – falls nicht – ob die vermeintlichen Vorteile dann nicht eher zum Nachteil gereichen.

Wäre REST nicht ebenfalls für Unternehmensprobleme geeignet, würden weltweit agierende Konzerne wie Amazon, Facebook, Google und Co. wohl kaum primär auf REST statt auf SOAP setzen.

Ein ebenfalls häufig genannter Einwand gegen die moderne Webentwicklung beruht auf dem gleichen Grund: Weil es kein statisches Typsystem gebe, müsse es unmöglich sein, große und komplexe Anwendungen zu schreiben!

Kaum ein C#-Entwickler kann sich vorstellen, wie man eine Anwendung mit 15 Millionen Zeilen Code auf einer Plattform aufbauen kann, die lediglich ein dynamisches Typsystem zur Verfügung stellt.

Der Denkfehler dabei liegt darin, von den bekannten Paradigmen auszugehen und diese blind auf die neue Plattform zu portieren. Schreibt man Code weiterhin so, wie man es von C# her gewohnt ist, wird das vermutlich schiefgehen – das liegt dann allerdings nicht an der neuen Plattform, sondern an fehlender Erfahrung und gegebenenfalls auch an dem fehlenden Blick über den Tellerrand.

Eine neue Technologieplattform ist wie eine neue Welt, und wer sich auf die Reise macht, sie zu entdecken, muss offen für Neues sein, und vor allem bereit, Altbekanntes und vermeintlich Bewährtes über Bord zu werfen.

Legt man die erforderliche Offenheit an den Tag, stellt man nämlich fest, dass viele Probleme aus einem anderen Blickwinkel betrachtet andere Lösungen erlauben, die dem bisher Bekannten gänzlich widersprechen.

Node.js ist nicht .NET

Alles Gesagte gilt unter anderem auch beim Wechsel von C# und .NET zu JavaScript und Node.js. Die beiden Sprachen C# und JavaScript sehen sich syntaktisch sehr ähnlich, doch ist das bereits die erste große Falle: JavaScript ist nicht C#. Sie entstammen noch nicht einmal der gleichen Sprachfamilie.

Während es sich bei C# um ein Mitglied der C-Familie handelt, ist JavaScript viel stärker mit Lisp verwandt, das (bewusst) ganz anders funktioniert.

Wer das nicht weiß, läuft Gefahr, die Sprache von vornherein falsch zu verwenden. Der entsprechende Misserfolg ist dann vorprogrammiert.

Daran ändert übrigens auch TypeScript nur sehr wenig: Es bringt JavaScript syntaktisch noch näher an C# heran, basiert aber immer noch auf den Strukturen und Konzepten von Lisp, die in JavaScript enthalten sind.

Das führt lediglich dazu, dass C#-Entwickler verstärkt das Gefühl bekommen, man müsse sich ja nicht richtig mit JavaScript auseinandersetzen, sondern könne mit TypeScript die Unterschiede einfach wegabstrahieren. Dass dem nicht so ist, wird sich erst langfristig durch das Auftreten von seltsamen Fehlern herausstellen.

Doch die Sprache ist nicht das Einzige, was sich im Hinblick auf einen potenziellen Wechsel zu Node.js unterscheidet.

Auch Node.js tickt ganz anders als .NET, da es im Gegensatz zur Microsoft-Plattform von vornherein auf die Cloud ausgelegt ist.

Node.js kennt kein Multithreading

Das zeigt sich beispielsweise hervorragend am fehlenden Multithreading. Kann man Code nur noch mit einem einzigen Thread ausführen, erzielt man zunächst zahlreiche Vorteile. Es gibt weniger Probleme hinsichtlich Race Conditions, Deadlocks und ähnlichen Situationen.

Man muss sich nicht mehr mit der Synchronisation von Datenzugriffen herumschlagen: Semaphoren, Mutexes und Co. gehören der Vergangenheit an. Auch das Debuggen wird deutlich einfacher.

