

JavaTMmagazin

Java • Architekturen • Web • Agile

www.javamagazin.de

AngularJS

Die Fallstricke ▶ 70

TensorFlow

Der Weg des maschinellen Lernens ▶ 11



It's a Match!

Finde die NoSQL-Datenbank, die zu dir passt

andrena

OBJECTS

Experts in agile software engineering

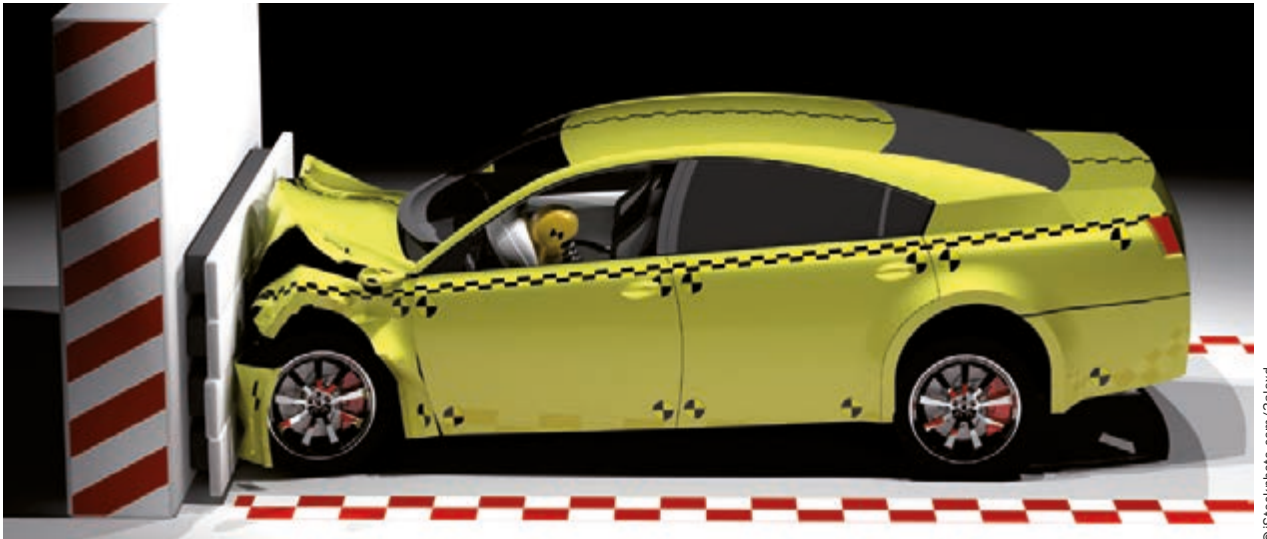
Sonderdruck für

AngularJS Projekte schnell an die Wand fahren

Technische Schulden
managen ▶ 30

Mobile Apps mit eigenem
Backend ▶ 54

AngularJS-Performance
verbessern ▶ 78



© iStockphoto.com/3alexid

Die Fallstricke von AngularJS

Projekte schnell an die Wand fahren

AngularJS ist ein tolles Framework, um eine dynamische Webanwendung aufzubauen. Die Einstiegshürde ist niedrig, Ergebnisse sind in kurzer Zeit zu erzielen. Dennoch ist Vorsicht geboten: So schnell eine Anwendung hochgezogen werden kann, so schnell versumpft der Code in überladenen Controllern und komplizierten Abhängigkeiten. Dieser Artikel analysiert häufige Fehler und gibt praktische Empfehlungen, um Projekte so aufzubauen, dass die Wartbarkeit langfristig erhalten bleibt.

von Rouven Röhrig und Michael Zugelder

Die Definition klingt unspektakulär: AngularJS ist ein JavaScript-Framework für Single-Page-Applikationen. Der dazugehörige Vorteils katalog liest sich indessen beachtlich: AngularJS unterstützt das bidirektionale Data Binding, ermöglicht „expressive“, also sprechendes HTML, und hilft beim Strukturieren, etwa durch das Model-View-ViewModel-Pattern (MVVM). Außerdem wurde schon beim Entwurf auf gute Testbarkeit geachtet. Es benötigt weder serverseitige Unterstützung noch einen Compilerschritt; also kann man einfach mit einem Texteditor ans Werk gehen. Damit bietet Angu-

larJS einen leichten Einstieg für alle, die mit HTML und JavaScript vertraut sind.

