SYRACOM

THE BUSINESS AND IT ARCHITECTS

Publikation | Solution Engineering | Marc Liebl

brand eins Thema B 2015 BESTE BERATER

Moderne Enterprise-Architekturen mit Java & JavaScript

Eine Architekturempfehlung auf Basis einer Konzeptstudie

 Part of Consileon Group

Im Gegensatz zu modernen Webseiten von Google, Apple, BMW usw. wirken webbasierte Enterprise-Anwendungen meist altbacken und auch die Usability, das heißt die Benutzbarkeit, birgt häufig Optimierungspotenzial. Sind Stakeholder aus dem Vertrieb involviert, bekommt die Unterstützung mobiler Geräte und somit Responsive Design (vgl. {WikiRWDJ) schnell einen hohen Stellenwert. Wie ist mit diesen und weiteren Herausforderung, wie Zukunftsfähigkeit, Flexibilität, Wartbarkeit und Testbarkeit, umzugehen? Der Artikel bietet eine Architekturempfehlung auf Basis einer Konzeptstudie.

Anforderungen an Enterprise Anwendungen

Betrachtet man die Entwicklung von Enterprise Anwendungen in den letzten Jahren, lässt sich ein deutlicher Wandel erkennen. Wurden diese Expertensysteme vor einiger Zeit für dedizierte Prozesse und Benutzergruppen entwickelt, stellen sie heute zentrale, prozessunterstützende Werkzeuge für das gesamte Unternehmen dar. Die Benutzeranzahl, sowie die Heterogenität der Benutzergruppen sind hierdurch stark gestiegen. Lag der Fokus zuvor eher auf der Funktionalität, bzw. Korrektheit der Anwendung, kommen heute weitere nichtfunktionale Anforderungen mit ebenso hohem Stellenwert hinzu. Diese betreffen vor allem die Erwartungshaltungen im Hinblick auf Bedienbarkeit, Fehlertoleranz, Barrierefreiheit, sowie der User Experience. Warum sollte die alltägliche Arbeit nicht genauso viel Spaß bereiten, wie die Konfiguration ihres Wunschautos?

Im Unternehmens-, sowie im Projektkontext ist es nicht immer möglich, den neuesten technologischen Trends zu folgen. Die Vorgaben in den Unternehmen z.B. durch den Betrieb oder der Unternehmenssicherheit sorgen häufig für starre und nicht selten konservative Rahmenbedingungen. Die Orientierung an offiziellen Standards, sowie die Verfügbarkeit von kommerziellem Support sind in der Regel harte Vorgaben. In diesem Bereich hat sich Java EE, als ausgereifte und den meisten technischen Anforderungen gewachsene Spezifikation etabliert. Ebenso existieren Plattformen wie z.B. die JBoss EAP, welche den organisatorisch benötigten kommerziellen Support bieten. Zusammengefasst gilt es, alle fachlichen, technischen und organisatorischen Anforderungen in eine solide, zukunftsfähige Software-Architektur zu überführen, welche sich den Stempel „Enterprise-Ready“ verdient.

