Testing-Methoden und -Technologien im Überblick

Tests an die Front

Test-driven Development ist keine Mode, sondern professionelle Entwicklung. Anders als der Zuckerguss einer Torte ist Qualität auch kein Attribut, das nachträglich ergänzt werden kann, sondern muss von Anfang an berücksichtigt werden. Dieser Artikel soll einen Überblick über die zum Erreichen dieser Qualität nötigen Testing-Methoden und -Technologien geben.

von Karsten Sitterberg

Wir werden uns im Wesentlichen die Situation bei Anwendungen mit einem SPA-Frontend anschauen, aber auch herkömmliche, serverseitig gerenderte Anwendungen kommen nicht zu kurz. Ziel ist es, das Vorgehen von testgetriebener Softwareentwicklung zu motivieren und uns dann einen Überblick über die zur Verfügung stehenden, aktuellen Werkzeuge zu verschaffen.

Der vorliegende Artikel richtet sich an alle Java-Entwickler, die klassische Webanwendungen oder auch Anwendungen mit einem Single Page Application Frontend (im Folgenden Browseranwendungen genannt) entwickeln. Es werden verschiedene moderne Test-Frameworks unter anderem anhand von Quellcodebeispielen erläutert. Ziel ist, dass jeder Entwickler die diversen Frameworks einordnen kann und ein Gefühl dafür bekommt, wie der Einsatz der Frameworks in der Praxis aussehen kann.

Automatisierte Testinfrastruktur

Viele Entwickler stellen sich die Frage, warum sie Zeit in automatisierte Testinfrastruktur stecken sollen. Schließlich müssen die Tests nicht nur geschrieben, sondern auch gewartet werden. Für die Antwort muss man ein wenig ausholen.

An jede Anwendung werden gewisse zu erreichende Qualitätsziele, sowohl während der Entwicklung als auch in Produktion, gestellt. Eine Anwendung soll etwa fehlerfrei arbeiten, zugleich für alle Entwickler gut verständlich sein, sich durch hohe Entwicklerproduktivität auszeichnen, um neue Features erweiterbar sein und –

sofern es sich nicht um einen Prototyp handelt – eine langfristige Wartbarkeit sicherstellen. Um diese Qualitätsziele zu erreichen, muss ein gewisses initiales Investment getätigt werden: Zerlegung und Transformation des Problems in ein Anwendungsdesign, Auswahl des richtigen API und der zu verwendenden Technologien. Parallel dazu wird die Entwicklungsinfrastruktur aufgebaut, zum Beispiel Build Pipeline, SCM und Ticketsystem. Um die oben genannten Qualitätsziele zu erreichen, hat sich als Entwicklungsprozess das Test-driven Development inklusive Code-Reviews und Pair Programming bewährt. Dieses Investment zahlt sich erfahrungsgemäß recht schnell und schon bei kleinen Projekten aus. Martin Fowler hat sich dazu ebenfalls geäußert und geht in seinem Artikel „Is High Quality Software Worth the Cost?“ [1] von Wochen aus. Besonders die Vorteile von automatisierten Tests zeigen sich schnell, da damit manueller Aufwand eingespart wird und Tests frühzeitig verborgene und unerwünschte (Laufzeit-)Fehler zum Vorschein bringen können.

Generell sollte man sich in der Entwicklungsphase einer Anwendung die „Rule of Ten“ [2] in Erinnerung rufen: Je nach Projektphase kostet es das Zehnfache, Fehler in einer Anwendung zu finden und zu beheben. Wird ein Fehler etwa nicht während Design, Entwicklung oder Unit Testing, sondern erst während der System-/Integrationstestphase gefunden, kostet der Bug das Zehnfache (statt 100 Dollar bei Unit-Tests muss er in Integrationstests mit 1 000 Dollar bepreist werden). Wird der Fehler erst in Akzeptanztests gefunden, kostet er wiederum das Zehnfache mehr (10 000 Dollar), und wird er schließlich erst in Produktion gefunden, können gar 100 000 Dollar veranschlagt werden.