Trotzdem ist das Einzige, was C#-Entwickler zunächst wahrnehmen, die vermeintliche Einschränkung, dass man dann ja nicht mehr skalieren könne, um beispielsweise mehrere Prozessoren oder Prozessorkerne auszureizen. Das kommt daher, dass sie gewohnt sind, Threads als Lösung für das Problem der Skalierbarkeit anzusehen, weil es in der .NET-Welt so gelebt wird. Doch es gibt eben noch andere Lösungen des Problems.

Tatsächlich kennt Node.js sehr wohl ein Konzept, um über mehrere Prozessoren skalieren zu können: Wenn ein Prozess lediglich einen Prozessor ausnutzen kann, gilt es eben entsprechend viele Prozesse zu starten. Die Anwendung läuft also nicht nur einmal, sondern gleich mehrfach.

Der nächste Einwand lässt jedoch nicht lange auf sich warten. Die Prozesse müssten schließlich auch synchronisiert werden, und Interprozesskommunikation (IPC) sei noch hässlicher als die über Threads.

Was dabei vergessen wird, ist, dass für wirklich große Anwendungen eine Skalierung innerhalb einer Maschine über mehrere Prozessoren gar nicht genügt.

Alleine schon, um beispielsweise Lastverteilung und Hochverfügbarkeit anbieten zu können, muss man über mehrere Maschinen skalieren. ▶

Dann ist jedoch IPC unvermeidlich. Wenn man also davon ausgeht, dass ohnehin verteilte Anwendungen entwickelt werden sollen, die maschinenübergreifend ausgeführt werden, warum dann nicht IPC zur Regel erheben und Threads als Ausnahme für den Spezialfall ansehen, dass lediglich eine einzige Maschine benötigt wird? Zugegebenermaßen dürfte das Szenario in der Cloud sehr selten anzutreffen sein.

Wenn Threads also lediglich die Lösung für einen seltenen Spezialfall sind und man alle anderen Fälle mit IPC lösen muss und Threads sich durch IPC ersetzen lassen, warum setzt man dann nicht von vornherein auf IPC?

Damit hätte man nur noch ein einziges Kommunikationsmodell und kann von Haus aus skalieren, sei es über Prozessoren- oder über Maschinengrenzen. Betrachtet man das Problem aus diesem Blickwinkel, erscheinen Threads auf einmal gar nicht mehr als so erstrebenswert.

Das Ökosystem von Node.js

Auch an anderer Stelle werden die Unterschiede zwischen .NET und Node.js rasch deutlich. Während .NET eine Klassenbibliothek mitbringt, die Tausende von Anwendungsfällen umfasst, enthält Node.js serienmäßig kaum Module. Die vollständige API-Dokumentation von Node.js [3] lässt sich problemlos an einem Wochenende lesen.

Ein ähnliches Vorhaben für die MSDN würde hingegen mehrere Jahre, wenn nicht Jahrzehnte in Anspruch nehmen. Was ist der Grund für die Sparsamkeit von Node.js im Hinblick auf eingebaute APIs?

Der Grund ist wieder philosophischer Natur. Node.js soll klein und stabil sein und sich gut pflegen lassen. Das funktioniert umso besser, je kleiner die Codebasis von Node.js selbst ist. Folgerichtig wird die Laufzeitumgebung so entwickelt, dass sie ausschließlich jene Module enthält, die zwingend im Kern enthalten sein müssen.

Alles andere obliegt den Entwicklern, die mit Node.js arbeiten. Auf diesem Weg bleibt Node.js allerdings nicht nur klein und schlank, sondern hat auch von vornherein dazu beigetragen, Entwickler zu motivieren, doch gefälligst eigene Module zu entwickeln.

Die Paketverwaltung von Node.js [4] macht es Entwicklern daher auch extrem einfach, eigene Module zu veröffentlichen beziehungsweise Drittmodule in ein eigenes Projekt einzubinden (Bild 1).

Anders als bei .NET kommt man bei Node.js gar nicht erst in die Versuchung, ein Problem rein mit Bordmitteln lösen zu wollen. Viel mehr als „Hallo Welt“ lässt sich auf diesem Weg nämlich gar nicht entwickeln.