AngularJS ist ein JavaScript-Framework, also stellt sich die Frage, wie AngularJS mit einigen der typischen Herausforderungen in Enterprise-JavaScript-Projekten umgeht. Damit meinen wir beispielsweise den Umstand, dass es keine natürliche Strukturierung der Dateien gibt und der Zusammenhang zwischen den Dateien nur lose ist. Darüber hinaus beziehen wir uns auf die Abwesenheit eines Compilers, die dynamische Typisierung und den Umstand, dass JavaScript kein Klassenkonzept aufweist.

Nun ist JavaScript eine Implementierung des ECMAScript-Standards. Einige der genannten Punkte

adressiert bereits dessen sechste Version – ECMAScript 2015, der veraltet auch noch ES6 genannt wird. Zukünftig soll der Standard jährlich aktualisiert werden. Ist damit also alles gut? Im Moment zumindest nicht. Denn der aktuell gängige Standard, also derjenige, den die aktuellen Browser verstehen, ist ES5, nicht ES2015. Wer also von der Möglichkeit Gebrauch macht, TypeScript und ES2015 einzusetzen, kommt häufig nicht umhin, beide in ES5 zu übersetzen. Noch ist ES5 in der Entwicklung viel verbreiteter als TypeScript und ES2015. Solange das gilt, bleiben die genannten Herausforderungen bestehen. Welche Möglichkeiten gibt es also, sie zu meistern?

Modularisierung schützt vor Fehlern

Zunächst zu den beiden ersten Punkten, der fehlenden natürlichen Strukturierung und dem losen Zusammenhang zwischen den Daten: Im Gegensatz zu Java mit seinen festen Konventionen, z. B. für den Zusammenhang zwischen Datei- und Klassennamen, verfügt JavaScript (JS) nicht über Komponenten, Namespaces oder Packages. Entwickler können in JS die Dateinamen individuell festlegen und die Dateien nahezu beliebig füllen, sei es mit Funktionen oder Variablen. Diese Individualität kann es schwer machen, später nachzuvollziehen, welchen Inhalt eine Datei hat und wohin sie gehört. Auch der lose Zusammenhang der Dateien kann schnell zu Fehlermeldungen führen: Da es in JavaScript nicht zwingend erforderlich ist, Abhängigkeiten vorab zu definieren, müssen alle benötigten Dateien im Voraus ausgelesen sein.

Modularisierung kann gegen Fehler aufgrund ungeklärter Abhängigkeiten helfen, allerdings wird sie auch gerne falsch verstanden. Controller stellen in AngularJS Daten und Methoden zur Verfügung und sonst nichts. Laut den AngularJS-FAQ [1] ist beispielsweise ein Controller nicht dazu gedacht, jQuery-Code zu kapseln, der das Document Object Model (DOM) manipuliert. Es gibt einige Diskussionen darüber, ob AngularJS eher

ein Model-View-Controller-Pattern oder ein MVVM-Pattern hat. Nach unserer Meinung entspricht Angular einem MVVM-Pattern, legt das Model selbst aber nicht fest. Dieser Umstand begünstigt vermutlich die Tendenz, Controller zu überfrachten. Dessen ungeachtet gehen beide Pattern davon aus, dass die einzelnen Bestandteile weitgehend unabhängig voneinander sein sollten. Unserer Ansicht nach sollte das Model überhaupt nicht vom ViewModel oder der View abhängen. Anders gesagt: Es sollte auch ohne beide funktionieren. Nach unserer Erfahrung lässt sich das Data Binding dann effektiv nutzen, wenn man die Daten an exakt zwei Stellen konvertiert: Beim Laden vom und dann wieder beim Wegschicken an den Server. Damit ergibt sich schematisch folgender Ablauf:

Daten vom Server laden → Konvertierung zum ViewModel → Data-Binding → beim Absenden eine Konvertierung → Sendung an den Server

Nicht empfehlenswert ist es dagegen, die Daten vom Server zu laden und zu versuchen, sie direkt an die Oberfläche zu binden, von wo aus sie dann ohne Konvertierung zurück an den Server gehen. Denn dafür sind die Datenformate meist zu unterschiedlich. Anstatt also das Data Binding effektiv zu nutzen, kämpft man in diesem Fall damit, Datenformate sehr nah an der Oberfläche hin und her zu konvertieren. Das ist kein Gewinn. Die strikte Trennung von Repräsentations- und Businesslogik ist einer der Grundbausteine in AngularJS.