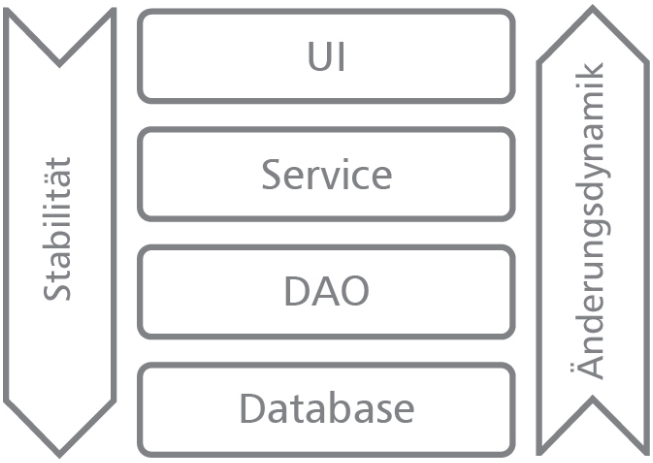

Abbildung 1:
Änderungsdynamik in
Webanwendungen

Ein kritischer Blick aus der Perspektive des Architekten

Aus Abbildung 1 lässt sich erkennen, dass die Änderungsdynamik stark zunimmt, sobald man sich in einer klassischen Schichtenarchitektur in Richtung Präsentationsschicht bewegt. Wie die Betrachtung gängiger Architekturmuster von Enterprise-Anwedungen zeigt, findet diese Tendenz zu wenig Berücksichtigung bei der Konzeption und Implementierung von Anwendungen. Eine DAO-Schicht für die Trennung der Datenzugriffsschicht auf die Datenbank wird integriert, für den Fall, dass diese sich ändert. Gängige Praxis stellt die Kapselung der Businesslogik in Services dar, welche die Datenzugriffsschicht orchestriert und die benötigte Fachlichkeit über eine klar getrennte Schnittstelle zur Verfügung stellt. Die Steuerung von Transaktionen findet in der Regel innerhalb dieser Schicht statt. Soweit sind die Aufgaben klar verteilt. In der Präsentationsschicht ist häufig eine enge Kopplung zwischen Model, View und Controller zu finden. Die einheitliche Verwendung von Domänenobjekten aus der Persistenz- bis zur Präsentationsschicht erzeugt ungewollte Abhängigkeiten. Sind Anpassungen für die Anzeige der Daten notwendig, werden diese häufig auf der obersten Schicht durchgeführt und nicht selten ist hier wieder Businesslogik zu finden. Es ist wichtig zu erwähnen, dass der skizzierte Ansatz viel Disziplin bei der Einhaltung architektonischer Vorgaben fordert. Im schlimmsten Fall ist die fachliche Logik in Stored-Procedures, über alle Schichten der Anwendung, sowie innerhalb der Domänenobjekte verteilt. Dies führt zu schlecht wartbaren und testbaren Anwendungen mit niedriger Qualität. Die klaren Trennungen die klassische Schichtenmodelle in der Theorie mit sich bringen, sind in der Praxis selten zu finden.

Beispiel einer modernen Software-Architektur

Um eine höchstmögliche Flexibilität zu erreichen, ist eine lose Kopplung zwischen der Businesslogik und der Präsentationsschicht, sowie der Darstellung von Daten essentiell. Dies kann durch die Trennung entlang der Systemgrenze zwischen Server und Client erreicht werden. Somit ist der Application Server für den serverseitigen Teil und der Web-Browser für den clientseitigen Teil der Anwendung als Ablaufumgebung verantwortlich. Für die Um-

setzung empfiehlt sich als Kombination die Verwendung des Java EE Standards, sowie JavaScript. Die Kommunikation erfolgt über das leichtgewichtige REST-Protokoll und JSON als Datenformat.

Aus architektonischer Sicht spricht vor allem die klare Trennung von Verantwortlichkeiten (Separation of Concerns, vgl. [Dijk74]) für diesen Ansatz. In vielen typischen Java EE Anwendungen basiert die Präsentationsschicht auf gängigen komponentenbasierten Frameworks (JSF), sowie ergänzender Komponentenbibliotheken (PrimeFaces, RichFaces, etc.). Der hierdurch vereinfachten Entwicklung stehen einige Risiken gegenüber. Das Konzept des clientseitigen Status als Beispiel, ist im HTTP-Protokoll nicht vorgesehen und findet durch diese Frameworks den Weg in die Anwendung. Eine Auseinandersetzung mit den originären Web-Technologien ist somit nicht nötig und führt häufig zu unschönen Nebeneffekten und Boilerplate-Code, wodurch vor allem die Wartbarkeit leidet.