Im Fall eines Refactorings können wir uns dann darauf verlassen, dass durch Tests abgesicherter Code auch nach dem Refactoring fehlerfrei funktioniert. Ansonsten wären die Tests nicht fehlerfrei durchgelaufen. Hinzu kommt neue Komplexität durch SPA-Architekturen und Endgeräte.

Testpyramide

Es gibt zahlreiche Arten von Tests, die unterschiedlichen Zwecken dienen und auch unterschiedlichen Umfang haben (sollten). Diese wollen wir zunächst anhand der Testpyramide einordnen (**Abb. 1**). Die Testpyramide symbolisiert das anzustrebende Verhältnis zwischen den verschiedenen Arten, Anwendungen zu testen.

Unit-Tests sichern einzelne technische Komponenten auf Codeebene ab. Die Tests sind dabei in der Regel so aufgebaut, dass sie die Komponenten in Isolation testen. Das bedeutet, dass die Funktion einer einzelnen Klasse ohne Kollaborationsobjekte getestet wird. Daher sind Unit-Tests mit verhältnismäßig geringem Aufwand zu implementieren. Das gilt allerdings nur, wenn die Tests rechtzeitig geschrieben werden: Schreibt man die Unit-Tests erst im Nachhinein, kann es passieren, dass sich die zu testende Komponente als schlecht testbar und damit schlecht wiederverwendbar herausstellt. Um schließlich

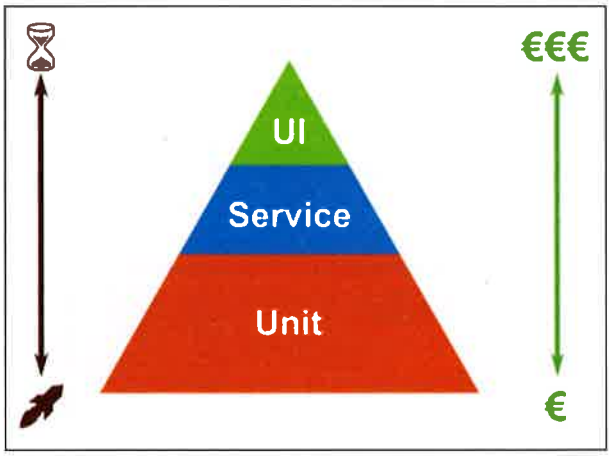

Abb. 1: Die Testpyramide

die gute Testbarkeit wiederherzustellen, wäre direkt ein Refactoring nötig. Treibt man die Entwicklung anhand der Tests voran, nennt man das Test-driven Development (TDD).

Unit-Tests lassen sich durch ihre Einfachheit relativ schnell ausführen. Da sie grundsätzliche Anwendungskomponenten bzw. -logik abtesten, sollte es von diesen Tests relativ viele geben. Dabei ist es vor allem wichtig, die richtige Menge an Tests zu schreiben. Gute Tests können zwar die Anwendungsqualität erhöhen, gleichzeitig erhöhen sie aber auch den Wartungsaufwand. Denn je mehr Tests geschrieben werden, desto mehr Tests müssen auch angepasst werden, falls sich die Anwendungslogik ändert. Ein Maßstab, der zur Messung der Güte von Tests zu Rate gezogen werden kann, ist die Testabdeckung des zu testenden Codes (Code Coverage). Erhöht ein Test diese Testabdeckung, ist das ein Indikator dafür, dass der Test gut sein könnte. Bleibt die Abdeckung gleich, kann es sein, dass bereits ein anderer Test die gleiche Funktionalität prüft und der Test somit überflüssig ist. Die Code Coverage werden wir im nächsten Teil dieser Serie näher betrachten.