JavaScript-Build-Prozesse automatisieren

Grundsätzlich spricht allein die Vielzahl der typischerweise notwendigen Schritte im Build-Prozess dafür, soweit möglich zu automatisieren. Gemeint sind zum Beispiel die Minifizierung oder Verkleinerung des Sourcecodes oder dessen Obfuskation, also Vernebelung, um Reverse Engineering zu erschweren. Dazu kommen das Zusammenführen der Dateien (Bundling), um die Zahl der benötigten HTTP-Anfragen zu senken, und weitere obligatorische Aufgaben, etwa die Ausführung der Tests,

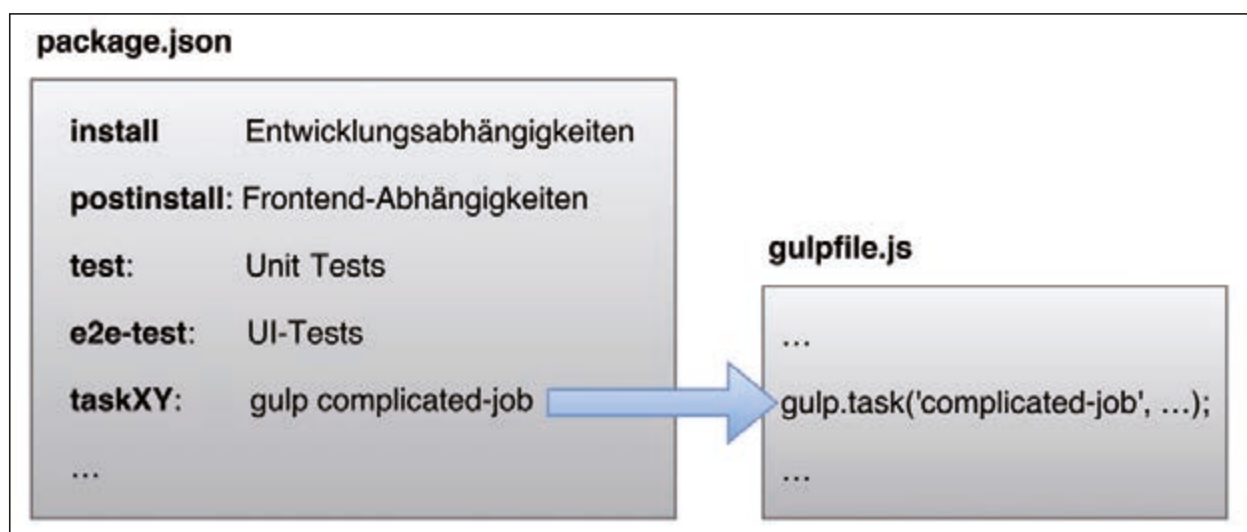


Abb. 1: npm als einziger Einstiegspunkt im Build-Prozess

Die Empfehlung lautet, im Build-Prozess Node.js und npm für die Automatisierung und Entwicklungsabhängigkeiten einzusetzen – vom ersten Moment an.

die Verwaltung von externen Abhängigkeiten, die Prüfung der Dateien auf Syntaxfehler oder Fehler des Linters. Hier sei nochmals auf die Abwesenheit eines Compilers hingewiesen, die es besonders wichtig macht, Fehler mittels Linting und automatisierten Tests früh aufzuspüren. Denn sonst tauchen sie erst zur Laufzeit auf.

Obwohl die Automatisierung klare Vorteile bringt – bessere Testbarkeit, höhere Produktivität, Kostenersparnis bei der Fehlerkorrektur – wird sie gerne vertagt, getreu dem Motto „Wir fangen erst einmal an und automatisieren später“. Ohne Automatisierung müssen Entwickler jedoch Abhängigkeiten gegenüber Dritten manuell pflegen. Tests, die schnelles Feedback geben, existieren nicht, genauso wenig wie ein automatisierter Build-Prozess. Obendrein muss der Build-Server ohne Build- und Testergebnis laufen. Generell besteht die Gefahr, dass aus dem „wir automatisieren später“ ein „wir automatisieren gar nicht“ wird. Deshalb lautet die Empfehlung dahingehend, im Build-Prozess Node.js und npm für die Automatisierung und Entwicklungsabhängigkeiten einzusetzen – vom ersten Moment an. npm als Command-Line-Tool sollte der einzige Einstiegspunkt für das Abhängigkeitsmanagement und die Installation, für den Build-Prozess sowie die Unit und die GUI-Tests sein. Bei Bedarf lässt sich der npm-Einsatz mit Grunt oder gulp ergänzen. Allerdings ist npm besser geeignet, die Komplexität zu senken. Denn Aufgaben wie die Ausführung eines Unit oder GUI-Tests kommen mit einem npm-Einzeiler aus, wohingegen Grunt oder gulp mehrere Zeilen JavaScript benötigen. Grunt-Files in diverse Dateien zu teilen, schafft zwar Abhilfe, aber eben für ein Problem, das mit npm gar nicht erst entstanden wäre.