Trennt man die Verantwortlichkeiten und überlässt allen Schichten bis zur Serviceschicht die Datenbeschaffung inkl. der Verarbeitungslogik und nutzt die Präsentationsschicht zur Darstellung der Daten ergibt sich eine klares Bild (vgl. Abbildung 2).

Serverseitig wird die fachliche Funktionalität mit technischen Querschnittsaspekten (Security, Logging, etc.) unter Zuhilfenahme etablierter Mechanismen (JPA / Hibernate, CDI, EJB, RESTEasy etc.) gebündelt. Der Client übernimmt die Aufgabe der Darstellung von Informa-

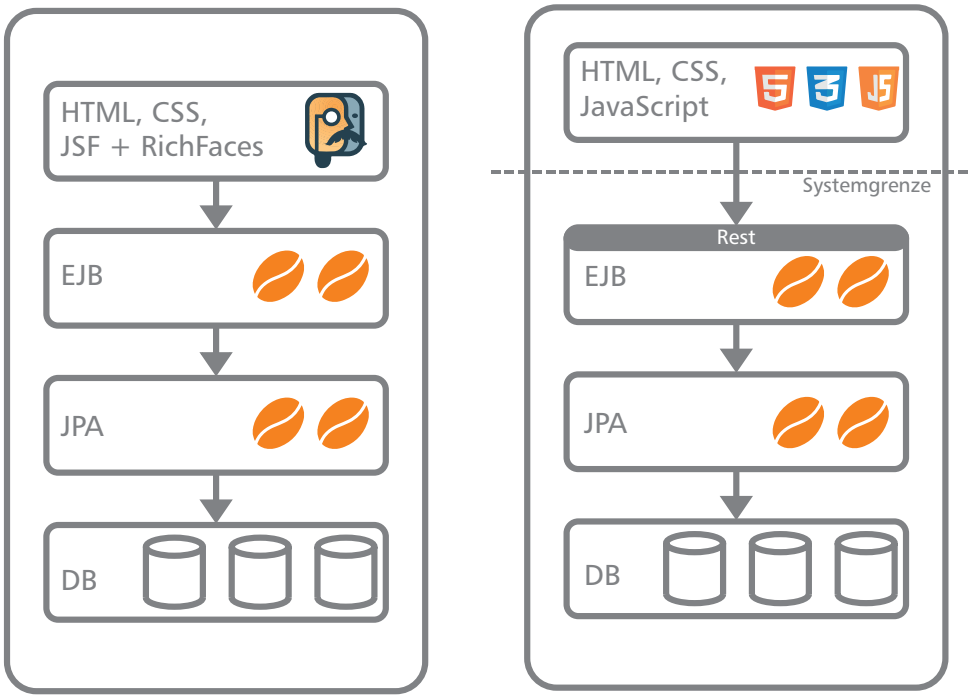

Abbildung 2:
Architekturansätze im Gegensatz

tionen. Die dafür ursprünglich vorgesehen Auszeichnungssprachen HTML5 und CSS3, sowie diverse Hilfswerkzeuge (z.B. JavaScript, JQuery) kommen hier zur Anwendung. Komplexere Layouts und Maskenflüsse lassen sich mit den genannten Techniken realisieren. Die Entwicklung findet näher an der Zielplattform, dem Web-Browser statt, sodass die die Abstraktionsschicht, welche durch komponentenbasierte Frameworks Einzug erhält, entfällt.
Ein weiterer Vorteil ergibt sich aus der hohen Flexibilität, der sich auf diesem Weg mehrstufig begegnen lässt. Durch die lose Kopplung zwischen dem server-, bzw. clientseitigen Teil der Anwendung, ist der Austausch eines kompletten Client-Frameworks mit verhältnismäßig wenig Aufwand möglich. Je nach Auswahl der Frameworkkombination ist eine noch feingranularere Flexibilität zu erreichen. Denkbar ist, die Darstellung der Informationen durch Templates zu erhalten und die Art der Datenbeschaffung auszutauschen. Als weiterer Vorteil erweist sich die Möglichkeit auf einfachem Weg eine Multikanalfähigkeit zu schaffen. Die Daten stehen als JSON-Objekte per REST zur Verfügung und erlauben vielseitige und flexible Nutzungsmöglichkeiten. Sie können von einem JavaScript-Client, einer nativen oder hybriden Applikation oder durch einen simplen Template-Generator für einen Seriendruck verwendet werden.

