

Heiko Spindler

# Single-Page- Web-Apps

JavaScript im Einsatz: Webseiten erstellen  
mit AngularJS, Meteor und jQuery Mobile

Heiko Spindler

**Single-Page-Web-Apps**



Heiko Spindler

# Single-Page- Web-Apps

JavaScript im Einsatz: Webseiten erstellen  
mit AngularJS, Meteor und jQuery Mobile

## Bibliografische Information der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Alle Angaben in diesem Buch wurden vom Autor mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. Der Verlag und der Autor sehen sich deshalb gezwungen, darauf hinzuweisen, dass sie weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernehmen können. Für die Mitteilung etwaiger Fehler sind Verlag und Autor jederzeit dankbar. Internetadressen oder Versionsnummern stellen den bei Redaktionsschluss verfügbaren Informationsstand dar. Verlag und Autor übernehmen keinerlei Verantwortung oder Haftung für Veränderungen, die sich aus nicht von ihnen zu vertretenden Umständen ergeben. Evtl. beigelegte oder zum Download angebotene Dateien und Informationen dienen ausschließlich der nicht gewerblichen Nutzung. Eine gewerbliche Nutzung ist nur mit Zustimmung des Lizenzinhabers möglich.

© 2014 Franzis Verlag GmbH, 85540 Haar bei München

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Das Erstellen und Verbreiten von Kopien auf Papier, auf Datenträgern oder im Internet, insbesondere als PDF, ist nur mit ausdrücklicher Genehmigung des Verlags gestattet und wird widrigfalls strafrechtlich verfolgt.

Die meisten Produktbezeichnungen von Hard- und Software sowie Firmennamen und Firmenlogos, die in diesem Werk genannt werden, sind in der Regel gleichzeitig auch eingetragene Warenzeichen und sollten als solche betrachtet werden. Der Verlag folgt bei den Produktbezeichnungen im Wesentlichen den Schreibweisen der Hersteller.

**Programmleiter:** Dr. Markus Stäuble

**Lektorat:** Christian Immler

**Satz:** DTP-Satz A. Kugge, München

**art & design:** [www.ideehoch2.de](http://www.ideehoch2.de)

**Druck:** CPI, Printed in Germany

**ISBN 978-3-645-60310-2**

# Vorwort

## Für wen ist das Buch gedacht?

Das Buch richtet sich an alle, die Interesse an einer neuen Art von Web-Entwicklung haben und den Nutzern mehr Dynamik bieten wollen. Um sinnvolle Single-Page-Web-Applikationen (SPWA) zu bauen, werden wir das Rad nicht neu erfinden. Die Zeiten, in denen man alles von Hand machen musste, sind für die Web-Entwicklung zum Glück vorbei. Der Hauptfokus des Buches liegt auf der Auswahl und dem Einsatz der richtigen Frameworks, für die Implementierung von SPWA. Dabei steht die Pragmatik und Einfachheit im Vordergrund. Gleichzeitig gibt das Buch mit vielen Beispielen Hilfestellung für die Entwicklung von unternehmenskritischen Applikationen mit JavaScript.

Erste Grundkenntnisse im Umgang mit HTML5, CSS 3 und einer Programmiersprache sind sehr sinnvoll und werden vorausgesetzt. Erfahrungen in der Entwicklung von JavaScript sind hilfreich, aber kein Muss.

## Wie ist das Buch aufgebaut?

Das Buch gliedert sich in vier Hauptteile:

### ① Einleitung in das Thema:

Die Einleitung führt in das Thema ein und erklärt die grundlegenden Konzepte und Herausforderungen beim Erstellen von Web-Applikationen.

### ② JavaScript für Entwickler:

Das zweite Kapitel enthält eine Einführung in die professionelle Programmierung mit JavaScript. Dabei werden vor allem die Unterschiede zu verbreiteten Programmiersprachen, wie zum Beispiel Java, aufgezeigt. Hilfreich sind erste Erfahrungen in der Programmierung.

### ③ Single-Page-Web-Apps erstellen:

Der Hauptteil zeigt Schritt für Schritt, wie dynamische Web-Applikationen mit AngularJS erstellt werden. Jedes Beispiel hebt dabei einzelne Aspekte hervor. Besonderes Augenmerk liegt auf dem effizienten Einsatz und Zusammenspiel von aktuellen Frameworks. Zusätzlich vermittelt der Abschnitt viele Tipps und Tricks für die tägliche Arbeit.

### ④ Von der DB bis ins Web mit Meteor:

Die Programmiersprache JavaScript eignet sich auch für die Implementierung von Server-Komponenten. Meteor stellt ein durchdachtes Applikationsframework bereit, das mit herausragenden Eigenschaften überrascht: Datenänderungen werden automatisch an alle Clients verteilt. Weil viele Funktionen und Module vorgefertigt sind, kann man sich ganz auf das Wesentliche konzentrieren. Mehrere nachvollziehbare Beispiele erklären den Einstieg in die Entwicklung mit Meteor.

**5 Mobile Single-Page-Web-Apps:**

Mobile Geräte stellen einen immer größeren Markt im Web dar, wobei die teilweise großen Unterschiede, zum Beispiel bei den Displays oder der Bedienung, besondere Anforderungen stellen. Das fünfte Kapitel zeigt, wie sich spezielle Web-Applikationen für diese Gerätelasse mit jQuery-Mobile erstellen lassen.

**Downloads zum Buch**

Alle im Buch erwähnten Quellcodes finden Sie auf [www.buch.cd](http://www.buch.cd) zum Download.

Ich bedanke mich bei meiner Familie für die Unterstützung. Ein besonderer Dank geht an Niko Köbler.

# Inhaltsverzeichnis

1	Einleitung .....	11
1.1	Das Web als Plattform .....	11
1.2	Was ist eine Single-Page-Web-Applikation? .....	11
1.2.1	Die Vorteile von Single-Page-Web-Applikationen .....	12
1.3	Ein Blick zurück in die Historie .....	14
1.4	Warum erst jetzt? .....	16
1.5	Für welche Applikationen sind SPWA geeignet? .....	17
1.6	Nachteile von Single-Page-Applikationen .....	18
1.7	Was sind die Herausforderungen? .....	18
1.7.1	JavaScript: Gehasst und geliebt? .....	20
1.7.2	Interne Strukturen von Applikationen .....	22
1.8	Architektur-Modell .....	22
1.9	Ein erstes Beispiel .....	24
1.9.1	»Hello World« mit AngularJS .....	24
2	JavaScript für Entwickler .....	27
2.1	Einleitung .....	27
2.2	Dynamisches Web mit JavaScript .....	27
2.3	Missverständnisse bei JavaScript .....	29
2.3.1	JavaScript kennt keine Datentypen .....	29
2.3.2	Die Funktion eval() ist böse .....	31
2.3.3	Verwirrungen mit Vergleichen: == oder ===? .....	32
2.3.4	Objektorientierte Programmierung .....	33
2.3.5	JavaScript ist in jedem Browser unterschiedlich .....	42
2.3.6	JavaScript kennt nur globale Variablen .....	42
2.4	Funktionen: First-Class Citizens .....	43
2.4.1	Definieren von Funktionen .....	43
2.4.2	Immediately Invoked Function Expression (IIFE) .....	45
2.4.3	Funktionen und Parameter .....	46
2.5	Einfacher Leben mit JavaScript .....	47
2.5.1	Konventionen .....	47
2.5.2	Behandlung von Fehlern und Ausnahmen .....	48
2.5.3	Guter Stil mit dem Strict Mode .....	50
2.6	JSON - JavaScript Object Notation .....	53
2.6.1	JSON im Detail .....	53
2.6.2	Parsen und Ausgaben von JSON .....	55
2.7	Gerüstet für die Zukunft: EcmaScript 6 .....	56

3	Single-Page-Web-Apps erstellen .....	59
3.1	Applikationen mit JavaScript – erste Schritte .....	59
3.2	Handarbeit oder Framework? .....	60
3.3	AngularJS: der Superheld an deiner Seite.....	61
3.4	Planung: Link-Verwaltung .....	69
3.4.1	Schritt 1: Tabellen-Ansicht.....	70
3.4.2	Schritt 2: Strukturen schaffen .....	74
3.4.3	Schritt 3: Detailansicht für Links.....	77
3.4.4	Schritt 4: Anlegen, Ändern, Kopieren und Löschen .....	80
3.4.5	Schritt 5: Filtern, aber richtig.....	87
3.4.6	Schritt 6: Sortieren mit mehr Komfort .....	93
3.4.7	Schritt 7: Daten im LocalStorage des Browsers.....	95
3.4.8	Schritt 8: Kommunikation mit dem Server per REST .....	104
3.5	Web-Anwendungen im Unternehmensumfeld.....	113
3.5.1	Fachliche Anforderungen.....	113
3.5.2	Herausforderungen.....	114
3.5.3	AngularJS-UI Bootstrap .....	115
3.5.4	Google Charts .....	120
3.5.5	Umsetzen der Anforderungen.....	123
3.5.6	Fazit zum zweiten AngularJS-Beispiel .....	143
3.6	Spiele mit AngularJS entwickeln .....	143
3.6.1	SVG-Grundlagen .....	144
3.6.2	Eine Sudoku-App mit AngularJS und SVG .....	147
3.6.3	Die Umsetzung der Sudoku-App .....	149
4	Von der DB bis ins Web mit Meteor.....	173
4.1	Das Meteor-Framework.....	173
4.2	Die sieben Prinzipien.....	174
4.3	Starten mit Meteor .....	175
4.3.1	Installation und erste Schritte .....	175
4.3.2	Mehr Struktur in umfangreicheren Projekten .....	180
4.4	Beispiel: Chat-Applikation .....	181
4.4.1	Schritt 1: Start und Setup .....	181
4.4.2	Schritt 2: Erste Erweiterungen des Chats .....	185
4.4.3	Schritt 3: Ein erstes Login.....	188
4.4.4	Schritt 4: Externe Login-Provider .....	194
4.5	Ausflug: Persistenz mit MongoDB.....	198
4.6	Beispiel: Kollektives Whiteboard.....	202
4.6.1	Publish/Subscribe .....	202
4.6.2	Remote Procedure Calls .....	203
4.6.3	Die Anforderungen des Whiteboards.....	204
4.6.4	Die Umsetzung.....	205

4.6.5	Wir erweitern das Whiteboard .....	223
4.6.6	Deployment & Packaging .....	230
4.7	Abschließende Anmerkungen zu Meteor .....	230
4.7.1	Meteor-Erweiterungen: Atmosphere .....	231
4.7.2	Bewertung von Meteor .....	233
5	Mobile Single-Page-Web-Apps .....	235
5.1	Mobile Anwendungen werden zum Standard.....	235
5.1.1	Einleitung zu jQuery Mobile .....	236
5.2	Eine mobile Zitate-Datenbank mit jQuery Mobile.....	253
5.2.1	Schritt 1: Backend und Anbindung.....	257
5.2.2	Schritt 2: Grundgerüst und zufällige Zitate .....	260
5.2.3	Schritt 3: Vorschläge für Autoren und Stichworte.....	264
5.2.4	Schritt 4: Ergebnisliste.....	268
5.2.5	Schritt 5: Feinarbeiten und Erweiterungen .....	269
5.3	Fazit zu jQuery Mobile .....	279
5.4	Testen von mobilen Applikationen .....	280
5.5	Zur nativen App mit PhoneGap .....	281
6	Anhang .....	285
	Stichwortverzeichnis .....	287





## **Einleitung**

### 1.1 Das Web als Plattform

In den letzten Jahren hat sich das Internet als die umfassende Kommunikationsplattform etabliert und bildet ebenfalls die Basis für den Erfolg mobiler Geräte. Es entstehen ständig neue Technologien. Ganze Branchen müssen sich in dieser »schönen neuen Welt« umorientieren und tragfähige Geschäftsmodelle finden. Die Software-Entwicklung im Zentrum dieser Veränderung muss neue Trends erkennen und die eigenen Kompetenzen immer wieder an diese Entwicklungen anpassen. Die Dynamik, die sich manchmal aus kleinen Neuerungen ergibt, ist erstaunlich. Ein neuer Begriff am Horizont ist das Paradigma von Single-Page-Web-Applikationen (SPWA): eine neue Art, Web-Anwendungen zu bauen.

### 1.2 Was ist eine Single-Page-Web-Applikation?

Eine Single-Page-Web-Applikation ist eine Web-Anwendung, die keinen Seitenwechsel (Refresh) durchführt, sondern die Oberfläche über dynamischen Austausch der HTML-Elemente mit JavaScript ändert.

Bei Aktionen in der Anwendung entfällt das kurze Flackern, das sonst die Anfrage beim Webserver und beim Neuaufbau begleitet. Das scheint auf den ersten Blick nicht

besonders spektakulär. Einem Benutzer fällt diese kurze Wartezeit vom Drücken eines Bestell-Knopfes bis zum Aufbau des Warenkorbes auf einer Shopping-Seite aber eventuell unangenehm auf.

Schon eine Verzögerung von mehr als zwei Sekunden wird oft als störend empfunden. Bei herkömmlichen Webseiten wird die gesamte Seite (HTML-Code, Bilder, Style-sheet und Scriptdateien) übertragen. Natürlich strengen sich die modernen Browser an und merken sich Elemente, die sich nicht geändert haben. Zumindest der HTML-Code einer Seite muss neu geladen werden. Diese Übertragung erfordert eine gewisse Zeit: Verbindungsaufbau, Übertragung und Darstellung summieren sich schnell zu Sekunden.

Was eine SPWA auszeichnet, in Kürze:

- Die Implementierung erfolgt mit den Technologien HTML5, CSS 3 und JavaScript.
- Die HTML-Seiten werden zum größten Teil dynamisch im Browser erzeugt.
- Der Datenaustausch mit einem Backend erfolgt meist in Form von JSON oder XML.
- Die Webseite verhält sich mehr wie eine Applikation.
- Es gibt keinen Reload oder Seitenwechsel.
- Der aktuelle Zustand der Applikation wird im Browser gehalten.
- Neben den klassischen Eingabe-Elementen bieten diese Applikationen meist erweiterte, aktivere Benutzeroberflächen an.

Interessant ist, dass HTML, JavaScript und CSS nicht neu sind, sondern schon lange verfügbar. JavaScript erhält einen deutlich höheren Stellenwert als bisher.

## 1.2.1 Die Vorteile von Single-Page-Web-Applikationen

Welche Vorteile entstehen aus diesem neuen Ansatz? Eines der Hauptziele vieler Anbieter ist es, Applikationen mit mehr Logik als bisher zu erstellen:

- Integration verschiedener Informationsquellen im Client
- Automatische Berechnungen und frühzeitige Validierung von Benutzereingaben
- Flexible Anpassung auf individuelle Benutzerwünsche und Vorlieben
- Mehr Interaktivität und Dynamik bei der Bedienung
- Aktive und automatische Benachrichtigung beim Auftreten von Ereignissen oder Änderungen an Daten

### Bessere Verteilung

Eine SPWA ist über eine URL im Browser universell erreichbar. Eine Installation ist nicht notwendig. Für den Privatgebrauch ist das nur ein nettes Feature. Im Unterneh-

mensumfeld reduziert diese Eigenschaft die Kosten für Installation und Verteilung. Damit einher geht auch die Möglichkeit, leicht mehrere Versionen parallel zu betreiben. Leicht können neue Funktionen während einer Testphase ausprobiert werden. Eine andere URL für die neue Version ist schnell per E-Mail kommuniziert.

### Die einheitliche Plattform

Eine heutige Applikation soll möglichst universell verfügbar sein. Viele Benutzer wollen Dienste auf unterschiedlichen Geräten nutzen. Der Zugriff muss vom heimischen PC genauso gut funktionieren wie vom Tablet oder Smartphone aus. Eine separate Entwicklung für jede Zielplattform ist teuer. Das Web wird die übergreifende Plattform für alle Betriebssysteme und Gerätearten.

Laut einer Studie von Cisco Systems ([bit.ly/19jj1Sw](http://bit.ly/19jj1Sw)) waren schon 2010 über 12,5 Milliarden Geräte mit dem Internet verbunden. Der größte Teil ist mit Browern und JavaScript ausgestattet. Wenn man als Anbieter eine große Zielgruppe erreichen möchte, muss man auf diese Technologien setzen.

Einige Smartphone-Hersteller entwickeln neue mobile Betriebssysteme, die auf eine reine HTML-Oberfläche setzen. Alle Apps sind damit formal Web-Anwendungen. Vertreter dieser Gattung sind Tizen von Samsung und Firefox OS, hinter dem die Mozilla-Organisation steht. Offensichtlich sind das nicht nur Experimente, erste Geräte sind auf dem Markt verfügbar.

### Reduzierung der Technologien

Die oben beschriebene Reduzierung auf die am meisten verbreiteten Technologien hat weitere positive Aspekte auf Seiten der Software-Entwicklung: weniger Know-how muss für die Entwicklung und Wartung bereit gehalten werden. Sicherlich muss in die effiziente Entwicklung mit dem neuen Paradigma einmalig investiert werden, langfristig reduziert es erheblich die Kosten.

### Integration über REST-Schnittstellen

SPWA bieten sich gut als Integrationspunkt an. Eine Backend-Komponente kann leicht per REST-Service angebunden werden. Als Datenformat haben sich JSON (JavaScript Object Notation) und XML durchgesetzt. Der Client kann unterschiedliche Datenquellen zusammenführen und in der Applikation integrieren und präsentieren.

### Offline-Fähigkeiten

Mit den neuen Fähigkeiten von HTML5 wie dem LocalStorage gibt es zum ersten Mal eine wirkliche Möglichkeit, effiziente Cache-Strategien und Offline-Fähigkeiten zu etablieren.

### Geringe Einstiegshürden: Start Easy

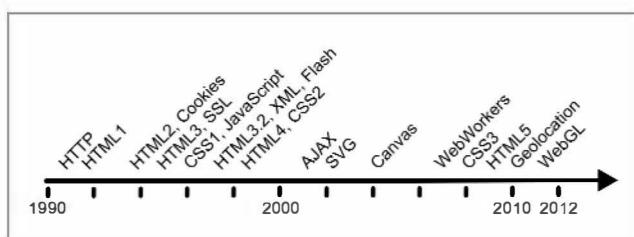
Es ist sehr leicht, mit den beschriebenen Technologien zu starten. Als Entwicklungs-Umgebung reicht ein guter Texteditor aus. Manche bieten sogar Syntax-Hervorhebung für HTML und JavaScript an. Zum Ausführen reicht ein Browser, der mit den entspre-

chenden Plug-ins sogar Debugging bereitstellt. Alle diese Komponenten sind für den Einsteiger kostenlos verfügbar. Kostenpflichtige Entwicklungsumgebungen bieten eventuell mehr Komfort und einige Arbeitserleichterungen, sind aber keine Voraussetzung.

Das Grundwissen über HTML und JavaScript ist bei vielen Entwicklern und Designern vorhanden. Man muss sicherlich tiefer in die Details eintauchen, wenn man komplexe Anwendungen entwickeln möchte.

### 1.3 Ein Blick zurück in die Historie

Die Ursprünge von HTML reichen zurück bis zum Ende des letzten Jahrhunderts. So gesehen handelt es sich um keine neue Technologie. Der ursprüngliche Fokus bei der Entstehung war, statische Inhalte in Form von wissenschaftlichen Texten und Bildern zu veröffentlichen. Über die Verlinkung sollten thematisch ähnliche Quellen leicht erreichbar sein.



**Bild 1.1:** Zeitliche Einordnung der Entstehung einiger grundlegenden Technologien für Web-Applikationen.

Das Erstellen von Applikationen war nicht das Ziel. Dieser Umstand drückt sich auch in dem zustandslosen http-Protokoll aus. Jede Anfrage ist isoliert und eine Applikation muss sich selbst um den logischen Zusammenhang kümmern, meist mit Hilfe von Session-Kennzeichen oder Cookies.

Eine Lösungsstrategie für den Bau von Applikationen hieß: Halte alle Informationen auf dem Server und behandle den Browser wie eine Darstellungseinheit für statische HTML-Inhalte. Die eigentliche Applikation läuft auf dem Server. Er verwaltet den Zustand und reagiert auf Benutzeraktionen. Als Ergebnis erhält der Benutzer eine neu generierte HTML-Seite.

#### Tipp: Evolution of the Web

Eine ansprechende und interaktive Darstellung der Historie bietet die Seite »Evolution of the Web« unter der Adresse [www.evolutionoftheweb.com](http://www.evolutionoftheweb.com). Die Seite dokumentiert ebenfalls alle gängigen Browser, die zeitliche Abfolge der Versionen und die implementierten Standards.

Diese Architektur funktioniert weiterhin sehr gut. Viele professionelle Angebote sind damit erfolgreich am Markt. Eigentlich wollen die Nutzer aber lieber mit einer Webapplikation arbeiten, die sich mehr wie eine native, installierte Anwendung anfühlt. Diese Programme zeichnen sich durch eine wesentlich aktiveren und intuitiveren Art der Benutzung aus. Eine Aktion führt schneller zur gewünschten Reaktion.

Aus diesen Forderungen entwickelten sich Plug-ins, die eine proprietäre Technologie im Browser bereitstellten. Die bekanntesten Vertreter sind Java Applets, Flash/Shockwave und Silverlight. Diese bieten mehr Freiheit und ermöglichen aktiveren Applikationen. Insbesondere Spiele und Multimedia-Applikationen wurden mit diesen Softwaretechnologien implementiert.

Die Nachteile der heterogenen Welt traten in den letzten Jahren immer stärker in den Vordergrund: Die Plug-ins sind nicht auf allen Plattformen verfügbar. Ständig müssen neuen Versionen installiert werden. In manchen Fällen stellen die Erweiterungen sogar Sicherheitsrisiken dar.

Seit 2005 wirbelte ein neuer Begriff viel Staub auf: AJAX (Asynchronous JavaScript and XML). Mit Hilfe von dynamischen und asynchronen Anfragen bringt AJAX mehr Dynamik in den Browser. Die HTML-Seite wird weiterhin auf dem Server erzeugt und der größte Teil bleibt statisch. Kleine Bereiche werden je nach Benutzeraktionen nachgeladen oder »on-the-fly« ausgetauscht. Partial Page Rendering (PPR) ist eines der Schlagworte.

Typische Beispiele sind Vorschlagslisten, die sich abhängig von der Eingabe anpassen und Vorschläge automatisch vervollständigen. In einigen populären Frameworks sind AJAX-Komponenten heute häufig zu finden. Frameworks wie JSF (Java Server Faces) gelingt damit der Spagat: Dynamik im Client zu bieten, ohne das Programmierparadigma grundlegend zu ändern. Für manche Aufgabenstellungen ist das ein akzeptabler Weg.

Aus diesen ersten Anfängen haben sich die aktuellen mächtigen Frameworks, allen voran jQuery, entwickelt. Ohne diese Hilfe bei der Manipulation des DOM-Baums einer HTML-Seite verzweifelt man schnell an den Unterschieden gängiger Browser und ihren verschiedenen Versionen.

Vor diesem Hintergrund stellt die aktuelle Entwicklung zur Single-Page-Web-Applikation den logisch nächsten Schritt dar. Das Web erwächst vom einfachen Container für statische Inhalte zur Plattform und Umgebung für komplexe, integrierte Anwendungen, die sich vor herkömmlichen Applikationen nicht mehr verstecken müssen.

Eine der ersten Applikationen mit diesem Ansatz war GMail, gefolgt von den anderen Office-Produkten von Google, die jetzt unter der Bezeichnung »Google Apps for Business« angeboten werden. Von der Textverarbeitung über Tabellenkalkulation bis hin zu Präsentationssoftware ist fast alles dabei, was der typische Anwender braucht.

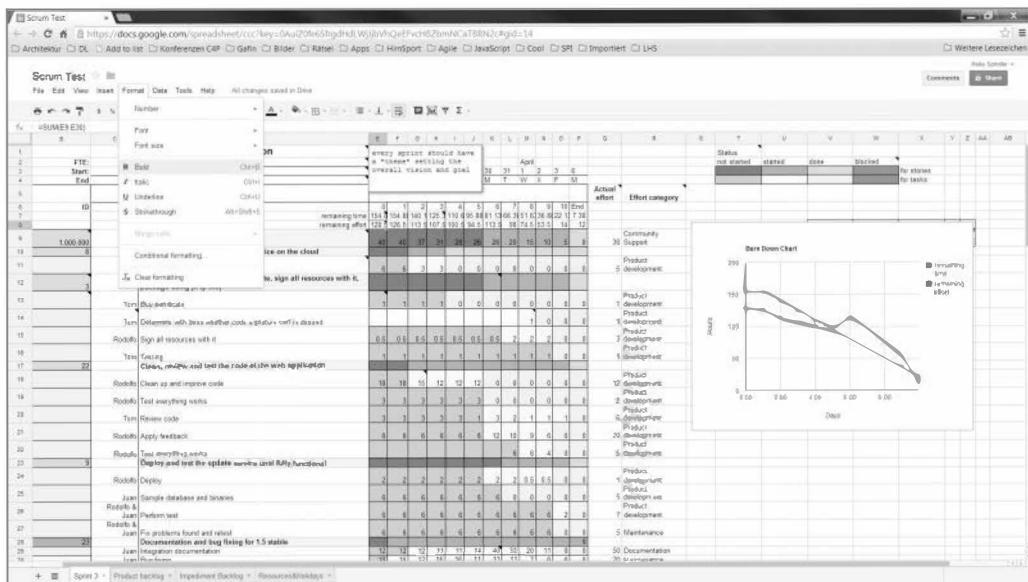


Bild 1.2: Tabellenkalkulation im Netz mit »Google Apps For Business«.

## 1.4 Warum erst jetzt?

Grundsätzlich sind diese Technologien schon seit langem verfügbar. Bereits im Jahr 1992 spendierte Netscape seinem Browser in der Version 2.0 JavaScript. Warum wird diese Technik erst jetzt für die Entwicklung so interessant?

Dafür gibt es eine Reihe von Gründen, die sich in den letzten zwei bis drei Jahren herausgebildet haben:

## JavaScript wird schneller

JavaScript wird interpretiert und nicht nativ auf der CPU des Rechners ausgeführt. Die JavaScript-Engines wurden erst in den letzten Jahren optimiert. Intensive Anwendungslogik und das Manipulieren von HTML-Seiten sind erst jetzt effizient möglich. Vor allem Google hat im eigenen Browser (Chrome) JavaScript deutlich beschleunigt, schon mit dem Ziel vor Augen, der Sprache den Stellenwert einer Plattform zu geben. Seit 2008 findet ein regelrechtes »Speed Race« zwischen den Entwicklern von Browsern und JavaScript-Umgebungen statt.

Stark im Team: HTML5, CSS 3 und JavaScript

Die Fähigkeiten der einzelnen Technologien ergänzen sich heute im Browser sehr gut. Aus der Erfahrung der letzten Jahre konnten mit den aktuellen Versionen gezielt Funktionslücken geschlossen werden. Mussten früher bestimmte Layout-Elemente von Hand zusammengestückelt werden, wie zum Beispiel der Schattenwurf oder

abgerundete Ecken, bieten heute CSS und HTML viele Möglichkeiten, um die Kreativität der Designer zu beflügeln.

Die folgende Liste ist nur eine kleine Auswahl der neuen Fähigkeiten, die HTML5 und die begleitenden Technologien bieten:

- Leichtes Integrieren und Abspielen von Multimedia-Inhalten
- Canvas-Objekt für das Zeichnen von Grafiken
- LocalStorage für Caching und Offline-Fähigkeiten
- Integrationen von SVG (Scalable Vector Graphic)
- Erweiterungen von CSS 3 (Transformationen)
- Erweiterungen für HTML-Formulare (Placeholder und Eingabevalidierung)

### **EcmaScript Version 5**

Aber auch die Sprache JavaScript selbst wurde erwachsener: Mit dem Sprachstandard EcmaScript 5 kommen Sprachfeatures hinzu, die JavaScript auf eine professionellere Ebene heben: Der »Strict Mode« erlaubt es, Prüfungen einzuschalten, die fehlerträchtige Programmkonstrukte melden. Der Interpreter moniert diese Stellen und zwingt Entwickler zu mehr Disziplin. Ziel ist es, mehr Lesbarkeit und Nachvollziehbarkeit zu erreichen. Da dieses Thema für die professionelle Entwicklung in Teams besonders wichtig ist, liefert das Kapitel 2 weitere Informationen.

### **Produktive Frameworks**

Eine Programmiersprache alleine reicht für die professionelle Entwicklung nicht aus. Notwendig sind auch Frameworks, die Strukturen vorgeben und Routineaufgaben abnehmen.

Frameworks wie jQuery, Prototype und »Script.aculo.us« haben den Zugriff auf den DOM-Baum vereinheitlicht und die problematischen Unterschiede zwischen den Browsern versteckt. Gerade jQuery hat daran einen großen Verdienst. Vielleicht sind diese neuen Frameworks sogar der Hauptgrund für den aktuellen Erfolg von JavaScript.

## **1.5 Für welche Applikationen sind SPWA geeignet?**

Bisher werden meist einfache und kleine Applikationen in dieser Form erstellt, die in ihrem Funktionsumfang überschaubar sind. Komplexere Beispiele sind noch die Ausnahme. Grundsätzlich sind SPWA nicht für alle Arten von Anwendungen geeignet. Sie haben ihre Stärken bei dynamischen Webseiten, die sich wie eine Applikation verhalten sollen und zum Beispiel Berechnungen ausführen, Eingaben prüfen oder den Nutzern viele Möglichkeiten der Anpassung bieten.

SPWA bieten sich als Ersatz für die Fälle an, in denen bisher Technologien wie Java Applets, Flash oder Silverlight zum Einsatz kamen. Browser-Spiele können somit als eine Spezialform von Web-Anwendungen gesehen werden.

Durch die vermehrte Dynamik auf dem Client scheinen ebenfalls neue Applikationskonzepte realisierbar: kollaborative Applikationen, die durch Nachrichten (anderer Benutzer oder automatisch erzeugt) reagieren. Beispiele für diese Anwendung könnten Gruppen-Whiteboards oder das gemeinsame Arbeiten an Dokumenten sein.

Der Ansatz passt ebenfalls gut für mobile Anwendungen: Die Kommunikation zwischen Client und Server reduziert sich auf den Transport von Daten, nachdem die Applikation initial geladen wurde. Das kommt den meist geringen Bandbreiten im mobilen Umfeld entgegen. Sinnvoll scheint der Einsatz für datengetriebene Business-Applikationen, die ein Frontend für umfangreiche Logik auf dem Server darstellen.

## 1.6 Nachteile von Single-Page-Applikationen

Wirklich verstanden hat man einen Ansatz erst, wenn man neben dessen Stärken auch seine Schwächen kennt. Weniger gut geeignet scheint der SPWA-Ansatz für folgende Arten von Applikationen zu sein:

- Applikationen, die sehr aufwendige Berechnungen durchführen oder direkt große und umfangreiche Datenmengen verarbeiten müssen.
- Web-Angebote, deren Schwerpunkt auf statischen Inhalten liegt. Beispiele hierfür sind Artikelseiten oder Blogs.
- Applikationen, die spezielle Hardware erfordern oder sehr betriebssystemnah sind. Hierzu zählen ebenfalls spezielle Bedienkonzepte oder Fähigkeiten der Geräte. Es gibt Werkzeuge, die dabei helfen, Web-Applikationen in native Apps zu verwandeln – ein Beispiel hierfür stellt das 5. Kapitel vor. Diese Frameworks erzielen gute Ergebnisse, erreichen aber nicht immer die volle Integration in das native System.

## 1.7 Was sind die Herausforderungen?

Single-Page-Web-Applikationen verhalten sich nicht wie typische Webseiten, woraus einige besondere Herausforderungen entstehen. Für die meisten dieser Herausforderungen gibt es aber recht einfache Lösungen. In den weiteren Kapiteln des Buches zeigen konkrete Beispiele, wie das funktioniert.

### Initiales Laden

Das erstmalige Laden der Applikation könnte länger dauern als bei einer klassischen Webseite. Die umfangreicheren JavaScript-Dateien inklusive eingesetzter Bibliotheken müssen über die Leitung. Hinzu kommt vermutlich eine längere Initialisierung durch

den Browser beim Starten. Eine statische HTML-Seite kann wahrscheinlich schon angezeigt werden, bevor sie komplett geladen ist. Bei einer SPWA muss das JavaScript komplett geladen sein, um es starten zu können. Wenn dann erst noch Daten vom Server zu holen sind, geht viel Zeit ins Land. Damit ist der erste Aufruf einer SPWA wahrscheinlich ein kritischer Punkt, der schon während der Entwicklung zu prüfen ist.

### Moderne Browser

Voraussetzung für den Einsatz von SPWA ist ein moderner Browser in einer aktuellen Version. Der Internet Explorer von Microsoft bietet erst ab Version 9 die volle Unterstützung des JavaScript-Standards EcmaScript Version 5. Bei vielen Nutzern und Unternehmen sind deutlich ältere Browser im Einsatz. Zum Glück hat sich dieses Problem aber in den letzten Jahren abgeschwächt. Für eine geplante Applikation empfiehlt es sich, die Zielgruppe und die exakten Anforderungen aufeinander abzustimmen.

### Number Crunching

Applikationen, die sehr umfangreiche Berechnungen durchführen, sollten nicht in einer interpretierten Sprache wie JavaScript erstellt werden. Die Anbieter, insbesondere Google, haben ihre Browser schon deutlich optimiert, aber der Unterschied zu einer kompilierten Sprache besteht weiter. Es gibt bessere Möglichkeiten für die Implementierung von rechenintensiven Algorithmen (wie Number Crunching). Eine Web-Anwendung kann natürlich eine wunderbare Oberfläche für eine solche Anwendung bieten.

Es ist trotzdem erstaunlich, was mittlerweile mit JavaScript möglich ist. So bieten die modernen Browser eine gute Integration mit WebGL und können (mit Unterstützung einer modernen Grafik-Karte) 3-D-Modelle flüssig anzeigen. Erste Spiele zeigen, dass dieses Vorgehen selbst auf vielen Tablets und Smartphones gut funktioniert.

### Browser History und Deep Links

Klickt man sich durch eine Web-Applikation, sieht man meist, wie sich die URL verändert und eventuell mit Parametern anreichert. Die gerade sichtbare Seite kann über diese URL direkt angesprungen werden (Deep Links).

Das ist eine nützliche Funktion, von der Benutzer bei Bookmarklisten regen Gebrauch machen. Eine SPWA durchbricht dieses Prinzip leider, da sich die Seite dynamisch ändert. Aus Sicht des Browsers gibt es keinen Grund, die URL zu ändern, obwohl sich die Applikation in einem neuen Zustand befindet.

Mittlerweile gibt es mit der History-API die Möglichkeit, das Verhalten des Browsers für die Anzeige der URL zu steuern. Damit können Applikationen den Nutzern den Status einer Applikation in der Eingabezeile für die URL zeigen und Bookmarks können wie gewohnt gespeichert werden.