Ingsesamt sorgt Node.js damit für ein enorm lebhaftes Ökosystem, das inzwischen die größte Sammlung von Open-Source-Code der Welt darstellt und das immens schnell wächst. Nuget, Maven und Co. wirken im Vergleich zu npm wie niedliche Sandkästen, in denen allerdings nicht ernsthaft gegraben und gebaut wird.

Kritiker wenden ein, dass Qualität und Quantität zwei paar Schuhe seien. Alleine die Tatsache, dass in der npm-Registry derzeit annähernd eine halbe Million Module enthalten seien, sage noch nichts über deren Alltagstauglichkeit aus.

Dieser Einwand ist berechtigt, trifft das Ziel aber nicht. Natürlich sind nicht alle Module innerhalb des Ökosystems von Node.js gleich gut. Aber zum einen gibt es genau dafür die Vielfalt, sodass man sich etwas zu den eigenen Bedürfnissen Passendes heraussuchen kann. Zweitens lebt Node.js von der Grundidee von Open Source, dem gegenseitigen Geben und Nehmen. Findet man in einem Modul einen Fehler oder eine andere Unzulänglichkeit, steht es jedem frei, an der Verbesserung mitzuwirken. Das ist einer der Gründe, warum der Code praktisch jedes npm-Moduls auf GitHub gehostet wird.

Node.js ist Open Source

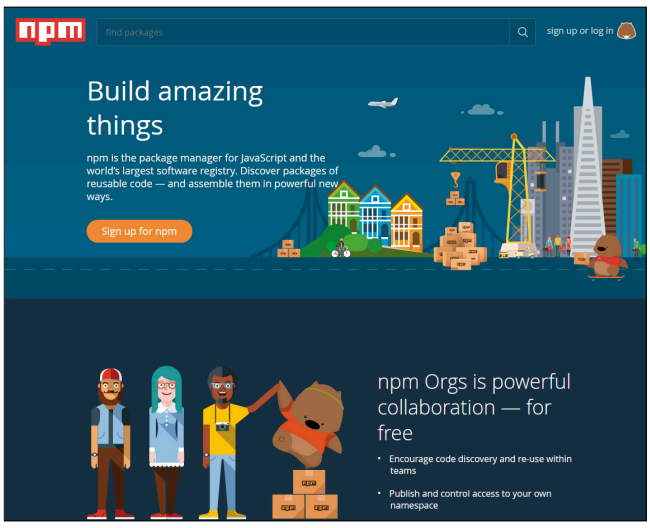
Häufig erwähnt wird in dem Zusammenhang das Fiasko um das left-pad-Modul [5], das eine Zeichenkette von links auf eine gewünschte Länge mit Füllzeichen auffüllt:

```
~~~~~~
console.log(leftPad('foo', 5));
// => '  foo'
~~~~~~
```

Das Modul wurde im Frühjahr 2016 aus rechtlichen Gründen von seinem Autor aus der npm-Registry entfernt. Die Folge davon war, dass sich Tausende andere Module nicht mehr installieren und verwenden ließen, die left-pad intern verwendeten. Das hat die Frage aufgeworfen, wie sinnvoll es ist, ein eigenständiges Modul zu veröffentlichen, das gerade einmal aus 11 Zeilen Code besteht. Zahlreiche Kritiker haben den Vorfall damals zum Anlass genommen, sich über Node.js und dessen Ökosystem lustig zu machen.

Was sie dabei allerdings gern vergessen, ist, dass die Größe des Moduls nichts mit den Gründen für das Entfernen aus der Registry zu tun hatte: Auch eine Million Zeilen Code hätten das Modul nicht davor bewahrt, aus rechtlichen Gründen entfernt werden zu müssen.

Der Haken an der Geschichte ist, dass das Gleiche auf jeder anderen Plattform ebenfalls passieren kann: Wer garantiert, dass eine Drittkomponente für .NET auch morgen noch zur Verfügung stehen wird?



npm: Der Ort an dem man Pakete findet und veröffentlicht (Bild 1)

Gerade in einer solchen Situation ist von Glück zu sprechen, wenn ein Modul als Open Source vorliegt. Dank der permissiven Lizenz von left-pad war es nämlich ein Leichtes, das Problem rasch zu beheben und einen Fork des Moduls zu veröffentlichen, dieses Mal auf einer rechtlich sicheren Basis. Hätte es sich um ein proprietär lizenziertes Modul von einem Unternehmen gehandelt, wie es bei vielen .NET-Komponenten der Fall ist, wäre der Ausfall weitaus gravierender und schmerzvoller gewesen.