npm einzusetzen bedeutet, beliebige JS-Projekte zu standardisieren und autark zu machen. Voraussetzung sind in der Regel lediglich Node.js, Git und die Ausführung der folgenden Schritte:

1. *git clone ...*
2. *npm install*: Dieser Befehl installiert alle Abhängigkeiten lokal.
3. *npm test*: Führt Tests aus.
4. *npm start*: Dieser Schritt ist optional, er startet den Webserver.

In der Folge sinkt die Komplexität massiv, weil diverse umfangreiche Arbeitsschritte stark vereinfacht werden. Beispiel Abhängigkeiten: Sie müssen zwar immer noch definiert werden, das System installiert sie jedoch

automatisch und erspart damit manuelles Laden und Einpflegen. Dadurch ergibt sich eine zentrale Stelle, an der alle Versionen definiert werden. Ein Umstand, der nebenbei auch die Update- und Upgrademoral fördert. Die Automatisierung senkt also deutlich das Risiko, viele veraltete Versionen in seinem Projekt zu haben. Nun steht der Begriff Automatisierung normalerweise in Zusammenhang mit den Tests. So auch hier: Wie verhält es sich mit den Unit und GUI-Tests?

Unit Tests sind auch fürs Frontend wichtig

Grundsätzlich begünstigt AngularJS das Testing, beispielsweise aufgrund der strikten Trennung von DOM und JavaScript und über die Dependency Injection, die gerade Unit Tests deutlich vereinfacht. Trotzdem ist die Überzeugung weit verbreitet, wonach Unit Tests für das Frontend eher keine Option sind. Also unterbleiben die Tests, mit der unschönen Folge, dass Fehler erst spät entdeckt werden. Die de facto praktizierte Entwicklung per „Trial and Error“ führt zu Code, der sich weder gut warten noch testen lässt – und damit zu einer langsamen und riskanten Art des Entwickelns. Deswegen gilt: Egal ob Front- oder Backend, Testbarkeit ist eine, wenn nicht die, Metrik für Qualität. Denn solange es möglich ist, gute Tests zu schreiben, ist der Code normalerweise gut isoliert. Übersetzt in AngularJS heißt das, dass sich Controller nur sinnvoll testen lassen, wenn die Logik sauber isoliert wurde. Explizit geht es hier um die strikte Trennung von View und Logik in Controller und Direktiven. Außerdem sei hier erneut die Unabhängigkeit des Controllers vom DOM betont.

Geeignete Werkzeuge für das Unit Testing sind Jasmine oder Mocha als Frameworks. Als Test-Runner empfiehlt sich Karma, das Testausführungen in vielen Browsern erlaubt und über zahlreiche Erweiterungen verfügt. Mit einer davon kann man auch die Codeabdeckung messen und das Ergebnis mittels eines weiteren Plug-ins für CI/Build-Systeme/Server aufbereiten. Der Charme beim Karma-Einsatz ist der hohe Automatisierungsgrad: npm führt Karma aus und Karma führt Jasmine-Tests aus, misst die Codeabdeckung und generiert das Ergebnis.

Häufig möchten Entwickler die Ergebnisse der Unit Tests möglichst schnell haben, um die Feedbackschleife kurz zu halten. Viele Projekte haben einen größeren Build-Prozess, z. B. Minification oder Sass Build. Muss der Code gebaut werden, ehe er getestet wird, ist das akzeptabel für den Build-Server, aber eben schlecht für Test-driven Development und schnelles Feedback. Um

trotzdem die Tests schnell ausführen zu können, muss der Build-Prozess gut modularisiert sein und sollte keine unnötigen Aktionen ausführen. Wird z. B. nur eine JavaScript-Datei geändert, sollte etwa der Linter nicht alle JavaScript-Dateien des gesamten Projekts neu analysieren. Dadurch kann das Projekt wachsen, ohne dass der jeweilige Build-Schritt den Entwicklungsprozess zunehmend verlangsamt.