### SEO: Die Kunst, gefunden zu werden

Was nützt die schönste Web-Applikation, wenn sie nicht gefunden wird? SEO (Search Engine Optimization) vereint alle Maßnahmen, mit denen eine Webseite

optimiert werden kann, damit diese für ausgewählte Begriffe bei Suchmaschinen einen hohen Stellenwert erhält. Ziel ist es, bei den gewünschten Begriffen möglichst weit vorne in der Ergebnisliste zu landen. Für statische Webseiten gibt es viele Tipps und Tricks für die erfolgreiche Optimierung.

Viele Maßnahmen, die das Gestalten der Seite betreffen (On-Site-Optimierung), widersprechen dem Konzept, eine Anwendung mit dynamischen Inhalten zu bauen. Diese Maßnahmen funktionieren bei SPWA nicht automatisch. Der Crawler hat eine andere Sicht auf die Seite als ein menschlicher Benutzer, weil er das JavaScript nicht interpretiert und nicht ausführt, um die dynamisch angepassten Seiten zu indexieren. Er findet also kaum Inhalt für die Indexierung.

Dieses Problem ist nicht leicht zu lösen. Eine Lösungsidee könnte sein, dem Crawler eine zusätzliche statische Version der Seite anzubieten. Dieses Vorgehen widerspricht leider den Vorgaben von Google, weil damit viel Missbrauch getrieben wurde (Cloaking). Im Extremfall kann das Vorgehen zu einer negativen Bewertung oder dem kompletten Ausschluss führen. Als Lösung bietet sich an, zusätzlich statische Inhalte bereitzustellen, die immer sichtbar sind.

### Alles sichtbar: Die gläserne Applikation

Bei einer SPWA liegt der JavaScript-Programmcode im Klartext auf dem Client-Rechner vor. Ein Blick in das Cache-Verzeichnis oder in die Browser-Konsole entblößt die komplette Applikation mit ihrer gesamten Logik. Ungewollt wird damit vielleicht eine Preissstaffel oder die Entscheidungs- beziehungsweise Validierungslogik offen gelegt. In einer serverseitigen Applikation hat man hingegen alles unter Kontrolle und unter Verschluss. Gesendet wird nur der generierte HTML-Code.

Es gibt Maßnahmen, mit denen das Problem abgeschwächt werden kann: Der JavaScript-Code kann »obfuscated« werden: Alle sinnvollen Namen für Variablen und Funktionen werden in kryptische, wenig aussagefähige, eventuell zufällige Bezeichner transferiert.

Dadurch verschwindet ebenfalls jede sinnvolle Formatierung. Das Lesen und Verstehen des Algorithmus wird deutlich aufwendiger, aber nicht vollständig verhindert. Alternativ können sensible Daten und Berechnungen auf den Server ausgelagert und per Web-Service vom Client aufgerufen werden. Mit allen Maßnahmen erkauft man sich zusätzlichen Aufwand für die Implementierung.

#### 1.7.1      JavaScript: Gehasst und geliebt?

An der Sprache JavaScript selbst scheiden sich die Geister vieler Entwickler. Wenn man sich der Sprache als Neuling nähert, erscheint sie auf den ersten Blick sehr einfach. Schnell kommt man zu ersten kleinen Programmen. Die Sprache bietet viele Möglichkeiten, sehr kompakt zu programmieren. Das kann leider auch leicht zu unverständlichen Codes führen. Außerdem bietet JavaScript einige außergewöhnliche

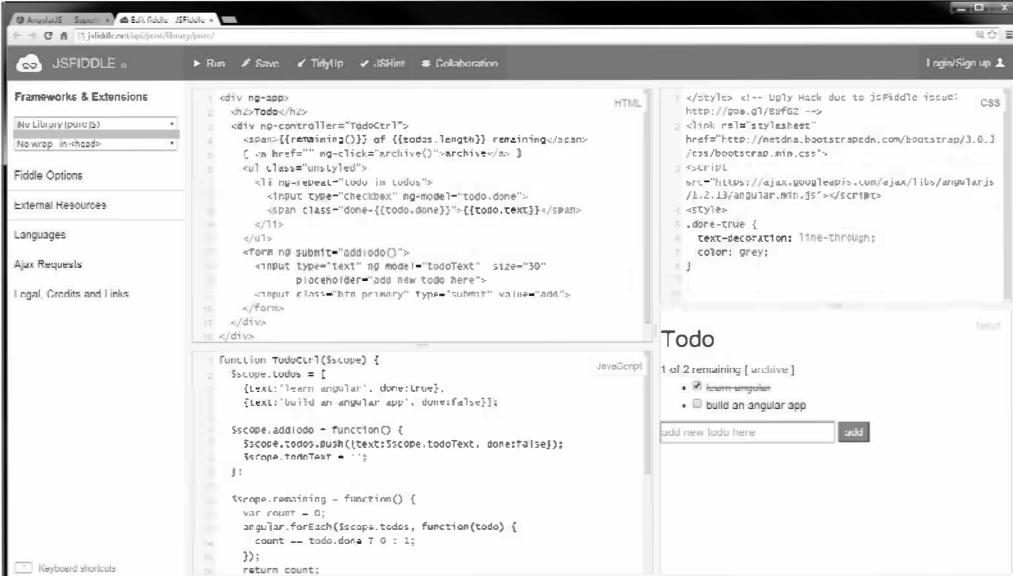
Sprachkonstrukte, die leicht missverstanden werden. Das zweite Kapitel stellt einige Beispiele im Detail vor.

Bei kleinen Projekten sind diese Fallstricke meist unproblematisch. In größeren Teams und umfangreichen Projekten hingegen erwächst daraus ein Risiko: Der Programmcode wird eventuell anfälliger für Fehler, der Wartungsaufwand könnte steigen. Ein verständlicher und wartbarer Code muss einen hohen Stellenwert haben. Ohne sinnvolle Konventionen und Disziplin bei allen Beteiligten im Team ist das nur schwer zu erreichen.

## Werkzeuge und Tools

Neben der eigentlichen Programmiersprache gehören zum professionellen Umfeld zusätzliche Werkzeuge. Für den Start in die Web-Entwicklung reicht ein besserer Texteditor aus; wenn es komplexer wird, sollte man Wert auf mehr Unterstützung legen. Von anderen Programmiersprachen sind Features wie Syntax-Hervorhebung, Auto-Vervollständigung oder Refactoring bekannt. Diese Funktionen zahlen sich schnell aus. Für eine dynamische Sprache scheint es aufwändiger zu sein, diese mächtigen Funktionen anzubieten. Insgesamt bleibt der Funktionsumfang meist hinter anderen Sprachen zurück.

Eine recht neue Entwicklung sind Programmierumgebungen, die direkt im Browser funktionieren. Gerade für die Web-Entwicklung können so Code und die Ergebnisse auf einen Blick bearbeitet werden. Vertreter dieser Gattung sind zum Beispiel jsFiddle ([jsfiddle.net](http://jsfiddle.net/)) und jsBin ([jsbin.com](http://jsbin.com/)/net).



The screenshot shows the jsFiddle interface with an AngularJS application example. The left sidebar contains 'Frameworks & Extensions' (No Library (pure.js)), 'Fiddle Options' (No wrap in <head>), 'External Resources' (Bootstrap 3.0.3), 'Languages' (JavaScript), 'Ajax Requests' (None), and 'Legal, Credits and Links'. The main area has tabs for 'HTML', 'CSS', and 'JavaScript'. The 'HTML' tab shows the following code:

```

<div ng-app>
  <h1>Todo</h1>
  <div ng-controller="TodoCtrl">
    <span>{{remaining()}} of {{todos.length}} remaining</span>
    <a href="#" ng-click="archive()"><archive></a>
    <ul class="unstyled">
      <li ng-repeat="todo in todos">
        <input type="checkbox" ng-model="todo.done">
        <span class="done-{{todo.done}}">{{todo.text}}</span>
      </li>
    </ul>
    <form no submit="addTodo()>">
      <input type="text" ng-model="todoText" size="30" placeholder="Add new todo here">
      <input class="btn-primary" type="submit" value="Add">
    </form>
  </div>

```

The 'JavaScript' tab shows the following code:

```

function TodoCtrl($scope) {
  $scope.todos = [
    {text:"Learn angular", done:true},
    {text:"Build an angular app", done:false}
  ];

  $scope.addTodo = function() {
    $scope.todos.push({text:$scope.todoText, done:false});
    $scope.todoText = '';
  };

  $scope.remaining = function() {
    var count = 0;
    angular.forEach($scope.todos, function(todo) {
      count += todo.done ? 0 : 1;
    });
    return count;
  };
}


```

The right side shows the resulting 'Todo' application with a list of todos and a form to add new ones. The CSS tab contains a single line of CSS:

```

<style> /* Ugly Hack due to jsFiddle issue: http://goog.l/eUFGZ -->
  <link rel="stylesheet" href="http://netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.13/angular.min.js"></script>
<style>
  .done-true {
    text-decoration: line-through;
    color: grey;
  }

```

**Bild 1.3:** Ein geöffnetes Beispiel in jsFiddle mit HTML-, JavaScript- und CSS-Code und dem ausgeführten Ergebnis unten rechts.

Werkzeuge zur Code-Analyse und für die Automatisierung entstehen gerade erst oder sind noch nicht völlig ausgereift. Hier gibt es im Umfeld der Web-Entwicklung Nachholbedarf.

## 1.7.2 Interne Strukturen von Applikationen

Nichttriviale Anwendungen kommen ohne eine gute interne Struktur nicht aus. Die Aufteilung in separate Module mit abgegrenzten Aufgabenbereichen ist ein kritischer Erfolgsfaktor. Erste Ansätze und Frameworks sind in den letzten Jahren entstanden, haben sich aber noch nicht vollständig durchgesetzt. Im Bereich der UI- und MVC-Frameworks (Model-View-Controller) hingegen schießen die Frameworks wie Pilze aus dem Boden und man verliert als Entwickler den Überblick.

Ein Hauptanliegen der folgenden Kapitel ist, in diesem Umfeld mehr Überblick zu liefern. Welche Frameworks gibt es und wie lassen sie sich zu einer sinnvollen und tragfähigen Anwendungsarchitektur kombinieren?

### Test-getriebene Entwicklung (TDD)

Sehr zu empfehlen ist die test-getriebene Entwicklung, die sich für andere Plattformen als sehr sinnvoll erwiesen hat. Für jede nichttriviale Funktion erstellt der Entwickler Unit-Tests, die immer wieder leicht ausgeführt werden können. Die Tests prüfen jede Funktion in Isolation und stellen sicher, dass eine Änderung keinen anderen Teil der Applikation negativ beeinflusst. Ein plötzlich fehlschlagender Test legt eine technische Abhängigkeit schnell offen. Das Team erhält die Chance, schnell zu reagieren. Das Vorgehen bringt dem Team mehr Sicherheit, auch bei wachsender Komplexität in größeren Projekten mit vielen technischen Abhängigkeiten.

## 1.8 Architektur-Modell

Wie sieht das grobe Architektur-Modell einer Web-Anwendung aus? Um diese Frage zu beantworten, betrachten wir die wesentlichen Komponenten mit ihren Aufgaben und Beziehungen untereinander. Einfach dargestellt, sind die wesentlichen Teile folgende: Client (in Form des Browsers), Server und Datenbank.

Die folgende Abbildung zeigt die Veränderung einer typischen Anwendungsarchitektur über die letzten Jahre. Dabei ist die wesentliche Logik immer weiter vom Server hin zum Client gewandert.

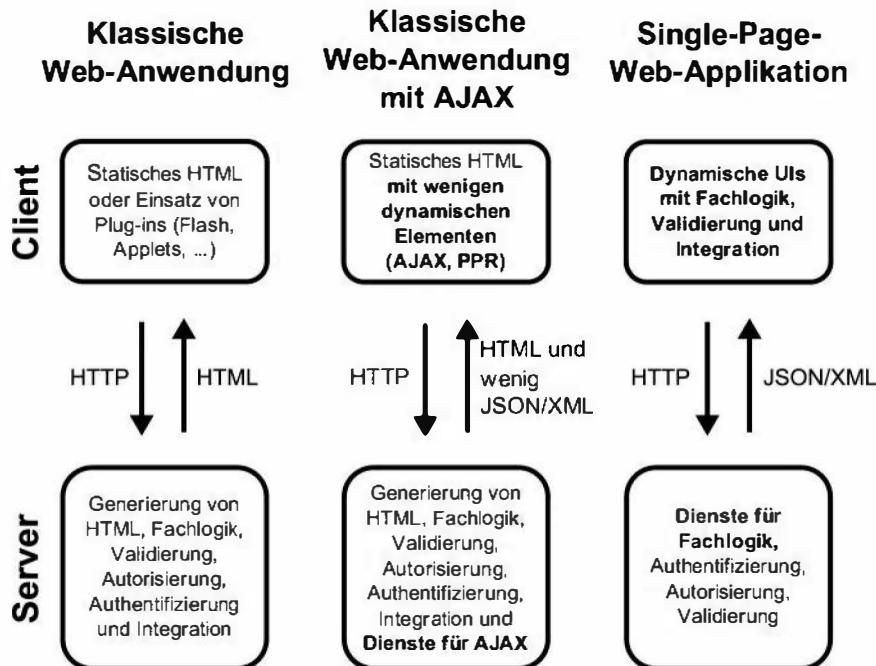


Bild 1.4: Grober Architekturvergleich von klassischer Web-Applikation und SPWA.

Die Datenbank dient hauptsächlich als Container für Daten. Trotzdem können hier wesentliche Aufgaben einer Anwendung realisiert sein, zum Beispiel Historisieren, Prüfung der Konsistenz oder das Aufbereiten von Daten bei großem Volumen.

Die zeitliche Abfolge (von links nach rechts) zeigt, wie sich der Server mehr zum Anbieter von Diensten verändert und den Client unterstützt. Typische Basisdienste sind Authentifizierung und Autorisierung. Es bietet sich an, die anwendungsspezifische Logik ebenfalls in Form von Diensten oder Services zu realisieren. Das könnten Berechnungen und Validierungen sein. Sinnvolle und generische Services können ebenfalls leicht in anderen Szenarien erneut Verwendung finden.

Der Client verwandelt sich vom Viewer für statisches HTML in eine Ausführungsumgebung und Plattform. Er führt die wesentliche Logik direkt aus, erzeugt die Oberfläche dynamisch und spricht Dienste auf dem Server an. Der Client integriert unter Umständen andere externe Dienste und führt die Daten zusammen.

Zusätzlich zeigt die Abbildung, wie sich das Datenformat für die Kommunikation zwischen Server und Client von HTML zu Rohdaten in Form von JSON oder XML gewandelt hat. Meist ändert sich im Zug dessen auch das Kommunikationsverhalten: von seltenen umfangreichen Übertragungen kompletter HTML-Seiten hin zu kleineren und häufigen Transfers.

## 1.9 Ein erstes Beispiel

An dieser Stelle soll ein erstes kleines Beispiel zeigen, wie leicht die Entwicklung mit JavaScript und modernen Frameworks sein kann. Dazu werfen wir einen Blick auf ein einfaches Beispiel mit AngularJS, das im dritten Kapitel tiefer mit mehreren praxis-relevanten Applikationen vorgestellt wird.

### 1.9.1 »Hello World« mit AngularJS

Eine der Hauptaufgaben ist die Verbindung der Daten (in Form eines Modells) und der Darstellung in der View. Der View-Teil wird, wie bisher üblich, als HTML umgesetzt. Das Programm soll unsere Eingabe in einem HTML-Eingabefeld an einer anderen Stelle in der Seite wieder ausgeben:

```
<!doctype html>
<html ng-app>
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.4/angular.min.js"><
/script>
</head>
<body>
<div>
<label>Name:</label>
<input type="text" ng-model="theName" placeholder="Enter a name here">
<hr>
<h1>Hello {{theName.toUpperCase()}}!</h1>
</div>
</body>
</html>
```



Bild 1.5: Ausgabe des ersten AngularJS-Beispiels.

Das Script-Tag bindet das Framework ein. Ein weiterer Code in Form von Bibliotheken oder handgeschriebenem JavaScript ist in diesem Fall nicht notwendig.

#### Content Delivery Network

Ein Content Delivery Network (CDN) speichert global oft angefragte Dateien (wie zum Beispiel populäre Frameworks) zwischen und liefert diese sehr optimiert aus. Eine Bibliothek muss nicht lokal im Projekt liegen, sondern kann über ein CDN bezogen werden. Die Adresse für AngularJS in der Version 1.2.4 lautet:

`ajax.googleapis.com/ajax/libs/angularjs/1.2.4/angular.min.js`

An dem äußersten umschließenden HTML-Tag befindet sich die Markierung `ng-app`. Damit erkennt AngularJS, dass es die Kontrolle für diesen Teil der Seite übernehmen soll. Im Beispiel ist das die komplette Seite. In einer umfangreichen Webseite könnte man so nur genau den Bereich auswählen, der durch AngularJS dynamisch verändert werden soll. Dadurch reduziert sich der Aufwand bei der Initialisierung und weniger HTML-Code ist durch das Framework zu analysieren.

Eine ähnliche Kennzeichnung findet sich am Eingabefeld `ng-model="theName"`. Sie definiert den Namen für die Modell-Variable, die mit dem Eingabefeld verbunden ist. Das reicht für unser einfaches Beispiel aus. Wir müssen keine zusätzliche Modellvariable von Hand erstellen. AngularJS überwacht alle Änderungen in diesem Eingabefeld und überträgt diese in das Modell. Wie die Modellvariable im JavaScript angesprochen wird, untersuchen wir später.

Im Beispiel soll die Eingabe an einer anderen Stelle ausgegeben werden: Hierzu bietet AngularJS einen Template-Mechanismus an: AngularJS sucht im HTML-Code nach Elementen in doppelten geschweiften Klammern der Form: `{} Ausdruck {}`. Der Ausdruck in den Klammern wird ausgewertet und durch sein Ergebnis ersetzt. Hier könnte jegliche Art von Logik angesprochen werden. Im Beispiel wandeln wir lediglich die Eingabe in Großbuchstaben um.





## JavaScript für Entwickler

### 2.1 Einleitung

Diese Kapitel richten sich an Leser, die Erfahrungen in der Programmierung in anderen Sprachen gesammelt haben. Wir starten nicht bei den absoluten Einsteigerthemen, sondern konzentrieren uns auf die Besonderheiten von JavaScript im Vergleich zu üblichen Programmiersprachen. Von diesen Besonderheiten hat JavaScript einige zu bieten, und darüber stolpern viele Entwickler beim Umstieg auf JavaScript.

#### Tipp: Starten mit JavaScript

Wer mit den Grundlagen, wie Syntax und Kontrollstrukturen, starten möchte, findet unter folgenden Links einen guten Einstieg:

Auf Wikipedia gibt es eine 5-Minuten-Einführung in JavaScript:  
[de.wikipedia.org/wiki/JavaScript#Sprachelemente](https://de.wikipedia.org/wiki/JavaScript#Sprachelemente)

Sehr zu empfehlen ist folgendes kostenlose Online-Buch:  
[eloquentjavascript.net](http://eloquentjavascript.net)

### 2.2 Dynamisches Web mit JavaScript

JavaScript wurde erfunden, um Webseiten dynamisch im Browser zu ändern und Interaktionen mit dem Benutzer zu realisieren. Das ist heute immer noch der Haupt-einsatz. Der Standard ECMAScript (ECMA-262, Edition 5: [www.ecma-international.org/](http://www.ecma-international.org/)

*publications/standards/Ecma-262.htm*) beschreibt die Sprache und ihre Elemente. An der Syntax ist die Verwandtschaft mit der Sprache »C« zu erkennen. Der Namensbestandteil »Java« ist eher als Marketing-Trick zu bezeichnen, denn die Gemeinsamkeiten mit Java sind gering.

JavaScript ist eine dynamisch typisierte, objektorientierte Scriptsprache. Das Besondere ist die hohe Flexibilität, die sich vor allem darin ausdrückt, dass es möglich ist verschiedene Arten der Programmierung anzuwenden: objektorientiert, prozedural und funktional. JavaScript entpuppt sich damit als Allesköninger. Das macht JavaScript aber leider nicht besonders einfach zu verstehen.

JavaScript kennt keine Sprachkonstrukte für Klassen, Interfaces und Vererbung, wie zum Beispiel Java. Alternativ bietet JavaScript Objekte (Instanzen) mit einem mächtigen Prototyp-Konzept. Sie sind ein flexibles Werkzeug, mit dem sich Vererbung nachbilden lässt.

Um komplexere Applikationen mit JavaScript bauen zu können, muss man sich intensiver mit der Sprache beschäftigen und mehr verstehen als die Deklaration von Variablen und die grundlegenden Kontrollstrukturen.

Die folgenden Abschnitte stellen einige der interessantesten Konzepte von JavaScript vor, um für die Implementierungen von Single-Page-Web-Apps in den nächsten Kapiteln gerüstet zu sein. Dabei werden wir auf einige Fehlerquellen und Fallstricke stoßen, auf die man bei der täglichen Arbeit immer wieder trifft.

Die meisten JavaScript-Beispiele können leicht in der Konsole von Google Chrome oder Firefox ausprobiert werden. Die Taste F12 öffnet die Entwickler-Tools und unter dem Reiter Konsole lassen sich direkt JavaScript-Befehle ausführen.

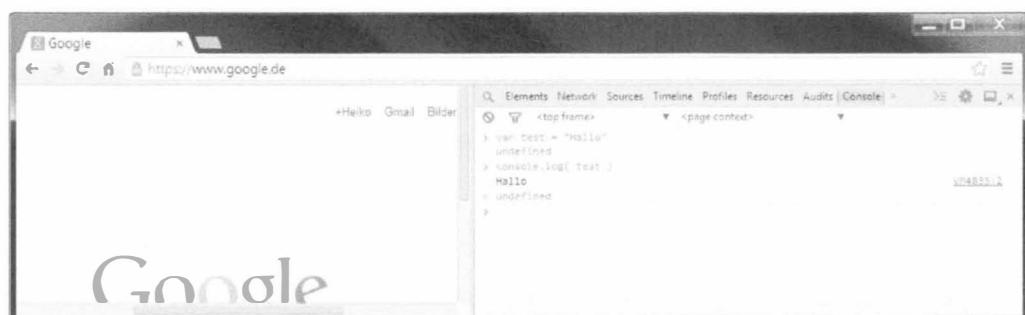


Bild 2.1: Die geöffnete Konsole in Google Chrome.

## 2.3 Missverständnisse bei JavaScript

JavaScript hatte lange Zeit einen schlechten Ruf innerhalb der Entwicklergemeinde. In vielen Projekten wurde es ausschließlich eingesetzt, wenn die Aufgabe nicht anders zu lösen war. Das schlechte Bauchgefühl ist durchaus begründet, wie ein kleines Beispiel deutlich macht:

```
//Plus ist für Strings definiert. Die Ziffer wird konvertiert:  
> console.log("42" + 7);  
"427"  
//Führt zur Konversion des 1. Operanden in eine Zahl:  
> console.log("42" - 7);  
35
```

Das Verhalten ist nicht konsistent. Der Plus-Operator ist für Zeichenketten sinnvoll einsetzbar: Die beiden Operanden werden als Strings verkettet. Der Minus-Operator verhält sich anders, da er für Zeichenketten nicht existiert. Der linke Operand wird in eine Zahl konvertiert und eine Subtraktion ausgeführt. Beides ist für sich genommen logisch, aber in der Kombination verwirrend.

Leider steht dieses Beispiel stellvertretend für viele weitere sonderbare Konstellationen, von denen einige in diesem Kapitel vorgestellt werden. Trotzdem ist JavaScript insgesamt besser als sein Ruf. Die nächsten Punkte stellen einige der typischen Missverständnisse bei JavaScript in ein klareres Licht.

### 2.3.1 JavaScript kennt keine Datentypen

Das ist eine der bekanntesten Fehleinschätzungen, die immer wieder gegen die Sprache ins Feld geführt werden. Die Behauptung ist schlichtweg falsch. Schon die Existenz des `typeof`-Operators sollte stutzig machen. Der Operator liefert den Datentyp eines Wertes oder einer Variable als Zeichenkette:

```
var bool1 = true;  
console.log( typeof bool1 );  
"boolean"
```

Dabei unterscheidet der `typeof`-Operator vier grundlegende Fälle:

- Für undefinierte Werte oder Variablen ist die Rückgabe: `undefined`.
- Die drei primitiven Datentypen liefern `boolean`, `number` oder `string`.
- Wendet man `typeof` auf eine Funktion an, erhält man `function`.
- Für alle anderen Elemente liefert die Funktion `object`. Dazu gehören ebenfalls Arrays und der Wert `null` für nicht gesetzte Referenzen.

Die folgende Tabelle listet einige typische Beispiele mit den entsprechenden Rückgabewerten auf:

Typ des Operanden	Beispiel	Rückgabe
Undefiniert	<code>typeof newvar;</code>	"undefined"
Boolean	<code>typeof true;</code>	"boolean"
Number	<code>typeof 42;</code> <code>typeof 9.81;</code> <code>typeof 2.817e-15;</code>	"number"
String	<code>typeof "Hallo";</code>	"string"
Funktionen	<code>typeof function add2() {};</code> <code>typeof Math.sin;</code>	"function"
Sonstige Objekte	<code>typeof new String("Hallo");</code> <code>typeof new Boolean(true);</code> <code>typeof {};</code> <code>typeof { x: 1, y: 2 };</code>	"object"
Null	<code>typeof null;</code>	"object"
Array	<code>typeof [ 1, 2, 3 ];</code>	"object"

Einige Besonderheiten springen ins Auge: Funktionen haben offenbar eine große Bedeutung und stehen auf derselben Ebene wie Objekte. Funktionen werden wir intensiv weiter unten in einem separaten Abschnitt untersuchen.

Intuitiv würde man für den Wert `null` die Rückgabe `undefined` anstatt `object` erwarten. Verwirrend ist die Unterscheidung zwischen den beiden Resultaten für den primitiven Typ `string` und das Objekt `String`, das über den Konstruktor (`new`) erzeugt wurde. Intuitiv würde man für beide dasselbe Ergebnis erwarten.

### Primitive Datentypen und Objekte

Die Unterscheidung zieht sich nicht in allen Fällen konsistent durch. So bietet das `String`-Objekt das Attribut `length`, das die Länge der Zeichenkette liefert. Erstaunlich ist, dass die Methode analog auf den primitiven `String` anwendbar ist. Offenbar wandelt JavaScript den primitiven Datentyp automatisch in ein Objekt um. Man erhält den Eindruck, das Zusammenspiel der Typen sei nicht vollständig konsistent:

```
variable2 = new String("Beispieltext");
alert(variable2.length); // Ausgabe: 12
var variable = "Beispieltext";
alert(variable.length); // Ausgabe: 12
```

Wichtig ist in diesem Zusammenhang, dass Objekte als Referenz (»by reference«) an Funktionen übergeben werden. Eine Änderung am Objekt wirkt sich auf das übergebene Objekt aus und ist somit außerhalb der Funktion sichtbar. Primitive Typen werden als Wert (»by value«) übergeben. Änderungen sind nur im Kontext der Funktion sichtbar und nicht nach außen.

## Schwache Typisierung

Damit ist das ursprüngliche Argument entkräftet. In diesem Zusammenhang muss man erwähnen, dass beim Einsatz von Variablen oder Parametern kein Zwang besteht einen Typ zu wählen. JavaScript ist dynamisch typisiert und die Zuweisung eines konkreten Wertes an eine Variable unterliegt keiner Einschränkung oder Prüfung.

Als Fazit bleibt festzuhalten: JavaScript bietet Typen an, verwirrt aber mit einigen schlecht nachvollziehbaren Eigenheiten.

### 2.3.2 Die Funktion eval() ist böse

Mit der Funktion `eval()` kann man sich wie der legendäre Lügenbaron Münchhausen an den eigenen Haaren aus dem Sumpf zu ziehen. Als Argument erwartet die Funktion eine Zeichenkette mit beliebigem JavaScript-Code. `eval()` interpretiert den Code und führt ihn direkt aus. Es lebe die Dynamik von JavaScript.

Damit scheint diese Funktion ein sehr mächtiger Verbündeter zu sein. Das ist richtig, sie hilft Aufgaben zu lösen, die in anderen Sprachen oder mit anderen Mitteln nur mit viel Aufwand umsetzbar sind. Das folgende Beispiel zeigt einen fragwürdigen Einsatz für die Funktion:

```
var object1 = { attribut1: 42 }
var attributName = 'attribut1';
var value = eval('object1.' + attributName);
```

Das Attribut (`attribut1`) der Objektinstanz (`object1`) soll angesprochen werden, dessen Name vielleicht erst zur Laufzeit dynamisch bekannt ist. Das Beispiel funktioniert ohne Probleme. Besser ist aber der Einsatz der Array-Schreibweise, um die Attribute zu nutzen:

```
var object1 = { attribut1: 42 }
var attributName = 'attribut1';
var value = object1[attributName];
```

Diese Lösung kommt zum selben Ergebnis, ist in den meisten Fällen aber schneller in der Ausführung und gleichzeitig besser nachvollziehbar.

In Anhang 1 findet sich ein einfaches Beispiel, wie man mithilfe der Funktion `eval()` dynamisch eine Applikation um neue Funktionen erweitern könnte.

#### Bewertung eval()

Es bleibt die Frage: Sollte man `eval()` einsetzen? Die Funktion erschwert in fast allen Fällen die Lesbarkeit und Verständlichkeit des Programms. Das Debugging und die Fehlersuche werden meist deutlich erschwert.

Grundsätzlich ist die Ausführung langsamer, da `eval()` den JavaScript-Code zuvor interpretieren muss. Es gibt weitere Tricks diesen Geschwindigkeitsnachteil zu mildern, was leider zu einem noch schlechter lesbaren Code führt.

In den meisten Fällen gibt es eine bessere Lösung und mittelfristig zahlt sich der Aufwand für die Suche nach einer Alternative ohne die Funktion `eval()` fast immer aus.

### 2.3.3 Verwirrungen mit Vergleichen: == oder ===?

Das einfache Gleichzeichen formuliert eine Zuweisung in JavaScript. Wie in anderen Sprachen bietet JavaScript das doppelte Gleichzeichen für das Formulieren eines Vergleichs zweier Werte. In JavaScript gibt es noch eine zweite Variante: mit drei Gleichheitszeichen. Kein Wunder, dass diese Situation immer wieder für Verwirrung sorgt.

Die Regel ist im Grunde einfach:

- ➊ Der Vergleichsoperator (==) vergleicht die beiden Elemente links und rechts des Operators und führt eventuell notwendige Typumwandlungen durch.
- ➋ Der Identitätsoperator (===) verhält sich analog, führt aber keine Typkonversionen durch. Passen die Typen nicht direkt zusammen, so schlägt der Vergleich sofort fehl und liefert `false`.

Das heißt, für den Programmieralltag ist in den meisten Situationen der Identitätsoperator das richtige Werkzeug. Man möchte die Werte vergleichen und falls sich Typen unterscheiden, ist fast immer der Wert automatisch nicht passend.

Sollte das nicht ausreichen, ist der zweitbeste Weg, die unterschiedlichen Typen und Werte in Fallunterscheidungen schrittweise und nachvollziehbar zu untersuchen.

Die automatische Typumwandlung des Vergleichsoperators (==) zu nutzen, ist meist sehr fehleranfällig. Der Operator ist nicht transitiv. So ist in vielen Situationen wichtig, welcher Typ auf welcher Seite steht. Die Regeln sind schwer nachvollziehbar und undurchsichtig. Um diese Regeln zu verstehen, sollte man sich viel Zeit nehmen und experimentieren. Die folgenden Beispiele dokumentieren diese Undurchsichtigkeiten:

```
' ' == '0'           // Ergebnis: false
' ' == 0             // Ergebnis: true
0 == '0'             // Ergebnis: true

100 == "1e2"         // Ergebnis: true
"100" == 1e2         // Ergebnis: true
"100" == "1e2"       // Ergebnis: false

false == 'false'     // Ergebnis: false
false == '0'          // Ergebnis: true
false == undefined  // Ergebnis: false
false == null         // Ergebnis: false
null == undefined   // Ergebnis: true

'\t\r\n' == 0          // Ergebnis: true

new String("Hallo") === new String("Hallo");
```

```
// Ergebnis ist false, da beide zwar denselben Inhalt haben,  
// aber es nicht um die gleichen Objekte geht  
  
"Hallo" === "Hallo";  
// Ergebnis: true  
  
// Einen Stringvergleich besser so formulieren:  
var s1 = new String("Hallo");  
var s2 = new String("Hallo");  
s1.toString == s2.toString; // Ergebnis true  
  
"abc" == new String("abc") // Ergebnis: true  
"abc" === new String("abc") // Ergebnis: false  
  
// Objekte müssen inhaltlich verglichen werden, indem alle  
// Attribute untersucht werden:  
[1,2,3] == [1,2,3];  
false  
[1,2,3] === [1,2,3];  
false
```

Für beide Operatoren existieren ebenfalls die negierten Varianten: != und !==.

## 2.3.4 Objektorientierte Programmierung

In JavaScript gibt es weder ein Schlüsselwort, um Klassen zu definieren, noch um Vererbung auszudrücken. Trotzdem ist es möglich, in JavaScript objektorientiert zu programmieren. JavaScript bietet ein Prototypkonzept, das teilweise flexibler ist als viele objektorientierte Sprachen.

### Anlegen von Objekten

Objekte können in JavaScript direkt anhand ihrer Eigenschaften erzeugt werden:

```
var neuesAnonymesObjekt = {  
    var geheimeZahl: 4711,  
    sagDieZahl: function () {  
        return this.geheimeZahl;  
    }  
};  
alert(neuesAnonymesObjekt.sagDieZahl()); // Ausgabe: 4711
```

Die zweite Variante, um Objekte zu definieren, ist formal die Erstellung einer Funktion. Der Kontext der Funktion dient als Container für die Instanz-Variablen. Die Instanz selbst erzeugt der Befehl new gefolgt von dem Funktionsnamen mit entsprechenden Parametern. Innerhalb der Funktion ist der Kontext mit dem Variablenamen this erreichbar:

```
function MeineObjektDefinition(x) { // Konstruktor
    this.zahl = x;
}
var objekt = new MeineObjektDefinition(3); // Instanz erzeugen
alert(objekt.zahl); // per Meldefenster ausgeben (3)
```

## Zugreifen auf Eigenschaften und Methoden

JavaScript bietet gleich mehrere Wege für den Zugriff auf Methoden und Eigenschaften von Objekten:

Die Punkt-Notation ist aus vielen anderen Sprachen bekannt. Zwischen Instanz-Name und Eigenschaft bzw. Methode steht ein Punkt:

```
// Zugriff auf die Eigenschaften und Methoden:
einObjekt.eigenschaft;
einObjekt.methode(parameter1, parameter2);
```

Die Klammer-Notation ist ein alternativer Weg. In eckigen Klammern steht der Name der Eigenschaft oder Methode. Diese Form ist nützlich, wenn man dynamisch zugreifen möchte, und der Name des Attributes in einer Variablen steht:

```
// Dynamischer Zugriff auf Eigenschaften oder Methoden:
einObjekt["eigenschaft"];
einObjekt["methode"](parameter1, parameter2);

// Auflisten aller Eigenschaften eines Objektes:
for(var eigenschaftsName in objekt){
    console.log(eigenschaftsName, '=', objekt[eigenschaftsName]);
}
```

## Verändern von Objekten zur Laufzeit

Die Flexibilität von JavaScript macht ebenfalls vor den Objekten nicht Halt. Eigenschaften und Methoden können zur Laufzeit hinzugefügt und wieder entfernt werden.