Konzeptionelle Überlegungen

Im Folgenden möchte ich mich auf die Realisierung des Clients fokussieren und den serverseitigen Java EE Teil etwas vernachlässigen. Wie im zu Beginn gezeigten Schaubild zu sehen, unterliegt dieser Teil der Anwendung einer geringeren Änderungsdynamik. Betrachtet man die benötigten Frameworks zur technischen Umsetzung, sowie die für den Entwicklungsprozess notwendigen Werkzeuge (Build, Qualitätssicherung, Erstellung automatisierter Tests), hat sich im Java EE Bereich eine solide Basis etabliert. Gegenteilig sieht es im JavaScript Umfeld aus. Hier existieren unzählige Alternativen für die jeweiligen technischen Anforderungen, sodass ein ganzheitlicher Überblick schwer fällt.
Zur Umsetzung eines auf JavaScript basierenden Clients existiert eine große Anzahl von Möglichkeiten. Diese reicht von leichtgewichtigen Frameworks, welche im Sinne des Baukasten-Prinzips Verwendung finden können, wie Backbone.js, Marionette.js, Require.js, Handlebars, bis hin zu All-In-One Lösungen, wie Ember.js und Angular.js. Je nach Anwendungsfall bieten alle Varianten Vor-und Nachteile, welche es im gegebenen Projektumfeld zu berücksichtigen gilt. Nimmt man die genannten zwei Kategorien als Basis, lassen sich folgende Vor-, bzw. Nachteile identifizieren:

	Vorteile	Nachteile
Modulare Lösungen (Backbone.js, Marionette.js, Knockout.js)	■ Flexibilität ■ Reifegrad ■ hohe Anzahl von Plugins ■ einfache Erweiterbarkeit	■ Auswahl geeigneter Plugins zeitaufwändig ■ flache Lernkurve
All-In-One Lösungen (AngularJS, EmberJS)	■ hoher Funktionsumfang ■ Lernkurve steil (zu Beginn) ■ aktuellere Dokumentation	■ geringere Flexibilität ■ Kompatibilität eingeschränkt ■ schnelle Änderungsdynamik

Tabelle 1
Vor- und Nachteile verschiedener Umsetzungslösungen

Die Praxis I: JavaScript-Client

Im aktuellen Projektkontext ist die Entscheidung auf den Einsatz von Backbone.js, Marionette.js, Require.js und Handlebars gefallen (vgl. Abbildung 3). Ausschlaggebend war der hohe Reifegrad, die hohe Flexibilität, sowie die damit einhergehende Zukunftssicherheit. Getreu dem Motto „think big, start small“, steht zu Beginn eine leichtgewichtige Basis zur Verfügung, welche mit steigenden Anforderungen stetig wächst. Da eine detaillierte Vorstellung aller genannten Frameworks und Werkzeuge den Rahmen dieses Artikels sprengen würde, möchte ich hier einen Überblick über die wichtigsten Bestandteile der Anwendung auf hoher Abstraktionsebene geben.