Hinzu kommt, dass selbst ein Modul wie left-pad, das aus gerade einmal 11 Zeilen Code besteht, getestet werden muss. Merke: Auch in wenigen Zeilen Code können gravierende Fehler stecken, die es zu finden und auszumerzen gilt. Dazu ist es hilfreich, Tests zu schreiben. Auch eine Dokumentation schadet nicht. Nimmt man nur diese drei Punkte zusammen (Code, Tests und Dokumentation), geht es auf einmal nicht mehr nur um 11 Zeilen Code, sondern um 100 Zeilen Code und 50 Zeilen Dokumentation.

Die Frage, die sich stellt, lautet daher: Will man auf die Tests und die Dokumentation wirklich verzichten, und den dann – hoffentlich – korrekt lauffähigen Code mehrfach duplizieren, um ihn in diverse Projekte zu verteilen? Oder ist ein npm-Modul, das genau diese eine Aufgabe erfüllt und das zentral weiterentwickelt und gepflegt wird, nicht eventuell doch die bessere Lösung?

Wie auch immer die individuelle Antwort auf diese Frage lautet, die Community um Node.js hat hierzu eine klare Meinung: Kleine Module sind gute Module, und vor allem besser als kein Modul.

Große Anwendungen schreiben

Wer lauter kleine Module einbindet, kann sich zudem viel gezielter aussuchen, was eingebunden wird, und auf diesem Weg die Größe der Anwendung reduzieren. Das mag in Zeiten von etlichen GByte RAM lächerlich wirken, doch wird der meiste JavaScript-Code immer noch geschrieben, um ihn im Webbrowser auszuführen. Dazu muss er jedoch erst einmal übertragen werden. Jedes Byte, das hierbei eingespart werden kann, beschleunigt den Seitenaufruf, und damit die Position in den Suchergebnissen und die Zufriedenheit der Besucher.

Node.js ist zwar zunächst eine Technologie zum Ausführen von JavaScript auf dem Server, es bildet inzwischen aber auch für clientseitige JavaScript-Projekte die technologische Grundlage, und die meisten npm-Module lassen sich auf dem Server und im Client gleichermaßen nutzen. Daher ist das Argument durchaus zugkräftig. Apropos: Alleine die Tatsache, dass Node.js als Programmiersprache gerade auf JavaScript setzt, ermöglicht dieses Vorgehen überhaupt erst. Eine in C# geschriebene Komponente kann man zwar per Nuget veröffentlichen und verteilen, aber niemals nativ im Webbrowser ausführen. Der gemeinsame Technologiestack ist einer der großen Vorteile von Node.js gegenüber .NET.

Das Verwenden von kleinen Modulen und die Modularisierung von Anwendungen ist übrigens zugleich auch die Antwort auf die Frage, wie man denn große Anwendungen mit beispielsweise 15 Millionen Zeilen Code in Node.js schreiben und warten könne. Die einfache Antwort lautet, dass man das in Node.js nicht macht. Wächst ein Modul tatsächlich auf 15 Millionen Zeilen Code an, ist das ein deutliches Zeichen für schlechtes Design – was im Übrigen für jeden Technologiestack gilt.

Jede Aufgabe, die mit 15 Millionen Zeilen Code gelöst werden kann, lässt sich in Unteraufgaben zerteilen. Das Gleiche gilt für jede dieser Unteraufgaben. Am Ende erhält man eine Vielzahl von kleinen, in sich überschaubaren Aufgaben, die zu einem großen Teil sehr generisch sind.

Insbesondere diese Aufgaben lassen sich hervorragend als npm-Module verpacken, die dann von anderen npm-Modulen verwendet werden können. Anstatt also einen monolithischen Klotz mit 15 Millionen Zeilen Code zu bauen, zerlegt man die Anwendung in zahlreiche kleine npm-Module, die alle einen speziellen Zweck erfüllen und jeweils auf ebendiesen Zweck spezialisiert sind.