Eine neue Eigenschaft wird direkt durch die Benutzung (zum Beispiel die Zuweisung) eines Wertes definiert. Die neue Eigenschaft ist nur in der veränderten Instanz selbst existent:

```
// Neue Eigenschaften definieren:
einObjekt.eigenschaftX = "ein Wert";
einObjekt["eigenschaftY"] = "ein anderer Wert";
```

Das Entfernen einer Eigenschaft erledigt der Befehl `delete`:

```
// Eigenschaften entfernen:
delete einObjekt.eigenschaftX;
delete einObjekt["eigenschaftY"];
```

Der Delete-Befehl funktioniert ebenfalls für das Löschen von normalen Variablen, die global definiert sind.

### Private Attribute

Im gerade beschriebenen Beispiel ist die Instanzvariable öffentlich direkt ansprechbar. Das entspricht nicht dem Geschlossen-Prinzip der Objektorientierung (Open-Closed-Prinzip). Typischerweise erfolgt der Zugriff auf den inneren Zustand einer Instanz nur über eine definierte Schnittstelle. Im Normalfall definiert man hierfür Methoden und versteckt die internen Variablen als private Eigenschaften. Leider bietet JavaScript hierfür kein explizites Sprachkonstrukt.

#### Tipp: S-O-L-I-D-Prinzipien

Diese Prinzipien beschreiben fünf einfache Grundregeln für ein gutes objektorientiertes Design. Weiterführende Informationen finden Sie hier:  
[de.wikipedia.org/wiki/Prinzipien\\_objektorientierten\\_Designs](https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs).

Die gute Nachricht ist, dass das Verhalten von privaten Attributen und Methoden (mithilfe von Funktionen) nachgebildet werden kann:

```
// Wir definieren ein Objekt: Sparschwein
var sparschwein = function () {
    var muenzen = 0;
    var log = function () {
        return "Muenzen: "+muenzen;
    };
    return {
        muenzeRein: function() {
            muenzen += 1;
            console.log( log() );
        }
    };
};

var schwein1 = sparschwein();

schwein1.muenzeRein();
> Muenzen: 1
schwein1.muenzeRein();
> Muenzen: 2
schwein1.muenzen; // Zugriff direkt auf das Attribut scheitert
> undefined
```

Die Variable `muenzen` ist privat und nur im Objekt-Kontext sichtbar. Von außen erreichbar ist dagegen die Methode `muenzeRein()`. Durch die Definition im `return`-Bereich ist sie Teil der öffentlichen Schnittstelle. Aus dieser Methode heraus ist der Zugriff auf

die Elemente möglich, die nur innerhalb des Objekt-Kontexts existieren, zum Beispiel die Methode `log()`.

## Vererbung in JavaScript durch Prototypen

Objektorientierte Sprachen bieten meist Schlüsselworte (wie `extends` und `implements`), um die Ableitung einer Klasse von einer Elternklasse zu formulieren. In JavaScript sucht man diese vergeblich. JavaScript bietet ein Prototyp-Konzept, das es erlaubt Vererbung zu simulieren.

### Vererbung und Klassen

Die Syntax für Vererbung und Klassen befindet sich aktuell in der Planung für den neuen Standard von JavaScript (EcmaScript 6). Preview-Versionen mancher Browser bieten diese Features für experimentierfreudige Entwickler schon an.

Jedes JavaScript-Objekt bietet standardmäßig die Eigenschaft `prototype`. Dieser Eigenschaft kann ein anderes Objekt zugewiesen werden, das als Vorlage dient. Beim Instanziieren neuer Objekte wird die Vorlage zurate gezogen und steuert Eigenschaften des Prototyps bei. Falls Attribute in der neuen Instanz und an dem Prototyp parallel existieren, wird die die Prototyp-Implementierung überdeckt.

Im folgenden Beispiel kann dies an der Eigenschaft `raeder` des Fahrrades und des Dreirades nachvollzogen werden:

```
var Fahrrad = function () {
    this.raeder = 2;
};

var meinFahrrad1 = new Fahrrad();

var Dreirad = function () {
    this.raeder = 3; // Ueberschreiben der Eigenschaft.
};

Dreirad.prototype = new Fahrrad(); // Dreirad erbt von Fahrrad.

var meinDreirad1 = new Dreirad();

// Das Objekt erhält drei Räder von seiner Klasse »Dreirad«.
console.log(meinDreirad1.raeder); // Ausgabe: 3

// Überschriebene Eigenschaft entfernen mit delete.
delete meinDreirad1.raeder;

// Das Objekt hat die Eigenschaft »raeder« nicht mehr selbst
// und es wird die Eigenschaft der Elternklasse (des
// Prototyps) »Fahrrad« sichtbar:
console.log(meinDreirad1.raeder); // Ausgabe: 2
```

Das Urobject (`Object`) von JavaScript vererbt ebenfalls einige Methoden als Standard. Dazu gehört eine Methode, mit der abfragbar ist, ob ein Objekt ein Attribut selbst definiert. Für das Dreirad aus dem Beispiel von oben prüft die folgende Anweisung, ob das Attribut `raeder` vom Objekt selbst definiert wird. Ist das nicht der Fall, muss einer der Vorfahren das Attribut beisteuern:

```
// Ergebnis ist false nach dem Löschen der Eigenschaft.  
meinDreirad1.hasOwnProperty(raeder); // Ausgabe: false
```

Welcher der Vorfahren das Attribut letztlich definiert, sagt die Methode nicht.

**Achtung:**

Beim Aufbau der Vererbung weisen wir eine Instanz der Elternklasse dem Prototyp-Attribut der abgeleiteten Klasse zu. Der Konstruktor des Prototyps wird ausgeführt. Durch die Vererbung selbst werden Instanzen erzeugt. In anderen objektorientierten Sprachen ist das nicht der Fall.

### InstanceOf: Zu welcher Klasse gehörst du?

Manchmal ist es erforderlich ein Objekt nach seiner Klasse zu fragen. Für diese Aufgaben gibt es den `instanceof`-Operator. Dieser prüft für das Objekt auf der linken Seite, ob es eine Instanz der Klasse auf der rechten Seite ist. Für das Beispiel von oben können wir folgende Abfragen formulieren:

```
console.log( meinDreirad1 instanceof Dreirad );  
// Ausgabe: true  
  
console.log( meinDreirad1 instanceof Fahrrad );  
// Ausgabe: true  
  
console.log( meinFahrrad1 instanceof Dreirad );  
// Ausgabe: false  
  
console.log( meinFahrrad1 instanceof Object );  
// Ausgabe: true
```

Die erste Abfrage ist leicht nachvollziehbar. `meinDreirad1` wurde mit `new Dreirad()` erzeugt. Deshalb muss es eine Instanz dieser Klasse sein. Die zweite Anweisung ist ebenfalls wahr, da unser Dreirad eine Ableitung der Klasse Fahrrad ist. `instanceof` prüft offenbar den gesamten Vererbungsbaum über mehrere Ebenen hinweg. Jedes Dreirad ist automatisch ein Fahrrad.

Die dritte Prüfung scheitert, da ein Fahrrad kein Dreirad ist. Die letzte Prüfung zeigt, dass jedes Objekt automatisch von der Basisklasse `Object` erbt.

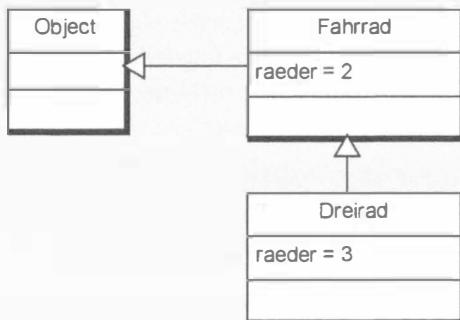


Bild 2.2:  
Das Klassendiagramm des Beispiels.

### Flexibilität mit Prototypen

Die prototypische Vererbung bietet interessante Möglichkeiten, die der klassischen Objektorientierung fremd sind. Nehmen wir uns eine Aufgabe vor: Alle Instanzen einer Klasse sind nach ihrer Erzeugung (zur Laufzeit) um eine neue Methode zu erweitern. Mit statischer Vererbung alleine ist das schwer möglich.

Mit JavaScript ist die Anforderung leicht zu lösen. Die folgenden Zeilen ergänzen den Prototyp (`String`) aller Zeichenketten. Wir erreichen so alle existierenden und zukünftigen Instanzen auf einen Schlag. Die neue `repeat`-Funktion wiederholt die Zeichenkette für eine im Parameter übergebene Anzahl:

```

// Eine neue Funktion für alle Zeichenketten:
String.prototype.repeat = function( count ) {
    var output = '';
    for ( var i=0; i<count;i++ ) {
        output = output + this.toString();
    }
    return output;
}

// Damit steht die Funktion zur Verfügung:
console.log( "e".repeat(11) );
"eeeeeeeeee"
console.log( "e".repeat(-1) );
""
// Das Entfernen der Funktion ist ebenfalls kein Problem:
String.prototype.repeat = undefined;

// Jetzt führt die Anwendung zu einem TypeError:
console.log( "e".repeat(1) );
  
```

JavaScript definiert neben den vorgestellten Prototypen für `String` und `Objekte` ebenfalls Prototypen für alle anderen Datentypen.

Kritisch anzumerken bleibt, dass sich die Formulierung von Objekten und Vererbung holprig anfühlt. Das syntaktische Gewand mutet umständlich an, trotzdem ist die Flexibilität ein mächtiges Hilfsmittel.

### Attribute und Properties

Im objektorientierten Sprachgebrauch bezeichnet der Begriff »Properties« Attribute, die nur über Zugriffsmethoden angesprochen werden. Meist sind dies Set- und Get-Methoden. Damit ist die technische Implementierung des Attributes hinter einer Schnittstelle versteckt und geschützt. Sinnvoll kann dieses Vorgehen sein, um in einer Set-Methode eine Prüfung oder Umwandlung der Werte vorzunehmen.

JavaScript erlaubt im neuesten Standard die Definition von Properties. Im folgenden Beispiel wird die Klasse `quadrat` erstellt. Sie besitzt ein Attribut `laenge`, das die Seitenlänge enthält. Die Fläche des Rechtecks ist über die Get- und Set-Methode zugreifbar, die somit das Property `flaeche` definiert:

```
Var quadrat = {  
    laenge: 10,  
    get flaeche() { return this.laenge * this.laenge },  
    set flaeche(wert) { this.laenge = Math.sqrt(wert) }  
}
```

Das Attribut `laenge` ist nach außen sichtbar und kann wie üblich genutzt werden. Die Zuweisung über die Methode `flaeche()` ändert das Attribut ebenfalls indirekt.

Die Nutzung der beiden Property-Methoden sieht auf den ersten Blick wie eine Zuweisung aus. Technisch gesehen ist es ein Funktionsaufruf:

```
// Auslesen der Länge:  
> quadrat.laenge  
10  
// Auslesen der Fläche (intern ein Funktionsaufruf):  
> quadrat.flaeche  
100  
// Zuweisen einer neuen Fläche:  
> quadrat.flaeche = 64  
64  
// Die Länge wurde entsprechend angepasst:  
> quadrat.laenge  
8
```

Die Kontrolle über die Properties geht noch weiter. Die folgenden Anweisungen lesen die Eigenschaften des Objektes aus und liefern Informationen zum Attribut in Form eines Property-Descriptors. Die Methoden sind am Prototyp `object` implementiert und stehen allen JavaScript-Objekten zur Verfügung:

```
// Meta-Informationen für Attribut "laenge" abfragen:  
Object.getOwnPropertyDescriptor(quadrat, "laenge")
```

```
// Ausgabe:
{
  configurable: true
  enumerable: true
  value: 20
  writable: true
}

// Meta-Information für das Property "flaeche" abfragen:
Object.getOwnPropertyDescriptor(quadrat, "flaeche")
```

```
// Ausgabe:
{
  configurable: true
  enumerable: true
  get: function flaeche() {return this.laenge * this.laenge }
  set: function flaeche(wert) {this.laenge =Math.sqrt(wert) }
}
```

Die Ausgaben für die beiden Aufrufe sind ähnlich, unterscheiden sich jedoch hinsichtlich der Methoden für die Property. Die folgende Tabelle zeigt alle Werte, die in einer Property-Description vorkommen:

Feldname	Bedeutung
writable	Ist der Wert des Attributes änderbar?
value	Liefert den Defaultwert des Attributes.
enumerable	Kann über das Attribut Object.keys iteriert werden?
configurable	Kann die Definition verändert werden?
Get- und Set-Methoden	Listet die Zugriffsmethoden im Falle einer Property auf.

Die Dynamik von JavaScript macht auch bei Objekten keine Ausnahme. Veränderungen sind mit der Methode: `Object.defineProperty()` möglich. So könnten wir auf die Idee kommen, nachträglich das Ändern eines Attributes zu verbieten:

```
// Festlegen der Meta-Informationen des Attributes laenge:
Object.defineProperty(quadrat, "laenge", {
  value: 12,
  writable: false
});

Rechteck.laenge
```

```
> 12
Rechteck.flaeche
> 144
Rechteck.laenge = 17
// Es wird kein Fehler ausgegeben. Der Wert bleibt bei 12.
Rechteck.laenge
> 12
```

Diese Anweisung setzt für das Attribut `laenge` einen neuen Defaultwert von 12 und verbietet weitere Zuweisungen. Das Attribut ist damit eine Konstante. Eine weitere Anwendungsmöglichkeit des Befehls ist das Anlegen neuer Attribute für ein existierendes Objekt.

### Objekte einfrieren

Was für einzelne Attribute funktioniert, erlaubt JavaScript auch für komplette Objekte. Möchte man den kompletten Zustand eines Objekts einfrieren, gibt es hierfür folgende Möglichkeiten. Man kann die Erweiterung von Objekten verbieten:

```
// Erweiterung für das Quadrat verhindern:
Object.preventExtensions(quadrat);

// Wir probieren es trotzdem und legen ein neues Property an:
Object.defineProperty(quadrat, "farbe", { value: "rot" });

// Ausgabe:
TypeError: Cannot define property: farbe, object is not extensible.
```

Das Erweitern ist nur eine Form der Veränderung. Weiterhin könnten Attribute hinzugefügt oder gelöscht werden. Der Befehl `Object.seal()` geht weiter und verhindert das Löschen und Erzeugen von Attributen:

```
// Objekt einfrieren:
Object.seal(quadrat);

// Wir versuchen das Attribut zu löschen:
> delete quadrat.laenge
false
// Die Aktion wird verweigert.
```

Jetzt ist die Struktur des Objektes vor Veränderungen geschützt. Attributwerte bleiben änderbar. Der Befehl: `Object.freeze()` macht alle Attribute zu Konstanten:

```
// Setzen der Länge:
quadrat.laenge = 12;

// Alle Attributwerte einfrieren:
Object.freeze(quadrat);

// Wir wollen den Wert trotzdem ändern:
```

```
quadrat.laenge = 17
>17
// Ausgabe: Keine Fehlermeldung, aber der Wert bleibt gleich.
quadrat.laenge
>12
```

Wenn es möglich ist, die Veränderung von Objekten so fein zu steuern, braucht man eine Möglichkeit, zu prüfen, was für ein Objekt aktuell erlaubt ist. Diese Informationen liefern folgende Funktionen:

```
Object.isSealed(obj)
Object.isFrozen(obj)
Object.isExtensible(obj)
```

Rückgabe ist jeweils ein Boolean-Wert, der anzeigt, ob das angefragte Element veränderbar ist.

Die Funktionen zum Einfrieren von Attributen und Objekten werden vorwiegend bei der Entwicklung von Frameworks und größeren Projekten genutzt, um sicherzustellen, dass keine unerwarteten Veränderungen vorkommen.

### 2.3.5 JavaScript ist in jedem Browser unterschiedlich

In der Vergangenheit brachte jeder Browser-Hersteller seine eigene Geschmacksrichtung von JavaScript auf den Markt. Um diese Unterschiede in den Griff zu bekommen, mussten oft Browser-Weichen in Webseiten implementiert werden. Diese fragen die exakten Informationen zu Betriebssystem, Version und Browser ab und verzweigen zu der passenden Implementierung. Diese Methode erlaubt es, Besonderheiten oder Fehler der einzelnen Implementierungen zu umgehen. Als Entwickler war man sich nie ganz sicher, ob alle Varianten korrekt berücksichtigt wurden und wie sich eine neue Browser-Version in der Zukunft verhält.

In den letzten Jahren hat sich die Lage deutlich entspannt und die Browser-Unterschiede sind in den neueren Versionen deutlich reduziert worden. Ganz aus der Welt ist das Problem nicht, denn die alten Browser sind noch im Umlauf.

Der Einsatz von Frameworks verringert die Unterschiede. Ein gutes Beispiel ist das Framework jQuery ([jquery.com](http://jquery.com)). Es vereinfacht den Zugriff auf den DOM-Baum einer HTML-Seite über alle Browser hinweg und versteckt deren Abweichungen.

### 2.3.6 JavaScript kennt nur globale Variablen

Das Konzept der Sichtbarkeiten (Scopes) von Variablen mutet in JavaScript etwas sonderbar an. Neben der globalen Sichtbarkeit bietet JavaScript lediglich Funktionen als Scopes an. Andere Scopes, zum Beispiel Blöcke in geschweiften Klammern, gibt es nicht. Verwirrend ist, dass Anweisungen wie **If-Then-Else** oder **While-Do** keinen neuen Bereich definieren. Das folgende Beispiel zeigt dieses Verhalten:

```
var a=1;
if (true) {
    var a=2; // Kein neuer Scope: a wird neu zugewiesen.
    alert(a); // Ausgabe: 2.
}
alert(a); // Ausgabe: 2.

function aNewFunction()
{
    var a=3;
    alert(a); // Ausgabe: 3.
}
aNewFunction();
alert(a); // Ausgaben: 2.
```

Die ersten beiden Ausgaben liefern jeweils den Wert 2. Das if-Statement erzeugt keinen neuen Bereich, sondern belegt die lokale Variable (a) neu. Erst die Deklaration der Funktion führt zu einem neuen Scope, wie die dritte Ausgabe aus der Funktion dokumentiert. Die Ausgabe nach dem Funktionsaufruf ist wieder im globalen Bereich und die Variable besitzt den ursprünglichen Wert.

## 2.4 Funktionen: First-Class Citizens

Funktionen sind ein ganz besonderes, mächtiges Werkzeug in JavaScript und gerade für das Steuern der Sichtbarkeit und das Verstecken von Interna eines Algorithmus unverzichtbar.

In den meisten anderen imperativen Programmiersprachen dienen Funktionen als Strukturierungsmittel, um Aufgaben in beherrschbare Teile zu zerlegen, gemäß dem Prinzip: »Teile und herrsche«. Diese Denkweise hat eine lange Tradition aus der strukturierten Analyse und der funktionalen Dekomposition. Dieser Ansatz ist weiterhin guter Stil und mit JavaScript gut umsetzbar. Die folgenden Abschnitte zeigen, was JavaScript-Funktionen zusätzlich zu bieten haben.

### 2.4.1 Definieren von Funktionen

Der ECMAScript-Standard beschreibt drei Wege, Funktionen zu definieren:

#### Der Funktionskonstruktor

Diese Definition mutet wie die Erzeugung eines Objektes an, da das Schlüsselwort `new` eingesetzt wird. Der Funktionsrumpf wird als String im letzten Argument übergeben. Die Argumente davor stellen die zukünftigen Parameter dar.

Der Aufruf der Funktion erfolgt über den Namen der Variablen, der die Funktion zugewiesen wurde:

```
// Deklaration mit Funktionskonstruktor:  
var addiere = new Function('a','b', 'return a + b;');  
alert(addiere(10, 20)); //Ausgabe ist 30.
```

### Funktionen deklarieren

Diese Form ist in vielen anderen Programmiersprachen üblich. Dem Schlüsselwort `function` folgt die Signatur mit dem Funktionsnamen und der Parameterliste. Der Funktionsrumpf steht in geschweiften Klammern danach:

```
// Klassische Deklaration einer Funktion:  
function addiere2(a, b)  
{  
    return a + b;  
}  
  
alert(addiere2(3, 5)); // Ausgaben 8.
```

### Deklarieren über Funktionsausdrücke

Dieser Weg erscheint mehr wie eine Variablenzuweisung. Letztlich wird hier eine anonyme Funktion definiert und einer Variablen zugewiesen. Der Funktionsname nach dem Schlüsselwort `function` ist optional. Wenn die Funktion doch einen Namen erhält, so ist dieser nur im Funktionsrumpf sichtbar und nicht außerhalb.

```
// Deklaration als Funktionsausdruck:  
var addiere3 = function(a, b) { return a + b; }  
  
console.log( addiere3(3, 5) ); // Ausgabe: 8.  
  
// Deklaration als Funktionsausdruck mit optionalem Namen:  
var myFunction = function fname(a, b) {  
    console.log( typeof fname );  
}  
  
console.log( typeof(fname) ); //Ausgabe 'undefined'  
  
myFunction();  
//Ausgabe: 'function'. fname ist im Rumpf bekannt.
```

Das Beispiel zeigt, dass der Name der Funktion `fname` nur im Rumpf bekannt ist.

## Die Unterschiede der Varianten

Es gibt einige feine Abweichungen der drei Alternativen, die zu viel Verwirrung führen können:

```
// f2() kann schon hier benutzt werden
console.log( "f1(): "+ typeof f1 ); // Liefert "undefined".
console.log( "f2(): "+ typeof f2 ); // Liefert "function"
console.log( "f3(): "+ typeof f3 ); // Liefert "undefined"

var f1 = function (a,b) { return a+b; }

function f2 (a,b) { return a+b; }

var f3 = new Function ('a','b', '{ return a+b; }');

// f1() und f3() sind erst jetzt bekannt.
console.log( "f1(): "+ typeof f1 ); // Liefert "function"
console.log( "f3(): "+ typeof f3 ); // Liefert "function"
```

Der Codeblock deklariert drei unterschiedliche Funktionen:

- Funktion **f1** mit der klassischen Deklaration.
- Funktion **f2** mit einem Funktionsausdruck.
- Funktion **f3** mit dem Funktionskonstruktor.

Wider Erwarten sind die Funktionen zu unterschiedlichen Zeitpunkten bekannt. Die klassische Deklaration (**f2**) erkennt der Interpreter schon beim Einlesen der Datei. Die Funktion ist von Beginn der Ausführung an verfügbar. Die beiden anderen Wege machen die Funktion erst nutzbar, wenn der Interpreter bei der Ausführung bis zu der Stelle im Code vorgedrungen ist, an der die Funktion formuliert ist.

Funktionen sollten grundsätzlich erst deklariert und danach genutzt werden. Insbesondere der Weg über den Funktionskonstruktor ist schlechter Stil, da der Funktionsrumpf (ähnlich wie die Funktion `eval()`) als String übergeben wird.

Bleibt noch zu bemerken, dass Funktionen geschachtelt werden können. In der Funktionsdeklaration können also weitere Unterfunktionen spezifiziert werden. Diese definieren selbst einen neuen Sichtbarkeitsbereich.

### 2.4.2 Immediately Invoked Function Expression (IIFE)

Ein beliebtes Stilmittel sind »Immediately Invoked Function Expressions« oder auch Closures genannt. In den meisten Fällen sind es anonyme Funktionen, die direkt einmalig ausgeführt werden. Zu erkennen ist das Konstrukt an der öffnenden und schließenden Klammer am Ende. Außerdem ist die komplette Funktion selbst in Klammern gruppiert:

```
(function(param1) {
    var meineVariable;
    ...
    return meineRueckgabe;
})(argument));

console.log(typeof meineVariable); // Ausgabe: "undefined"
```

Ein Anwendungsfall ist das Nutzen eines lokalen Scopes, der keine unerwünschten globalen Variablen zurücklässt. Die Variablen innerhalb der Funktion (zum Beispiel: `meineVariable`) sind nach der Ausführung nicht mehr sichtbar. Die Rückgabe könnte ein Objekt sein, das trotzdem die internen Variablen kennt und eine öffentliche Schnittstelle anbietet.

### 2.4.3 Funktionen und Parameter

JavaScript geht sehr locker mit Signaturen von Funktionen um. So findet keine explizite Prüfung bei einem Aufruf statt, ob die Anzahl der definierten Parameter einer Funktion zu der tatsächlich übergebenen Anzahl passt. Es gilt eine einfache Regel:

- ➊ Erhält eine Funktion mehr Argumente, als sie Parameter definiert, ignoriert sie die überzähligen Werte.
- ➋ Erhält eine Funktion weniger Argumente als definierte Parameter, so werden die nicht bestückten Parameter mit `undefined` gefüllt.

Das Beispiel stellt beide Situationen vor:

```
function addiere( a, b ) {
    var add = a + b;
    console.log(add);
}

addiere( 2, 5, 9 );
// Überzählige Argumente werden ignoriert. Ergebnis ist 7.

addiere("2","3");
// Ausgabe: "23". Addition funktioniert mit String.

addiere("2");
// Ausgabe: "2undefined". Fehlende Argumente sind »undefined«. // Trotzdem
werden die Zeichenketten verknüpft.

addiere( 5 );
// Ergebnis ist NaN: »Not A Number».
// Undefined konnte nicht in eine Zahl gewandelt werden.
```

Ruft man die Funktion mit zu wenigen Argumenten auf, erhält man keinen Fehler, sondern formal funktioniert der Aufruf. Im letzten Fall ist das Ergebnis »NaN (not a

Number)». Da JavaScript den zweiten Parameter mit `undefined` belegt, funktioniert das Plus als Verkettung von Strings.

Wie zu erwarten war, findet ebenfalls keine Typprüfung der Parameter statt. Dieser Umstand fordert vom Entwickler in vielen Situationen mehr Voraussicht und Disziplin, da man sich nicht sicher sein kann, wie eine Funktion später im Projekt von anderen Entwicklern eingesetzt wird und welche unerwarteten Werte sie erhält.

### Flexibler Zugriff auf Argumente

Genau genommen könnte man sich das Deklarieren von Parametern komplett sparen, denn JavaScript bietet in der Funktion eine flexible Möglichkeit auf die Argumente zuzugreifen.

JavaScript definiert das Feld `arguments`, in dem alle Argumente des aktuellen Aufrufs zugreifbar sind. Die Methode `arguments.length` liefert die Anzahl der Parameter zurück. Damit implementieren wir die Additionsfunktion von eben sehr viel flexibler:

```
function addiereAlle() {  
    var summe = 0;  
    for (var i = 0; i < arguments.length; i++) {  
        summe += arguments[i];  
    }  
    console.log(summe);  
    return summe;  
}  
addiereAlle( 2, 5, 7, 11 ); // Ergebnis ist 25.
```

## 2.5 Einfacher Leben mit JavaScript

Einige der Fallstricke im Umgang mit JavaScript haben wir im letzten Kapitel untersucht. Umso wichtiger ist es, sich Gedanken zu machen, mit welchen Maßnahmen sich die Nachteile reduzieren lassen. Dieses Kapitel stellt einige konkrete Tipps und Tricks aus der Praxis vor. Viele davon lassen sich ohne großen Aufwand umsetzen.

### 2.5.1 Konventionen

Wie in anderen Programmiersprachen helfen Konventionen in JavaScript die Lesbarkeit des Codes zu erhöhen. So mancher Freigeist fühlt sich vielleicht in seiner Kreativität und Entfaltung eingeschränkt, aber sobald mehrere Personen mit demselben Quellcode arbeiten, sollten Konventionen eingehalten werden, damit sich alle besser zurechtfinden. Ein guter Start sind die Konventionen von Google (JavaScript Style Guide) unter der Adresse: [bit.ly/14npCZ](http://bit.ly/14npCZ)). Auf dieser Basis kann sich das Team leicht eigene Regeln ableiten. Dieser Abschnitt umfasst ebenfalls die Begründungen und Negativbeispiele.

## 2.5.2 Behandlung von Fehlern und Ausnahmen

Die Auslöser für Fehlerzustände und Ausnahmen (Exceptions) in einer Applikation sind vielfältig. Oft treten diese auf, weil Systemaktionen fehlschlagen oder Fremdsysteme nicht erreichbar sind. Der korrekte und einheitliche Umgang mit diesen Situationen ist eine Herausforderung in jeder umfangreicheren Applikation.

Das JavaScript-Konstrukt **try-catch-finally** sammelt die definierten Ausnahmen, die innerhalb des Try-Blocks auftraten. Die normale Ausführung wird abgebrochen und der Catch-Bereich ausgeführt. Tritt kein Fehler auf, wird der Catch-Bereich ignoriert.

Im optionalen Finally-Bereich können Aufräumarbeiten, zum Beispiel das Freigeben von Ressourcen, stattfinden. Dieser wird immer angesprungen, ganz gleich, ob ein Fehler auftrat oder nicht.

```
try {  
    // Hier könnten Fehler oder Ausnahmen auftreten:  
    ...  
} catch (exception) {  
    // Im Falle eines Fehler geht die Verarbeitung an  
    // dieser Stelle weiter und die Behandlung des Fehlers  
    // kann erfolgen.  
} finally {  
    // Dieser Block wird in jedem Fall ausgeführt.  
}
```

Über den Befehl `throw` kann man einen Fehler selbst kreieren. Dieser bietet zwei Varianten: Die erste Variante definiert eine Zeichenkette als »Fehlermeldung«, die an den ausführenden Catch-Bereich übertragen wird. Die zweite Variante liefert ein Exception-Objekt. Dieses kann weitere Informationen als Attribute umfassen, zum Beispiel technische Details. Der Code zeigt beide Varianten:

```
// Bricht die Verarbeitung in einem try-catch-Block ab und  
// übergibt die »Fehlermeldung«.  
throw "Fehlermeldung";  
  
// Bricht die Verarbeitung in einem try-catch-Block ab und  
// und übergibt ein Error-Objekt.  
throw new Error("Meldungstext");
```

Leider unterstützen nicht alle JavaScript-Umgebungen das Error-Objekt gleich. Manche Umgebungen liefern neben Nachricht und Name (die im ECMA-Standard vorgeschrieben sind) zusätzlich Zeilenummern, Dateinamen und sogar den Stack-Trace mit. Der folgende Codeblock zeigt, wie eine Fehlermeldung erzeugt wird und im Catch-Bereich die Attribute ausgelesen werden.

```
try {  
    throw new Error("Fehlermeldung");  
} catch (e) {
```

```
    alert(e.name + ": " + e.message);
// Ausgabe: "Error: Fehlermeldung".
}
```

In manchen Situationen können mehrere Fehler auftreten. Leider bietet JavaScript hierfür keine typisierten Catch-Blöcke. Stattdessen behilft man sich meist mit einer Fallunterscheidung, um die unterschiedlichen Fehler separat zu behandeln:

```
try {
  // Hier könnte ein EvalError oder ein RangeError auftreten.
} catch (e) {
  if (e instanceof EvalError) {
    console.log(e.name + ": " + e.message);
  } else if (e instanceof RangeError) {
    console.log (e.name + ": " + e.message);
  }
  // ... etc.
}
```

Die folgende Tabelle zeigt alle vordefinierten Namen, die im Attribut `name` des Error-Objektes vorkommen können.

Fehlername	Beschreibung
EvalError	Bei der Ausführung einer eval()-Funktion trat ein Fehler auf.
RangeError	Ein numerischer Wert ist außerhalb seines zulässigen Bereiches.
ReferenceError	Eine illegale Referenz wurde angesprochen. Der Grund ist meist eine nicht definierte Variable oder Funktion.
SyntaxError	Während der Interpretation des Codes einer eval()-Funktion wurde ein Syntaxfehler gefunden.
TypeError	Es ist ein Fehler bei der Typumwandlung oder Prüfung aufgetreten.
URIError	Im Rahmen des Kodierens und Dekodierens einer URI trat ein Fehler auf, zum Beispiel bei dem Aufruf der Funktion: encodeURI().

### Eigene Fehlerklassen definieren

Für umfangreiche Projekte ist es sehr sinnvoll, eigene Fehlerklassen zu definieren. Diese können zusätzliche Informationen, wie eine anwendungsbezogene Fehlernummer oder eine sprechende Fehlerbeschreibung, enthalten. Über eine Hierarchie von Fehlern und klaren Regeln ist es leichter zu prüfen, welcher Fehler auftrat und wie zu reagieren ist. Das folgende Beispiel definiert eine neue Fehlerklasse `MyError`:

```
// Definieren einer neuen Fehlerklasse: MyError
function NewError(message) {
    this.name = "NewError";
    this.message = message || "Dummy Message";
}
NewError.prototype = new Error();
NewError.prototype.constructor = NewError;

try {
    throw new NewError();
} catch (e) {
    // Ausgabe: NewError: Dummy Message
    console.log (e.name + ": " + e.message);
}

try {
    throw new NewError("Andere Fehlermeldung");
} catch (e) {
    // Ausgabe: NewError: Andere Fehlermeldung
    console.log (e.name + ": " + e.message);
}
```

### 2.5.3 Guter Stil mit dem Strict Mode

In der JavaScript-Version ECMAScript 5 wurde ein neuer »Ausführungsmodus« vorgestellt. Die Intention des Strict Mode ist es, die Qualität und Lesbarkeit von JavaScript zu erhöhen.

Es handelt sich nicht um ein Subset von Befehlen, sondern die Ausführungsumgebung interpretiert einige Sprachkonstrukte auf eine strengere Art. In manchen Fällen verhindert der Strict Mode den Einsatz fehlerträchtiger Konstrukte. Der Modus ist abwärts-kompatibel und man kann sogar sehr punktuell steuern, für welche Code-Teile diese Prüfungen aktiv sind.

Der Modus kann sowohl für eine komplette Datei als auch nur für eine einzelne Funktion aktiviert werden. Das Einschalten funktioniert durch die Anweisung `"use strict";` oder `'use strict';`.

```
/* Strict Mode für die komplette Datei einschalten */
"use strict";
var stringVar = "Das ganze Script im Strict Mode ausführen";

function funktionImStrictMode(){
    "use strict";
    // ... Code der Funktion ...
}
```

Soll der Modus nur für eine Funktion gelten, muss die Deklaration vor der ersten Anweisung in der Funktion stehen. Die Einstellung gilt automatisch für weitere geschachtelte Funktionen. Sinnvoll ist diese Möglichkeit, wenn eine längere Datei mit mehreren Funktionen in Etappen nach und nach umgestellt wird. Kennt ein älterer Browser diesen Modus nicht, interpretiert er das Einschalten als simple Zeichenkette ohne Wirkung.

### Auswirkungen des Strict Mode

Der Strict Mode verändert die Semantik der Ausführung. Zum einen werden einige Warnungen (Silent Errors) des Interpreters zu echten Fehlermeldungen. Damit wird der weitere Ablauf des Scripts abgebrochen. Diese Fehlermeldungen sollten ernsthaft untersucht werden, denn sie zeigen Missverständnisse an oder sind oft auch Hinweise auf Programmierfehler.