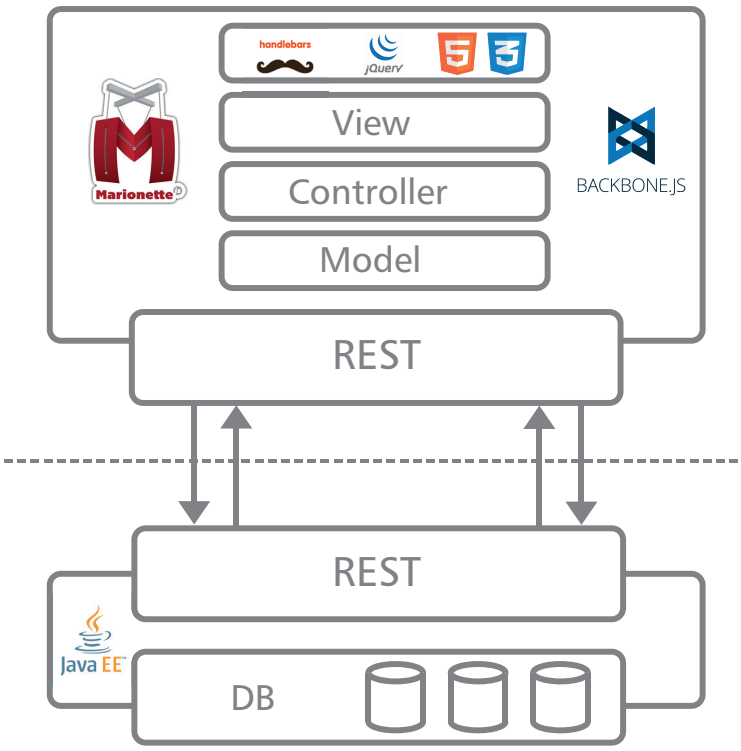

Abbildung 3:
Softwareübersicht des JavaScript-Clients

- Backbone.js

Backbone.js stellt den Kern der clientseitigen Anwendung dar und ist verantwortlich für die REST-Kommunikation, die Definition des Domänenmodells, sowie das Routing. Gerade im zuletzt genannten Bereich spielt Backbone seine Stärken aus. Es protokolliert die Benutzernavigation und ermöglicht somit die Nutzung des Vor-und Zurück Buttons im Browser.

- Marionette.js

Marionette.js erweitert Backbone.js um zahlreiche Funktionalitäten. Es erleichtert die Erstellung komplexer Layouts, die Darstellung von Massendaten, sowie das Speichermanagement. Eine Aufteilung der Anwendung in einzelne Module verbessert die interne Struktur, sowie die Wartbarkeit.

- Handlebars

Handlebars findet als Templating-Mechanismus Verwendung. Dieser ermöglicht die Wiederverwendung einzelner HTML-Bausteine, sodass eine Zusammensetzung ganzer Seitenhierarchien aus diesen realisierbar ist.

- JQuery

Als eine transiente Abhängigkeiten von Marionette.js steht JQuery in vollem Funktionsumfang zur Verfügung und bietet umfangreiche Gestaltungsmöglichkeiten zur Erstellung moderner Oberflächen.

- Require.js (optional)

Bei größeren Anwendungen ist es sinnvoll, die Modularität nicht nur innerhalb der Anwendung abzubilden, sondern zusätzlich auf Dateiebene zu erweitern. Hierunter wird die Erstellung asynchroner Moduldefinitionen (vgl. [DocReqJS]) zum dynamischen Nachladen einzelner Teile der Anwendung auf Basis von Require.js verstanden.

Die Praxis II: Test-Werkzeuge

Die Sicherstellung eines hohen Qualitätsniveaus innerhalb der Anwendung muss durch die eingesetzten Frameworks und passender Werkzeuge gleichermaßen unterstützt werden. Ist dies im Java EE Teil mit JUnit, Arquillian, Selenium, Cucumber, etc. sichergestellt, gilt es für den JavaScript Teil adäquate Werkzeuge zu integrieren. Hier existiert analog der Auswahl von Frameworks für den JavaScript-Client eine hohe Anzahl möglicher Alternativen. Im Folgenden ist die Auflistung einer bewährten Kombination von Werkzeugen zu finden, welche allen Anforderungen an einen den testgetriebenen Entwicklungsprozess optimal unterstützt (vgl. Abbildung 4).