In der Testpyramide liegen die Unit-Tests ganz unten und bilden das Fundament der Anwendung. Darüber befinden sich die Service- oder auch Integrationstests. Je nach Autor werden auch andere Begriffe wie End-to-End/E2E-Tests verwendet, außerdem wird teilweise noch feiner zwischen den Begriffen unterschieden, sodass die Testpyramide dann aus vier und mehr Ebenen besteht. Unter dem Begriff „Integrationstest“ fassen wir hier alle automatisierten Tests zusammen, die die Anwendung nicht auf Codeebene, sondern aus der Perspektive eines Nutzers testen. Da diese Tests den normalen Anwendungsfluss abtesten, sind sie typischerweise langsamer als Tests, die auf Codeebene testen. Da für solche Tests auch die Umsysteme und eine Datenbasis vorhanden sein müssen, ist ihr Set-up aufwendiger. Aus diesem Grund sollten nicht übermäßig viele Integrationstests geschrieben werden, in der Regel reicht es aus, sich auf die wichtigen Anwendungsfeatures zu beschränken. Sowohl Unit- als auch Integrationstests sollten automati-

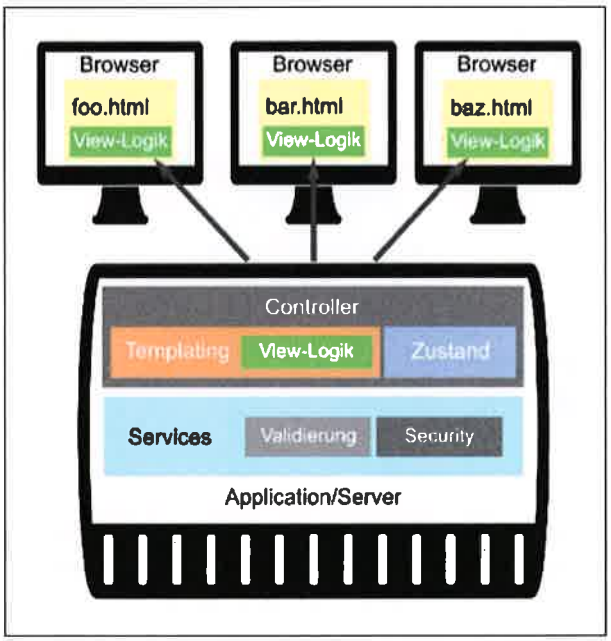

Abb. 2: Aufbau klassischer, serverseitig gerenderter Webanwendungen

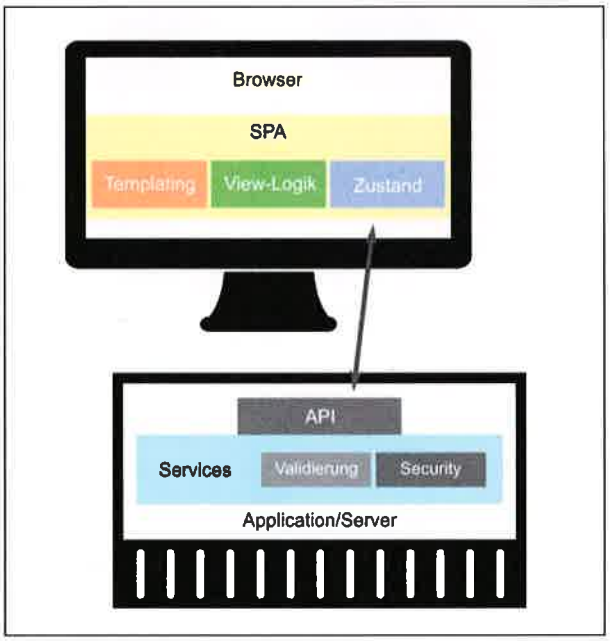

Abb. 3: Aufbau von Browseranwendungen mit einem SPA Frontend

siert in einer Continuous Integration (CI) Pipeline, etwa für jeden Commit ins SCM, ausgeführt werden.

Über den automatisierten Tests liegen letztlich noch die manuellen UI- bzw. Akzeptanztests. Da hierfür menschliche Tester nötig sind, die sich von Hand durch die Anwendung klicken, sind diese Tests sehr langsam und gleichzeitig sehr teuer. Daher sollte diese Art zu testen nur sehr sparsam eingesetzt werden, etwa bei Entwicklung oder Integration neuer Features.