GUI-Tests mit Page Objects

„Spät ist besser als nie“ dürfte auf die Entdeckung von GUI-Fehlern nicht zutreffen. Häufig offenbart die späte Automatisierung schlecht testbares HTML – und zieht dann massiven Aufwand nach sich. Was also stattdessen tun? GUI-Tests ab der ersten Seite, obwohl sie selbst aufwendig sind und schnell veralten? Die Antwort ist ein klares „Jein“. Unserer Erfahrung nach reicht es, sich auf wichtige Use Cases zu konzentrieren und Smoke-Tests zu erstellen: Erste grundlegende Probeläufe einer Software, die simple Probleme offenlegen soll, deren mögliche Auswirkungen allerdings ernst genug sind, um das Programm nochmals zu überarbeiten. Außerdem empfehlen wir zwei grundlegende Dinge: Erstens HTML-Komponenten eine ID zuzuweisen, weil Klassen fehleranfälliger sind. Das ist nicht ganz einfach, da Komponenten wiederverwertbar sein sollten und dennoch eindeutige IDs benötigen. Umso wichtiger ist es, sich darüber von Anfang an Gedanken zu machen. Zweitens Page Objects zu verwenden, egal, ob man mit Protractor, Selenium, Java oder JavaScript arbeitet. Das Page-Objects-Pattern versucht, GUI-Tests wartbarer zu machen, indem Seitenzugriffe – z. B. das Anklicken eines Buttons oder das Ausfüllen eines Felds – in Page Objects gekapselt werden. In diesem programmatischen Ansatz repräsentiert jedes Page Object eine logische Seite. Technisch hat eine Single-Page-Applikation natürlich nur eine Seite, aus Sicht der Nutzer aber quasi mehrere, beispielsweise für das Log-in, Artikeldetails oder Suchergebnisse. Ein Page Object für eine Log-in-Seite besitzt dann Methoden, um die Seiten aufzurufen, den Benutzernamen und das Passwort einzugeben sowie den Log-in-Button zu klicken. Jeder Test, der durch das Log-in muss, verwendet dann das Page Object, um diese Operationen auszuführen. Ändert sich nun die Log-in-Seite, etwa durch einen neuen Log-in-Button, muss nur noch genau ein Test angepasst werden. Eine weitere Empfehlung lautet dahingehend, dass Page Objects ihre benachbarten Page Objects kennen, beispielsweise liefert dann die Log-in-Page das Dashboard Page Object, sobald der Log-in-Vorgang erfolgreich war.

Page Objects repräsentieren eine Webseite in zweierlei Richtung: Dem Entwickler zeigen sie die Dienste oder Funktionalität, die eine spezielle Seite anbietet. Hinsichtlich der Webseite wissen sie, wie sie auf die Elemente zugreifen, idealerweise über deren IDs. Für das Beispiel eines E-Mail-Systems würde das bedeuten, dass die Services umfassen, eine neue Mail zu schreiben, Mails zu lesen oder die Betreffzeilen neuer Mails anzuzeigen. Der Test

würde sich also darauf konzentrieren, ob diese Services wunschgemäß funktionieren. Wie sie implementiert werden, ist nicht Gegenstand des Tests. Page Objects kennen auch die Page Objects verlinkter Seiten und wissen, wie man auf die jeweilige Seite zugreift. Damit reduzieren sie die Menge an dupliziertem Code und stellen sicher, dass Änderungen in der Benutzeroberfläche nur an einer Stelle abgebildet werden müssen (Listing 1). Mangelnde (Test-) Automatisierung ist ein großer Fallstrick, allerdings bei Weitem nicht der Einzige. Nachfolgend also vier weitere – und wie sie sich umgehen lassen.

Keine einheitlichen Codekonventionen

Codekonventionen vermindern das Risiko von Flüchtigkeitsfehlern und einfachen Programmierfehlern. Um sie durchzusetzen, empfehlen wir den Einsatz eines Linters. Die statische Codeanalyse überprüft Codekonventionen und findet simple Fehler. Bekannte JavaScript-Linter sind: JSLint (feste Regeln), JSHint (konfigurierbare Regeln, Fork von JSLint) und ESLint (erweiterbare Regeln, Plug-in-basiert). Gute IDEs zeigen die Fehler des Linters direkt im Quellcode an.