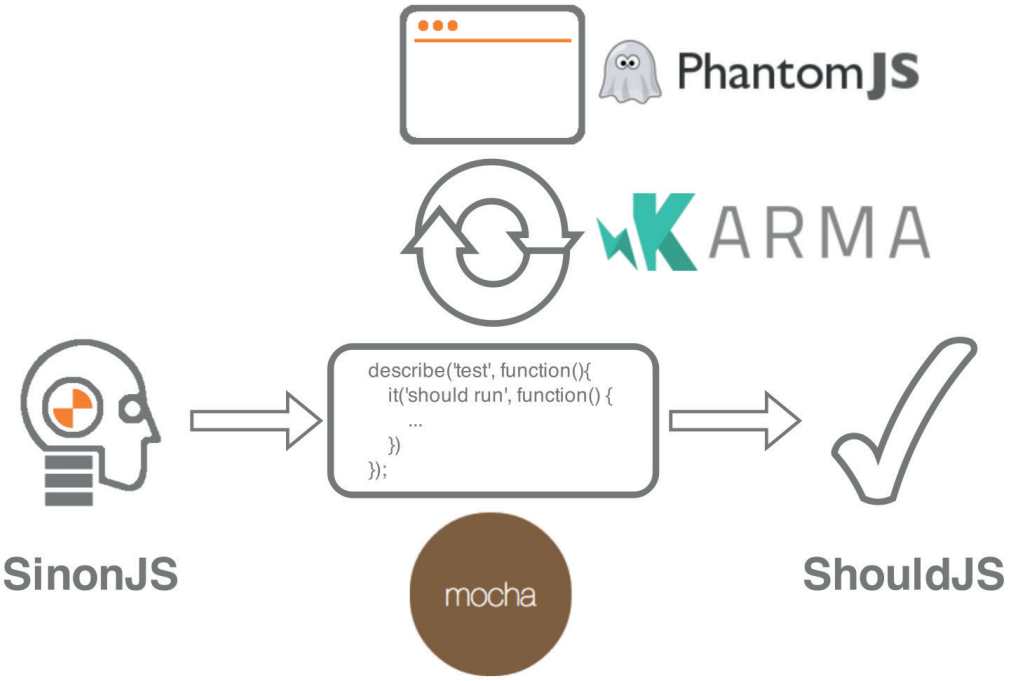

Abbildung 4:
Testwerkzeuge

- Karma.js

Karma.js dient als Ausführungsumgebung der Test-Suite. Per Konfiguration lässt sich z.B. die Ablaufumgebung, d.h. die zu verwendenden Browser definieren, sowie das Reporting über die Testresultate steuern.

- Mocha

Die Implementierung der Test-Suite erfolgt mit Mocha. Der Funktionsumfang geht weit über das Java-Pendant JUnit hinaus und bietet z.B. eine einfache Unterstützung bei der Ausführung asynchroner Tests. Eine rudimentäre Assertion-Library, sowie das Reporting von Testresultaten ist in Mocha bereits integriert

- Sinon.js

Sinon.js kommt bei der Erstellung von Test-Doubles (Spies, Stubs, Mocks) zur Geltung. Weiterhin bietet es die Möglichkeit einen Fake-Server zu implementieren, welcher auf einen HTTP-Request eine vordefinierte HTTP-Response liefert. Um die Qualität und Kompatibilität von REST-API's sicherzustellen ist dies das ideale Werkzeug.

- Should.js (optional)

Zur Verbesserung der Lesbarkeit der Tests bietet sich Should.js als Assertion-Library an. Die textuelle Beschreibung der Prüfung von Bedingungen orientiert sich am sehr natürlichsprachigen Behaviour-Driven-Development Stil (vgl. [North06]).