Wir wollen uns die verschiedenen Werkzeuge anschauen, die zum Testen verwendet werden können. Um entscheiden zu können, wo welches Werkzeug am sinnvollsten einzusetzen ist, werden im Folgenden zunächst einmal die beiden wesentlichen Architekturen für Webanwendungen vorgestellt.

Typischer Stack bei klassischen Webanwendungen

Klassische Webanwendungen, wie in **Abbildung 2** gezeigt, arbeiten nach dem Request-Response-Verfahren: Bei jeder Aktion des Nutzers wird eine komplette Seite neu beim Server angefragt. Auf dem Server werden die Daten, die vom Client kommen, vom Controller entgegengenommen und in die internen Datentypen transformiert. Der Controller delegiert die Anfragen seinerseits weiter an die fachlichen Services, die zum einen die eingegebenen Daten validieren, zum anderen auch die Sicherheitsaspekte der Anwendung abdecken. Mit Hilfe der Services und der entgegengenommenen Daten baut der Controller für diesen Request einen Anwendungszustand auf, in dem dann auch solche Dinge wie Berechtigungen enthalten sind. Sowohl Zustand als auch Rendering-Logik liegen auf Serverseite. View-Logik ist potenziell sowohl auf Server- wie auch auf Clientseite vorhanden.

Der Controller nutzt schließlich diese Informationen für das Rendering der HTML-Seite. Für das Rendering selbst kommt eine Template-Engine zum Einsatz. Häufig eingesetzte Template-Engines sind etwa JSF oder Thymeleaf. Innerhalb der Templates wird serverseitige View-Logik, wie zum Beispiel Schleifeniterationen, Anzeige von Validierungsfehlern oder Conditional Rendering, umgesetzt. So können etwa Features, die nur einem Admin zugänglich sein sollen, für normale Nutzer ausgeblendet werden.

Die durch die Template-Engine erzeugte Seite wird dann vom Controller an den Browser des Clients weitergeleitet und dort zur Anzeige gebracht. Da im Extremfall für jeden Klick somit ein Server-Roundtrip notwendig wird, führt das dazu, dass die Anwendung sich aus Benutzersicht langsam und sperrig anfühlt. Aus dem Grund werden oft Teile der View-Logik zusätzlich auf dem Client mit JavaScript umgesetzt. Man spricht hier auch von Progressive Enhancement, wenn diese JavaScript-Implementierung optional ist und nur das Ziel der verbesserten Benutzbarkeit hat. Beispielsweise werden ein schicker, dynamischer Date Picker oder auch Animationen hier als View-Logik aufgefasst. Häufig wird für die Umsetzung einer solchen View-Logik eine JavaScript-Bibliothek wie JQuery eingesetzt. Auch Hinweise über den Status der Anwendung, etwa, um dem User mitzuteilen, welche Pflichtfelder in einem Formular er noch nicht ausgefüllt hat (einfache clientseitige Formularvalidierung), sind View-Logik. Wichtig ist natürlich, dass eine solche Logik immer auch auf dem Server implementiert sein muss, da ein JavaScript-Client keine vertrauenswürdige Umgebung darstellt.

Tests werden bei klassischen Anwendungen vor allem für die Serverlogik geschrieben. Dabei wird in der Regel für Unit-Tests des fachlichen Codes JUnit verwendet. Je nach Framework gibt es auch Testunterstützung für Controller- und Service-Tests. Im Spring Framework gibt es hierfür beispielsweise MockMVC. Damit kann

Die Startzeit einer SPA-Anwendung kann dadurch optimiert werden, dass initial nur die Module geladen werden, die notwendig sind, um die Anwendung in der jeweils aufgerufenen Ansicht zu starten.

geprüft werden, ob die passende Fehlermeldung geliefert wird, wenn ein User versucht, ein Admin-Feature aufzurufen.