Ein dritter Aspekt ist die Zukunftsfähigkeit des Programmcodes. So werden Syntaxänderungen für zukünftige JavaScript-Versionen schon vorweggenommen. Der folgende Abschnitt listet die wichtigsten Änderungen des Strict Mode auf.

### Deklaration von Variablen

In JavaScript können Variablen ohne explizite Deklaration genutzt werden und sind damit automatisch global bekannt. Der Strict Mode erzwingt den Einsatz des Schlüsselwortes `var` für die Deklarationen. Damit können Tippfehler nicht unbemerkt zur Definition einer neuen Variablen führen:

```
"use strict";
xyz = 4711; // Es fehlt "var": ReferenceError.
```

### Überschreiben von vordefinierten Bezeichnern

In JavaScript sind viele Konstrukte zwar erlaubt, haben aber keinen Sinn. So ist es möglich, der vordefinierten Variablen `undefined` einen neuen Wert zuzuweisen. Zum Glück hat diese Anweisung keinen Effekt, da `undefined` (zumindest ab ECMAScript 5) schreibgeschützt ist. Aus diesem Grund macht der Strict Mode diesen Missbrauch sichtbar und wirft einen Fehler aus:

```
"use strict";

//TypeError: "undefined" darf nicht umdefiniert werden.
undefined = 42;

//TypeError: Object.prototype ist schreibgeschützt.
delete Object.prototype;
```

JavaScript bietet weitere vordefinierte Bezeichner, wie das `arguments`-Array. Dieses speichert die Aufruf-Parameter, die eine Funktion erhält, mit ihren ursprünglichen Werten. Die Parameter-Variable kann im Strict Mode nicht mehr verändert werden:

```

function f1_NotStrict(x){
    x++;
    console.log(arguments[0]); //Ergebnis ist 4712 !
}
f1_NotStrict(4711);

function f2_StrictMode(x){
    "use strict";
    x++;
    //Ergebnis bleibt 4711.
    console.log(arguments[0]);
    //Die direkt Zuweisung ist aber möglich.
    arguments[0]=4788;
    console.log(arguments[0]);
}
f2_StrictMode(4711);

```

Der Strict Mode erzwingt die Einzigartigkeit von Attributnamen und Parametern. So müssen sich die Eigenschaften in einem Objektliteral unterscheiden. Das gleiche gilt für die Parameter einer Funktion. Welchem Zweck sollte es dienen, einen Parameternamen doppelt zu deklarieren?

```

"use strict";

// SyntaxError: feld1 kann nicht zweimal vorkommen
var obj = { feld1: "x", feld1: "y" };

// SyntaxError: Parameter müssen unterscheidbar sein
function newFunction(a, b, b){}

```

Der Strict Mode moniert den Einsatz von vordefinierten Schlüsselwörtern für Bezeichner. Dazu zählen ebenfalls einige reservierte Wörter, die aktuell noch keine Funktion haben, sondern erst für die nächsten Versionen geplant sind. Beispiele dieser reservierten Wörter sind: `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static` und `yield`:

```

"use strict";
var package=42;
// Ergibt einen Fehler: "package" ist reserviert.

```

Neue Funktionen können nur auf der Hauptebene oder innerhalb von Funktionen definiert werden. Ohne Strict Mode ist es möglich, neue Funktionen ebenfalls in If-Statements und Schleifen zu erstellen, was in verschiedenen Browsern sogar zu abweichenden Ergebnissen führen kann.

## 2.6 JSON - JavaScript Object Notation

Im Unternehmenskontext spielt der Austausch von Daten zwischen unterschiedlichen Systemen eine große Rolle. Dabei müssen oft Technologie und Zuständigkeitsgrenzen überwunden werden. Die Liste der Formate, die in Projekten eingesetzt werden, ist lang und oft überleben die Formate sogar die Technologie, für die sie ursprünglich erfunden wurden.

In den Jahren ab 1990 entwickelte sich XML (Extensible Markup Language) als das universelle Format für den Austausch von strukturierten Daten. Aus dem Alltag vieler Unternehmen ist es gerade im Umfeld von B2B (Business To Business) nicht mehr wegzudenken. Typische Einsatzbeispiele sind Produktkataloge, Bestellungen oder Prozesse (mit der Business Process Execution Language kurz BPEL). Es gibt viele gute Gründe, es einzusetzen. Die Struktur der Daten kann mit den Tags sehr flexibel und frei als Baum definiert werden. Über DTD (Document Typ Definition) und Schemata ist die Prüfung der Struktur und der Datentypen sehr einfach. Leider ist das Format etwas »geschwäztig« und bläht eine Nachricht schnell auf eine enorme Größe auf, wie schon das einfache Beispiel andeutet:

```
<Buch>
  <ISBN>3839154057</ISBN>
  <Preis>8.50</Preis>
  <Währung>EURO</Währung>
  <Titel>HirnSport.de - Rätsel für das tägliche Gehirnjogging</Titel>
  <Autor>
    <Name>Spindler</Name><Vorname>Heiko</Vorname>
  </Autor>
  <Kategorien>
    <Kategorie>Gehirnjogging</Kategorie>
    <Kategorie>Rätsel</Kategorie>
    <Kategorie>Freizeit</Kategorie>
  </Kategorien>
</Buch>
```

Kein Wunder, dass sich trotz der Vorteile von XML Alternativen etabliert haben, die leichtgewichtiger sind.

### 2.6.1 JSON im Detail

Eine der bekanntesten leichtgewichtigen Varianten für den Datenaustausch ist JSON ([www.json.org/json-de.html](http://www.json.org/json-de.html)). Der Name steht für JavaScript Object Notation. Dieses Textformat ist sehr kompakt, flexibel und sowohl für Menschen als auch Maschinen gut lesbar. Es ist eine ideale Kombination und hat darüber hinaus noch mehr zu bieten:

Jedes JSON-Dokument stellt gleichzeitig einen gültigen JavaScript-Ausdruck dar. Das Einlesen und die Interpretation sind im Handumdrehen erledigt. Davon abgesehen ist JSON in anderen Sprachen sehr gut nutzbar. Die offizielle JSON-Seite dokumentiert

dies eindrucksvoll: Es gibt Parser für alle ernsthaften Programmiersprachen, selbst Urgesteine wie COBOL und RPG sind vertreten. Das XML-Beispiel von oben lässt sich in JSON deutlich kürzer formulieren, ohne dass Struktur oder Übersicht verloren geht:

```
{
  "ISBN": "3839154057",
  "Preis": 8.50,
  "Waehrung": "EURO",
  "Titel": "HirnSport.de - Rätsel für das tägliche Gehirnjogging",
  "Seiten": 120,
  "Lieferbar": true,
  "Kategorien": [ "Gehirnjogging", "Rätsel", "Freizeit" ],
  "Autor": {
    "Name": "Spindler",
    "Vorname": "Heiko",
    "Alter": 42
  }
}
```

Die JSON-Nachricht besteht aus einer Liste von Eigenschaften, umrahmt von geschweiften Klammern. Die Eigenschaften selbst bestehen jeweils aus einem Schlüssel (Key) und einem Wert (Value), getrennt durch einen Doppelpunkt. Jede Eigenschaft muss eindeutig sein. Es darf jeder Schlüssel nur genau einmal vorkommen. (Zumindest pro Kontext). Schlüssel sind gültige Bezeichner von JavaScript. Leerzeichen können zur besseren Lesbarkeit für die Formatierung genutzt werden.

Der Wert kann ein Ausdruck der folgenden Datenformate sein:

Format	Beschreibung
Zahlen	Diese bestehen aus einer Folge von Ziffern und einem optionalen negativen Vorzeichen und Dezimalpunkt. Auch die Angabe in der Exponentenschreibweise ist möglich: 3.14 oder 1e+10.
Zeichenketten (Strings)	Zeichenketten stehen in Anführungszeichen und können Escape- und Unicode-Zeichen enthalten.
Boolesche Werte	Für boolesche Werte sind die Ausprägungen true und false gültig. Achtung: Diese stehen nicht in Anführungszeichen.

Format	Beschreibung
Arrays (Felder)	Ein Array beginnt mit einer eckigen Klammer [ und endet mit einer eckigen Klammer ]. Die einzelnen Elemente sind jeweils durch ein Komma getrennt und können jeden beliebigen JavaScript-Typ annehmen. Außerdem ist es erlaubt, in einem Array verschiedene Typen zu mischen.
Objekte	Diese werden von geschweifter Klammer eingeschlossen und enthalten eine Liste von Attributen und zugewiesenen Werten. Im Grunde kann hier wieder eine komplette neue JSON-Nachricht enthalten sein. Objekte können beliebig geschachtelt werden und auch leere Objekte sind erlaubt.
Nullwerte	Mit dem Bezeichner »null« definiert man einen Nullwert.

Standardmäßig nutzt JSON UTF-8 zur Kodierung der Zeichen. Andere Formate, wie zum Beispiel UTF-16 und UTF-32 können ebenfalls verwendet werden.

## 2.6.2 Parsen und Ausgaben von JSON

Für das Einlesen und Ausgeben von Objekten bietet JavaScript ab Standardversion EcmaScript 5 eigenständige Funktionen:

- ➊ `JSON.parse( String )` liest einen JSON-String ein und erzeugt ein Objekt.
- ➋ `JSON.stringify( Object )` wandelt das Objekt in einen JSON-String um.

Der folgende Codeblock zeigt die Funktionen in Aktion:

```
// JSON-Objekt als String definieren...
var s2 = '{ "ISBN": "3839154057", "Titel": "HirnSport.de - Rätsel für das
tägliche Gehirnjogging", "Seiten": 120 }'

// ...und einlesen.
var asObject = JSON.parse( s2 );

// Ein Attribut des Objekts ausgeben
console.log( asObject.Titel );
«HirnSport.de - Rätsel für das tägliche Gehirnjogging»

// Objekt in JSON-String wandeln ...
var asString = JSON.stringify( asObject );
```

```
// ... und ausgeben.
console.log( asString );
'{"ISBN":"3839154057","Titel":"HirnSport.de - Rätsel für das tägliche
Gehirnjogging","Seiten":120}'
```

Jeder JSON-String ist ein gültiger JavaScript-Ausdruck. Aus diesem Grund kann er alternativ direkt mit der Funktion `eval()` eingelesen werden. Damit bietet sich auch für ältere Browser eine Notlösung, wenn sie nicht über die oben beschriebenen Funktionen verfügen:

```
var obj = eval ("(" + jsonString + ")");
```

## 2.7 Gerüstet für die Zukunft: EcmaScript 6

Die Weiterentwicklung des Sprachstandards ist in vollem Gange. Version 6 (ES6) von EcmaScript hat den Beinamen »Harmony« erhalten. Der Umfang der neuen Funktionen und Erweiterungen reicht von einfachen Kleinigkeiten, die sich nur auf die Syntax beziehen und wenig Einfluss auf den Ablauf haben, bis hin zu umfangreichen Änderungen. Hier sollen nur einige wenige Highlights kurz angerissen werden.

### JavaScript erbt echte Klassen

Zukünftige Versionen werden gerade die Formulierung von Objekten und Vererbung deutlich eleganter abbilden. Vorschläge sind schon ausgearbeitet und Schlüsselwörter definiert:

Das Schlüsselwort `class` definiert Klassen und `extends` benennt die Elternklasse. Die Zukunft wird ebenfalls das Schlüsselwort `constructor` bringen. Diese Erweiterungen werten die Objektorientierung in JavaScript deutlich auf und erhöhen die Lesbarkeit. Das folgende hypothetische Beispiel zeigt, wie Klassen in JavaScript in Zukunft aussehen könnten:

```
class KindKlasse extends ElternKlasse {
    constructor() {
        // Initialisierung:
        super(); // Ruft den Konstruktor der Elternklsse.
    }

    eineMethode(arg) {
        // Hier könnte Logik implementiert werden.
    }
}
```

Die Implementierung von Klassen sieht im Vergleich zur Zuweisung der Elternklasse über das Prototyp-Attribut sehr viel aufgeräumter und verständlicher aus. Unter der Haube wird vermutlich das Prototyp-Konzept der Vererbung weiterleben. Bleibt zu hoffen, dass damit keine Instanz der Elternklasse notwendig ist, um die Vererbung zu formulieren.

### Feinere Sichtbarkeit: Blockscope

Eine andere Erweiterung mutet auf den ersten Blick unscheinbar an. Das Schlüsselwort `let` wird für die Definition von Variablen eingeführt. Es erlaubt, ähnlich wie das Schlüsselwort `var`, das Definieren von Variablen. `Let` verhält sich in Bezug auf die Sichtbarkeit anders. Die mit `let` definierte Variable ist nur in dem Block sichtbar, in dem sie definiert wurde. Damit ist eine feinere Steuerung der Sichtbarkeiten auch für Schleifen und Verzweigungen möglich.

Es ist leider nicht genau abzusehen, wann die Version von EcmaScript 6 final verabschiedet ist und vor allem, wann die Browser diese Version unterstützen werden. Vorab- und Test-Implementierungen sind schon aktuell verfügbar, was auf eine schnelle Adaption hindeutet.





# Single-Page-Web-Apps erstellen

Ziel dieses Kapitels ist es, einen praxisnahen Einstieg in das konkrete Erstellen von Single-Page-Web-Applikationen mit dem Framework AngularJS vorzustellen. Dazu erstellen wir Schritt für Schritt drei Beispiele aus unterschiedlichen fachlichen Bereichen.

## 3.1 Applikationen mit JavaScript – erste Schritte

Bei der Entwicklung von Webseiten hat man es immer mit einer Troika folgender Technologien zu tun:

- HTML beschreibt das Grundgerüst der Seite und integriert im Headerbereich die anderen Applikationsteile.
- CSS definiert das Layout und Styling der Elemente, beginnend mit Farben, Schriftarten und Verhalten in Bezug auf Positionierung.
- JavaScript formuliert die Logik der Applikation und steuert die Darstellung sowie das Zusammenspiel der anderen Teile.

### Klare Zuordnung der Aufgaben

Diese drei Bereiche und ihre Aufgaben sollten möglichst klar abgegrenzt bleiben. Es gibt immer wieder Situationen, in denen es einfach erscheint, in JavaScript dynamisch HTML-Elemente zu definieren und auszugeben. In geringem Umfang und in klar abgegrenzten Bereichen kann das sehr sinnvoll sein. Gerade in größeren Projekten sollte der Grund klar dokumentiert sein, damit sich alle Teammitglieder zurechtfinden und die Gründe nachvollziehen können. Das Durchbrechen der klaren Abgrenzung hat schnell negative Folgen und rächt sich mittel- und langfristig. Die Auswirkungen sind meist eine höhere Fehlerrate bei Änderungen und der Weiterentwicklung, was letztendlich zu höheren Kosten führt.

## 3.2 Handarbeit oder Framework?

In den Anfangsjahren der Web-Entwicklung waren Entwickler darauf angewiesen, alles selber zu programmieren. Der Ansatz stößt sehr schnell an seine Grenzen und schränkt die Möglichkeiten der erstellten Applikationen ein. Erst in den letzten Jahren (ab etwa 2008) entstanden JavaScript-Frameworks, die sich auf breiter Front durchsetzten. Der Haupteinsatz in der Entstehungsphase konzentrierte sich auf die folgenden Punkte:

- die Vereinheitlichung der Unterschiede der diversen Browser, vor allem im Zugriff auf die DOM-Elemente einer Webseite.
- das Durchführen von asynchronen Anfragen (AJAX) aus einer Webseite gegen ein Backend.
- das dynamische Manipulieren von weitestgehend statischen Webseiten zum Beispiel in Form von Ein- und Ausblenden von Bereichen.

In den letzten Jahren hat gerade jQuery gezeigt, wie man die Web-Entwicklung verbessern kann. jQuery hat durch eine einheitliche API die Browser-Vielfalt versteckt und zum anderen mit vielen Plug-ins und nützlichen Features die Web-Entwicklung auf eine neue Stufe der Produktivität gehoben. Das gilt insbesondere für die Dynamik in der Seite und mehr Benutzer-Interaktion.

Der Bekanntheitsgrad von jQuery spricht Bände. Die Statistik von Google Trends verzeichnet mehr Suchanfragen nach dem Begriff »jQuery« als nach der zugrunde liegenden Programmiersprache »JavaScript« selbst.

Umfangreiche Projekte, die eine gewisse Komplexität überschreiten, erfordern fast immer das Definieren von Strukturen, um den erhöhten Umfang an Programmarte-fakten und Quellcode in den Griff zu bekommen. Zum einen müssen sich alle beteiligten Entwickler zurechtfinden, zum anderen müssen diese Strukturen in einer späte- ren Wartungsphase verstanden werden.

Parallel dazu treten weitere wichtige Fragestellungen in den Vordergrund, die das Design einer Applikation betreffen und grundsätzlich gelöst werden müssen. Die folgende Liste ist nur eine kleine Auswahl:

- Wie kommuniziert das Frontend mit dem Server?
- Wie verwaltet man Module und deren Abhängigkeiten?
- Wie verbindet man das Modell (Daten) mit der Darstellung (View)?
- Wie behandelt man Datenänderungen im Umfeld dieser Verbindung?
- Wie kann Logik mit den Kontrollelementen sinnvoll verknüpft werden?

### 3.3 AngularJS: der Superheld an deiner Seite

Ein Framework, das viele dieser wichtigen Aufgaben adressiert, ist AngularJS ([angularjs.org](http://angularjs.org)). Das Projektteam bezeichnet es selbst wenig bescheiden als »super-heroic Web-Framework«.

Das Projekt wird von Google vorangetrieben und sehr aktiv weiterentwickelt. AngularJS steht unter einer Open-Source-Lizenz und ist in der Entwicklungsgemeinde sehr populär. Die Tutorials sind in einem guten Zustand und machen den Einstieg recht leicht. Die Dokumentation könnte für manche Funktionen umfangreicher ausfallen und mehr Erklärung zum Hintergrund liefern. Das Entwicklungsteam hofft offenbar auf mehr Zuarbeit aus der globalen Entwicklungsgemeinde.

The screenshot shows the official AngularJS website. At the top, there's a navigation bar with links for Home, Learn, Develop, Discuss, and Search. Below the header is the AngularJS logo, which consists of a stylized 'A' icon followed by the word 'ANGULARJS' and 'by Google'. A tagline 'HTML enhanced for web apps!' is prominently displayed. Below the tagline are two buttons: 'View on GitHub' and 'Download (1.2.6)'. Social sharing links for Follow on GitHub, G+, Twitter, and Facebook are also present. The main content area is divided into three sections: 'Why AngularJS?' (explaining its expressiveness and extensibility), 'Alternatives' (mentioning other frameworks like Backbone.js and React.js), and 'Extensibility' (describing how AngularJS can be integrated with existing tools). On the right side, there's a live code editor showing an 'index.html' file with AngularJS code, and a preview window showing a 'Hello!' application. A video player for an 'AngularJS Tutorial' is also visible.

**Bild 3.1:** Die Projektseite von AngularJS.

AngularJS ist ein mächtiges Werkzeug. Trotzdem sind der Start und das Umsetzen von einfachen Aufgaben sehr leicht, wie das einführende Beispiel am Ende des ersten Kapitels gezeigt hat.

## MVC-Pattern als Grundlage

AngularJS nutzt als Grundlage das schon lange eingeführte MVC-Pattern. Die View wird in HTML als Templates realisiert. Den Controller formuliert man in JavaScript. Als Modell definiert AngularJS Sichtbarkeitsbereiche, sogenannte Scopes, und folgt an dieser Stelle dem MVVM (Model View ViewModel). Der aktive Scope wird in den aktuellen Controller injiziert. Elegant ist, dass ein Scope nicht von einer Basis-Klasse abgeleitet ist oder andere Vorgaben erfüllen muss, ein simples JavaScript-Objekt reicht aus.

Eine der grundlegenden Funktionen ist eine effiziente Datenbindung zwischen Model und View. Ähnlich wie bei anderen populären Frameworks (zum Beispiel Knockout) funktioniert das Mapping in beide Richtungen ohne weiteres Zutun. Das heißt, eine Eingabe durch den Benutzer landet sofort im verbundenen Modell-Attribut. Ebenso führt eine automatische oder berechnete Modifikation des Modells zur Aktualisierung der Darstellung.

### Aufbau des MVC

Das MVC (Model-View-Controller)-Pattern ist eines der bekanntesten und ältesten Muster der Software-Entwicklung. Die Anwendung in der Entwicklung von Benutzeroberflächen ist sehr verbreitet. Es wurde schon Ende der 1970er Jahre im Rahmen von Smalltalk-Systemen eingesetzt.

Die wesentlichen Teile sind:

- **Das Model** hält die Daten unabhängig von ihrer Darstellung.
- **Die View** stellt Daten des Models nach außen zum Benutzer dar.
- **Der Controller** verbindet View und Model und implementiert die Logik für das Zusammenspiel, die Validierung und die allgemeine Geschäftslogik.

Die Abbildung zeigt das typische Zusammenspiel:

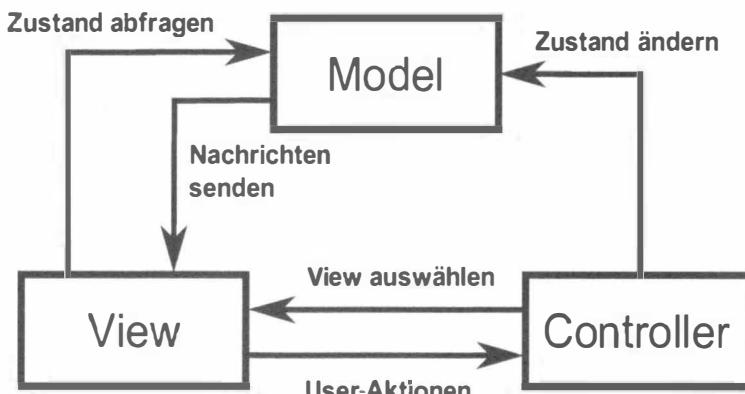


Bild 3.2: Zusammenspiel der Elemente im MVC-Pattern.

## Controller

Die View aus dem Einführungsbeispiel kam ohne eigenständigen Controller aus, da das Binding eine Standardfunktion ist. Im Normalfall würde man für jede View einen eigenen Controller anbieten. Dieser dient als Hort der Logik, die an die UI-Elemente gebunden ist und würde außerdem zwischen dem UI und dem Modell vermitteln.

Im folgenden Beispiel ist der Controller mit dem Attribut `ng-controller` und dem Namen »BookController« an das Formular geknüpft:

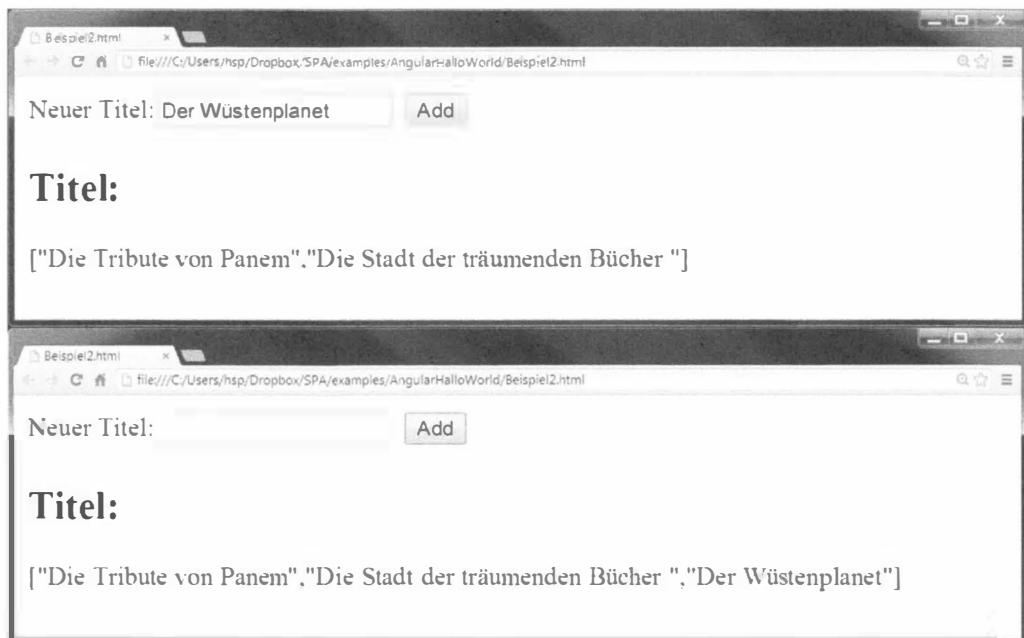
```
...
<div ng-controller="BookController">
    Neuer Titel:<input type="text" ng-model="newtitle" />
    <button ng-click="add()">Add</button>
    <h2>Titel:</h2>
    {{ titles }}
</div>
...
```

Dieser stellt die Logik für die Schaltfläche (`ng-click="add()"`) in seiner Funktion `add()` bereit und initialisiert das Modell-Attribut `titles`. Das Modell instanziert AngularJS automatisch mit den Attributen `newtitle` und `titles` und injiziert es in den Controller als `$scope`:

```
function BookController($scope) {
    $scope.titles = [
        "Die Tribute von Panem",
        "Die Stadt derträumenden Bücher "];

    $scope.add = function() {
        $scope.titles.push($scope.newtitle);
        $scope.newtitle = "";
    }
}
```

Gibt man einen neuen Buchtitel ein und aktiviert den Knopf, fügt die Funktion den neuen Titel zum Array hinzu und leert das Eingabefeld für die nächste Eingabe. Die Änderung am Modell wird automatisch an die Darstellung kommuniziert.



**Bild 3.3:** Das Controller-Beispiel in Aktion: Ein Klick erweitert die Liste.

### Filter

Im einführenden Beispiel wurde eine Eingabe in Großbuchstaben umgewandelt. Mit Filtern formuliert man die Aufgabe noch eleganter:

```
 {{theName | uppercase}}
```

Der Name des Filters folgt nach dem Pipe-Zeichen (|). Der Filter verändert den Wert der Variable, bevor er zur Anzeige kommt. Es ist möglich, mehrere Filter hintereinander zu schalten. Die Ausgaben des ersten Filters fließen als Eingaben in den folgenden Filter, bis zum Schluss ein Resultat vorliegt. Ein sehr mächtiges und trotzdem einfach anzuwendendes Konzept.

AngularJS bringt eine ganze Reihe vordefinierter Filter ab Werk mit, wie die folgende Tabelle aufzeigt:

Filtername	Beschreibung
currency	Formatiert einen Betrag in eine Währungsdarstellung, entweder mit dem Währungssymbol gemäß der Locale-Einstellung oder einem als Parameter übergebenen Symbol.

Filtername	Beschreibung
date	Stellt einen Datumswert flexibel nach einem gewünschten Format dar.
filter	Filtert die Entitäten einer Liste, indem alle Attribute dieser Entitäten nach dem Vorkommen des Suchbegriffs verglichen werden. Der Filter beachtet ebenfalls Teile von Zeichenketten.
json	Wandelt den Input in eine JSON-Darstellung.
limitTo	Beschränkt die Ausgabe einer Liste von Einträgen auf eine maximale Anzahl.
lowercase, uppercase	Wandelt eine Zeichenkette entsprechend in Klein- oder Großschreibung um.
number	Formatiert eine Zahl mit der gewünschten Anzahl an Nachkommastellen.
orderBy	Sortiert eine Liste von Einträgen auf- oder absteigend nach einem definierten Attribut.

Noch beeindruckender als die Liste der verfügbaren Filter ist die Einfachheit, mit der sich eigene, neue Filter definieren lassen. Dazu sind nur wenige Zeilen JavaScript-Code nötig, wie die konkreten Beispiele in diesem Kapitel weiter unten zeigen.

## Services

Services sind wichtige Strukturmittel, um eine Applikation in sinnvolle und nachvollziehbare Module zu zerlegen, um Teile der Geschäftslogik oder die Anbindung an Backendsysteme anzubieten. Die Services können ihre Funktionalität den Controllern (und somit den Views) oder anderen Services zur Verfügung stellen.

Jeder Service sollte gemäß dem Prinzip der Trennung der Zuständigkeiten (»Separation of Concerns«) eine klar umrissene Aufgabe erfüllen. Und jede Aufgabe innerhalb der Applikation sollte nur in einem Service angesiedelt sein. AngularJS bietet mehrere Wege an, um Services zu definieren. Dazu erstellt man zuerst eine neue Applikationsinstanz mit der Anweisung:

```
var app = angular.module('app', []);
```

Für diese Applikation können weitere Module, wie Services, Controller und andere Elemente, erstellt werden. Der Name der Applikation (im ersten Parameter) ist die Verbindung zum Attribut `ng-app` in der HTML-Seite.

Eine Factory liefert ein JavaScript-Objekt als Ergebnis, das die Funktionen als Attribut bereitstellt. Das folgende Quellcode-Beispiel erstellt einen einfachen Service als Factory mit dem Namen »AuftragsService«:

```
app.factory('auftragsService', [ function () {
  return {
```

```
'berechneUst': function (betrag, ustSatz) {
    var ergebnis = betrag * ustSatz;
    return ergebnis;
}
};

}]);

```

Dieser Service bietet eine Funktion an, um die Umsatzsteuer für einen Betrag zu errechnen.

Den erstellten Server wollen wir als Nächstes in einem Controller bekannt machen und verwenden. Dazu erstellen wir einen Controller über die Funktion `controller()`, die AngularJS anbietet und übergeben den Scope und den notwendigen Auftragsservice:

```
app.controller('AuftragsController', [
    '$scope', 'auftragsService',
    function ($scope, auftragsService) {
        $scope.add = function () {
            $scope.endBetrag =
                betrag + auftragsService.berechneUst(betrag, 0.19f);
        };
    }
]);

```

Die zweite Möglichkeit, einen Service zu erstellen, zeigt der folgende Quellcode. AngularJS bietet eine Funktion `service()`, die eine JavaScript-Funktion liefert.

```
app.service('auftragsService', function(){
    this.berechneUst = function(betrag, ustSatz) {
        var ergebnis = betrag * ustSatz;
        return ergebnis;
    }
});

```

In beiden Fällen werden die Services als Singleton erzeugt. Es existiert nur jeweils eine Instanz des Services, die immer wieder erneut in die Konsumenten der Abhängigkeit injiziert wird. Formal liefern beide Methoden der Erzeugung unterschiedliche JavaScript-Strukturen zurück.

Vielleicht ist das Factory-Verfahren etwas besser geeignet, um Details der internen Service-Implementierung und die Schnittstelle nach außen zu trennen.

### Dependency-Injection (DI)

Ein weiteres Highlight von AngularJS ist die konsequente Nutzung von Dependency-Injection, wie es in anderen Frameworks und Programmiersprachen als Best-Practice Standard ist. Der Entwickler vergibt für seine Komponenten, wie Services, Controller oder Filter, eindeutige Namen. Über diese sprechenden Namen können Webentwickler die Komponenten nutzen. So wie zum Beispiel in den letzten Abschnitten.

AngularJS erkennt die Abhängigkeit an den aufgeführten Parametern. Die Verwaltung der Abhängigkeiten (Instanziierung und Injizierung) erledigt AngularJS.

Das Vorgehen ist nicht nur elegant, da man auf eine programmatische Verknüpfung verzichtet, es erleichtert ebenfalls die Testbarkeit der Komponenten enorm. In einem Unit-Test erhält die zu testende Komponente eventuell Mock- oder Dummy-Implementierungen der Komponenten, von denen sie abhängig ist. Die Anwendung im Detail erklären die Beispiele später in diesem Kapitel.

#### K(I)leine Probleme bei der Minifizierung

Durch die Übergabe der notwendigen Services als JavaScript-Variablen bekommt man ein Problem, wenn man den Code aus Sicherheitsgründen minifizieren möchte. Die Namen der Parameter würden auf zufällige und kürzere Namen verändert. Da AngularJS bei der Deklaration eines Services den Namen als Zeichenkette erhält, passen beide nicht mehr zusammen. Als Notlösung definiert man die Parameter mit den abhängigen Komponenten zusätzlich als Zeichenkette, die bei der Minifizierung unverändert bleibt. So wie der Auftrags-Controller im Abschnitt Services.

### Vordefinierte Services von AngularJS

AngularJS bietet zusätzlich eine ganze Reihe von vordefinierten Services an. Dazu gehört zum Beispiel die Möglichkeit, eine Verbindung zu einem Backend per HTTP zu öffnen oder Timer zu erzeugen. Es hebt sich damit von anderen Frameworks, wie zum Beispiel Knockout ([knockoutjs.com](http://knockoutjs.com)) ab. Die folgende Liste zeigt einige nützliche vordefinierte Services:

Servicename	Beschreibung
\$location	Bietet Zugriff auf den Browser-Verlauf (History) und die URL-Anzeige. So kann man die sichtbare URL manipulieren oder auslesen.
\$http	Service für die Kommunikation mit einem Backend über HTTP, um zum Beispiel JSON-Requests zu senden.
\$timeout	Hilft bei der Erstellung von zeitgesteuerten oder wiederkehrenden Aufgaben und kapselt die beiden JavaScript-Funktionen <code>setTimeout()</code> und <code>setInterval()</code> .

### AngularJS-Direktiven

AngularJS treibt den Komponenten-Ansatz wesentlich weiter als andere Tools: Der Web-Entwickler soll nicht nur vorhandene DIV-Elemente umdefinieren und an Logik im Controller binden können, so wie das zum Beispiel jQuery sehr intensiv betreibt. Vielmehr bietet AngularJS mit Direktiven eine Möglichkeit, neue HTML-Elemente zu erfinden. Ein Webdesigner könnte diese neudefinierten Tags für die Konstruktion einer Web-Applikation nutzen. Wie das Element heißt, welche Attribute dieses konfigurieren und welche Semantik dahintersteckt, ist frei definierbar. Dieser Ansatz könnte die Kluft zwischen Entwicklung und Design nachhaltig reduzieren.

Im zweiten AngularJS-Beispiel: »Web-Anwendungen im Unternehmensumfeld« nutzen wir dieses Feature von AngularJS, um eine mächtige Chart-Komponente für Geschäftsgrafiken zu definieren.

#### HTML-Attribute ng-app und ng-model

Die HTML-Attribute `ng-app` und `ng-model` sind nicht ganz HTML5-Standard-konform. HTML erlaubt Erweiterungen mit dem Präfix `data-`. AngularJS kommt mit diesen Attributen ebenfalls zurecht. Es hat sich eingebürgert, das Präfix `data` wegzulassen.

## 3.4 Planung: Link-Verwaltung

Das war nur die Spitze des Eisberges. Was AngularJS noch alles zu bieten hat und wie angenehm es ist, damit Applikationen zu erstellen, sollen mehr Beispiele aufzeigen.

Dazu bauen wir eine Applikation, in der alle Links aus dem Buch auf der Webseite zugänglich sind: den LinkManager mit einer Listendarstellung und typischen Pflegefunktionen zum Anlegen, Ändern und Löschen von Links. Zuerst brauchen wir eine Listendarstellung. Diese soll Filtern, Sortieren und Blättern anbieten. Eine Detailansicht erlaubt es, jeden Link zu modifizieren oder zu löschen.