Die Praxis III: Build und QA-Werkzeuge

Um den Entwicklungsprozess optimal zu unterstützen ist ein schneller und flexibler Build-Prozess notwendig. Die Erstellung von JavaScript-Anwendungen, sowie das Management der Abhängigkeiten unterscheidet sich eminent zu Java-basierten Anwendungen und wird von Maven nicht unterstützt. Hierfür hat sich ein Pendant zu Maven, bzw. Ant Namens Grunt positioniert. Die Definition der Abhängigkeiten wird ähnlich zu Maven in einer Konfigurationsdatei (package.json) vorgenommen und mit Hilfe von Node.js und NPM als Paketmanager aufgelöst. Grunt versteht sich als sehr flexibles und durch Plugins erweiterbares Werkzeug zur automatisierten Ausführung von Aufgaben. Exemplarisch sind einige mögliche Aufgaben gelistet:

- Kompilierungsvorgänge (z.B. der Template-Dateien)
- Minifizierung der CSS-Dateien
- Konkatenierung der JavaScript-Dateien
- Aufruf von Karma zur Ausführung der Test-Suite
- Ausführung von Werkzeugen zur statischen Code-Analyse

Ein weiterer wichtiger Aspekt stellt die Kompatibilität mit einem eventuell bereits existierenden Build-Prozess auf Basis von Maven und Jenkins dar. Hier empfiehlt sich die Nutzung des Frontend-Maven-Plugins. Dieses ermöglicht die einfache Integration und Ausführung von Grunt in Maven-Projekte. Ebenso existiert eine Erweiterung für den Jenkins CI Server um die durch Karma aufgezeichneten Testresultate visuell darzustellen.

Die Praxis IV: Client-Server Kommunikation

Zur Kommunikation zwischen dem serverseitigen und dem clientseitigen Teil der Anwendung eignet sich das REST-Protokoll. Dies erlaubt mit Hilfe wenigen Annotationen auf Serverseite die Daten zur Visualisierung im JSON-Format zu liefern. Da die clientseitig verwendeten Frameworks auf den Umgang mit JSON ausgerichtet sind, ist die perfekte Symbiose geschaffen. Ist die Schnittstelle klar definiert, kann auf beiden Seiten testgetrieben gegen diese Spezifikation entwickelt und somit eine hohe Qualität und Kompatibilität sichergestellt werden.

Ein besonderes Augenmerkt muss hier allerdings auf das Schnittstellendesign gelegt werden. REST orientiert sich an Ressourcen (vgl. [Field10]) und stellt über die HTTP-Methoden GET, PUT, POST, DELETE die entsprechenden Operationen bereit. Dies sollte ebenso wie die korrekte Verwendung von HTTP-Statuscodes, bzw. die Nutzung des HTTP-Cachings bei der Spezifikation Beachtung finden. Auch die Änderungshäufigkeit der Schnittstelle und die damit eventuell notwendige Strategie zur Service-Versionierung ist zu beachten.

Die Praxis V: Herausforderungen

Allerdings gibt es bei jeder nicht trivialen Anpassung der Software-Architektur Herausforderungen zu meistern. Folgt man bereits jahrelang dem klassischen Java EE Standard und JSF, ist ein gravierendes Umdenken notwendig. Galt JavaScript lange als untauglich für Enterprise-Anwendungen und wurde oft belächelt, bietet es heute Konzepte, die dessen Einsatz rechtfertigen. Grundlegend ist ein gutes Verständnis von Asynchronität im Browser, generellen Web-Technologien, sowie der funktionalen Programmierparadigmen. Bei der Transformation von einer Multi-Page-Applikation hin zu einer Single-Page-Applikation ist ein geschultes Auge hinsichtlich der Performanz und möglicher Memory-Leaks notwendig.