Für die Ende-zu-Ende-Tests wird oft Selenium als Framework verwendet, wobei diese Tests aufgrund des hohen Erstellungs- und Wartungsaufwands häufig vernachlässigt werden. Dies ist akzeptabel, wenn kein oder wenig Progressive Enhancement genutzt wird, da somit im Browser keine weitere Logik die Anwendung verändern kann.

Typischer Stack bei Browseranwendungen

Abbildung 3 zeigt den Aufbau einer Webanwendung mit einem Single Page Application (SPA) Frontend. Diese unterscheidet sich in ihrer Aufteilung deutlich von einer klassischen Webanwendung. Bei klassischen Webanwendungen wird der Browser im Wesentlichen dafür genutzt, ganze Seiten beim Backend abzufragen. Der Client hat keine wesentliche Logik, die Intelligenz liegt auf dem Server. Stattdessen ist es bei einem SPA Frontend so, dass nur einmal, nämlich beim Starten der Anwendung, eine ganze HTML-Seite geladen wird. Diese Seite bindet dann die Anwendungslogik in Form von einer oder mehreren JavaScript-Dateien ein.

Um die Ladezeit der Anwendung zu optimieren, ist es vorteilhaft, wenn die initial zu ladenden JavaScript-Dateien so klein wie möglich sind. Daher werden SPA-Anwendungen oft in separate Module aufgeteilt. Die Startzeit der Anwendung kann dann dadurch optimiert werden, dass initial nur die Module geladen werden, die notwendig sind, um die Anwendung in der jeweils aufgerufenen Ansicht zu starten. Alle weiteren Module können zu einem späteren Zeitpunkt – spätestens, wenn sie benötigt werden – dynamisch nachgeladen werden (Lazy Loading). Das geschieht oft dann, wenn die SPA von einer View das erste Mal in eine View wechselt, die in einem anderen Modul definiert ist.

Nachdem die Anwendung einmal geladen ist, findet die Kommunikation mit dem Server, abgesehen von dynamisch geladenen Modulen, nur noch auf Basis eines reinen Datenaustauschs statt. Serverseitig wird für die Kommunikation ein Remote API bereitgestellt, etwa als HTTP JSON API. Dann erfolgt, wie auch bei einer klassischen Anwendung, die Verarbeitung im Backend mit Hilfe von Services. Ebenfalls wie bei klassischen Anwendungen werden hier auch Querschnittsaspekte wie Validierung und Security umgesetzt, selbst wenn der Client vergleichbare Funktionalität für die View umsetzt. Schließlich wird das Ergebnis dann auf eine Anfrage zurück an das SPA Frontend geschickt. Als Format hierfür hat sich JSON bewährt, da es eine nahtlose Integration mit JavaScript ermöglicht. In der SPA sorgen die neuen Daten nun für eine Zustandsaktualisierung. Zuständig für die Darstellung des neuen Zustands sind dann View-Komponenten.

Eine typische SPA-Anwendung, wie sie etwa mit Angular, Vue oder React erstellt werden kann, ist aus einzelnen Komponenten zusammengesetzt. Komponenten erfüllen zwei Aufgaben: Erstens bekommen sie einen Ausschnitt des Anwendungszustands hineingereicht und stellen ihn im Browser dar. Zweitens sind Komponenten dafür zuständig, Userinput entgegenzunehmen und zu verarbeiten. Diese Verarbeitung kann sowohl rein clientseitig erfolgen, beispielsweise durch das Triggern von View-Logik (Öffnen eines Date Picker), als auch in Form von Kommunikation mit dem Server.

Für eine Gegenüberstellungen der grundlegenden Eigenschaften von klassischen Anwendungen und Browseranwendungen siehe Tabelle 1.