Das Beispiel wird Schritt für Schritt um neue Funktionen erweitert. Der grobe Ablauf hat folgende Meilensteine:

- View für die Listenansicht erstellen und mit Beispiellinks füllen.
- Implementieren der Mask für das Pflegen der Links.
- Sortieren und Filtern einbauen und in die Listendarstellung integrieren.
- Seitenweises Blättern (Paging) ergänzen.
- Daten im LocalStore des Browsers speichern.
- Kommunikation mit dem Backend über REST implementieren.

Die Masken der Applikation sollen folgendem Aufbau folgen:

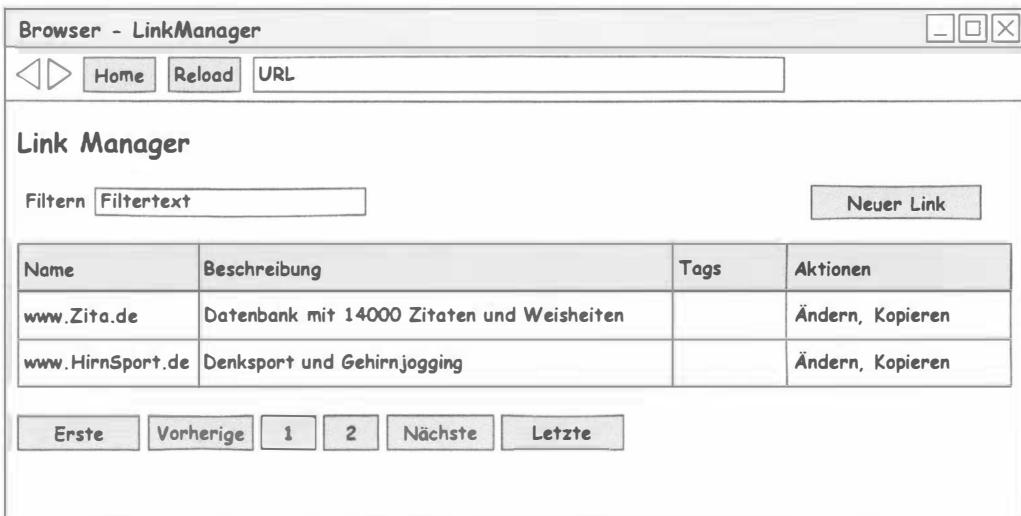


Bild 3.4: Maskenprototypen für die Übersichtsmaske.



Bild 3.5:  
Maskenprototypen für die Detailansicht.

### 3.4.1 Schritt 1: Tabellen-Ansicht

Starten wir mit der View für die Tabelle. Der HTML-Code hält auf den ersten Blick wenige Überraschungen bereit. Im Header kommt zu dem Einbinden von AngularJS eine weitere Bibliothek hinzu: Twitter Bootstrap ([getbootstrap.com](http://getbootstrap.com)). Dies ist eine sehr populäre Designvorlage, die ohne viel manuelle Arbeit ein »responsive« Webdesign

zaubert. Damit verhält sich die Seite auf Geräten mit unterschiedlicher Auflösung und Orientierung sinnvoll: Abhängig von der Bildschirmbreite orientieren sich die Hauptblöcke nebeneinander oder untereinander. Damit ist eine Webseite auf vielen Geräten gleichermaßen nutzbar.

Der Applikationscode ist in eine eigene Datei mit dem Namen `project.js` gewandert. Die `ng-app`-Direktive kennen wir aus dem einleitenden AngularJS-Beispiel. Hinzugekommen sind neue Elemente aus der Familie der `ng`-Attribute.

Das Attribut `ng-view` definiert, an welcher Stelle später dynamische Views eingebunden werden. Zurzeit existiert nur eine View, die in einem Template in Form eines Script-Bereiches abgelegt ist. Das Template ist mit dem Attribut `type="text/ng-template"` und mit einem Namen: `id="list.html"` versehen und ansprechbar:

```
<!DOCTYPE html>
<html>
<head>
    <title>LinkManager</title>
    <meta charset="UTF-8">
    <link rel="stylesheet"
        href="../bootstrap/css/bootstrap.min.css">
    <script src=".angular.min.js"></script>
    <script src=".project.js"></script>
</head>
<body>
    <div ng-app="linkManager">
        <h1>LinkManager</h1>
        <div ng-view></div>

        <script type="text/ng-template" id="list.html">
            <table class="table table-hover">
                <thead>
                    <tr>
                        <th>Name</th>
                        <th>Beschreibung</th>
                        <th>Tags</th>
                        <th>Aktionen</th>
                    </tr>
                </thead>
                <tbody>
                    <tr ng-repeat="link in links | orderBy:'name'">
                        <td><a href="{{link.site}}" target="_blank">{{link.name}}</a></td>
                        <td>{{link.description}}</td>
                        <td>{{link.tags}}</td>
                        <td>
                            &lt;i class="icon-pencil"&gt;</i>
                        </td>
                </tbody>
            </table>
        </script>
    </div>
</body>
```

```

        </tr>
    </tbody>
</table>
</script>
</div>
</body>
</html>

```

Das Template enthält die Tabellendefinition für unsere Links inklusive Header und Zeilen. Besondere Aufmerksamkeit zieht die `ng-repeat`-Direktive auf sich:

```
<tr ng-repeat="link in links">
```

Diese iteriert über alle Einträge in einer Liste mit dem Namen `links` und erzeugt für jeden Eintrag eine lokale Variable mit dem lokalen Namen `link`. Die Ausgabe der Attribute der einzelnen Links erfolgt in gewohnter Manier, mit geschweiften Klammern: `{{link.name}}`. Mit demselben Mechanismus können ebenfalls die HTML-Links aufgebaut werden, über die ein Benutzer zum Ziel des Links springt:

```
<a href="{{link.site}}" target="_blank">{{link.name}}</a>
```

Bleibt die spannende Frage offen: Woher kommen die Daten der Link-Liste? Ein Blick in die JavaScript-Datei lüftet das Geheimnis.

## Ein erster Controller

Unser erster Controller ist sehr übersichtlich. Als Parameter erhält er einen Scope (`$scope`), mit dem AngularJS einen Container für Daten und Funktionen bereitstellt, die in der View verfügbar sein sollen. Besondere Logik brauchen wir in diesem einfachen Fall nicht. Der Controller legt lediglich die Links als statische Liste im Scope an. Für jeden Link stellen wir ein Set von Attributen bereit: Name, Site (Link), Description und Tags. Hinzu kommt ein technischer Identifier (`Id`) zur eindeutigen Kennzeichnung jedes Eintrages:

```

function ListController($scope) {
  $scope.links = [ {
    Id: 1,
    name: 'Zita.de',
    site: 'http://www.zita.de',
    description : 'Beschreibung...',
    tags: 'Zitate'  },
  ...
];
}

```

## Konfiguration einer AngularJS-Applikation

Zunächst wird eine Applikation mit dem Namen `linkManager` über das AngularJS-API erzeugt:

```
var app = angular.module('linkManager', []);
```

Dieser Name stellt die Verbindung zum Attribut `ng-app` in der HTML-Seite her.

Als Nächstes konfigurieren wir die Applikation und stellen eine Verbindung zwischen dem Template und der URL her. Der übergebene Provider `$routeProvider` ist eine vordefinierte AngularJS-Komponente. Dieser bietet die Möglichkeit, über Regeln die Navigation in der Applikation zu definieren. Die Regeln entsprechen der Form: Wenn die URL / aufgerufen wird, instanziere den passenden Controller und das zugehörige View-Template.

In unserem Fall sind das der ListController und das Template mit dem Namen `list.html`.

```
app.config(function($routeProvider) {
  $routeProvider.
    when('/', {controller:ListController, templateUrl: 'list.html'}).
    otherwise({redirectTo:'/'});
});
```

Die abschließende `otherwise`-Regel leitet als Fallback alle sonstigen Pfade ebenfalls auf unsere Hauptansicht um. Wir werden später weitere Routen in die Konfiguration aufnehmen.

Startet man die Applikation, so erscheint sofort die Anzeige der Liste mit den Beispiel-daten aus dem Controller:

Name	Beschreibung	Tags	Aktionen
AngularJS	Superheroic JavaScript MVW Framework.	JavaScript	E.
BrainBrix.com	Besser als Sudoku.	Denksport	E.
HirnSport.de	Rätsel für das tägliche Gehirnjogging.	Denksport	E.
Zita.de	Datenbank für Zitate und Sprüche.	Zitate	E.

Bild 3.6: Ergebnis aus dem ersten Schritt der Link-Verwaltung.

### 3.4.2 Schritt 2: Strukturen schaffen

Damit haben wir den ersten Schritt gemacht. Unschön ist, dass die Daten direkt in dem ListController liegen. Das Ziel ist, eine klare Trennung der Zuständigkeiten aufzubauen für die wesentlichen Elemente:

- View kümmert sich um die Darstellung mithilfe von HTML.
- Model stellt die fachlich motivierten Datenstrukturen bereit.
- Controller steuert das Zusammenspiel zwischen View und Services.
- Services enthalten die Geschäftslogik und integrieren Datenquellen.

In dem kleinen Projekt scheint das auf den ersten Blick übertrieben, aber unsere Applikation wird weiter wachsen. In umfangreichen Projekten sind diese Strukturen eine wichtige Grundlage für eine wartbare und erweiterbare Basis. Als grobe Faustregel sollte pro View ein eigener Controller zuständig sein.

#### Ein Service für die Links

Die Aufgabe für den nächsten Evolutionsschritt ist damit klar: Für die Verwaltung der Links schaffen wir einen Service.

AngularJS bietet Methoden für die Erzeugung von Services an, wie in der Einleitung vorgestellt. Wichtig ist, dass der Service einen fachlich sinnvollen Namen erhält (`linkService`), mit dem er gut referenziert ist.

Die Liste der Links ist unverändert aus dem Controller in den Service gewandert. Für den Zugriff auf die Liste bieten wir eine Methode: `getLinks()`, damit wir die Implementierungsdetails nicht direkt offen legen:

```
app.service('linkService', function () {  
  var data = [  
    {  
      Id: 1,  
      name: 'Zita.de',  
      site: 'http://www.zita.de',  
      description : 'Beschreibung...',  
      tags: 'Kap1'  
    },  
    ...  
  ];  
  
  this.getLinks = function () {  
    return data;  
  }  
});
```

## Verwendung des Service

Benutzen wollen wir den Service in dem bekannten ListController. Dazu erweitern wird die Signatur um eine Variable mit dem Namen des Service `linkService`. AngularJS erkennt, dass der Controller vom LinkService abhängt und stellt diesem automatisch eine Instanz bereit:

```
// ListService in den Controller übergeben:  
function ListController($scope, linkService) {  
    $scope.links = linkService.getLinks();  
}
```

## Filtern und Sortieren

Bisher wurden nur interne Strukturen verändert. Um zu zeigen, wie elegant die Arbeit mit AngularJS ist, erweitern wir die View um eine komfortable Filtermöglichkeit für unsere Linkliste.

Als Erstes definieren wir ein HTML-Eingabefeld über der Listendarstellung, das die Zeichenkette abfragt, nach der gefiltert werden soll:

```
<input type="text" ng-model="search" class="search-query"  
placeholder="Search">
```

Das Attribut `ng-model` definiert wieder die Variable, die AngularJS für den Filterausdruck nutzen wird. Im Attribut `class` wird eine CSS-Klasse von Bootstrap angesprochen, die das Eingabefeld etwas hübscher darstellt. Die Angabe im Attribut `placeholder` erscheint, solange der Benutzer keine Eingabe tätigt. Diese Platzhalter sind ein Standard-Feature von HTML5.

Um die Daten zu filtern, könnten wir im Controller oder im Service die notwendige Logik in JavaScript von Hand implementieren. AngularJS bietet diese Möglichkeit schon an. Dazu erweitern wir die Angabe in der `ng-repeat`-Direktive um einen Filter:

```
<tr ng-repeat="link in links | filter:search | orderBy:'name'">
```

Getrennt von Pipe-Zeichen erwartet AngularJS nach der Angabe `filter` die Variable mit dem Suchbegriff. Diese referenziert unmittelbar die Modellvariable des Eingabefeldes für den Filter.

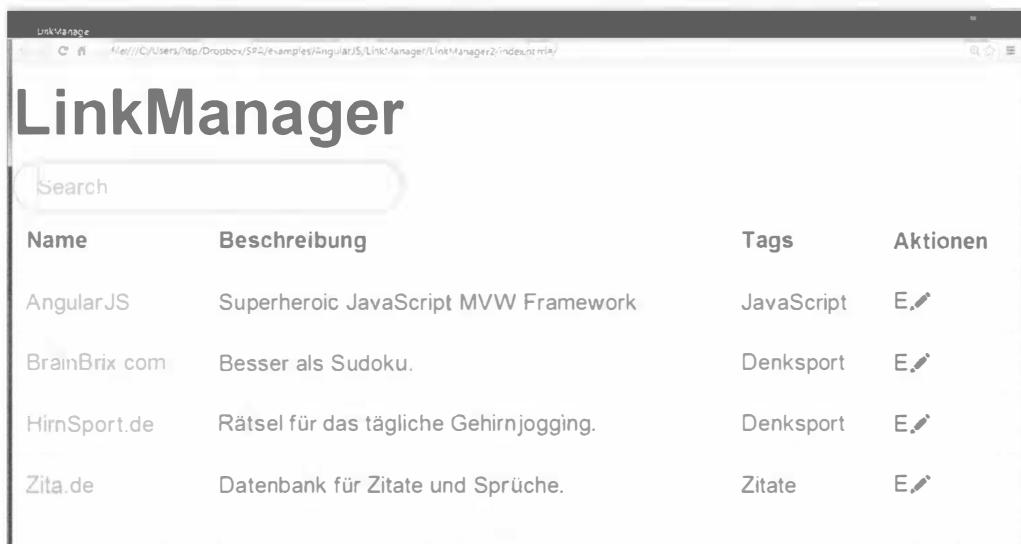
Eine Schaltfläche für das Aktivieren der Filterung ist überflüssig. AngularJS überwacht das Eingabefeld, ändert das Modelfeld und aktualisiert bzw. filtert bei jeder Änderung automatisch.

**Filterung bezieht sich auf alle Attribute**

Die Filterung bezieht sich auf alle Attribute der Links, auch wenn diese nicht angezeigt werden oder nur aus technischen Gründen existieren, wie zum Beispiel der technische Identifier (`Id`).

In unserem Beispiel stört dieses Verhalten nicht sonderlich, ist aber nicht immer erwünscht. In einem späteren Abschnitt reduzieren wir den Filter auf die gewünschten Attribute.

Mit der Angabe für die Sortierung `orderBy` gehen wir einen Schritt weiter und nutzen einen AngularJS-Filter für die Sortierung. In diesem Fall geben wir den Namen des Attributes für die Sortierung fest vor. Aus diesem Grund steht die Angabe in Hochkommas: '`'name'`'. Mit geringem Aufwand könnte man den Namen des Attributes, nach dem sortiert werden soll, dynamisch angeben.



The screenshot shows a web browser window titled "LinkManager". The URL in the address bar is "file:///C:/Users/tsp/Dropbox/SAP/examples/angularjs/LinkManager2/index.html". The page content is a table with the following data:

Name	Beschreibung	Tags	Aktionen
AngularJS	Superheroic JavaScript MVW Framework	JavaScript	E.
BrainBrix.com	Besser als Sudoku.	Denksport	E.
HirnSport.de	Rätsel für das tägliche Gehirnjogging.	Denksport	E.
Zita.de	Datenbank für Zitate und Sprüche.	Zitate	E.

Bild 3.7: Der LinkManager mit Sortierung.

Name	Beschreibung	Tags	Aktionen
BrainBrix.com	Besser als Sudoku.	Denksport	
HirnSport.de	Rätsel für das tägliche Gehirnjogging.	Denksport	

Bild 3.8: Der LinkManager mit gefilterten Daten.

### 3.4.3 Schritt 3: Detailansicht für Links

Bisher spielt sich die gesamte Ansicht in einer einzigen View ab. Als Nächstes erstellen wir eine zweite View für die Detailansicht der Links. Hierzu müssen wir einige Schritte durchführen. Der LinkService erhält eine neue Methode, die einen Link gezielt per Identifier liefert. Die Funktion iteriert über alle Datensätze und vergleicht das Identifier-Attribut, um den gesuchten Link zu finden:

```
app.service('linkService', function () {
    ...
    // unverändert

    this.findById = function (id) {
        for (var i = 0; i < data.length; i++) {
            if (data[i].Id == id) {
                return data[i];
            }
        }
        return null;
    }
    this.getLinks = function () {
        return data;
    }
});
```

Für die neue View legen wir ein zweites Template in der HTML-Datei an. Als Name passt `detail.html` gut. Das Form-Element sollte ebenfalls einen Namen erhalten, damit wir es später leicht referenzieren können. Für jedes Attribut in dem gewählten

Link (`alink`) existiert ein eigener DIV-Bereich, mit jeweils einem Label für die Beschreibung mit den Attributnamen. Zu guter Letzt darf natürlich das eigentliche Eingabefeld für das jeweilige Attribut nicht fehlen. Für die Attribute `name`, `site` (mit der Ziel-URL) und `tags` verwenden wir einzeilige Eingabefelder. Die Beschreibung kann längere Texte mithilfe einer Textarea aufnehmen:

```
<script type="text/ng-template" id="detail.html">
<form name="myForm" class="form-horizontal">
<div class="control-group">
    <label class="control-label">Name:</label>
    <input type="text" ng-model="alink.name" required>
</div>
<div class="control-group">
    <label class="control-label">Website:</label>
    <input type="url" ng-model="alink.site" required>
</div>
<div class="control-group">
    <label class="control-label">Description:</label>
    <textarea ng-model= "alink.description"> </textarea>
</div>
<div class="control-group">
    <label class="control-label">Tags:</label>
    <input type="text" ng-model="alink.tags">
</div>
<div class="control-group">
    <label class="control-label"></label>
    <a href="#" class="btn" >Cancel</a>
</div>
</form>
</script>
```

Die Verbindung zwischen UI-Element und Modell-Attribut stellen wir wie bisher mit dem Attribut `ng-model` her.

Unter dem Formular im letzten DIV-Bereich legen wir einen Link mit der Bezeichnung `Cancel` an, der die Maske schließt. Der Link wird mit der CSS-Klasse `btn` von Bootstrap ausgezeichnet, damit er als Schaltfläche und nicht als Link erscheint. Das Ziel des Links ist die Hauptseite mit der Root-URL im Attribut `href`.

### Neue Pfade im RouteProvider

Die Detailseite soll aus der Linkliste aufgerufen werden. Dazu fügen wir in der letzten Spalte der Übersicht einen neuen HTML-Link ein. Für die Darstellung bedienen wir uns bei den Icons, die Bootstrap mitliefert.

Das Ziel dieses HTML-Verweises definieren wir als `/edit/{{id}}`. Der Identifier des anzuseigenden Links ist in der URL enthalten. Das ist notwendig, damit der Controller für die View die richtige Link-Entität aus unserem Datenbestand suchen und anzeigen kann.

```
<a href="#/edit/{{link.Id}}"><i class="icon-pencil"></i></a>
```

Jetzt können wir aus der Liste zwar einen Link anklicken, unsere Applikation hat aber noch keine Informationen, was bei der Aktivierung passieren soll. Um das zu ändern, erweitern wir die Konfiguration und fügen dem RouteProvider eine neue Regel hinzu:

Wenn die URL `/edit/` (gefolgt von einem Identifier) aktiviert, sollen der EditController und das Template `detail.html` aufgerufen werden.

```
app.config(function($routeProvider) {
  $routeProvider.
    when('/', {controller:ListController, templateUrl:'list.html'}).
    when('/edit/:linkId', {controller>EditController, templateUrl:'detail.html'}).
    otherwise({redirectTo:'/'});
});
```

Mit dem Platzhalter (`linkId`) in der URL nimmt AngularJS den Identifier aus der URL und übergibt diesen an den EditController im zweiten Parameter: `$routeParams`. Mit dieser Information fragt der Controller genau den gesuchten Link beim LinkService über der Methode `findById()` an und erhält das gewünschte Objekt zurück.

```
function EditController($scope, $routeParams, linkService) {
  $scope.alink =
    angular.copy(linkService.findById( $routeParams.linkId ));
```

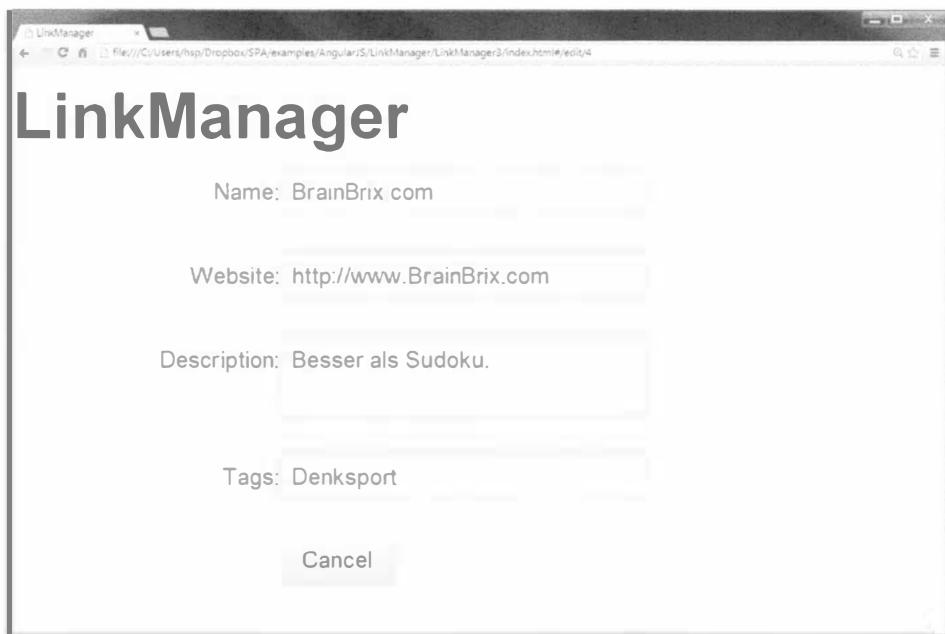
Der Link wird an dieser Stelle mithilfe einer AngularJS-Funktion kopiert. Damit erhält die Detailansicht eine Kopie und keine Referenz auf eine Link-Entität.

Da AngularJS für die Eingabeelemente das Binding automatisch in beide Richtungen anbietet, können wir sogar ohne Mehraufwand eine Editierfunktion bereitstellen. Wir müssen lediglich die Funktionen für das Kopieren entfernen. Dazu lassen wir den EditController keine Kopie erstellen, sondern liefern direkt das gefundene Objekt über den Scope an die View weiter:

```
function EditController($scope, $routeParams, linkService) {
  $scope.alink = linkService.findById( $routeParams.linkId );
```

Die View arbeitet mit derselben Instanz, die im internen Datenbestand liegt. Als Ergebnis sind Änderungen in der Detailansicht direkt in der Listenansicht sichtbar.

Damit ist unsere Detailansicht fertig und es ist möglich, in der Applikation zwischen beiden Ansichten zu wechseln.



**Bild 3.9:** Detailansicht der Link-Applikation.

### 3.4.4 Schritt 4: Anlegen, Ändern, Kopieren und Löschen

Bisher können wir uns eine statische Liste ansehen, nun wird es Zeit, eine vollständige Pflege der Link-Entitäten zu implementieren. Um die neuen Funktionen gut zu strukturieren, ist es sinnvoll, diese wieder in Controller-Klassen zu kapseln und damit klar abzugrenzen. Die folgende Tabelle zeigt die Controller, deren Aufgaben, den Pfad und die verwendeten Views:

Controller	Aufgabe	Pfad	Template
ListController	Die Liste der Links anzeigen.	/	List.html
EditController	Controller für die Detailansicht für einen Link mit den Funktionen: Ändern, Löschen und Speichern.	/edit/:linkId	Detail.html
CopyController	Einen vorhandenen Link duplizieren.	/copy/:linkId	List.html
CreateController	Einen neuen Link anlegen.	/new	Detail.html

Damit sind die Änderungen für die Konfiguration der Routen klar. Wir definieren jeweils für das Kopieren und Anlegen eine neue Route und einen neuen Controller:

```
app.config(function($routeProvider) {
  $routeProvider.
    when('/', {controller:ListController, templateUrl:'list.html'}).
    when('/edit/:linkId', {controller>EditController, templateUrl:'detail.html'}).
    when('/new', {controller>CreateController, templateUrl:'detail.html'}).
    when('/copy/:linkId', {controller:CopyController, templateUrl:'list.html'}).
    otherwise({redirectTo:'/'});
});
```

Beim Kopieren muss wieder ein Identifier des Links übergeben werden, der als Vorlage für die Kopie dienen soll, analog zum Editieren. Ergebnis ist in diesem Fall wieder die Listenansicht und nicht die Detailansicht.

### Änderungen am EditController

Der EditController ist deutlich umfangreicher als im letzten Abschnitt.

```
function EditController($scope, $location, $routeParams, linkService) {

  $scope.alink = angular.copy(linkService.findById( $routeParams.linkId ));
  $scope.original = angular.copy($scope.alink);

  $scope.isClean = function() {
    return angular.equals($scope.original, $scope.alink);
  };
  $scope.close = function() {
    $scope.original = null;
    $scope.alink = null;
    $location.path('/');
  }
  $scope.destroy = function() {
    linkService.deleteLink( $scope.alink );
    $scope.close();
  };
  $scope.save = function() {
    linkService.updateLink( $scope.alink );
    $scope.close();
  };
}
```

In den ersten beiden Zeilen wird die zu editierende Entität gesucht, kopiert und im Scope deponiert. Einmal als Modell, auf dem die Änderungen erfolgen und einmal als

Kopie des ursprünglichen Zustandes. Die Methode `isClean()` nutzt die AngularJS-Funktion `equals()`, die beide Links inklusive der Attribute auf Gleichheit prüft. Sind beide Link-Entitäten gleich, wurde in der Maske offenbar keine Änderung vorgenommen. Damit können wir in der Maske die Schaltfläche zum Speichern der Daten deaktivieren.

Die Funktion `close()` räumt die Variablen auf und springt zurück auf die Listenansicht, indem der entsprechende Pfad über den `$location`-Service gesetzt wird.

Die Methode `destroy()` löscht den Link über den LinkService und kehrt mit der Methode `close()` zur Hauptansicht zurück. Analog verhält sich die `save()`-Methode, mit dem Unterschied, dass eine andere Service-Methode angesprochen wird: `updateLink()` mit der veränderten Kopie des Links.

### Die neuen Controller für das Pflegen

Der CreateController ist für die AngularJS-Konfiguration ebenfalls mit der Detailansicht verknüpft. Damit erzeugt AngularJS automatisch eine neue Modellinstanz für die Pflege der Eingabefelder. Beim Speichern übertragen wir dieses Modell über den Service in unsere interne Datenstruktur und kehren zur Listenansicht zurück.

```
function CreateController($scope, $location, linkService) {
  $scope.save = function() {
    linkService.addLink( angular.copy( $scope.alink ) );
    $location.path('/');
  }
}
```

Der CopyController ist nicht mit der Detailansicht verknüpft, sondern erhält seine Vorlage über den Identifier in der URL und erzeugt selbst eine Kopie der Link-Entität.

Bei der Kopie verändern wir lediglich das Attribut für den Namen um den Zusatz `New`. Damit ist die Kopie in der Listenansicht am neuen Titel erkennbar. Die anderen Attribute muss der Benutzer über die Editierfunktion selbst ändern. Die Kopie fügt der LinkService zur internen Datenstruktur hinzu.

```
function CopyController($scope, $location, $routeParams, linkService)
{
  var p =
    angular.copy(linkService.findById( $routeParams.linkId ) );
  p.name = p.name + 'New';
  linkService.addLink( p );
  $location.path('/');
}
```

### Änderungen in der Detail-Darstellung

In der Listendarstellung erscheinen nur kleine Änderungen: Der Button für die Neuanlage wird ergänzt und ruft den Pfad zur Editierfunktion ("#/new") in der Detailansicht auf.

```
<a href="#/new">New<i class="icon-plus-sign"></i></a>
```

In der Detailansicht haben wir mehr Arbeit vor uns. Wir ergänzen die Eingabeelemente um die Anzeige von Fehlermeldungen. In die DIV-Bereiche, die das Label und das Eingabeelement enthalten, nehmen wir jetzt einen SPAN-Bereich für die Anzeige von Fehlermeldungen auf. Diese Fehlermeldungen resultieren aus der Validierung der Eingaben des Benutzers. Folgende Regeln sind definiert:

- Der Name wird zu einem Pflichtfeld, indem das Input-Element ein `required`-Attribut erhält.
- Der Verweis (im Attribut `alink.site`) muss eine gültige URL enthalten, das heißt komplett im Format `http://www.domain.tld`. HTML bietet dafür das Attribut `type` und den Wert `URL` an.

Diese Möglichkeiten sind Funktionen von HTML5, die grundsätzlich auch ohne AngularJS funktionieren. Um die Details der Prüfung brauchen wir uns nicht selbst zu kümmern. In unserem Fall wertet AngularJS diese Meldungen zusätzlich aus und steuert die Sichtbarkeit der Fehlermeldung. Der folgende Quellcode zeigt einige Bereiche der vollständigen View als Auszug:

```
...
<form name="myForm" class="form-horizontal">
  <div class="control-group" ng-class="{error: myForm.name.$invalid}">
    <label class="control-label" >Name:</label>
    <input type="text" name="name" ng-model="alink.name" required>
    <span ng-show="myForm.name.$error.required" class="help-inline">
      Required</span>
  </div>

  <div class="control-group" ng-class="{error: myForm.site.$invalid}">
    <label class="control-label" >Website:</label>
    <input type="url" name="site" ng-model="alink.site" required>
    <span ng-show="myForm.site.$error.required" class="help-inline">
      Required</span>
    <span ng-show="myForm.site.$error.url" class="help-inline">
      Not a URL</span>
  </div>
...
</form>
...
```

Project	Description	Tags	New
AngularJS	Superheroic JavaScript MVW Framework.	JavaScript	E C A
BrainBrix.com	Besser als Sudoku.	Denksport	E C A
HirnSport.de	Rätsel für das tägliche Gehirnjogging.	Denksport	E C A
Zitate.de	Datenbank für Zitate und Sprüche.	Zitate	E C A

Bild 3.10: Die Listenansicht zeigt die Links für Aktionen.

The screenshot shows a modal dialog titled "LinkManager" for editing a link. The form contains the following fields:

- Name:  Required
- Website:  Required
- Description:  Neue Beschreibung
- Tags:  Neuer Tag

At the bottom are three buttons: Cancel, Save (highlighted in grey), and Delete.

Bild 3.11: Detailansicht der Link-Applikation mit Validierung.

## Neue Schaltflächen für die Detail-Darstellung

Bisher hatte die Schaltfläche nur eine Aufgabe, die Detailansicht zu schließen. Über neue Aktionsbuttons wollen wir jetzt mehr Komfort für den Benutzer anbieten:

```
<a href="#" class="btn" ng-click="close()">Cancel</a>
```

Der Cancel-Button ruft die Controller-Methode `close()` auf und verwirft die Änderungen. Dieser Button unterscheidet sich in seiner Darstellung nicht von den anderen Schaltflächen. Im HTML-Code ist er als Link realisiert. Durch diesen Kniff umgehen wir die Prüfung der Eingabe. Es ist wenig sinnvoll, den Benutzer zur Eingabe einer korrekten URL für den Link zu zwingen, wenn dieser seine Eingaben verwerfen möchte.

Der Button zum Speichern ruft die `save()`-Methode des Controllers auf. Der Knopf ist nicht immer aktivierbar. Über das Attribut `ng-disabled` ist die Logik hinterlegt. Der Button ist deaktiviert, wenn eine der beiden Bedingungen wahr ist:

- Die Linkdaten sind noch unverändert. Die Controller-Methode `isClean()` liefert uns diese Information.
- Eines der Eingabefelder ist nicht korrekt gefüllt. Diese Information stellt uns AngularJS für die Formular-Variablen `myForm.$invalid` zur Verfügung.

```
<button ng-click="save()"  
       ng-disabled="isClean() || myForm.$invalid"  
       class="btn btn-primary">Save</button>
```

Neu hinzugekommen ist der Delete-Button. Über das `ng-show`-Attribut wird er nur angezeigt, wenn der Link ein gesetztes Identifier-Feld besitzt. Beim Pfad »/neu« wird die Detail-View ohne Identifier dargestellt und kennzeichnet somit (ohne Vorbelegung) eine neu angelegte Link-Entität, die nicht gelöscht werden kann:

```
<button ng-click="destroy()" ng-show="alink.Id" class="btn btn-  
danger">Delete</button>
```

Alternativ könnte man dem Controller eine Methode spendieren, die die Prüfung übernimmt, ob das Löschen in der Detailanzeige sichtbar sein soll. Das ist sinnvoll, wenn die Logik für die Prüfung umfangreicher wird.

## Änderungen im LinkService

Die Grundlage für die neuen Aktivitäten in den Masken sind entsprechende Funktionen im Service. Irgendwie müssen wir die Daten letztlich in unserer (wenn auch sehr einfachen Datenstruktur) ablegen oder verändern. Der Service erhält einige neue Methoden, um die Linkdaten zu verwalten:

```
app.factory('linkService', function () {  
  var data = [  
    // ... wie bisher  
  ];  
  
  // Private Methode zum Kopieren eines Objekts
```

```

function copy(quelle, ziel) {
  if (!ziel)
    ziel = {};
  Object.keys(quelle).forEach(function (val) {
    ziel[val] = quelle[val];
  });
  return ziel;
}

return {
// ... findById() und getLinks() wie bisher

  addLink: function (item) {
    // Generate a unique id.
    item.Id = new Date().getTime();
    data.push(item);
  },
  deleteLink: function (item) {
    var tempItem = this.findById( item.Id );
    data.splice(data.indexOf(tempItem), 1);
  },
  updateLink: function (item) {
    var p = this.findById( item.Id );
    if (!p)
      this.addLink( item );
    copy( item, p );
  }
};
});

```

`AddLink()` fügt einen neuen Link hinzu. Unsere Datenstruktur für die Links ist ein Array und mit dem `push()`-Befehl legen wir neue Elemente an. Um die Links eindeutig zu kennzeichnen, erzeugt die Methode einen Identifizierer für jeden neuen Link. In einem produktiven System würde dieser Mechanismus eventuell über eine Datenbank-Sequenz generiert. Wir simulieren das Erzeugen von eindeutigen technischen Schlüsseln, indem wir über die JavaScript-Funktion `getTime()` das aktuelle Datum als Wert in Millisekunden erfragen.