Die hohe Menge an Zeilen in einem Projekt wird also dadurch verringert, dass man das Projekt in kleinere Projekte zerteilt. Dabei muss gar nicht der Überblick über alle Module zu jedem Zeitpunkt gegeben sein, da man sich auf einer Ebene stets nur mit den Modulen beschäftigen muss, die direkte Abhängigkeiten haben. Auf dem Weg lässt sich ein gemeinsames Abstraktionsniveau viel einfacher erreichen, als wenn man versuchen würde, das ausschließlich auf Klassen- und Methodenbasis durchzusetzen.

Abhängigkeiten verwalten

Wichtig dafür ist aber, sämtliche Abhängigkeiten klar zu deklarieren und sie vor allem auch eindeutig zu versionieren. Zu dem Zweck greift Node.js auf ein Schema namens Semantic Versioning [6] zurück, das sich auch in anderen Technologien mehr und mehr durchsetzt. Hinterlegt werden die Abhängigkeiten sowie deren Versionen in der Datei *package.json*, die – zumindest für kurze Zeit – als Vorlage für die *project.json* von .NET Core gedient hat. Microsoft hat sich dann allerdings aus Gründen der Abwärtskompatibilität dagegen entschieden, das Dateiformat weiter zu verwenden, und setzt daher anstelle des leichtgewichtigeren Formats wieder auf die SLN- und CSPROJ-Dateien.

Auch bei der Installation der Abhängigkeiten zeigt sich bei Node.js wiederum die Ausrichtung auf die Cloud. Dinge wie der Global Assembly Cache (GAC) existieren gar nicht erst. Stattdessen werden alle Abhängigkeiten automatisch in den Kontext der lokalen Anwendung installiert, sodass jeglicher Code lokal vorliegt und sich eine Anwendung einfach dadurch verschieben oder kopieren lässt, indem man ihr Verzeichnis verschiebt oder kopiert.

Die einfache Portabilität der Anwendung ist nur deshalb möglich, weil auf systemweite Einstellungen oder global installierten Code verzichtet wird. Das gilt sogar für Node.js selbst, das sich problemlos mit einem einfachen Benutzerkonto installieren lässt, ohne dass man dazu administrative Berechtigungen bräuchte. Auf diese Weise wird das Aufsetzen eines neuen Servers zu einem Kinderspiel.

Vergleicht man die genannten Punkte, erkennt man ein wiederkehrendes Muster, auf das sich Node.js immer wie- ▶

der zurückführen lässt. Die Laufzeitumgebung legt großen Wert auf Leichtgewichtigkeit.

Dinge dürfen nicht kompliziert sein, und die Community trägt ihren Teil dazu bei, passende Erweiterungen und Module zu entwickeln, die diesem Gedanken Rechnung tragen. Insgesamt folgt Node.js damit dem gleichen Motto wie auch JavaScript: „Make simple things simple, and complex things possible."

Es ist nicht erforderlich, einem Einsteiger zuerst zu erklären, was alleine die Schlüsselwörter *public*, *static*, *void*, *using*, *namespace* und *class* bedeuten. Stattdessen genügt eine simple Zeile Code:

```
console.log('Hallo Welt!');
```

Das alles ist aber erforderlich, wenn man jemandem die Sprache C# erklären will:

```
using System;

namespace HelloWorld
{
  public class Program
  {
    public static void Main()
    {
      Console.WriteLine("Hallo Welt!");
    }
  }
}
```

Leichtigkeit siegt

Alleine dieses Beispiel genügt, um zu zeigen, wie weit C# von der Leichtigkeit von JavaScript entfernt ist, und wie aufwendig das Schreiben von Software mit .NET im Vergleich zu Node.js ist.

Natürlich handelt es sich dabei nur um ein Hallo-Welt-Programm, aber wenn der Faktor hier schon 1:12 beträgt, wie ist das dann erst bei richtigen Anwendungen? Wie viel unnötiger Boilerplate-Code lässt sich hier einsparen?