Da in einer SPA jede Komponente eine eigene Einheit bildet, lassen sie sich gut isoliert durch entsprechende Unit-Tests absichern. In der Regel ist die Anwendung in JavaScript bzw. TypeScript geschrieben, so bietet es sich an, die Komponenten auch mit entsprechenden JavaScript-/TypeScript-Test-Frameworks zu testen. Hierfür werden gerne Jasmine mit Karma oder auch Jest als Unit-Test-Frameworks genutzt. Komponententests decken dabei zwei wesentliche Felder ab: Zum einen muss sichergestellt werden, dass die Komponente die korrekten HTML-Elemente rendert, zum anderen muss

	Klassische Anwendung	Browseranwendung
Templating	Serverside (JSP, JSF, Thymeleaf, ...)	Angular/React/Vue
Zustand	Server	Client/Browser
Kommunikation	Vollständige Seiten	Daten (HTTP JSON, WebSocket, ...)
View-Logik/Komfortfunktionen	JQuery/Server	Angular/React/Vue

Tabelle 1: Vergleich der grundlegenden Eigenschaften von klassischen und Browseranwendungen

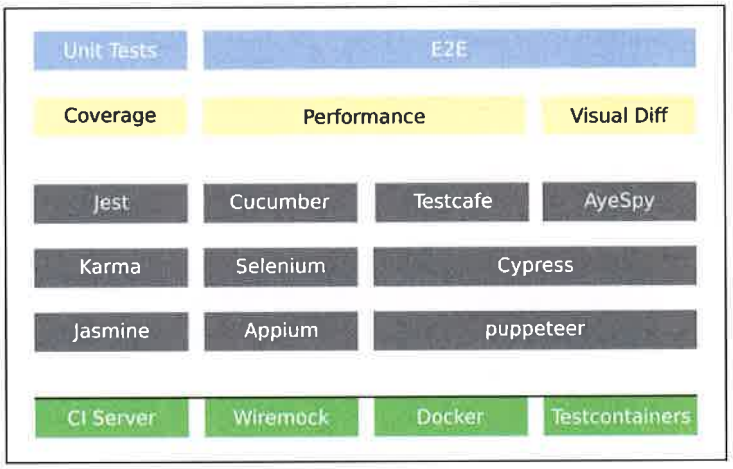

Abb. 4: Ausschnitt aus dem Testuniversum

die Komponente bei Nutzerinteraktion die korrekte Verarbeitungslogik aufrufen. Für den Serverteil werden entsprechend einer klassischen Anwendung separate Unit-Tests z. B. mit JUnit als Teil der Backend-Anwendung geschrieben.

Im Gegensatz zu klassischen Anwendungen ist es bei Anwendungen mit SPA Frontend deutlich wichtiger, in E2E-Tests zu investieren. Das liegt unter anderem daran, dass es sich bei solchen Browseranwendungen um verteilte Anwendungen handelt: Sowohl Client als auch Server enthalten nicht zu vernachlässigende Logik und sind lediglich durch ein Remote API miteinander verbunden. Durch diese verteilte Logik ergibt sich allerdings auch eine gewisse Komplexität, die durch E2E-Tests abgesichert werden sollte.

Für das SPA Frontend können E2E-Tests ferner absichern, dass die SPA an sich korrekt arbeitet: Wir hatten oben die Möglichkeit erwähnt, dass bestimmte JavaScript-Module dynamisch – zur Laufzeit des Clients – nachgeladen werden können. Dabei kann die Reihenfolge, in der die Module geladen werden, davon abhängen, wie der Nutzer sich durch die Anwendung bewegt. Mit E2E-Tests kann nun sichergestellt werden, dass die Anwendung korrekt arbeitet, unabhängig davon, ob der Nutzer erst Modul A lädt und dann Modul B, oder ob er direkt mit Modul B startet. Je nachdem, welche Browserfeatures genutzt werden sollen, ist es auch sinnvoll, die Anwendung in unterschiedlichen Browsern und insbesondere auf verschiedenen Devices zu testen, da die unterschiedlichen Browser jeweils eigene JavaScript-Laufzeit-Implementierungen und -Features anbieten.

E2E-Tests werden klassischerweise so geschrieben, dass sie einen Nutzer simulieren, der einen bestimmten Anwendungsfall im Client abarbeiten möchte. Je nach Testfall muss die Anwendung dafür mehr oder weniger realitätsnah angebunden sein.