Die Funktion `deleteLink()` nutzt eine andere JavaScript-Arrayfunktion: `splice()` entfernt eine Anzahl von Elementen aus einem Array. Wir identifizieren den zu löschen Link-Eintrag mit `indexOf()` und entfernen von dieser Position genau ein Element. Da im `EditController` zuvor eine Kopie des Links erstellt wurde, suchen wir die Entität, damit die Funktion `indexOf()` mit der korrekten Referenz arbeiten kann. Alternativ könnten wir eine eigene `indexOf()`-Funktion erstellen, die mit dem Identifier-Attribut der Links arbeitet. Dann könnten wir eine Iteration über alle Links einsparen.

Die Funktion `updateLink()` ändert einen vorhandenen Link ab. Dazu muss dieser erst über den Identifier und unsere schon bekannte Funktion `findById()` gefunden werden. Danach kopieren wir alle Attribute mit einer eigenen, privaten Kopierfunktion. Die private Copy-Funktion erhält zwei Objekte: eine Quelle und ein Ziel. Ist das Ziel nicht definiert, erzeugt die Funktion ein leeres Objekt. Danach liest die Funktion mit `Object.keys` alle Attribute der Quelle aus und überträgt die Attribute mit ihrem Inhalt in das Zielobjekt.

Eine einfache Zuweisung des Quellobjektes auf das Zielobjekt würde eine weitere Referenz auf dasselbe Objekt erzeugen. Wir wollen an dieser Stelle aber eine vollständige neue Instanz erzeugen, die unabhängig geändert werden kann. Die Copy-Methode geht davon aus, dass unser Objekt nur primitive Attribute enthält. Wenn das nicht der Fall wäre, müssten wir alle Attribute selbst mit der Copy-Methode rekursiv kopieren und ein Deep-Copy implementieren.

Alternativ könnten wir die AngularJS-Funktion für das Kopieren nutzen, die wir im letzten Abschnitt gesehen haben. Dieses Vorgehen baut leider für den LinkService eine neue Abhängigkeit zu AngularJS auf. Bisher ist der LinkService möglichst unabhängig und könnte ebenfalls in einem ganz anderen Kontext genutzt werden. Aus diesem Grund wurde die Entscheidung für eine eigene Copy-Methode getroffen. Für die Controller-Klassen ist die Lage anders: Diese sind ohnehin mehr mit AngularJS verbunden und weitere Abhängigkeiten sind nicht störend.

### 3.4.5 Schritt 5: Filtern, aber richtig

Für große Datenmengen (deutlich jenseits der 1000 Einträge) sollte das Suchen, Filtern oder Blättern auf die Datenbank oder ein anderes datenlieferndes Backend abgewälzt werden. Diese sind (meist) durch spezielle Funktionen oder Datenstrukturen besser für diese Aufgaben geeignet. In unserer Größenordnung reicht der Browser vollkommen aus.

Das Filtern mit dem Standardverhalten von AngularJS ist elegant und schnell eingebaut. Es ist für viele Fälle gut geeignet und ausreichend. Für unseren Einsatz hat es leider einen gravierenden Nachteil:

AngularJS filtert über alle Attribute der Entitäten. Das bedeutet für uns leider, dass ebenfalls das technische Attribut `Id` durchsucht wird. Und die Lage ist noch schlimmer. AngularJS selbst erweitert in manchen Fällen Entitäten um temporäre, technische Felder. So könnte im Debugger eventuell ein Feld mit dem Namen `$$hashKey` erscheinen (je nach eingesetzter Version von AngularJS). Auch vor diesem Feld macht der Standardfilter nicht halt. Unsere Ergebnisse sind somit in manchen Situationen verfälscht: Zur Lösung erstellen wir einen eigenen Filter.

Damit der Filter generisch einsetzbar ist, soll konfigurierbar sein, welche Attribute für den Vergleich herangezogen werden. So haben wir die Freiheit, vielleicht später einen reinen Tag-Filter zu bauen.

Im Zusammenhang mit den anderen Aufgaben der Übersichtsliste müssen wir folgende Reihenfolge einhalten:

- Filtern oder Suchen in der gesamten Datenmenge.
- Sortieren der erhaltenen Ergebnismenge.
- Die sortierte Ergebnisliste auf Seiten verteilen, zwischen denen der Benutzer blättern kann.

Insbesondere das Aufteilen auf die Seiten beim Blättern ist erst zum Schluss sinnvoll. Durch die Sortierung kann sich die Aufteilung der Treffer auf Seiten vollständig ändern. Die folgende Abbildung zeigt die drei Schritte im Überblick. Im nächsten Abschnitt widmen wir uns den Aufgaben Filtern und Blättern.

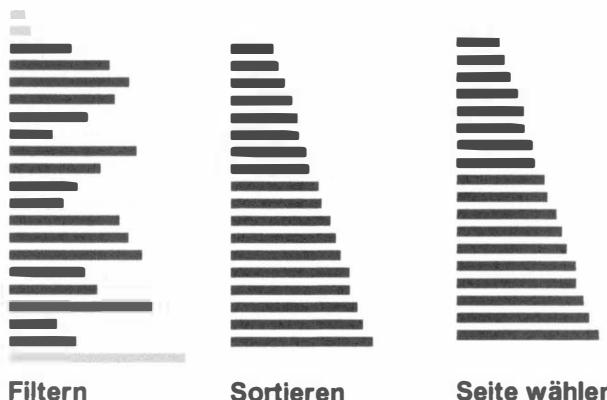


Bild 3.12: Der Ablauf für das Erstellen der Ergebnisliste.

### Filtern: Ein zweiter Blick

AngularJS bietet Funktionen für das Definieren von eigenen Filtern. Dabei müssen wir dem neuen Filter folgende Parameter übermitteln:

Fachliche Information	Name des Parameters im Filter	Beschreibung
Linkliste	data	Der gesamte Datenbestand unserer Links, der gefiltert werden soll.
Filterstring	queryStr	Die Zeichenkette, nach der gesucht werden soll.
Zu filternde Attribute	includedAttributes	Eine Liste der Feldnamen der Link-Entitäten (als Array von Strings), die der Filter beachten soll. Alle anderen Felder ignoriert der Filter.

Ergebnis ist die Liste der Links, in denen der Suchbegriff in einem der gewünschten Attribute erscheint. Der folgende Quellcode-Block enthält die Implementierung des neuen Filters. AngularJS bietet eine eigene Funktion an, um neue Filter zu erstellen: `app.filter()`. Dieser AngularJS-Funktion übergeben wir den Namen des neuen Filters (`customFilter`) und eine neue Funktion, die die eigentliche Logik des Filters durchführt. Diese Funktion erhält die oben beschriebenen Parameter:

```
app.filter('customFilter', function () {
  return function (data, queryStr, includedAttributes) {
    var resultArray = [];
    for (var i = data.length; i--;) {
      var item = data[i];
      var found = false;
      for (var attr in item) {
        if (includedAttributes.indexOf( attr ) >-1) {
          if (searchMatch(item[attr], queryStr)) {
            found = true;
            break;
          }
        }
      }
      if (found) {
        resultArray.push(item);
      }
    }
    return resultArray;
  }
});
```

Der grundsätzliche Ablauf ist nicht außergewöhnlich: Der Filter iteriert über alle Einträge der erhaltenen Data-Liste. Für jedes Element prüft die innere for-Schleife alle Attribute des Elements, ob diese für das Filtern zu berücksichtigen sind. Das ist der Fall, wenn der Feldname in der Liste der gewünschten Felder enthalten ist. Dann liefert `indexOf()` einen Wert größer -1.

Den eigentlichen Vergleich übernimmt die separate Funktion `searchMatch()` und erhält das zu prüfende Attribut (`item[attr]`) und die Filterzeichenkette, nach der wir suchen:

```
function searchMatch(haystack, needle) {
  if (typeof( haystack ) != "string" )
    return false;
  if (!needle) {
    return true;
  }
  return haystack.toLowerCase().
    indexOf(needle.toLowerCase()) !== -1;
};
```

Diese Funktion prüft lediglich, ob der zweite Parameter in dem ersten enthalten ist. Dabei wird die Groß- und Kleinschreibung ignoriert. Durch den Einsatz von `indexOf()` findet unsere Implementierung die gesuchten Zeichenketten ebenfalls, wenn diese als Wortteile vorkommen.

### Den neuen Filter anwenden

Damit ist der Filter erstellt. Zur Anwendung kommt der Filter im ListController und wir führen hierfür eine eigene Funktion mit dem Namen `search()` ein. Der Filter wird programmatisch aufgerufen und nicht aus der HTML-Seite angesprochen:

```
$scope.search = function () {
  if (!$scope.query || $scope.query === "") {
  }
  $scope.filteredItems = linkService.getLinks();
} else {
  $scope.filteredItems = $filter('customFilter')(linkService.getLinks(), $scope.query, ["name", "desciption", "tags" ] );
  $rootScope.query = $scope.query;
}

// take care of the sorting order
// will be next step...

// now group by pages
$scope.groupToPages();

// check current page
if( !$scope.currentPage)
{
  $scope.currentPage = 0;
}
$scope.currentPage = Math.min(
  $scope.currentPage, $scope.pagedItems.length-1 );
$rootScope.currentPage = $scope.currentPage;
};
```

Zuerst prüft die Methode, ob ein Filter vorliegt. Ohne Filter reichen wir die Liste der Links unverändert als Ergebnis zurück.

Wenn ein Filterbegriff vorliegt, übergeben wir diesen an unseren neuen Filter und versorgen ihn mit den Werten für die Linkliste, den Filterbegriff und die Liste der Attributnamen, die zu filtern sind. Im aktuellen Fall sind das die Attribute `Name`, `Beschreibung` und `Tags`.

Die gefilterte Ergebnisliste wandert in die Variable `filteredItems`. Diese werden wir nicht direkt anzeigen lassen, sondern am Ende der Funktion vorher in Seiten aufteilen. Die Funktion zum Aufteilen auf die Seiten wird aufgerufen.

Zum Schluss setzt die Search-Funktion die aktuelle Seite neu, falls diese nicht schon existiert. Durch die Filterung könnten jetzt weniger Seiten sichtbar sein. Aus diesem Grund wird das Minimum aus der aktuellen Seite und der maximalen Anzahl an Seiten gebildet.

Der Filterbegriff wird zwischenzeitlich in den `$rootScope` in die Variable `query` abgelegt. Dadurch ist die Information für die Applikation global und steht bei einem Wechsel oder einer neuen Initialisierung des Controllers wieder zur Verfügung. Denn AngularJS vererbt alle Variablen aus dem `$rootScope` auf den `$scope` eines konkreten Controllers.

In unserer Applikation wandern folgende Informationen in den `$rootScope`:

- der Filterbegriff
- die ausgewählte Seite
- die gewählte Sortierung (folgt im nächsten Abschnitt)

Damit der ListController den `$rootScope` nutzen kann, müssen wir nur die Signatur des Controllers erweitern. AngularJS übergibt den `$rootScope` automatisch:

```
function ListController( $scope, $rootScope, $filter, linkService ) {  
  ...  
}
```

Diese Fähigkeit von AngularJS sollte nur in geringem Umfang genutzt werden. Sonst besteht schnell die Gefahr, den Überblick zu verlieren, in welchem Sichtbarkeitsbereich die aktuellen Informationen liegen.

Alternativ könnten wir für den Datenaustausch einen Service erstellen oder eine globale JavaScript-Variablen nutzen. Der `$rootScope` scheint an dieser Stelle ausreichend.

#### Daten zwischen Scopes austauschen:

AngularJS kann Nachrichten (inklusive Parameter) zwischen Scopes senden mithilfe folgender Funktionen:

Für das Senden an einen untergeordneten Scope:  
`$scope.$broadcast('aMessage', { key: 'value' });`  
Soll die Nachricht an alle übergeordneten Kontexte:  
`$scope.$emit('aMessage', { key: 'value' });`

Mit einer Broadcast-Nachricht aus dem `$rootScope` erreicht man alle Scopes, da dem `$rootScope` alle anderen Kontexte untergeordnet sind.

Ein Controller oder Service könnte auf Nachrichten vom Typ »aMessage« lauschen, indem er eine `$on()`-Funktion anbietet:

```
$scope.$on('aMessage', function (data) {
  console.log('aMessage erhalten mit : '+data.result);
});
```

### Anpassung in der View für die Filterfunktion

In der View stellen wir das Feld für die Filtereingabe auf die neu erstellte Methode um:

```
<input type="text" ng-model="query" ng-change="search()" class="search-query" placeholder="Search" >
```

Jede Änderung durch den Benutzer führt durch die Angaben im Attribut `ng-change` zu einem Aufruf der Methode `search()` im ListController und zur Filterung der Liste, wie oben beschrieben.

Für die Aufteilung der Ergebnisliste in Seiten ändern wir die Struktur der Ergebnisliste in eine Liste von Seiten. Jede Seite ist eine Liste von Ergebniseinträgen. Die Seitengröße definiert die Variable `pageSize`:

```
$scope.pageSize = 4;
```

Das Aufteilen der Ergebnisse erledigt eine eigene Funktion, `groupToPages()`, die im folgenden Quellcode beschrieben ist:

```
$scope.groupToPages = function () {
  $scope.pagedItems = [];

  for (var i = 0; i < $scope.filteredItems.length; i++) {
    if (i % $scope.pageSize === 0) {
      $scope.pagedItems[Math.floor(i / $scope.pageSize)]
        = [ $scope.filteredItems[i] ];
    } else {
      $scope.pagedItems[Math.floor(i / $scope.pageSize)]
        .push($scope.filteredItems[i]);
    }
  }
};
```

Die Funktion wandert über alle Einträge der Ergebnisliste. Die aktuelle Seite für einen Eintrag ergibt sich, indem man die Position durch die Seitengröße teilt und abrundet. In der Funktion erledigt folgende Rechnung diese Aufteilung:

```
Math.floor(i / $scope.pageSize)
```

Eine neue Seite ist immer erreicht, wenn der Rest der Division des aktuellen Index durch die Seitengröße genau den Wert null ergibt. Mit der JavaScript-Funktion `push()` können leicht neue Seiten an die Liste angehängt werden.

Diese neue Struktur der seitenorientierten Ergebnisliste muss sich ebenfalls in der View widerspiegeln. Das `ng-repeat`-Attribut der Ergebnisliste greift auf die aktuelle Seite zu und läuft nur über die Links der aktuellen Seite im Feld `pagedItems`:

```
<tr ng-repeat="link in pagedItems[currentPage] | orderBy:'name' ">
```

Für die Darstellung der Attribute in den Spalten ändert sich nichts. Das Sortieren werden wir im nächsten Abschnitt einbauen. Die aktuelle Implementierung sortiert die Einträge der sichtbaren Seite und ist damit nicht vollständig.

Da die Ergebnisse jetzt seitenweise aufgeteilt sind, sind neue UI-Elemente für das Wechseln der Seite notwendig. Diese fügen wir unter der Tabelle hinzu:

```
<button
  ng-disabled="currentPage == 0"
  ng-click="setPage( currentPage=currentPage-1 )">
Previous
</button>
Page: {{currentPage+1}}/{{numberOfPages()}}
<button
  ng-disabled="currentPage>=filteredItems.length/pageSize - 1"
  ng-click=" setPage( currentPage=currentPage+1 ) " >
Next
</button>
```

Zwei Buttons erledigen das Vor- und Zurückblättern. Mit einem Klick ändern wir den Wert der Variable `currentPage` und damit den sichtbaren Bereich der Tabelle. Die Methode `setPage()` speichert die aktuelle Seite zusätzlich im `$rootScope`. Ist die erste oder letzte Seite erreicht, deaktiviert das Attribut `ng-disabled` jeweils den entsprechenden Button und verhindert weiteres Blättern über die Grenzen hinaus. Zwischen den beiden Schaltflächen erscheinen die aktuelle Seitennummern und die Gesamtzahl der Seiten. Diesen letzten Wert errechnet eine einfache Hilfsfunktion:

```
$scope.numberOfPages=function(){
  return Math.ceil(
    $scope.filteredItems.length/$scope.pageSize);
}
```

### 3.4.6 Schritt 6: Sortieren mit mehr Komfort

Im letzten Abschnitt harmonierte das Sortieren nicht mit der Funktion zum Blättern. Als zusätzliche Aufgabe erlauben wir dem Benutzer die Wahl, nach welchem Attribut sortiert werden soll.

Damit der Benutzer die Sortierung wählen kann, ergänzen wir die Spaltenüberschriften der Linkliste um jeweils ein Icon für auf- bzw. absteigendes Sortieren:

```
<th>Project
<a ng-click="sortBy('name')"><i class="icon-circle-arrow-up"></i></a>
```

```
<a ng-click="sortBy('-name')"><i class="icon-circle-arrow-down"></i></a>
</th>
<th>Description
<a ng-click="sortBy('description')"><i class="icon-circle-arrow-up"></i></a>
<a ng-click="sortBy('-description')"><i class="icon-circle-arrow-
down"></i></a>
</th>
```

Jedes Icon löst bei einem Klick die Funktion `sortBy()` aus. Übergeben wird jeweils der Name des Attributes, nach dem sortiert werden soll. Für eine absteigende Reihenfolge setzen wir ein Minuszeichen vor den Attributnamen.

Die Funktion `sortBy()` ist im `ListController` definiert, nimmt die ausgewählte Sortierung an und stößt eine neue Suche an, die die Sortierung einschließt:

```
$scope.sortBy = function(newSortingOrder) {
  $scope.sortingOrder = newSortingOrder;
  $scope.search();
};
```

Die Neuerungen in der `search()`-Methode für die Sortierung sind übersichtlich. Existiert eine Auswahl für die Sortierung, bemühen wir die Standardfunktionen von AngularJS des `orderBy`-Filters. Dieser erhält die zuvor gefilterte Ergebnisliste und den Attributnamen des Sortierkriteriums. Der letzte Parameter ist immer `false`. Hier könnte man die Richtung der Sortierung umdrehen. Wir übergeben die Richtung schon durch ein optionales Minuszeichen.

```
...
// take care of the sorting order
if ($scope.sortingOrder !== '')
{
  $scope.filteredItems = $filter('orderBy')($scope.filteredItems,
$scope.sortingOrder, false);
  $rootScope.sortingOrder = $scope.sortingOrder;
}
...
```

Project	Description	Tags	
Zita de	Datenbank für Zitate und Sprüche.	Zitate	
HirnSport.de	Rätsel für das tägliche Gehirnjogging.	Denksport	
DenkTipps.com	Informationen über MindMapping.	MindMapping	
BrainBrix.com	Besser als Sudoku.	Denksport	

Bild 3.13: LinkManager mit Sortierung und Blättern.

### Wunschliste für mehr Komfort

Damit haben wir die geforderten Grundfunktionen der Oberfläche implementiert. Weiterer Komfort wäre schön und könnte mit folgenden typischen Funktionen erreicht werden:

- Sortieren nach mehreren Kriterien gleichzeitig
- Wählen der Seitengröße für das Blättern
- Filtern nach exakten Treffern und inklusive der Beachtung von Groß- und Kleinschreibung

An dieser Stelle wollen wir uns um die Kommunikation mit einem Backend und um das Ablegen der Daten im Client kümmern.

#### 3.4.7 Schritt 7: Daten im LocalStorage des Browsers

Der Service, der die Linkdaten verwaltet, ist bisher leider nicht sonderlich langlebig. Die Daten sind bei jedem neuen Besuch der Seite im Originalzustand. Das ändern wir jetzt, indem wir die Daten im lokalen Speicher des Browsers permanent ablegen. Bei einem erneuten Besuch bekommt der Benutzer den letzten gespeicherten Zustand präsentiert.

## Lokale Daten im Browser

In den frühen Zeiten des Internets gab es keine Möglichkeit, Informationen auf dem Clientrechner zu speichern. Als sich Webseiten zu Anwendungen mit Logik entwickelten, wurde es notwendig, den Zustand zu merken. Die ersten Lösungen waren Cookies und URL-Parameter, diese helfen in manchen Situationen weiter. Trotzdem musste der vollständige Zustand meist auf dem Server gespeichert werden. Die Verbindung zwischen Client und Zustand auf dem Server wurde mithilfe von Session-IDs erreicht. Bei jeder Anfrage sendet der Client diese Session-ID mit und der Server kann damit den Zustand, in dem der Client ist, auf dem Server rekonstruieren. Das entlastet den Client, dieser verhält sich als einfache Anzeige-Komponente. Dieses Vorgehen erzwingt leider eine kontinuierliche Verbindung zwischen Client und Server, denn jede Aktion des Benutzers muss zum Server übertragen und dort ausgeführt werden. Dieses Verhalten verträgt sich nicht mit dem Wunsch, eine Applikation offline zu nutzen.

Zwar steigen die Abdeckung und die Bandbreite von drahtlosen Netzen weiter an, aber trotzdem sollen die Applikationen mit kurzfristigen Ausfällen oder schmalen Bandbreiten zureckkommen. Ein Hilfsmittel ist es, Daten auf dem Client (also im Browser) lokal zu halten.

Über die Jahre entstanden mehrere Konzepte, die sich aber nicht vollständig etablieren konnten. HTML5 bietet jetzt eine einheitliche Lösung für alle Browser an:

### LocalStorage

Dieser Speicher erlaubt es, größere Mengen von Daten auf dem Clientrechner zu speichern. Die Speicherung ist nicht zeitlich begrenzt und kann je nach Browser und Betriebssystem mehrere Megabyte umfassen.

### SessionStorage

Auch der Session-bezogene Speicher wurde erweitert. Cookies haben ausgedient und werden durch ein neues API ersetzt. Der SessionStore bietet eine ähnliche API wie der LocalStore an, ist aber zeitlich begrenzt und wird gelöscht, wenn die Session abläuft.

### Nachteile von Cookies

Cookies waren immer nur eine Notlösung. Denn die Größe und die Anzahl der Cookies sind stark begrenzt. Der Standard fordert, dass 20 Cookies pro Domain mit je 4 Kilobyte gespeichert werden können. Der Haupteinsatzzweck ist, eine Session-ID oder eine User-ID zu merken, aber keine umfangreichen Daten der Anwendung.

## HTML5 und Local Storage

JavaScript bietet Funktionen für den Zugriff auf den lokalen Speicher an. Es gibt drei Methoden für das Speichern, Abfragen und Löschen der Inhalte. Zusätzlich existiert eine Funktion, die alle Daten, die für eine Domain vorliegen, entfernt. Alle modernen Browser bieten diese Funktionen an.

Der Zugriff erfolgt über Key-Value-Paare. Das folgende Beispiel legt einen Wert mit der Funktion `setItem()` und liest diesen über den Key später wieder aus:

```
localStorage.setItem("key", "value");
var value = localStorage.getItem("key");
```

Eine typische Anwendung könnte das Speichern von Benutzereingaben für ein HTML-Formular sein, das ein Benutzer später vervollständigen möchte. Das Formular könnte wie im Beispiel formuliert sein:

```
<form onsubmit="lokalspeichern(); return false">
  <input type="text" name="vorname" />
  <input type="text" name="nachname" />
</form>
```

Die Funktion `lokalspeichern()` legt die Werte aus den Formularelementen in den lokalen Speicher:

```
function lokalspeichern() {
  localStorage.setItem("vorname",
    document.forms[0]["vorname"].value);
  localStorage.setItem("nachname",
    document.forms[0]["nachname"].value);
}
```

Für das Entfernen gibt es mehrere Wege. Zum einen könnte der Wert einfach mit einem leeren Inhalt belegt werden:

```
localStorage.setItem("key", null);
```

Zum anderen gibt es spezielle Funktionen für das Löschen. Die Funktion `removeKey("key")` entfernt den übergebenen Schlüssel inklusive des verknüpften Inhaltes. Die Funktion `localStorage.clear()` entfernt alle vorhandenen Informationen der Domain, von der die gerade ausgeführte JavaScript-Datei geladen wurde.

```
localStorage.removeItem("key");
localStorage.clear();
```

Die Daten bleiben so lange im lokalen Speicher, bis sie über eine der gerade vorgestellten Funktionen entfernt werden. Zusätzlich existiert ein Attribut, mit dem die Anzahl der gespeicherten Werte erfragt werden kann: `localStorage.length`.

Alternativ existiert der Session-Storage. Dieser hält die Daten nur, solange die Session existiert. Die Funktionen bieten dieselben Signaturen, die wir gerade besprochen haben.

Der Haupteinsatzzweck für die Funktionen liegt im Offlinebetrieb oder dem Zwischen-speichern (als Cache) von intensiv genutzten Daten.

#### Online oder Offline?

Um festzustellen, ob der Browser gerade Zugriff auf das Internet hat, gibt es das Attribut `navigator.onLine`. Es liefert den Wert `true`, wenn eine Internetverbindung existiert und `false`, wenn der Zugriff nicht möglich ist.

#### Fallstricke im Umgang mit LocalStorage

Aktuell unterstützt der LocalStorage nur Zeichenketten. Wie der folgende Test zeigt, wirft der Browser keine Fehlermeldung aus und legt Daten ab, auch wenn wir ein anderes Format als die unterstützten Zeichenketten nutzen. Leider ändert sich der Datentyp - aus einem Array wird eine Zeichenkette:

```
> localStorage.setItem( 'key', [1,2, 3]);
> localStorage.setItem( 'key2', '[1,2, 3]');

> localStorage.getItem( 'key' );
"1,2,3"
> localStorage.getItem( 'key2' );
"[1,2, 3]"
```

Dieses Verhalten ist schlecht, wenn man Arrays oder Objekte speichern möchte. Zum Glück gibt es eine einfache Lösung, indem wir Felder und Objekte mit den Befehlen `JSON.stringify()` und `JSON.parse()` als JSON-Objekte behandeln und diese somit in Zeichenketten serialisiert werden.

Es gibt weitere Fallstricke beim Einsatz des LocalStorage:

- ➊ Die Funktion kann durch den Benutzer in den Einstellungen des Browsers deaktiviert werden. Letztlich kann sich eine Applikation nicht sicher sein, dass der LocalStorage verfügbar ist.
- ➋ Die verschiedenen Browser stellen unterschiedlich viel Speicher für eine Applikation bereit. Das Volumen bewegt sich typischerweise zwischen 5 und 10 Megabyte, was für die meisten Applikationen ausreichen sollte. Überschreitet eine Anwendung dieses Volumen, fragen manche Browser den Benutzer, ob zusätzlicher Speicher bereitgestellt werden soll. Insgesamt ist das Verhalten nicht klar und eindeutig.
- ➌ Öffnet ein Benutzer dieselbe Applikation mit unterschiedlichen Browsetypen, wird der Speicher nicht gemeinsam genutzt. Der Speicher ist jedem Browser separat zugeordnet.
- ➍ Der Befehl `clear()` löscht den gesamten lokalen Speicher der Domäne. Das heißt, selbst wenn man die Applikation über Verzeichnisse in Bereiche unterteilt hat, nach dem Muster `www.mydomain.de/bereich1` und `www.mydomain.de/bereich2`, so bezieht sich der `clear()`-Befehl auf alle Bereiche der Domain. Es empfiehlt sich, die Bereiche über die geschickte Benennung von Schlüsseln zu nutzen und die Anwendungsbereiche gezielt zu löschen.

Es führt kein Weg an ausgiebigem Testen beim Einsatz der Funktionen vorbei.

## LocalStorageModul

Um den lokalen Speicher in der Link-Applikation anzusprechen, verpacken wir die Funktionen in ein eigenes AngularJS-Modul. Hier die Anforderungen an das Modul im Überblick:

- Methoden für das Ablegen und Auslesen von Werten
- Auflisten aller Schlüssel
- Löschen aller Schlüssel und Daten
- Zuordnen der Daten in verschiedene Themenbereiche
- Funktion zum Prüfen, ob der LocalStorage aktiv bzw. verfügbar ist.

Einer der großen Vorteile von AngularJS ist die Modularisierung und Erweiterbarkeit. Das Modul für den LocalStorage ist ein gutes Beispiel hierfür. Die Implementierung ist hinter einer Schnittstelle verborgen. Der Benutzer des Moduls braucht kein Wissen über die Details der Implementierung zu besitzen.

Außerdem schützt die Schnittstelle unsere Applikation vor Änderungen der Browser-API. Würde sich die Implementierung der Browser ändern, könnten die Funktionen, die unsere Applikation im Modul aufruft, unverändert bleiben, sofern sich dieses Zusammenspiel nicht grundlegend ändert.

### Die Schnittstelle des SimpleLocalStorageModul

Die folgende Tabelle listet die Funktionen, die das Modul anbietet:

Funktion	Beschreibung	Rückgabe
isSupported	Prüft, ob der LocalStorage im Browser verfügbar und aktiviert ist. Rückgabe ist true oder false.	true oder false
put(Key,Data)	Legt die Daten (data) unter dem Schlüssel key ab. Es wird das Präfix »ls_« vorangestellt, wenn bei der Initialisierung des Moduls kein anderes Präfix definiert wurde.	true oder false
get(Key)	Liefert die Daten zurück, die unter dem Key gespeichert wurden.	Daten oder null, falls der Key nicht existiert.
keys()	Listet alle existierenden Keys auf.	Array mit allen Keys

Funktion	Beschreibung	Rückgabe
remove(Key)	Entfernt den übergebenen Schlüssel und löscht die damit verknüpften Daten.	true oder false
clearAll()	Löscht alle Daten und Schlüssel.	true oder false

### Das SimpleLocalStorageModul im Detail

Das Modul erstellen wir in einer eigenen JavaScript-Datei (`simpleLocalStorageModule.js`), um es später vielleicht in anderen Applikationen zu nutzen. Interessant ist, dass sich die Definition kaum von der Definition unserer Applikation selbst unterscheidet.

```
var angularLocalStorage =
  angular.module('SimpleLocalStorageModule', []);
angularLocalStorage.value('prefix', 'ls_');
```

Als Nächstes wird das Default-Präfix mit dem Wert »ls\_« vorbelegt. Dies wird allen Schlüsseln automatisch vorangestellt. So können wir die Schlüssel mit verschiedenen Präfixen leicht in thematische Bereiche aufteilen. Es folgt die Definition des Services. Am Ende wird in einem Return-Statement die Schnittstelle mit allen öffentlichen Funktionen aufgeführt:

```
angularLocalStorage.service('simpleLocalStorageService', [
  'prefix', function( prefix) {

    var hasLocalStorage = function () {
      try {
        return ('localStorage' in window && window['localStorage'] !== null);
      } catch (e) { return false; }
    };

    var putToLocalStorage = function (key, value) {
      if (!hasLocalStorage()) {
        return false;
      }
      if (typeof value == "undefined") value = null;

      try {
        if (angular.isObject(value) || angular.isArray(value)) {
          value = angular.toJson(value);
        }
        localStorage.setItem(prefix+key, value);
      } catch (e) { return false; }
      return true;
    };
  };
}];
```

```
var getFromLocalStorage = function (key) {

    var item = localStorage.getItem(prefix+key);
    if (!item) return null;
    if (item.charAt(0) === "{" || item.charAt(0) === "[") {
        return angular.fromJson(item);
    }
    return item;
};

var removeFromLocalStorage = function (key) {
    if (!hasLocalStorage()) {
        return false;
    }

    try {
        localStorage.removeItem(prefix+key);
    } catch (e) { return false; }
    return true;
};

var getKeysForLocalStorage = function () {
    if (!hasLocalStorage()) {
        return [];
    }

    var prefixLength = prefix.length;
    var keys = [];
    for (var key in localStorage) {
        if (key.substr(0,prefixLength) === prefix) {
            try {
                keys.push(key.substr(prefixLength))
            } catch (e) { return []; }
        }
    }
    return keys;
};

var clearAllFromLocalStorage = function () {
    if (!hasLocalStorage()) {
        return false;
    }

    var prefixLength = prefix.length;
    for (var key in localStorage) {
        if (key.substr(0,prefixLength) === prefix) {
```

```
        try {
            removeFromLocalStorage(key.substr(prefixLength));
        } catch (e) { return false; }
    }
}

return true;
};

return {
    isSupported: hasLocalStorage,
    put: putToLocalStorage,
    get: getFromLocalStorage,
    keys: getKeysForLocalStorage,
    remove: removeFromLocalStorage,
    removeAll: removeAllFromLocalStorage
};
]);
});
```

Der `return`-Befehl am Ende definiert die Schnittstelle, die nach außen hin sichtbar ist. In diesem Fall sind die Namen der Methoden kürzer und prägnanter formuliert als die Internet-Implementierung. Der Service ist damit ein Beispiel, wie man mit JavaScript Schnittstellen definiert und Interna versteckt. Er könnte weitere Hilfsfunktionen oder Datenstrukturen enthalten, die nach außen unsichtbar sind. Der folgende Abschnitt erklärt die Funktionen im Detail:

● **hasLocalStorage():**

Die Funktion prüft, ob das globale `window`-Objekt des Browsers ein Attribut mit dem Namen `localStorage` enthält. Ist dieses nicht vorhanden, bietet der Browser keinen lokalen Speicher an. Die meisten Funktionen des Services nutzen diese Funktion.

● **putToLocalStorage():**

Diese Funktion legt die übergebenen Daten unter dem Schlüssel ab. Das Präfix wird ergänzt. Falls die Nutzdaten ein Objekt oder eine Array sind, werden diese mit der AngularJS-Funktion `toJson()` in das JSON-Format umgewandelt, da der lokale Speicher nur Zeichenketten erlaubt.

● **getFromLocalStorage():**

Die Funktion liest den übergebenen Schlüssel auf. Wird etwas im lokalen Speicher gefunden, untersucht die Funktion, ob es ein JSON-Objekt (Darstellung eines JavaScript-Objektes im JSON-Format) oder Array ist. Das ist an dem ersten Zeichen sichtbar. Bei Arrays ist das erste Zeichen eine eckige Klammer und bei einem Objekt eine geschweifte Klammer. In diesen Fällen werden die Daten mit der Funktion `fromJson()` wieder in ein JavaScript-Objekt deserialisiert.

● **removeFromLocalStorage():**

Diese Funktion ist einfach und ruft letztlich nur die Funktion `removeItem()` auf.

**getKeysForLocalStorage():**

Die Funktion listet alle Schlüssel in einer Liste auf, wobei nach dem passenden Präfix gefiltert wird.

**clearAllFromLocalStorage():**

Die Funktion löscht alle Schlüssel mit dem passenden Präfix.

## Einbinden des Moduls

Wir binden das JavaScript-File des neuen Moduls wie gewohnt im HTML-Dokument ein. Neu ist jetzt, dass unsere AngularJS-Applikation eine Abhängigkeit zum neu erstellten Modul aufweist. Diese nehmen wir bei der ursprünglichen Definition unserer Applikation am Beginn der JavaScript-Datei `project.js` auf:

```
var app = angular.module('linkManager',
  ['SimpleLocalStorageModule'])
```

Bisher war die Liste der Abhängigkeiten (im letzten Parameter) leer und enthält jetzt den Namen des neuen Moduls. Damit können wir dieses Modul ebenfalls dem ListController übergeben, indem wir es als neuen Parameter aufnehmen:

```
function ListController($scope, $rootScope, $filter, linkService,
simpleLocalStorageService) {
  ...
  $scope.storeLinks = function()
  {
    simpleLocalStorageService.put('localLinks',
      linkService.getLinks());
  }

  $scope.loadLinks = function()
  {
    linkService.setLinks(
      simpleLocalStorageService.get('localLinks'));
    $scope.search();
  }
}
```

Die beiden neuen Funktionen sind recht übersichtlich. Die Funktion `storeLinks()` soll die komplette Liste der Linkdaten speichern. Dazu übergeben wir alle Linkdaten, die der LinkService liefert, unter dem Schlüssel `localLinks` an den StorageService.