Insgesamt erhöht sich die Anzahl der verwendeten Frameworks enorm. Zu JPA, EJB, RestEasy, etc., gesellen sich Backbone.js, Marionette.js, Require.js, Handlebars, JQuery, HTML5, CSS3, Grunt, Node, NPM, Karma, u.v.m. Diese Masse an unterschiedlichen Frameworks und Werkzeugen mit ihren jeweils unterschiedlichen Konzepten und Paradigmen, gilt es zu beherrschen und zu verstehen. Die Lernkurve zu Beginn ist nicht nur für erfahrene Java EE erprobte Kollegen eher flach, sodass eine finanzielle Investition in Aus-und Fortbildung gegebenenfalls notwendig ist.

Eine temporäre Anpassung der Strukturen innerhalb des Teams kann unter Umständen helfen, den Technologiewechsel zu unterstützen. Ist es auch erstrebenswert vertikal entwickelnde Teams zu formen, ist zu überlegen, in einer Übergangsphase die Entwicklung horizontal zu organisieren. Ein Teil des Teams fokussiert sich auf die neuen Tools, sowie den JavaScript-Client, während ein weiterer Teil sich der serverseitigen Realisierung widmet und die Schnittstellen klar definiert. Durch regelmäßige Meetings findet ein Know-How-Transfer statt, sodass im Laufe der Zeit eine vertikale Entwicklung möglich ist. Unerlässlich für den Erfolg, den dieser technologische Wandel mit sich bringt, ist ein eindeutiges Commitment aller Stakeholder. Die aufgezeigten Hürden stellen ein nicht zu unterschätzendes Risiko dar und müssen intern und extern transparent gemacht werden.

Fazit

Ist die Entscheidung auf den skizzierten Architekturansatz gefallen, gewinnt man im Vergleich zu „traditionellen“ Ansätzen Flexibilität, Erweiterbarkeit, Zukunftssicherheit und einen größeren Gestaltungsspielraum zur Realisierung der fachlichen Anforderungen. Die Architektur wird durch die zusätzliche REST-Schnittstelle entkoppelt und ermöglicht es, die Trennung von Businesslogik und Darstellung von Daten separat mit den jeweils hierfür vorgesehenen, ausgereiften Mechanismen zu implementieren. Alle Voraussetzungen für einen nach agilen Methoden ausgerichteten Entwicklungsprozess sind gegeben. Einfache Integrierbarkeit in den CI-Prozess, Modularisierbarkeit, gute Testbarkeit und die Möglichkeit der Erstellung moderner Oberflächen sprechen für diese Kombination. Das dem Artikel zugrundeliegende Projekt wurde vor wenigen Monaten gestartet. Die bisher gesammelten Erfahrungen unterstreichen die Validität des gewählten Ansatzes. Sollten Sie Fragen oder Anmerkungen haben, würde ich mich sehr auf einen Dialog freuen!

Der Autor

Marc Liebl (marc.liebl@syracom.de) ist Senior Consultant bei Syracom. Zu seinen Schwerpunkten gehören Enterprise Anwendungen auf Basis von Java EE und entsprechenden Öko-Systemen. Dabei liegt sein Fokus auf der ganzheitlichen Betrachtung des Softwareentwicklungsprozess unter Verwendung agiler Methoden.

Literatur und Links

[Dijk74] Edsger W. Dijkstra, Manuskript „On the role of scientific thought“ 1974

[Field10] Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures 2000, siehe: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[DocReqJS] Dokumentation Require.js, Why AMD, siehe http://requirejs.org/docs/whyamd.html

[North06] Dan North, Introducing BDD, siehe http://dannorth.net/introducing-bdd

[WikiSPA] Wikipedia, Single Page Applications, siehe http://en.wikipedia.org/wiki/Single-page_application

[WikiRWD] Wikipedia, Responsive Webdesign, siehe http://de.wikipedia.org/wiki/Responsive_Webdesign

SYRACOM

THE BUSINESS AND IT ARCHITECTS

Ansprechpartner:

Marc Liebl

Otto-von-Guericke-Ring 15

65205 Wiesbaden

Tel: +49 6122 9176-0

www.SYRACOM.de

marc.liebl@syracom.de