Wenn zum Beispiel nur einfache CRUD-Operationen notwendig sind, besteht die Option, ein einfaches, auf den Test ausgelegtes Fake-Backend anzubinden und bereitzustellen. Mit WireMock steht hierfür ein Tool mit vielfältigen Möglichkeiten zur Verfügung. Um echte fachliche Use Cases in E2E-Tests nachzuvollziehen, ist es jedoch meist unumgänglich, einen echten Backend-Server anzubinden, der auf einem Testdatenbestand arbeitet.

Für E2E-Test von Browseranwendungen gibt es sehr viele verschiedene Tools. Unter anderem kann, wie bei klassischen Webanwendungen auch, Selenium genutzt werden. Im Fall von Anwendungen, die einen Angular-Client haben, wird häufig Protractor als Test-Framework verwendet, das Jasmine als Matcher Library und Selenium zur Browseransteuerung nutzt. Außerdem erfreuen sich Testing-Tools wie Cypress oder TestCafe wachsender Beliebtheit. Das liegt unter anderem daran, dass sie schneller sind als Selenium und obendrein Videos und Screenshots vom Testverlauf automatisiert aufnehmen können. Außerdem gibt es noch Testing-Tools, die aufgenommene Screenshots der Anwendung mit einer Vorgabe abgleichen, um darüber die Korrektheit auf optischer Ebene abzusichern (Visual Diff/Visual Regression Testing).

Da in der heutigen Welt diverse Geräte mit Webanwendungen arbeiten, gibt es auch Tools, mit denen man Webanwendungen auf nativen oder auch simulierten Endgeräten (wie Smartphones oder Tablets) testen kann.

Zunächst mag die Entwicklung und speziell das Testen von Browseranwendungen für den einen oder anderen Java-Entwickler ungewohnt erscheinen. Nachdem man sich allerdings mit den notwendigen Technologien und Werkzeugen vertraut gemacht hat, treten schon bald Erfolgserlebnisse ein, die schlussendlich dazu führen, dass man keine Anwendung mehr ohne moderne, browserbasierte Technologien und Frameworks entwickeln möchte. Für einen guten Einstieg verschafft der nächste Abschnitt eine Orientierungshilfe im modernen Testuniversum.

Orientierung im Testuniversum

Bei all der Vielfalt an Optionen, wenn es um die Auswahl der richtigen Testinfrastruktur für Browseranwendungen geht, kann es schnell unübersichtlich werden. Bei der Fülle von Möglichkeiten stellt sich die Frage, welches der Testwerkzeuge für welchen Zweck zum Einsatz kommen soll. Dazu kommt, dass man nicht unbedingt mit den oft in JavaScript entwickelten Werkzeugen vertraut ist. Ebenso gibt es auch nicht immer eine gute Integration mit der bestehenden Java-(Entwicklungs-) Umgebung und den bekannten Build-Werkzeugen, wie z. B. Maven.

Als Orientierungshilfe für Entwickler zeigt **Abbildung 4** beispielhaft, wie die verschiedenen Werkzeuge eingeordnet werden können: die verschiedenen Testarten (blau), damit verbundene Konzepte (gelb) und die damit verbundenen Werkzeuge (grau). Von den anderen Ebenen unabhängig sind unten in Grün querschnittliche Werkzeuge zur Testunterstützung dargestellt. Der Fokus liegt dabei auf den eher ungewohnten Tools aus der JavaScript-Welt, die für das Testen der browserseitigen Logik nötig sind.

Die für Unit-Tests gebräuchlichen Tools sind Jest, Karma und Jasmine. Dabei ist eine wichtige Metrik die Code Coverage, die angibt, wie gut der vorliegende Quellcode mit Tests abgedeckt ist.