Die Methode zum Laden der Links aus dem lokalen Browserspeicher ist ähnlich einfach: Aus dem StorageService werden mit der Get-Methode die Daten über den Schlüssel geholt und an den LinkService übergeben. Dieser bietet hierfür eine neue Methode, `setLinks()` an. Diese überträgt die Linkliste in seine interne Datenstruktur. Hinzu kommt lediglich der Aufruf der `Search()`-Funktion des ListControllers, damit die Filterung und Sortierung erneut aktiviert werden.

Über die Konsole können wir prüfen, ob die Daten wirklich in dem lokalen Speicher angekommen sind:

```
> localStorage
Storage {ls_localLinks: "[{"Id":1,"name":"Zitate, Sprueche,
Weisheiten","site":"http://www.zita.de","description":"","tags":""}]"}
```

Im übergeordneten Schlüssel ist das vordefinierte Präfix `ls_` automatisch ergänzt und die abgelegten Daten sind als JSON-String sichtbar.

### Erweiterungen in der UI

Für das Laden und Speichern ergänzen wir in der HTML-Seite zwei Knöpfe am unteren Rand der Liste.

```
<a href="#" class="btn" ng-click="storeLinks()">StoreLinks (LS)</a>
<a href="#" class="btn" ng-click="loadLinks()">ReadLinks (LS)</a>
```

## 3.4.8 Schritt 8: Kommunikation mit dem Server per REST

Zwar können wir seit dem letzten Schritt Daten speichern, bisher liegen diese aber nur im Client. Der nächste Schritt soll die Kommunikation mit einem Server sein, der die Daten langfristig in einer Datenbank vorhält. Außerdem kann der Server seine Daten mehreren Clients zur Verfügung stellen.

### Plattformübergreifende Kommunikation

Wenn wir die überschaubare Welt eines Rechners verlassen, treffen wir auf neuartige Herausforderungen: Mit welcher Technik sollen die einzelnen Komponenten kommunizieren? Wie sieht das Datenformat aus? Was kann alles schief gehen und wie reagiert die Applikation angemessen auf diese Situation?

Das sind Themen, mit denen sich viele Projekte Tag für Tag beschäftigen müssen. In den letzten zehn Jahren war ein großes Thema die Kommunikation mithilfe von Web-Service und XML. Diese haben geholfen, die technischen Unterschiede zwischen den Kommunikationspartnern zu reduzieren und fachlich wohldefinierte Schnittstellen zu entwerfen. Mit der Schema-Definition von WebService werden die Daten mit ihrer Struktur exakt beschrieben. Jede Nachricht kann leicht gegen dieses Interface geprüft werden. Gerade im Geschäftsumfeld etablierten sich WebServices als Standard und mit Serviceorientierten Architekturen (SOA) und bildeten eine neue Unternehmensarchitektur. Nicht mehr isolierte Applikationen stehen im Vordergrund, sondern die Funktionalität wird von eigenständigen Services angeboten und in Geschäftsprozessen und Applikationen orchestriert und zusammengeführt.

Leider bringen WebServices einige Nachteile mit: So ist XML eine gute Möglichkeit, die Struktur und den Inhalt einer Nachricht exakt zu beschreiben, leider ist das Format aber mit viel Overhead, in Form der Tags, überladen. Das bläht viele Nachrichten

enorm auf, was schließlich dazu führte, dass WebServices den Ruf erhielten, schwerfällig und unhandlich zu sein.

Das ist ein Grund für den Erfolg, den das Format JSON in den letzten Jahren für sich verbuchen konnte. Für viele Anwendungen besonders im Web-Bereich ist es das Format der Wahl, da es im Vergleich zu XML wesentlich einfacher strukturiert ist.

## Kommunikation über REST

Die Abkürzung REST steht für den Ausdruck »Representational State Transfer« und meint, eine bestimmte Art des Zustands von Objekten oder Entitäten über eine URL eindeutig zugreifbar zu machen. Die URL wird klassisch mit HTTP an einen REST-Service gesendet und dieser liefert (oder modifiziert) ein Objekt. Über das Format, in dem die Daten geliefert werden, macht Rest keine Aussage, üblich sind JSON oder XML. Andere Formate für spezielle Daten, wie Texte, Bilder oder Videos, sind ebenfalls möglich. Da REST nicht in einer Norm oder einem Standard festgelegt ist, finden sich in der Literatur und im Internet unterschiedliche Definitionen.

Für die Beispielapplikation könnten die URLs für die REST-Abfragen den folgenden Aufbau haben; im Browser würde jeweils ein JSON-Dokument geliefert:

Alle Links abfragen:

```
http://localhost/rest/api/links
```

Alle Links nach einem Begriff »zitate« durchsuchen und liefern:

```
http://localhost/rest/api/links/search/zitate
```

Den Link mit dem Identifier 47 laden:

```
http://localhost/rest/api/links/47
```

Mit den Kommandozeilen-Werkzeugen wie cURL oder WGET können auf Linux-Systemen sogar Links (Dateien) über den Identifier gelöscht werden.

Dazu wird eine HTTP-Nachricht mit der Methode DELETE übertragen:

```
curl -i -X DELETE http://localhost/rest/api/links/19
```

Parameter können ebenfalls mit versendet werden, mit dem Parameter -d. Das Beispiel legt einen neuen Link an:

```
curl -i -X POST -H 'Content-Type: application/json' -d '{"name": "Linkname", "site": "http://www.zita.de", ... }' http://localhost/rest/api/links
```

## Frameworks für das Implementieren von REST-Schnittstellen

Mittlerweile existieren viele Frameworks in fast allen Sprachen und für nahezu alle Plattformen, um REST-Schnittstellen zu implementieren. Meist ist die Programmierung sehr einfach und konzentriert sich darauf, vorhandene Entitäten mit Konfigurationen oder Annotationen zu markieren. Da die Schnittstelle von REST einfach ist, können große Teile der Funktionen von einem Framework automatisiert werden. Die

nächste Tabelle listet nur eine kleine Auswahl von Bibliotheken auf, die auf REST-Schnittstellen fokussiert sind:

Name des Frameworks	Plattform	Link
Jersey	Java	<a href="http://jersey.java.net">jersey.java.net</a>
Restify	JavaScript / Node.js	<a href="http://mcavage.me/node-restify">mcavage.me/node-restify</a>
Slim	PHP	<a href="http://www.slimframework.com">www.slimframework.com</a>

### Implementierung eines REST-Servers

Für die Link-Verwaltung nutzen wir das PHP-Framework Slim. Es ist ein Microframework und sein Funktionsumfang ist voll und ganz auf das Anbieten einer REST-Schnittstelle fokussiert.

#### Apache, PHP, MySQL

Tipp: Ein lauffähiges System, mit Apache, PHP und MySQL ist mit EasyPHP (für Windows) oder einer der unter dem Wikipedia-Link aufgeführten Erweiterungen schnell installiert:  
[www.easyphp.org](http://www.easyphp.org)  
[en.wikipedia.org/wiki/LAMP\\_\(software\\_bundle\)](http://en.wikipedia.org/wiki/LAMP_(software_bundle))

Der REST-Service kommt mit wenigen Zeilen Konfiguration aus. Zu Beginn werden einige Einträge für den HTTP-Header der Antwort gesetzt, damit Zugriff von anderen Domains möglich ist:

```
<?php
header('Access-Control-Allow-Origin: *');
header('Access-Control-Allow-Methods',
      'GET, POST, PUT, DELETE, OPTIONS');
header('Access-Control-Allow-Headers',
      'accept, origin, content-type');

require 'Slim/Slim.php';
$app = new Slim();
```

Danach laden wir die notwendige Bibliothek und initialisieren diese in der Variable \$app. Jetzt können wir die erwarteten REST-Pfade mit den Antwort-Methoden verknüpfen. Dynamische Teile in den URLs, wie die Identifier der Links, beginnen mit einem Doppelpunkt. Das Framework löst diese heraus und stellt sie als Variablen mit gleichem Namen bereit.

Die erste der folgenden Zeilen veranlasst den Service, nach einem HTTP-Request mit HTTP-GET und der URL `/links` zur Funktion `getLinks()` zu springen. Dies implementiert die fachliche Logik. Die weiteren Funktionen werden analog erstellt. Nach der Konfiguration startet die Anweisung `$app->run()` den Service:

```
// Alle Links abfragen:  
$app->get('/links', 'getLinks');  
// Eine Link per Id abfragen:  
$app->get('/links/:id',      'getLink');  
// Links durchsuchen:  
$app->get('/links/search/:query', 'findByName');  
//Link einfügen:  
$app->post('/links', 'addLink');  
//Link ändern:  
$app->put('/links/:id', 'updateLink');  
//Link löschen:  
$app->delete('/links/:id', 'deleteLink');  
  
... // Deklaration für Anfragen mit der Methode OPTIONS  
$app->run();  
  
function getLinks() {  
    $sql = "select * FROM links ORDER BY name";  
    try {  
        $db = getConnection();  
        $stmt = $db->query($sql);  
        $links = $stmt->fetchAll(PDO::FETCH_OBJ);  
        $db = null;  
        echo '{"link": ' . json_encode($links) . '}';  
    } catch(PDOException $e) {  
        echo '{"error":{"text":"' . $e->getMessage() . '"}}';  
    }  
}  
  
function getLink($id) {  
    $sql = "SELECT * FROM links WHERE id=:id";  
    try {  
        $db = getConnection();  
        $stmt = $db->prepare($sql);  
        $stmt->bindParam("id", $id);  
        $stmt->execute();  
        $wine = $stmt->fetchObject();  
        $db = null;  
        echo json_encode($wine);  
    } catch(PDOException $e) {  
        echo '{"error":{"text":"' . $e->getMessage() . '"}}';  
    }  
}  
  
function updateLink($id) {  
    $request = Slim::getInstance()->request();  
    $body = $request->getBody();
```

```

$link = json_decode($body);

$sql = "UPDATE links SET name=:name, site=:site, tags=:tags,
description=:description WHERE id=:id";
try {
    $db = getConnection();
    $stmt = $db->prepare($sql);
    $stmt->bindParam("name", $link->name);
    $stmt->bindParam("site", $link->site);
    $stmt->bindParam("tags", $link->tags);
    $stmt->bindParam("description", $link->description);
    $stmt->bindParam("id", $id);
    $stmt->execute();
    $db = null;
    echo json_encode($link);
} catch(PDOException $e) {
    echo '{"error":{"text":' . $e->getMessage() . }}';
}
}
}

```

Der größte Teil des REST-Servers umfasst die Implementierung der Funktionen für das Manipulieren der Daten, die angefragt oder geliefert wurden. Als Beispiele sind im Quellcode drei Funktionen aufgeführt.

Die Methode `getLinks()` definiert ein SQL-Select-Statement, um alle Links in der Tabelle zu lesen. In dem try-Catch-Block öffnen wir eine Verbindung zur Datenbank und führen die Abfrage aus. Die Fetch-Anweisung legt alle vorhandenen Zeilen in eine Liste (`$links`) ab. Die Methode `json_encode()` verwandelt die komplette Liste in einen JSON-String, den die PHP-Anweisung `echo` im Content der Antwort (HTTP-Response) zurücksendet.

Die Funktion `getLink($id)` fragt einen einzelnen Link ab. Dazu erhält sie über die Anfrage-URL den gesuchten Link-Identifier als Parameter `$id`. Der Parameter wird im SQL-Statement in eine Where-Klausel verpackt und dem Datenbank-Statement als Variable übergeben. Der weitere Ablauf unterscheidet sich kaum von der vorherigen Methode.

Die Update-Methode `updateLink($id)` erhält im HTTP-Content ein vollständiges Link-Objekt. Diese wird zuerst aus dem Request-Body gelesen und aus dem JSON-Format deserialisiert. Danach erstellen wir wieder ein SQL-Statement und übergeben die Parameter gemäß ihren Namen. Das Link-Objekt umfasst einen eindeutigen Identifier für die Selektion der korrekten Datenbankzeile im Update-Statement.

### **Erweiterung des Clients um REST-Anfrage**

Nachdem der Service implementiert ist, wenden wir uns der Client-Implementierung zu. Da der Service per HTTP erreichbar ist, könnten wir direkt den eingebauten `$http`-Service von AngularJS nutzen und die URL von Hand erstellen.

Für REST-Services gibt es Frameworks, die mehr Arbeit abnehmen, wie zum Beispiel Restangular ([github.com/mgonto/restangular](https://github.com/mgonto/restangular)). Das Framework versteckt die Kommunikation und die Links fast vollständig und erlaubt es, mit Objekten zu arbeiten. Wir fügen in der HTML-Datei einen neuen Script-Eintrag ein, um Restangular zu laden:

```
<script type="text/javascript" src="restangular.js"></script>
```

Zusätzlich müssen wir es in der Applikation bekannt machen und nehmen es als weitere Abhängigkeit zur Applikation hinzu:

```
var app = angular.module('linkManager',
  ['SimpleLocalStorageModule', 'restangular']);
```

Damit wir über Domain-Grenzen den Server ansprechen können, setzen wir das Attribut `default.useXDomains` am `$httpProvider` und löschen ein Header-Attribut. Das Header-Attribut würde die Adresse des Absenders enthalten, die der Server prüfen könnte. Dadurch könnte der Server die Kommunikation mit unbekannten oder unerwünschten Clients abweisen.

Um Restangular zu nutzen, ist lediglich die Angabe der Basis-Adresse notwendig, die wir am `RestangularProvider` angeben können. Beide fügen wir als Abhängigkeiten in unserer Applikation hinzu:

```
app.config(function($routeProvider, RestangularProvider, $httpProvider) {
  ...
  $httpProvider.defaults.useXDomain = true;
  delete $httpProvider.defaults.headers.common['X-Requested-With'];

  RestangularProvider
    .setBaseUrl('http://localhost/spa2/rest3/api');
})
```

Danach sind die Links mit Restangular über folgende Befehle erreichbar:

```
var theLink = Restangular.one('links', link.id);
theLink.name = link.name;
theLink.site = link.site;
theLink.description = link.description;
theLink.tags = link.tags;
theLink.put();
```

Die Funktion `one()` führt einen GET-Zugriff mit dem übergebenen Identifier durch und liefert als Rückgabe das Link-Objekt. Der String »links« definiert den Zugriffspfad (genannt Route) auf die Entitäten. Routen helfen, den Service in unterschiedliche thematische Bereiche zu trennen.

Der erfragte Link wird über Attributzugriffe verändert und über die Funktion `put()` zurück an den REST-Service gesendet. Restangular erweitert die gelieferten Objektentitäten um zusätzliche Funktionen für die Kommunikation und macht den Umgang

dadurch sehr viel einfacher. Einige dieser neuen Funktionen nutzt ebenfalls der LinkService.

### Neue Aufgaben für den LinkService

Der LinkService wird die gesamte REST-Kommunikation erledigen und vor den anderen Applikationsteilen, wie den Controllern, verstecken.

Zwei neue Abhängigkeiten sind notwendig:

- ➊ Über den Parameter `Restangular` erhalten wir den Zugriff auf das Framework.
- ➋ Den `$rootScope` nutzen wir, um die Controller über Datenaktualisierungen zu benachrichtigen.

Zuerst initialisiert der Service seine Daten, indem er direkt alle Einträge mit der Rest-Funktion `all('links')` abfragt. Die Ergebnisfunktion nimmt die gelieferten Daten entgegen und legt diese in der bekannten Variable `data` ab. Die anderen Applikationsteile (der ListController, siehe unten) werden über eine Nachricht über die Aktualisierung der Links informiert:

```
app.factory('linkService', function ( $location, Restangular, $rootScope ) {

  var data = [ { ... } ];

  Restangular.all('links').getList().then(function(result){
    data = result.link ;
    $rootScope.$broadcast('links:update');
  });

});
```

Der LinkService erhält in seiner Schnittstelle neue Funktionen für die REST-Kommunikation. Um diese besser von den eingeführten Funktionen zu unterscheiden, erhalten diese neuen Funktionen den Namenszusatz `rest` vorangestellt:

```
...
  return {
// ... Bekannte Methoden bleiben unverändert
  restGetLinks : function()
  {
    Restangular.all('links')
      .getList().then(function(data2){
        data = data2.link;
        $rootScope.$broadcast('links:update');
      });
  },
  restRemove : function(link)
  {
    Restangular.one("links", link.id).remove();
    this.deleteLink(link);
    $rootScope.$broadcast('links:update');
  }
};
```

```
},
restSave : function(link)
{
    this.updateLink( link );
    var theLink = Restangular.one('links', link.id);
    theLink.name = link.name;
    theLink.site = link.site;
    theLink.description = link.description;
    theLink.tags = link.tags;
    theLink.put( );
},
restAdd: function(newlink)
{
    Restangular.all('links')
        .post(newlink)
        .then(function(newResource){
            data.push( newResource );
            $rootScope.$broadcast('links:update');
        })
    }
},
});
```

Die Funktion `restGetLinks()` haben wir in der Einleitung zu Restangular vorgestellt. Die folgenden Funktionen sind ebenfalls neu im ListService:

● `restGetLinks():`

Die Funktion enthält dieselbe Logik, die für die Initialisierung des ListService ausgeführt wurde.

● `restRemove():`

Holt genau einen Datensatz über den Identifier und löscht diesen mit Remove auf dem Backend.

● `restAdd():`

Diese Funktion legt einen neuen Link an. Dazu nutzt sie die Operation `post()` und sendet das neue Link-Objekt.

In allen drei Funktionen ist folgendes Verhalten gleich: Sie setzen ebenfalls eine Broadcast-Nachricht über die Aktualisierung der Daten ab, wie wir es schon vorgestellt haben. Als zweite Gemeinsamkeit aktualisieren sie die lokalen Daten parallel zum Aufruf des entfernten Services, damit sparen wir uns das neue Übertragen der gesamten Liste.

## Fehlerhandling für Restangular

An jede Restangular-Funktion können wir neben der Callback-Funktion für den Erfolgsfall zusätzlich eine zweite Funktion für den Fehlerfall anhängen:

```
Restangular.all("links").getList().then(function() {
  console.log("Anfrage war Erfolgreich.");
}, function(response) {
  console.log("Ein Fehler trat auf:", response.status);
});
```

Wie die sinnvolle Reaktion auf die Fehlersituation aussieht, ist vom Anwendungsfall abhängig. Wahrscheinlich ist es eine gute Idee, den Benutzer zu informieren. Die zweite Möglichkeit ist das Erstellen einer globalen Fehlerbehandlung. Restangular bietet dafür Error-Interceptoren an.

Mit der Funktion `setErrorInterceptor()` definiert man eine Callback-Funktion. Diese erhält im Fehlerfall den Rückgabewert der Anfrage (`response`) als einen Parameter. Die Funktion entscheidet mit ihrem Rückgabewert, wie Restangular mit der fehlerhaften Anfrage weiter umgehen soll:

- ➊ Die Rückgabe des Wertes `false` beendet die weitere Verarbeitung der Anfrage. Sie wird nicht weitergeleitet.
- ➋ Alle anderen Werte leiten die Antwort der fehlerhaften Anfrage normal an andere Callback-Funktionen weiter. Eine andere Funktion der Applikation könnte zusätzlich auf den Fehler reagieren.

## Änderungen an den Controllern

In den Controllern werden fast alle Aufrufe von Funktionen des LinkService durch die hinzugefügten Funktionen ersetzt. Die einzige Ausnahme ist die Suchfunktion im ListController, diese wird sehr oft (auch bei jeder Sortierung und Filterung) ausgeführt. Um nicht bei jeder Aktion Netzwerzkurzgriff zu forcieren, soll diese Funktion weiterhin auf den zwischengespeicherten Links arbeiten. Sie nutzt weiterhin `linkService.getLinks()`.

Um trotzdem eine Datenaktualisierung nach den Modifikationen (Ändern, Löschen und Neuanlegen) nicht zu verpassen, senden wir aus dem LinkService heraus Nachrichten mit der Broadcast-Funktion.

Der ListController lauscht auf diese Nachrichten und führt daraufhin die `search()`-Funktion aus, die sich die aktuellen Daten (wie bisher) holt und anzeigt. Dafür nutzen wir die `$on`-Funktion am Scope-Objekt und registrieren den Namen der Nachrichten, die uns interessieren:

```
$scope.$on('links:update', function() {
  console.log("links:update!");
  $scope.search();
});
```

## 3.5 Web-Anwendungen im Unternehmensumfeld

Das zweite Beispiel ist fachlich in der Finanzwelt als Fachdomäne angesiedelt. Ziel der Applikation ist es, die langfristige Entwicklung von Handelsstrategien für Aktien optisch ansprechend zu visualisieren. Die Applikation ist die Vorstudie eines realen Projektes. Uns interessieren an dieser Stelle die technische Implementierung und nicht alle fachlichen Hintergründe im Detail dazu, wie diese Handelsstrategien funktionieren.

### 3.5.1 Fachliche Anforderungen

Im oberen Bereich der Seite ist eine Liste von Handelsstrategien angezeigt. In der Tabelle werden für jeden Eintrag bestimmte Eigenschaften sichtbar:

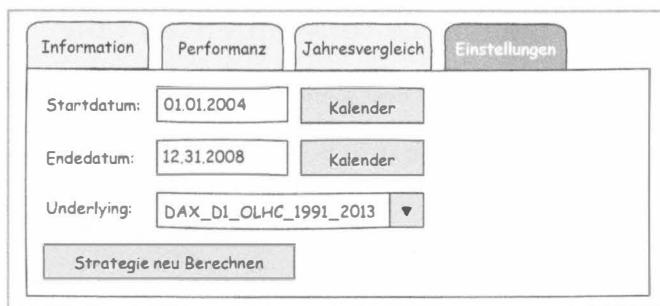
- Name der Strategie
- Erstellungsdatum
- Grobe Entwicklung der Strategie

Der Benutzer kann aus dieser Tabelle eine Strategie wählen, indem er die Spalte anklickt. Daraufhin bekommt er weitere Stammdaten, einen Performanzvergleich und eine Jahresübersicht angezeigt. Die Screenprototypen zeigen den Aufbau der geplanten Applikation.

The screenshot displays the main interface of the EvoTrader application. At the top, there's a browser-like header with buttons for Home, Reload, and URL. Below it, the title "EvoTrader" is displayed above a table. The table has four columns: Strategename, Erstellt, Underlying, and Performanz. It contains two rows of data: one for "MaxPerformanz" (Erstellt: 1.10.2013, Underlying: DAX\_D1\_OHLC, Performanz: 9,45%) and one for "ReduziertesRisiko" (Erstellt: 4.11.2013, Underlying: DAX\_D1\_OHLC, Performanz: 12,83%).

Below the table, a section titled "Strategie: MaxPerformanz" is shown. It includes a message box stating "Aktuelle Meldung: Auswertung erfolgreich durchgeführt." and a navigation bar with tabs for Information, Performanz, Jahresvergleich, and Einstellungen. The "Information" tab is currently active. Under the "Information" tab, there are two sections: "Basis" and "Parameter". The "Basis" section shows "Underlying: DAX\_D1\_OHLC", "MaxBefehle: 500", and "Periode: D1". The "Parameter" section shows "Transaktionskosten (%): 0,1", "Gap (Buy): 200", and "Gap (Sell): 200".

Bild 3.14: Die Hauptmaske des EvoTraders.



**Bild 3.15:** Der Reiter für die Einstellungen der Neuauswertung.

Unterhalb der Liste befindet sich ein Bereich, der über Registerkarten gewechselt werden kann. Im jeweiligen Detailbereich des Registers befinden sich thematisch gruppiert die Informationen der ausgewählten Handelsstrategie:

- ➊ Die erste Seite zeigt grundlegende Informationen der Strategie wie zum Beispiel Name, wichtige Parameter, Startzeitpunkte, Startkapital und viele weitere fachliche Details.
- ➋ Sehr interessant sind die beiden folgenden Reiter (Performanz und Jahresvergleich). Sie sollen Charts mit grafischen Informationen über den Erfolg der gewählten Handelsstrategie im zeitlichen Verlauf über die letzten Jahre enthalten. Dazu verwendet die Applikation unterschiedliche Charttypen.
- ➌ Im letzten Reiter gibt es die Möglichkeit, die Simulation der Strategie neu zu starten. Dazu kann der Zeitraum – für die Auswertung – mit dem Startdatum und dem Endatum gewählt werden. Die Auswahl erfolgt über eine komfortable Datumsauswahl.

Bei Wechsel der Selektion in der Strategieliste müssen die Charts und Informationen in den Registerkarten aktualisiert werden. Das erfolgt ohne kompletten Neuaufbau der Seite. Die Applikation holte die Rohdaten und aktualisiert lediglich die notwendigen Anzeigen.

### 3.5.2 Herausforderungen

Die Applikation soll ein professionelles Design erhalten. Dazu setzen wir das CSS-Framework Twitter Bootstrap ein. Für AngularJS gibt es UI-Erweiterungen, die auf Bootstrap aufbauen und neue UI-Elemente bereitstellen. Im Projekt nutzen wir mehrere Komponenten, zum Beispiel die Datumsauswahl, Registerkarten und das Accordeon.

Der Umfang und die Struktur der Daten sind größer als im letzten Beispiel. Mithilfe von AngularJS-Direktiven erstellen wir eine Komponente, die bestehende JavaScript-Bibliotheken von Google für die Anzeige von Business Chart integriert.

Eine Herausforderung ist in diesem Fall das Zusammenspiel der Frameworks. Im Falle der Bootstrap-Erweiterungen ist die Integration einfach, da dieses Framework als ein

AngularJS-Modul bereitsteht und sich so harmonisch einfügt. Im Fall von Google Charts ist die Ausgangslage anders. Das Framework ist ein eigenständiges JavaScript-Framework. In der Projektpraxis sind solche Integrationen nicht immer leicht und stellen oft ein Risiko dar. Das konkrete Beispiel zeigt die Erweiterungsmöglichkeiten von AngularJS für solche typischen Situationen.

Bevor wir in die Implementierung des Beispiels einsteigen, stellt der nächste Abschnitt die beiden Frameworks näher vor.

### 3.5.3 AngularJS-UI Bootstrap

Die UI-Elemente, die wir einsetzen wollen, gehen weit über die vordefinierten HTML-Elemente hinaus und betten die beliebten Elemente von Twitter Bootstrap in AngularJS-Direktiven ein. Die Bibliothek bietet Zeit- und Datumsauswahl, Registerkarten, Dialogboxen, Fortschrittsbalken, Tooltips und vieles mehr.

Zur Reife des Projektes ist zu sagen: Ein finaler Stand ist noch nicht erreicht, somit könnten Änderungen der Funktionalität oder Schnittstelle möglich sein. Die Projektseite macht den Einstieg leicht und stellt alle UI-Elemente mit Beispielen und Sourcecode zum Ausprobieren vor. Das Projekt ist unter folgendem Link erreichbar: [angular-ui.github.io/bootstrap](https://angular-ui.github.io/bootstrap)



Bild 3.16: Beispiele der Twitter-Bootstrap-Komponenten für AngularJS.

### Installation im Projekt

Die Abhängigkeit vom neuen Modul UI-Bootstrap erstellen wir durch die Angaben beim initialen Erzeugen der Applikation:

```
angular.module('myModule', ['ui.bootstrap']);
```

In der HTML-Seite müssen folgende Dateien aufgenommen werden:

```
...
<script
  src="http://angular-ui.github.io/bootstrap/ui-bootstrap-tpls-0.5.0.js">
</script>
<link rel="stylesheet"
  href="../lib/bootstrap-combined.min.css">
...
```

Der nächste Abschnitt zeigt an einigen Beispielen, wie die einzelnen neuen Komponenten funktionieren, bevor wir diese in der Applikation einsetzen.

### Erstellen von Dialogfenstern

Dialoge sind frei gestaltbare Fenster, die sich als Pop-up über der Applikation in einem separaten Fenster öffnen. Sie steuern das Benutzerverhalten und erfragen wichtige Informationen.

Bei der Gestaltung hat man alle Freiheiten und kann beliebige HTML-Codes verwenden. Im Beispiel nutzen wir ein separates HTML-Template. Es ist ebenfalls möglich, den HTML-Inhalt direkt in der Dialogbox-Definition als JavaScript-Zeichenkette abzulegen. Das Beispiel zeigt einen Dialog, der einen Wert mit einem Input-Element vom Benutzer erfragt.

Das Template verwendet spezielle CSS-Klassen für Header, Body und Footer des Dialoges, um diesen Bereich grafisch abzusetzen.



Bild 3.17: Darstellungen des Dialogbeispiels.

Der folgende Codeblock enthält das Template für den Dialog:

```
...
<script type="text/ng-template" id="dialog.html">
<div class="modal-header">
  <h3>Das ist der Titel.</h3>
```

```
</div>
<div class="modal-body">
    <p>Geben Sie einen Wert ein:<input ng-model="result"/></p>
</div>
<div class="modal-footer">
    <button ng-click="close(result)" class="btn btn-primary">Close</button>
</div>
</script>
...

```

Die Bibliothek Bootstrap-UI liefert einen neuen Service mit dem Namen `$dialog`. Dieser bietet die Funktion `dialog()`, der die Konfiguration erwartet und eine Instanz des Dialoges liefert. Sichtbar wird der Dialog durch einen Aufruf von `open()` auf dieser Instanz. Falls eine Rückgabe wichtig ist, bietet die Callback-Funktion `then()` die Möglichkeit, die Logik anzugeben, die nach dem Beenden angestoßen wird.

```
function DialogSampleController($scope, $dialog){
    $scope.dialogOptions = {
        backdrop: false,
        keyboard: true,
        backdropClick: true,
        //template: // HTML-Inhalt als Zeichenkette oder
        templateUrl: 'dialog.html',
        controller: 'TestDialogController'
    };

    $scope.openDialog = function(){
        var theDialog = $dialog.dialog($scope.dialogOptions);
        theDialog.open().then(function(result){
            if(result) {
                console.log('Dialog closed with result: ' + result);
            }
        });
    };
}
```

Die Funktion des Kontrollers `openDialog()` könnten wir an einen Button binden und damit das Öffnen des Dialoges starten.

Das Konfigurationsobjekt `dialogOptions` nutzt einige Attribute, die das Verhalten im Detail steuern.

Ein Blick in die folgende Tabelle erklärt die wichtigsten Attribute und ihre Bedeutung:

Attribut	Bezeichnung
backdrop	Überlagert den Hintergrund, sodass der Dialog modal wird und keine anderen Aktionen im Hauptfenster der Applikation möglich sind.
backdropClick	Wenn dieses Attribut aktiviert ist, kann der Benutzer den geöffneten Dialog mit einem Klick in den Hintergrund (neben den Dialog) beenden. Er muss nicht die vorgesehene Schaltfläche für das Schließen nutzen.
keyboard	Definiert, ob der Dialog zusätzlich zur Maus über die Escape-Taste geschlossen werden kann.
templateUrl template	TemplateUrl benennt das Template-Fragment für die Darstellung. Alternativ kann der Inhalt im JavaScript definiert werden. Das Attribut template erhält in diesem Fall den vollständigen HTML-Code für die Anzeige.
controller	Enthält den Namen der Kontrollerklasse für den Dialog.

Für den Dialog stellen wir einen eigenen Kontroller bereit. Dieser erhält den \$scope und die Dialoginstanz als Argumente. In unserem Fall definiert er nur eine Funktion (dialog.close()) für das Schließen des Dialoges.

```
function TestDialogController($scope, dialog){
  $scope.close = function(result){
    dialog.close(result);
  };
}
```

### Beispiel Meldungsfenster

Meldungsfenster sind etwas einfacher aufgebaut und bieten weniger Freiheit als ein Dialog. Für viele Fälle ist das ausreichend. Eine häufige Anforderung an Applikationen besteht darin, den Nutzern Meldungen und Informationen zu präsentieren und spezielle Eingaben abzufragen. Dabei ist es hilfreich, Dialoge oder Pop-up-Fenster zu nutzen. Typische Beispiele sind Warnungen oder Sicherheitsabfragen. Der altbekannte Alert-Dialog aus dem Standard von JavaScript bietet nur eingeschränkte Möglichkeiten und passt selten in ein professionelles Design. Die Bootstrap-UI-Komponente bietet Abhilfe und definiert flexible Meldungsfenster.

Die MessageBox eignet sich dafür, kurze, wichtige Nachrichten exklusiv an den Benutzer zu übermitteln. So kann man den Benutzer auf die Konsequenzen hinweisen und fragen, ob er diese Aktion wirklich durchführen möchte.

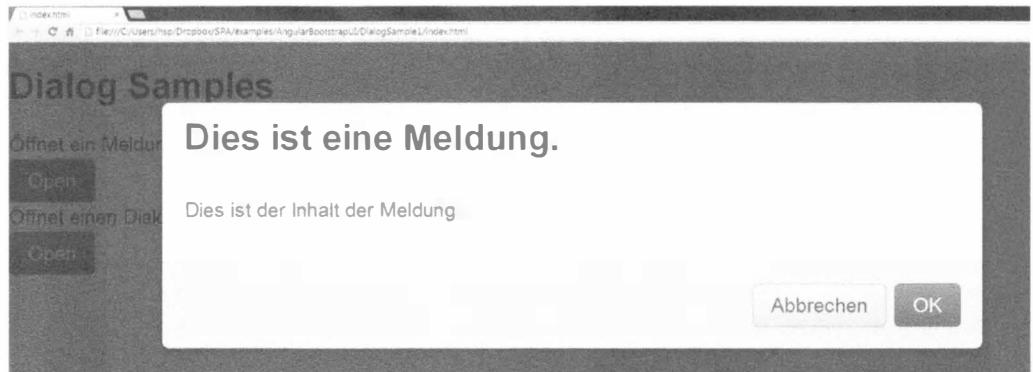


Bild 3.18: Beispiel für ein Meldungsfenster.

Im Quelltext definiert die Funktion zuerst Variablen für die Konfiguration des Meldungsfensters. Darin sind die gewünschten Knöpfe in einem Array aufgeführt. Für jede Schaltfläche bietet das Array ein Objekt mit dem Rückgabewert, der Beschriftung und einer optionalen CSS-Klasse für die Darstellung:

```
...
$scope.openMessageBox = function(){
    var title = 'Dies ist eine Meldung.';
    var message = 'Dies ist der Inhalt der Meldung.';
    var buttons = [
        {result:'cancel', label: 'Abbrechen'},
        {result:'ok', label: 'OK', cssClass: 'btn-primary'} ];

    $dialog.messageBox(title, message, buttons)
        .open()
        .then(function(result){
            console.log('Geschlossen mit Resultat:' + result);
        });
};

...
```

### 3.5.4 Google Charts

In den frühen Tagen der Webentwicklung war das Erstellen von individualisierten Grafiken oder Bildern fast immer eine Aufgabe, die auf dem Server erledigt werden musste. Zum einen lagen die eventuell umfangreichen Rohdaten auf dem Server vor und dieser bot sowohl die notwendigen Frameworks als auch die Leistungsfähigkeit für diese Aufgabe.