Um wie viel einfacher ist eine Anwendung zu warten, die mit 1,25 Millionen Zeilen Code auskommt statt mit 15 Millionen, einfach nur dadurch, dass die Sprache Dinge weniger kompliziert macht?

Selbstverständlich hinkt der Vergleich, und die Zahlen kann man nicht einfach so hochrechnen. Die grundlegende Aussage bleibt aber, dass zur Lösung des gleichen Problems in Node.js weitaus weniger Zeilen Code erforderlich sind als in .NET, ohne dass die Lesbarkeit oder Verständlichkeit darunter leiden würde.

Wer diese Effizienz einmal kennen- und wertschätzen gelernt hat, hat schon viel von der Denkart von JavaScript und Node.js verstanden: Dinge müssen einfach sein. Sind sie kompliziert, hat man noch nicht lange genug nachgedacht.

Das zieht sich wie ein roter Faden überall durch, sei es bei der Verteilung von Logik über Prozessor- und Maschinengrenzen, sei es bei der Verwaltung von Abhängigkeiten, oder bei der Sprache selbst: Stets ist Einfachheit ein hohes und erstrebenswertes Gut.

Gewöhnt man sich daran, beginnt man über kurz oder lang, seine eigene Software auf die gleiche Art zu entwickeln: Warum eine Klasse exportieren, wenn eine Funktion genügt? Warum aufwendig eine Factory schreiben, wenn eine Funktionen höherer Ordnung genügt? Warum mit vielen Umständen eine Singleton-Klasse aufsetzen, wenn man auch einfach ein einzelnes Objekt exportieren kann?

Fazit

Hat man sich daran einmal gewöhnt, kommt einem C# und .NET ungemein umständlich und schwerfällig vor. Wird das dann noch kombiniert mit der Aussage, dass man ja mit C# und .NET aber im ernsthaften Unternehmenskontext unterwegs sei und nicht nur im Web, zeigt sich daran, wie wenig Ahnung von Softwareentwicklung eigentlich gegeben ist: Wer glaubt, schwierige Probleme seien nur kompliziert zu lösen, hat vieles nicht verstanden.

Wer das aus Überzeugung vertritt, wird aber auch nie aus dieser komplizierten Welt ausbrechen können und die anderen stets nur verlachen.

Am Ende bleibt derjenige dann als Dinosaurier zurück, der es nie gewagt hat, aus seiner Komfortzone auszubrechen und neuen Technologien eine Chance zu geben. Im Moment mag das Node.js sein, morgen wird es etwas anderes sein. Gefährlich ist aber in jedem Fall, sich dem Neuen zu verschließen, weil man vermeintlich ach so komplizierte Probleme zu lösen habe. Dann kann das böse Erwachen deutlich schneller kommen, als man heute glauben mag.

Daher ist es wichtig, anders zu denken, und sich für Neues zu öffnen. Das muss dann, wie gesagt, noch nicht einmal zwingend Node.js sein, aber Node.js hilft enorm, den über 15 Jahre angesammelten Ballast von .NET zu hinterfragen und ihn gegebenenfalls abzuwerfen. ■

[1] Golo Roden, Oft kopiert, doch nie erreicht, dotnetpro 4/2016, S. 44 ff, www.dotnetpro.de/A1604Lisp
[2] Golo Roden, Stack, Heap & GC, dotnetpro 5/2017, S. 86 ff, www.dotnetpro.de/A1705StackHeap
[3] API-Dokumentation von Node.js, www.dotnetpro.de/SL1708NodePhilosophie1
[4] npm, https://www.npmjs.com
[5] left-pad, www.dotnetpro.de/SL1708NodePhilosophie2
[6] Semantic Versioning, www.dotnetpro.de/SL1708NodePhilosophie3

Golo Roden
ist Gründer, CTO und Geschäftsführer von the native web, einem auf native Webtechnologien spezialisierten Unternehmen. Er ist zweifacher MVP für C# und Autor von „Node.js & Co.", dem ersten deutschsprachigen Buch zu Node.js.
www.thenativeweb.io

dnpCode A1708NodePhilosophie