Die linke Spalte von **Abbildung 4** zeigt die für Unit-Tests gebräuchlichen Tools Jest, Karma und Jasmine, wobei Karma und Jasmine gerade bei Angular-Anwendungen zusammenarbeiten. Ins Detail gehen wir im nächsten Teil der Serie. Eine wichtige Metrik bei Unit-Tests ist die Code Coverage, in Gelb dargestellt. Sie gibt an, wie gut der vorliegende Quellcode mit Tests abgedeckt ist.

Die drei rechten Spalten enthalten Tools, die für E2E-Testing genutzt werden. In diesem Bereich tauchen auch Werkzeuge auf, die dem einen oder anderen Entwickler klassischer Anwendungen bereits bekannt sein dürften. Dazu gehört vor allem Selenium, ein Tool zur Browserautomatisierung. Cucumber, eigentlich ein Tool, um die korrekte Umsetzung von Use Cases zu testen und zu dokumentieren, ist dem einen oder anderen möglicherweise ebenfalls bekannt. Es ist ein Tool aus dem Bereich Behaviour-driven Development und hat eine sehr spezielle, an echten Text erinnernde Grammatik. Um Tests auf iOS- oder Android-Devices ausführen zu können, gibt es Appium. TestCafe und Cypress sind beides sehr moderne, eigenständige und vor allem performante Testing-Tools. TestCafe hat eine sehr breite Browserunterstützung, während Cypress zwar im wesentlichen Chrome unterstützt, dafür aber einen breiten Funktionsumfang fürs Debugging der Tests besitzt. Puppeteer ist eigentlich ein Tool zur Automatisierung speziell des Chrome-Browsers, es kann aber auch zum Testen verwendet werden. AyeSpy ist ein Werkzeug, das als visuelles Testtool verstanden werden kann und das Erzeugen und Vergleichen von Screenshots im Rahmen von Visual Diff Testing vereinfacht.

Separat zu den oberen Tools sind unten in Grün noch einige weitere Hilfsmittel aufgeführt, die wir uns im Verlauf dieser Artikelserie noch anschauen werden. Sie existieren alle, um das Testen zu vereinfachen, oder sind, wie im Fall des CI-Servers, sogar notwendig für eine automatisierte Ausführung der Tests. Darüber hinaus sind die Tools nicht nur beim Testen von Frontends interessant. Vor allem Testcontainers, die für Java-Anwendungen Umsysteme wie Datenbanken oder Message-Queues bereitstellen können, helfen, Integrationstests zu automatisieren. Auch WireMock, mit dem Remote APIs ohne tatsächliche Anwendung bereitgestellt werden können, ist ein Werkzeug, das für Integrationstests eine wichtige Rolle spielen kann. Docker wiederum ist eine Lösung zur Prozessvirtualisierung, mit der zum Beispiel ganze Build- und Testumgebungen durch einen Container bereitgestellt werden können. Die Isolation macht sich Testcontainers zu Nutze, ebenso wie viele moderne CI-Server Docker nutzen, um für jeden CI-Durchlauf eine isolierte Umgebung zur Verfügung zu stellen.

Fazit und Ausblick

Tests sind der Schlüssel zu langfristiger Wartbarkeit von Softwareanwendungen, egal ob klassische Webanwendung oder moderne Browseranwendung. Mit modernen Tools ist Testen einfach und kann bisweilen – wie wir noch sehen werden – sogar Spaß machen. Im nächsten Artikel lernen wir zunächst verschiedene Werkzeuge für Unit-Tests von modernen SPA Frontends kennen und sehen anhand von Codebeispielen, wie der praktische Einsatz aussehen kann. Dabei wird uns eine kleine Angular-Anwendung zu Demonstrationszwecken dienen.

Karsten Sitterberg ist als freiberuflicher Entwickler, Trainer und Berater für Java und Webtechnologien tätig. Karsten ist Physiker (MSc) und Oracle-zertifizierter Java Developer. Seit 2012 arbeitet er mit trion zusammen.

@kakulty

Links & Literatur

[1] https://martinfowler.com/articles/is-quality-worth-cost.html

[2] https://www.standishgroup.com/sample_research_files/RuleTen.pdf