IDEs, die JavaScript unzureichend unterstützen

„Wir haben schon immer mit IDE XY entwickelt“ ist ein typischer Satz, der dann problematisch wird, wenn der JavaScript-Support, den XY bietet, schlecht bis nicht existent ist. Konkret bedeutet das, dass beispielsweise keine sinnvolle Autovervollständigung vorhanden ist. Automatisierte Refactorings wie Rename oder Extract Method gibt es auch nicht, genauso wenig wie die Möglichkeit zur Linter-Integration. Zur Deklaration oder

Listing 1

```
class InboxPage {

  constructor() {
    this.title = element(by.css("#title"));
    this.emails = element.all(by.css("#email li"));
    this.newMailButton = element(by.css("#email li button.newMail"));
  }

  open() {
    return browser.get("/inbox");
  }

  writeNewMail() {
    return this.newMailButton.click()
      .then(() => new WriteMailPage());
  }

  subjectLine(index) {
    return this.emails.get(index).getText();
  }

  readMail(index) {
    return this.emails.get(index).click()
      .then(() => new ReadMailPage());
  }
}
```

Definition von Variablen zu springen erweist sich – vorsichtig ausgedrückt – als schwierige Aufgabe. Genauer gesagt ist es gar nicht möglich, das Problem mittels statischer Analyse zu lösen, solange Definitionen und Deklarationen beliebig sein können. Es liegt eben kein Klassenkonzept vor, die Strukturierung fehlt und der Zusammenhang zwischen den Dateien ist nur lose. Objekte und Methoden können an beliebigen Stellen erzeugt werden. Außerdem können sich Objekte auch strukturell ohne große Hürden ändern, und das zu jeder Zeit. Bei prototypischer Vererbung ist dafür nicht einmal ein Verweis auf das spezifische Objekt erforderlich. Etwa dann,

wenn durch das Ändern des Object Prototype plötzlich neue Felder auf allen Objekten auftauchen. Die wenig schönen Folgen der mangelhaften Unterstützung bestehen in der niedrigen Geschwindigkeit der Entwicklung und dem Risiko, vermeidbare Fehler zu machen.

Wir empfehlen daher, eine IDE mit guter JavaScript-Integration zu wählen. Von den kommerziellen Angeboten sind das beispielsweise Visual Studio oder IntelliJ/WebStorm. WebStorm ermöglicht unter anderem automatisiertes Refactoring, die Build-Tool-Integration, die Test-Tool-Integration und die Integration eines Linters. Auch nicht kommerzielle, Plug-in-basierte IDEs bieten eine gute Unterstützung, etwa Brackets, Atom oder Sublime Text. Oft amortisieren sich Aufwand und Kosten des IDE-Wechsels schnell, schon allein aufgrund der signifikant steigenden Entwicklungsgeschwindigkeit.

Listing 2: emailBlock.directive.js

```
class EmailBlockController {
  constructor(emailService) {
    this.emailService = emailService;
  }
  get subject() {
    return this.email.subject;
  }
  get text() {
    return this.email.text;
  }
  reply() {
    this.emailService.replyTo(this.email);
  }
  delete() {
    this.emailService.delete(this.email);
  }
}

angular
  .module("beispiel")
  .directive("emailBlock", () => ({
    scope: {
      email: "="
    },
    controller: EmailBlockController,
    controllerAs: "emailBlock",
    bindToController: true,
    templateUrl: "emailBlock.template.html"
  }));
```

Listing 3: emailBlock.template.html

```
<section>
  <header>
    <h3>{{emailBlock.subject}}</h3>
    <button ng-click="emailBlock.reply()">Reply</button>
    <button ng-click="emailBlock.delete()">Delete</button>
  </header>
  <pre>{{emailBlock.text}}</pre>
</section>
```

Schlechte Codequalität

Abstriche an der Codequalität sind dagegen ein schlechter Weg, um die Entwicklung zu beschleunigen. Code-Duplikationen sind ein typisches Beispiel für unsauberen Code, sie machen Änderungen aufwendig, weil an vielen Stellen nachgebessert werden muss, bereits gefixte Fehler plötzlich wieder auftauchen oder Features nicht an allen Stellen implementiert sind. In AngularJS-Anwendungen treten Codeduplikationen primär innerhalb von HTML oder in den Controllern auf.

Wie gut sich Unit Tests für Controller schreiben lassen, hängt meist von ihrer Größe ab. Damit Controller nicht überfrachtet werden, bieten sich verschiedene Werkzeuge an. Alle beruhen auf dem Prinzip, wiederverwendbare Teile überhaupt nicht in den Controller zu packen. Die Logik beispielsweise lässt sich in Services auslagern. Allgemein lassen sich die wiederverwendbaren Aspekte der Benutzeroberfläche in Direktiven kapseln, um Coderedundanzen zu vermeiden.

Selbst definierte Direktiven mit sinnvollen Namen sollten grundsätzlich das Mittel der Wahl sein, um AngularJS zu verwenden, anstatt nur einen Controller zu definieren und diesen anschließend über die *ng-controller*-Direktive zu verwenden. Denn sonst folgen HTML-Duplikationen oder nicht wiederverwendbare Controller. Außerdem drohen komplizierte Scope-Abhängigkeiten und die Gefahr, dass sich ähnliche oder duplizierte Controller einschleichen.

Direktiven sind, wenn sie konsequent angewendet werden, das Feature schlechthin innerhalb von AngularJS. Eine Direktive sollte normalerweise in sich abgeschlossen sein und besteht aus einem HTML-Template, dem Controller, dem Styling und idealerweise einem Satz an Unit Tests. Ihr Hauptkennzeichen besteht darin, dass sie nur ihr eigenes DOM manipuliert.

Mit dem Einsatz der Direktiven mindern Entwickler daher einen der großen Nachteile typischer JavaScript-Anwendungen ab: Gemeint ist, dass jeder Teil einer Anwendung potenziell jeden Teil der Seite beeinflussen kann. Abgeschlossene Direktiven führen dagegen zu Komponenten, die explizite Grenzen haben und

damit in großen Anwendungen kombiniert und wiederverwendet werden können. Unser Beispiel illustriert so eine Komponente, es lässt sich über `<email-block email="beispielmail"></email-block>` verwenden (Listing 2 und 3).

Fazit

AngularJS ist nach unserer Erfahrung wirklich ein tolles Framework, wenn man exakt darauf achtet, einige Voraussetzungen zu erfüllen: unter anderem ein hohes Maß an Automatisierung, testbaren Code, Modularisierung und eine strikte Trennung von Repräsentations- und Businesslogik. Damit lassen sich die typischen Herausforderungen der JavaScript-Projekte gut meistern.

Einen interessanten Ausblick bildet der Einsatz von zukünftigen ECMAScript-Features und TypeScript. Die kommenden ECMAScript-Neuerungen ermöglichen es, gängige Probleme einfacher und mit weniger Code zu lösen. TypeScripts Aushängeschild sind das Typsystem und der dazugehörige Compiler. Dieser deckt viele Fehler schon in Sekunden auf, die selbst erfahrene Entwickler sonst oft erst viele Minuten später finden. Mittels Codegenerierung ist es auch möglich, die Kommunikation mit dem Server weitgehend statisch zu verifizieren. Wird etwa auf dem Server ein Attributname geändert, listet der Compiler dann direkt die auf dem Client zu ändernden Stellen auf. Noch gibt es für den TypeScript-Einsatz allerdings einige Einschränkungen, beispielsweise den Zeitbedarf, den der Compilerschritt in Anspruch nimmt, oder die Tatsache, dass AngularJS in Kombination mit TypeScript teilweise schwierig benutzbar ist.

Angular 2 stattdessen wird direkt mit TypeScript entwickelt, was hinsichtlich der genannten Einschränkungen signifikante Verbesserungen verspricht.



Rouven Röhrig ist als Entwickler bei andrena objects ag tätig. Er hat Expertise in der Frontend- und Backend-Entwicklung. Nach mehreren Projekten mit AngularJS hat er an der Entwicklung des ASE-Kurses für JavaScript mitgewirkt und anschließend bei mehreren Kunden als Trainer und ASE-Coach gearbeitet.



Michael Zugelder ist Softwareentwickler der andrena objects ag und Technologieenthusiast. Nach einigen Jahren mit Schwerpunkt auf Desktopsoftware ist er heute hauptsächlich als Full-Stack-Webentwickler und gelegentlich als Trainer tätig.

Links & Literatur

[1] AngularJS-FAQ: <http://bit.ly/1zNtHT6>