Andere Technologien wie Applets und Flash lösten das Problem durch die Bilderzeugung auf dem Client, werfen aber neue Probleme der Integration und Sicherheit auf.

Die letzten Jahre haben die Rolle des Clients verändert. Google Charts ist ein gutes Beispiel, wie grafisch aufwendige Aufgaben mittlerweile gut in JavaScript im Browser lösbar sind. Google bietet mit seiner Chart-Bibliothek schon seit einigen Jahren ein Framework an, um professionelle Geschäftscharts mit reinem JavaScript und HTML zu erstellen. Dabei können umfangreiche Daten visualisiert werden. Das Framework besticht durch eine große Bandbreite an Charttypen, eine hohe Konfigurierbarkeit und Interaktivität.

Vorhanden sind alle gängigen Typen wie Linien-, Balken- und Tortengrafiken. Diese können in hohem Maße angepasst und teilweise in einer Grafik kombiniert werden. Hinzu kommen weitere interessante Typen wie Organisations-Charts, Geo-Charts, Candlestick-Charts oder Treemaps. Die Galerie auf der Projektseite bietet einen schönen und bunten Überblick:

[google-developers.appspot.com/chart/interactive/docs/gallery](http://google-developers.appspot.com/chart/interactive/docs/gallery)

**Chart Gallery**

Our gallery provides a variety of charts designed to address your data visualization needs. These charts are based on pure HTML5+SVG technology (adopting VML for old IE versions), so no plugins are required. Adding these charts to your page can be done in a very simple way.

Some additional community-contributed charts can be found on the [Additional Charts page](#).

- Overview
- Chart Gallery
- Playground
- Miscellaneous Examples
- Area Charts
- Bar Charts
- Bubble Charts
- Candlestick Charts
- Column Charts
- Combo Charts
- Gauge Charts
- Geo Charts
- Line Charts
- Maps
- Org Charts
- Pie Charts
- Scatter Charts
- Stacked Area Charts
- Table Charts
- Timelines
- Tree Map Charts
- Trendlines
- Advanced Usage
- Customizing Charts
- Chart Types
- Axes Options
- TecNcs
- Formatters
- Contributing a New Chart
- Development Tools
- Library Loading
- Interacting with Charts
- Chart Data
- Commands
- API Reference
- GoogleChartTools
- Related Chart Tools
- Terms and Conditions

**Pie Chart**

**Scatter Chart**

**Gauge**

**Geo Chart**

**Table**

Name	Salary	FullTime
Mane	\$24.700	✓
Aberd	\$24.200	✗
Ennco	\$25.700	✓
Usa	\$26.600	✓

**Treemap**

**Combo Chart**

**Line Chart**

**Bar Chart**

**Column Chart**

**Area Chart**

**Candlestick Chart**

**Timeline**

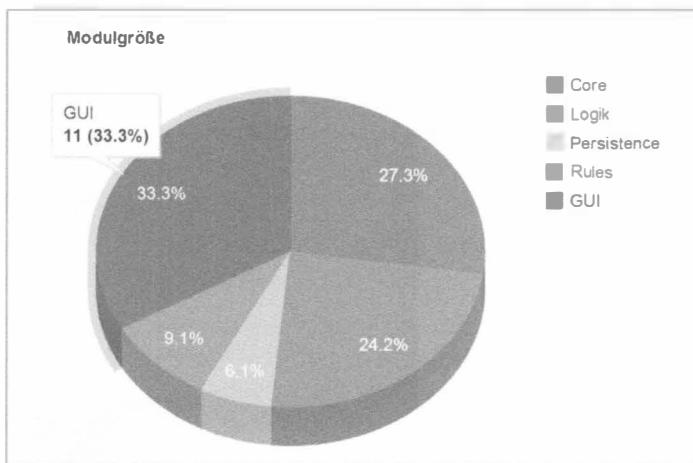
**Bubble Chart**

**Donut**

**Bild 3.19:** Google Charts bietet eine große Auswahl unterschiedlicher Diagrammarten.

### Wir bauen einen ersten Chart

Wie leicht sich die Bibliothek nutzen lässt, wollen wir an einem einfachen Beispiel testen. Wir erstellen ein Tortendiagramm (PieChart).



**Bild 3.20:** Ein einfaches Tortendiagramm als erstes Beispiel.

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript"
        src="https://www.google.com/jsapi"></script>
    <script type="text/javascript">
        google.load(
            "visualization", "1", {packages:["corechart"]});

        google.setOnLoadCallback(drawChart);

        function drawChart() {
            var data = google.visualization.arrayToDataTable(
                [
                    ['Modulname', 'Codezeilen je Modul'],
                    ['Core',      9],
                    ['Logik',     8],
                    ['Persistence', 2],
                    ['Rules',   3],
                    ['GUI',     11]
                ]);
            var options = {
                title: 'Modulgröße',    is3D: true,
            };

            var chart = new google.visualization.PieChart(
                document.getElementById('chart_3d'));
            chart.draw(data, options);
        }
    </script>
</head>
<body>
    <div id="chart_3d" style="width: 100%; height: 100%;></div>
</body>
</html>
```

```
</script>
</head>
<body>
  <div id="chart_3d" style="width: 900px; height: 500px;">
  </div>
</body>
</html>
```

Zuerst müssen die Bibliotheken geladen werden. Google definiert ein übergreifendes JavaScript-API, das in der ersten Script-Anweisung geladen wird. Die Chart-Bibliothek selbst wird erst mit dem Befehl `google.load()` nachgeladen. Da das Übertragen etwas dauern kann, wird das Zeichnen des Charts asynchron über die Anweisung `google.setOnLoadCallback()` zu dem Zeitpunkt gestartet, an dem die Bibliothek vollständig übertragen und initialisiert ist.

Die Callback-Funktion `drawChart()` enthält die fachlichen Werte für das Diagramm: Die Daten liegen in einer DataTable. Google bietet Funktionen an, um diese DataTable aus einem herkömmlichen JavaScript-Array zu erzeugen. Die Namen der Spalten befinden sich in der ersten Zeile des Feldes, gefolgt von den eigentlichen Daten der Spalten.

Google Charts bietet viele detaillierte Einstellungen für die individuelle Anpassung der Charts. Die Dokumentation listet die Parameter detailliert auf und erklärt sie. Die Bandbreite der Einstellungen umfasst Beschriftungen, Farbwahl, Positionen von Achsen und Legenden und vieles mehr.

Im Beispiel enthält die Variable `options` nur Titel und das Kennzeichen für eine 3-D-Optik. Für die Erzeugung des Charts müssen wir das HTML-Element identifizieren, das den Chart aufnehmen soll. Der Konstruktor des PieCharts erhält den Identifier als Parameter. Andere Charttypen bieten ähnliche Konstruktoren an, wie das Beispiel für einen Liniendiagramm zeigt:

```
new google.visualization.LineChart(
  document.getElementById('chart_3d'));
```

Das Zeichnen selbst übernimmt die Funktion `draw()`. Diese erwartet die Daten in Form einer DataTable und die Optionen für den Chart. Im überschaubaren HTML-Body ist lediglich das DIV-Element mit dem Identifier (`chart_3d`) zu sehen, das als Container für unser Diagramm dient.

Ausführlichere Beispiele und eine ausführliche Dokumentation inklusive der umfangreichen Optionen finden sich auf der Projektseite: [google-developers.appspot.com/chart](http://google-developers.appspot.com/chart)

### 3.5.5 Umsetzen der Anforderungen

Die Anforderungen für das nächste Beispielprojekt und die beiden wichtigsten Frameworks wurden vorgestellt. Jetzt müssen wir diese Elemente nur richtig zusammenführen. Dabei kann AngularJS mit weiteren Stärken glänzen.

## Eine AngularJS-Direktive für Diagramme

HTML wurde gerade in der Version 5 um neue Tags und Features erweitert. Die Charts, die wir in der Applikation nutzen wollen, sind leider nicht im HTML-Standard zu finden und es ist nicht abzusehen, dass ein solches Feature für eine nächste Version geplant ist.

Ideal wäre es, wenn wir HTML selbst um neue projektbezogene Tags ergänzen könnten. Wie würden wir eine solche Chart-Komponente in einer HTML-Seite idealerweise nutzen? Hier ein Vorschlag für einen solchen neuen HTML-Tag:

```
<chart height='400' width='900' varname='jahresvergleichChartConfig'>  
</chart>
```

Wir definieren die Abmessungen des Charts mit Höhe und Breite in Pixeln. Der Rest der Konfiguration – wie zum Beispiel die Daten, Beschriftungen, Dialogrammtyp u.v.a. – liegt in einer Struktur mit dem Namen `jahresvergleichChartConfig`, die wir uns im JavaScript-Teil genauer anschauen.

AngularJS kann nicht zaubern. Aber mit AngularJS-Direktiven können wir unseren Wunsch von dem neuen HTML-Tag erfüllen. Der Erweiterungsmechanismus erlaubt es, zusätzliche Semantik zum Standard von HTML zu definieren. Grob gesagt sind wir damit in der Lage, eigene Markup-Erweiterungen zu erstellen. Diese Erweiterungen können sehr projektspezifisch oder sehr allgemein sein.

Oft sind solche Erweiterungsmöglichkeiten kompliziert und erfordern viel Detailwissen über das Framework selbst. AngularJS löst diese Herausforderung vorbildlich und nimmt dem Entwickler viel Arbeit ab. In HTML wird weiterhin die Struktur definiert. Die Logik findet sich komplett in JavaScript wieder.

## Implementierung der Charts-Komponente im Detail

Im nächsten Abschnitt erstellen wir schrittweise die neue Chart-Direktive. Wie schon erwähnt, bieten die Diagramme sehr viele Einstellungsmöglichkeiten. Hinzu kommen unsere fachlichen Daten, die im Diagramm visualisiert werden sollen. Wegen des Umfangs der Informationen sollten wir diese nicht in einzelnen Parametern, sondern in einer Datenstruktur an die Konfiguration übergeben. Das folgende Beispiel zeigt, wie diese Struktur aussehen kann:

```
$scope.jahresvergleichChartConfig = {  
    title: $scope.strategy.name,  
    data: dataService.getYearData($scope.strategyid),  
    columns: dataService.getJahresColumns(),  
    type: "ColumnChart",  
    options: {  
        hAxis: {  
            slantedText: true,  
            slantedTextAngle: 30,  
            showTextEvery: 1,  
            minTextSpacing: 100,  
        }  
    }  
};
```

```

        allowContainerBoundaryTextCutoff: true
    }
}
}
}
```

Zwei wesentliche Elemente fallen sofort auf: Das title-Attribut trägt die Überschrift des Diagramms und das type-Attribut enthält den Namen des Typs.

Die folgende Tabelle zeigt alle Attribute der Konfigurationsstruktur.

Attributname	Beschreibung
title	Definiert die Überschrift des Diagramms.
type	Definiert den Diagrammtyp und entspricht dem Namen des Konstruktors.
data	Enthält die fachlichen Daten in Form eines Arrays von Zeilen. Die Anzahl der Elemente in jeder Zeile entspricht der Anzahl der Spalten im Attribut columns.
columns	Hält die Spaltennamen und die Datentypen für das Diagramm in Form eines Feldes bereit.
options	Definiert weitere Detaileinstellungen für das Diagramm, so wie es der Diagrammtyp laut Dokumentation von Google Charts beschreibt.

Werfen wir einen weiteren Blick in die Konfiguration der Spalten selbst. Wir müssen den Datentyp und die Bezeichnung jeder Spalte bereitstellen:

```

var columnsJahresvergleich = [
  {type: 'number', name: 'Jahr'} ,
  {type: 'number', name: 'Underlying'} ,
  {type: 'number', name: 'Strategie'} ,
  {type: 'number', name: 'Change'}
]
```

Die Direktive ist in der HTML-Seite verbaut und wir wissen, wie die Konfiguration aussieht. Damit haben wir alle Vorbereitungen getroffen, um die AngularJS-Direktive selbst zu erstellen. Die neue Direktive erzeugen wir über die AngularJS-Funktionen directive() und übergeben den Namen chart und eine JavaScript-Funktion als Herzstück der Direktive:

```

chartApp.directive('chart', function () {
  return {
    restrict: 'E',
    link: function ($scope, $element, $attr) {

      // Chart-Konfiguration als Name der Variablen übergeben:
```

```

var varname = $attr['varname'];
if (!$scope[varname]) {
    console.log("Chart: No config found in scope with name :" + varname);
    return;
}

// Check for changes on the configuration and data
$scope.$watch(varname,
    function (newValue, oldValue) {
        if (newValue) {
            config = $scope[varname];

            // Aus den fachlichen Daten eine DataTable erzeugen
            var data = new google.visualization.DataTable();
            console.log('Chart: ' + varname);
            for (var i = 0; i < config.columns.length; i++) {
                var col = config.columns[i];
                data.addColumn(col.type, col.name);
            }
            data.addRows(config.data);

            // Set chart options
            var options = config.options;
            if (!options)
                { options = {}; }
            options['title'] = config.title;
            options['width'] = $attr['width'];
            options['height'] = $attr['height'];

            var type = config.type;
            if (!type) {
                type = "LineChart";
            }
            // Instantiate and draw our chart
            var jscommand = "new google.visualization." + type +
"($element[0]);";
            var chart = eval(jscommand);

            chart.draw(data, options);
        }
    }, true);
}
);

```

Die übergebene Funktion muss laut Kontrakt festgelegte Attribute veröffentlichen:

Das Attribut **restrict** definiert, wie sich die Direktive in HTML einbettet und genutzt werden kann. Es gibt folgenden Möglichkeiten:

<b>Wert für das Attribut restrict</b>	<b>Beschreibung</b>	<b>Beispiel</b>
E	Eigenständiges HTML-Element	<neuedirektive> </neuedirektive>
A	HTML-Attribut an einem HTML-Tag	<div neuedirektive="exp"> </div>
C	CSS-Klasse innerhalb eines Tags	<div class="neuedirektive:exp;">
M	HTML-Kommentar	<!-- directive: neuedirektive exp -->

Die Angaben in dem Attribut **restrict** lassen sich kombinieren: Die Angabe **restrict: 'EAC'** würde somit drei Alternativen erlauben. Wie im Quellcode dokumentiert, nutzen wir die Direktive als eigenständigen HTML-Tag.

Das Attribut **link** muss eine Funktion enthalten, die die Logik hinter der Direktive beschreibt. Diese ist für unseren Fall etwas aufwendiger. Zuerst lesen wir den Namen der Konfigurationsvariablen aus. Wenn wir den Namen kennen, überwachen wir Wertänderungen dieser Variable mit dem Watch-Mechanismus von AngularJS. Die Funktion im zweiten Parameter der **watch**-Anweisung wird bei jeder Wertänderung aufgerufen und transportiert jeweils den alten und neuen Wert:

```
$scope.$watch(varname, function (newValue, oldValue) {
  ...
})
```

Die Logik der Direktive besteht aus folgenden Schritten:

- Die fachlichen Daten in eine DataTable übertragen.
- Die Konfiguration inklusive Beschriftungen in eine Struktur übertragen, die für den Charttyp notwendig ist.
- Aus dem Charttyp leiten wir den Konstruktor ab und erzeugen dynamisch ein JavaScript-Statement, das ein Chart-Objekt mit der **eval()**-Funktion instanziert.
- Diese Instanz wird mit der Konfiguration und den fachlichen Daten versorgt und zum Schluss mit der Funktion **draw()** gezeichnet.



Bild 3.21: Die Strategieliste und ein ausgewähltes Diagramm im EvoTrader.

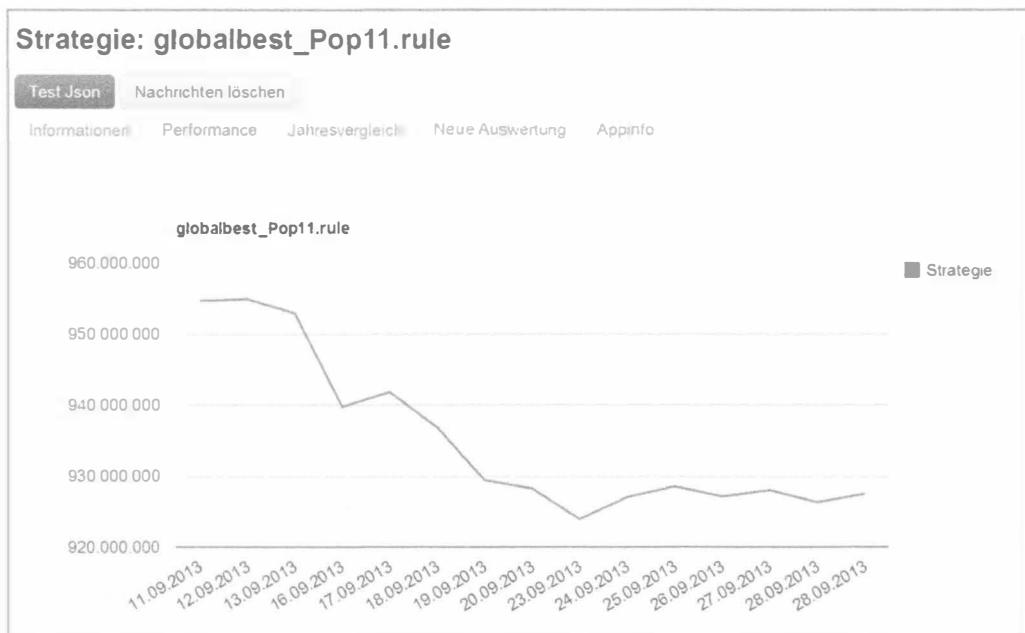


Bild 3.22: Ein zweiter Diagrammtyp im Detailbereich.

## Tabs und Registerkarten

Registerkarten sind eine sehr hilfreiche und übliche Strukturierung für Applikationen. Eine große Anzahl von Informationen oder Steuerelementen kann in thematische Bereiche aufgeteilt werden, ohne auf neue Seiten oder separate Dialoge zu verweisen.

Die Beispielapplikation nutzt Registerkarten, um die Informationen von den Charts zu trennen. Die Bootstrap-UI-Erweiterung bringt schon eine funktionsfähige und flexible Komponente mit. Das folgende Beispiel demonstriert den Einsatz:

```
<tabset>
  <tab heading="Informationen">
  ...
  </tab>
  <tab heading="Performance" >
  ...
  </tab>
  <tab heading="Einstellungen">
  ...
  </tab>
</tabset>
```

Das Tabset stellt einen Container für die Tabs (Reiter) bereit. Innerhalb der einzelnen Tag-Elemente können weitere HTML-Elemente eingebettet sein.

Auf der Projektseite von UI-Bootstrap befinden sich erweiterte Beispiele für den Einsatz von Tabs. So ist es möglich, ein Tabset programmatisch mit Reitern zu füllen oder beim Wechseln der Tabs können Aktionen wie die Rückfrage beim Benutzer angestoßen werden.

## Accordion und dynamische Feldausgaben

Eine weitere Möglichkeit, mehr Übersichtlichkeit in einer Applikation zu schaffen, ist das Accordion. Es wird meist vertikal eingesetzt und gruppiert Inhalte zu thematischen Gruppen. Der Name passt, da sich die Bereiche wie ein Akkordeon zusammenziehen, wenn ein neuer Bereich geöffnet wird. Meist ist die Komponente so konfiguriert, dass nur jeweils ein Bereich geöffnet ist.

Da die Strategien in unserem Beispiel über viele Informationen verfügen, werden diese in dem Accordion gruppiert. Anders als bei den Registerkarten ist die Implementierung hier stark konfigurierbar. Sowohl die thematischen Gruppen als auch die einzelnen Felder in jeder Gruppe werden im JavaScript-Code konfiguriert und dynamisch angezeigt. Außerdem wird unterschieden zwischen dem internen Namen, den das Attribut in der Datenstruktur der Strategie hat, und dem sprechenden Namen, den der Benutzer sieht.

Ein solches Vorgehen lohnt sich nur, wenn es sehr wahrscheinlich ist, dass sich die Felder oder Gruppen und deren Zuordnung in der Zukunft ändern werden.

Das folgende Listing zeigt die Struktur zur Definition der Gruppen und der Zuordnung der einzelnen Felder zu einer Gruppe im Attribut `fields`:

```
$scope.fieldDefinition = [
  { group: "basic", label: "Allgemein",
    fields: [
      { label: "Chartname", fname: "ChartName" },
      { label: "Taktung", fname: "ChartPeriod" },
      { label: "MaxBefehle", fname: "MaxBefehle" }
    ]
  },
  { group: "performance", label: "Auswertung",
    fields: [
      { label: "Gesamte Transaktionkosten", fname:
        "GesamtTransaktionkosten" },
      { label: "Wert", fname: "Wert" },
      { label: "Max. Einbruch", fname: "MaxEinbruch" },
      { label: "Max .Einbruch (%)", fname: "MaxEinbruchProzent" },
      { label: "Anzahl Käufe", fname: "Kauf" },
      { label: "Anzahl Verkäufe", fname: "Verkauf" },
      { label: "Anzahl Käufe (short)", fname: "KaufShort" },
      { label: "Anzahl Verkäufe (short)", fname: "VerkaufShort" }
    ]
  },
  ...
];
```

Jede Gruppe hat einen internen Name (`group`), einen sprechenden Namen für die Darstellung (`label`) und eine Liste(`fields`), in der die zugeordneten Attribute der Strategie-Entität enthalten sind. Jedes Element dieser Liste enthält ebenfalls zwei Attribute: ein Label für die Anzeige und ein Attribut `fname`, das auf das fachliche Feld in der Strategie verweist.

### Dynamische Darstellung des Accordion

In der HTML-Seite definieren wir ein Accordion als übergeordneten Container. Das Attribut `close-others` legt fest, ob beim Öffnen eines Bereichs der letzte sichtbare Bereich automatisch geschlossen wird. Wir setzen den Wert auf `false` und lassen damit zu, dass alle Bereiche geöffnet bleiben.

In der Accordion-Group iterieren wir mit dem ersten `ng-repeat` über die Gruppen (`fieldDefinition`) und erstellen so für jede Gruppe einen Accordion-Group-Tag. Als Bezeichnung im Heading-Attribut nutzen wir den entsprechenden Namen aus dem Label-Attribut der Gruppdefinition.

Innerhalb der Gruppe wandern wir über die Felder in der Gruppe (`fd.fields`) und füllen eine Tabelle mit allen Feldnamen `f.label`. Mit dem Attribut `f.fname` wird die

fachliche Information direkt aus der Strategie gelesen und in der zweiten Spalte positioniert.

```
<accordion close-others="false">
  <accordion-group ng-repeat="fd in fieldDefinition" heading="{{fd.label}}">
    <table>
      <tr ng-repeat="f in fd.fields">
        <td>{{f.label}}:</td>
        <td>{{strategy[f.fname]}}</td>
      </tr>
    </table>
  </accordion-group>
</accordion>
```

**Allgemein**

Chartname:	Dax_D1_OLHC_1991_2013
Taktung:	D1
Fitness Fkt Name:	Gewinn vs. Risiko
Fitness Fkt Beschreibung:	Fitness = (Gewinn / Groesster Einbruch)
MaxBefehle:	500

**Auswertung**

Gesamte Transaktionskosten:	971 354,94
Wert:	74.490.752,00
Max. Einbruch:	3.855 383,75
Max. Einbruch (%):	1 6942204
Anzahl Käufe:	689
Anzahl Verkäufe:	688
Anzahl Käufe (short):	89
Anzahl Verkäufe (short):	89
Längste Zeit ohne Aktion:	721

Parameter

Training

**Bild 3.23:** Das Accordion in der Applikation.

### Komfort bei der Datumsauswahl

Für Business-Applikationen müssen oft Datumswerte erfasst werden. Die Fehlermöglichkeiten sind im Vergleich zu einem Text oder Zahlenfeld vielfältig. Die Prüfung, ob der 29. Februar eine gültige Eingabe ist, hängt bekanntlich zusätzlich vom Jahr ab.

Vorgefertigte UI-Komponenten bieten Komfort und reduzieren die Wahrscheinlichkeit von Fehleingaben. Meist hilft dem Benutzer schon eine Darstellung in Form eines Kalenders weiter.

Die Beispielapplikation nutzt eine Datumskomponente im Reiter `Einstellungen` für die Eingabe des Start- und Enddatums für die Auswertung der Strategien. Zum Einsatz kommt hier der Datepicker der Bootstrap-UI-Komponenten.

Klickt der Benutzer in das Eingabefeld für das Datum, öffnet sich ein Pop-up-Dialog mit einer Kalenderansicht. Der Anwender kann die Jahre und Monate vor- und zurückblättern und ein Datum auswählen. Die Auswahl erscheint im Eingabefeld im numerischen Format.

Ein hilfreiches Feature ist die Angabe für gültige Datumsbereiche. So kann das Enddatum für die Auswertung nicht in der Zukunft liegen. Im Pop-up kann der Benutzer in die Zukunft blättern, die Tage sind aber deaktiviert und nicht auswählbar.

Der folgende Codeblock zeigt den Einbau der Datumsauswahl in die Applikation für die beiden Eingabefelder Startdatum und Enddatum:

```
...
<tab heading="Neue Auswertung">

    <div class="form-horizontal">
        Startdatum:
        <input type="text"
            datepicker-popup="dd.MM.yyyy" ng-model="startDate"
            open="openedStartDate"
            datepicker-options="dateOptions"
            ng-required="true"
            min="'1.1.1990'" max="'9.1.2013'" />
        <button class="btn" ng-click="openStartDate()">Open
        </button>
    </div>

    <div class="form-horizontal">
        Enddatum:
        <input type="text"
            datepicker-popup="dd.MM.yyyy" ng-model="endDate"
            open="openedEndDate"
            datepicker-options="dateOptions"
            ng-required="true"
            min="'1.1.1990'" max="'9.1.2013'" />
        <button class="btn" ng-click="openEndDate()">Open
        </button>
    </div>
    ... //Weiter Felder im Formular
</tab>
...
```

Für jedes Datumsfeld können wir das Verhalten über einige Attribute genauer steuern:

Attributname	Beschreibung
Datepicker-Pop-up	Gibt das Format für die Anzeige des Datums vor.
min und max	Legen das minimale und maximale Datum fest, in welchem Bereich eine gültige Eingabe akzeptiert wird. In der Anzeige des Pop-up-Fensters sind die Tage außerhalb des Bereiches grau hinterlegt und deaktiviert (nicht wählbar).
open	Das Attribut verweist auf eine Variable, die steuert, ob das Pop-up-Fenster mit der Kalenderansicht gerade sichtbar ist.
datepicker-options	Definiert zusätzliche Attribute wie das Jahresformat oder mit welchem Wochentag eine Woche beginnt. Der Wert 1 legt den Montag als ersten Wochentag fest: <pre>\$scope.dateOptions = [   'year-format': 'YYYY',   'starting-day': 1 ];</pre>

Die Attribute `ng-model`, `ng-click` und `ng-required` sind klassische AngularJS-Attribute, wie wir sie schon häufiger benutzt haben.



**Bild 3.24:** Die Datumsauswahl in Aktion.

## Meldungen und Warnungen

Im ersten Beispiel haben wir uns im Formular für die Detailpflege mit Meldung im Rahmen der Validierung beschäftigt. In dieser Anwendung wollen wir allgemeine Meldungen noch flexibler visualisieren.

Die Nachrichten können unterschiedliche fachliche Hintergründe haben:

- Die Kommunikation mit dem Server war erfolgreich und die Daten sind aktualisiert.
- Der Benutzer hat fachlich invalide Daten in ein Formular eingegeben, wobei in diesem Fall mehrere Eingabefelder logisch zusammenhängen. Die Validierung eines Feldes alleine greift daher zu kurz.

Hinzu kommen folgende Anforderungen im Hinblick auf die Meldung:

- Meldungen sollen sich farblich in Fehler, Erfolgsmeldungen und neutrale Meldungen unterscheiden.
- Meldungen sollen nach einer gewissen Zeit verschwinden.
- Es können mehrere Meldungen gleichzeitig erscheinen.
- Meldungen sollen durch den Benutzer einzeln entfernbbar sein.

Für einen Teil der Anforderung bieten die Bootstrap-UI-Komponenten eine Lösung an: Alert-Messages erfüllen die geforderten Anforderungen hinsichtlich der Darstellung.

Das zeitliche Verhalten, dass Meldungen automatisch verschwinden, müssen wir selbst implementieren. Mithilfe eines AngularJS-Filters lösen wir die Aufgabe elegant und flexibel:

Die Meldungen selbst legen wir im Kontroller im Feld `$scope.messages` ab. Die enthaltenen Einträge folgen dem Muster:

```
$scope.messages = [
  { type: 'error',
    msg: 'Oh, da hat etwas nicht geklappt!',
    timestamp : new Date() },
  { type: 'success',
    msg: 'Diese Aktion war erfolgreich.',
    timestamp : new Date() }
];
```

Jeder Eintrag erhält die folgenden drei Attribute:

Attributname	Beschreibung
type	Den Typ der Nachricht als einen der Werte: 'error', 'info' oder 'success'. Bei allen anderen Werten soll eine neutrale Meldung erscheinen. Der Wert steuert ebenfalls die farbliche Darstellung: Rot: error Blau: info Grün: success Gelb: Sonst
msg	Enthält den eigentlichen Text der Meldung als Zeichenkette.
timestamp	Ein Date-Objekt mit der Zeit (in Millisekunden), in der die Nachricht aufgetreten ist.

Um eine Nachricht zu erstellen, definieren wir eine neue Funktion, die gleichzeitig das Attribut Timestamp automatisch füllt. Somit reichen zwei Parameter für den Nachrichtentext und den Typ aus:

```
$scope.addMessage = function ( msg, type ) {
  if ( !type )
  {
    type = 'info';
  }
  $scope.messages.push(
    {type: type.toLowerCase(),
     msg: msg, timestamp: new Date()
    });
};
```

Für das Entfernen einer Meldung erstellen wir eine Funktion, die den Index der Meldung erhält, die zu entfernen ist. Eine zweite Funktion leert das gesamte Array, indem sie ein leeres Feld zuweist.

```
$scope.clearMessage = function (index) {
  $scope.messages.splice(index, 1);
};

$scope.clearAllMessages = function() {
  $scope.messages = [];
};
```

Für die Darstellung nutzen wir das Alert-Element der Bootstrap-UI-Bibliothek und iterieren über das Feld der Nachrichten. Somit wird pro Eintrag im Nachrichtenfeld ein Alert-Tag erzeugt und mit den Attributwerten versorgt:

```
<alert ng-repeat="amessage in messages | newest:60"
      type="amessage.type"
      close="clearMessage($index)">
  {{amessage.msg}} ({{amessage.timestamp | date:'HH:mm:ss'}})
</alert>
...
<button
  class="btn"
  ng-click="clearAllMessages()">Nachrichten löschen
</button>
```

Den Typ für das Alert-Element entnehmen wir dem Attribut `amessage.type` und das Attribut `amessage.msg` kommt als Textinhalt zur Anzeige. Das Attribut `amessage.timestamp` formatieren wir mit einem AngularJS-Filter und zeigen nur die Uhrzeit an.

Das Alert-Element bietet bereits die Möglichkeit für den Benutzer, ein einzelnes Element zu entfernen. Im Element erscheint eine Schaltfläche zum Entfernen. Die Methode im Kontroller, um ein Element zu löschen, wurde oben vorgestellt. Jetzt verknüpfen wir diese Funktion mit dem Close-Attribute des Alert-Elements. Den Index aus dem Repeat-Scope (`$index`) erhält die Close-Methode und reicht ihn an die `clearMessage()`-Methode weiter.

Damit sind wir fast fertig: Es fehlt die Funktion zum Ausblenden von Meldungen nach einer gewissen Zeitspanne. Für diese Anforderung definieren wir einen neuen Filter mit dem Namen `newest`, der im `ng-repeat`-Tag schon referenziert ist.

Als Parameter erhält der Filter die Anzahl an Sekunden, die eine Meldung mindestens sichtbar bleiben sollen. Im Beispiel oben entspricht der Wert 60 somit genau einer Minute. Den Filter können wir mit wenigen Zeilen implementieren:

```
chartApp.filter( 'newest', function () {
  return function ( messageItem, timeToShow ) {
    var arr = [];
```

```

var now = new Date().getTime()-(timeToShow * 1000);
for (var i = messageItem.length; i--;) {
    var item = messageItem[i];

    if ( item.timestamp.getTime() > now )
    {
        arr.push(item);
    }
}
return arr;
});
});

```

Zuerst ermittelt der Filter die aktuelle Zeit minus die übergebene Mindestanzeigezeit in Millisekunden. Alle Meldungen, bei denen der Timestamp unter diesem Wert liegt, können ausgeblendet werden. Genau diesen Vergleich führt der Filter im zweiten Schritt durch, indem er alle Einträge im Feld der Nachrichten prüft und ein neues Feld als Ergebnis liefert.

Ein kleiner Nachteil dieser Realisierung soll an dieser Stelle zur Sprache kommen: Die Nachrichten verschwinden nicht von selbst, sondern der Filter wird nur angewendet, wenn eine Benutzeraktion stattfindet und AngularJS andere Elemente aktualisiert. Erst in diesem Fall wird der Filter geprüft und die Liste aktualisiert. Eine alternative Implementierung (mithilfe eines JavaScript-Timers) könnte regelmäßig die Meldungen automatisch ohne Benutzeraktion verbergen.

### Kommunikation mit dem Backend per JSON

Die inhaltlichen Daten der Strategien können über HTTP-Requests im JSON-Format von einem Service abgerufen werden. In diesem Fall liegt keine REST-Schnittstelle vor und auf die Daten wird nur lesend zugegriffen. Für solche Szenarien reicht der Standard AngularJS \$http-Service völlig aus. Das folgende Beispiel erstellt einen einfachen GET-Request an die im Parameter url definierte Adresse:

```

$http({method: 'GET', url: '/someUrl'}).
success( function(data, status, headers, config) {
    // Diese Funktion wird asynchron aufgerufen, wenn
    // die Antwort eintrifft.
}).
error(function(data, status, headers, config) {
    // Diese Funktion wird im Fehlerfall asynchron
    // angesprungen.
});

```

Eine der beiden Callback-Funktionen für den Erfolgs- oder Fehlerfall übernimmt die Arbeit, wenn die Antwort eintrifft. Liegt der HTTP-Statuscode im Bereich 200 bis 299, interpretiert AngularJS die Antwort als Erfolg.

AngularJS erlaubt es, globale Header-Werte zu definieren, die das Framework bei jedem Request automatisch mit versendet. Diese können über das Objekt `$httpProvider` flexibel eingestellt werden:

```
// Definiert einen Header für alle GET-Anfrage:  
$httpProvider.defaults.headers.get  
= { 'Accept' = 'application/json, text/plain' }  
  
// Definiert einen Header für alle HTTP-Anfrage:  
$httpProvider.defaults.headers.common  
= { 'Content-Type' = 'application/json' }
```

### Die Implementierung der Kommunikation

Die Daten für die Anzeige in der View werden immer aus dem lokalen DataService entnommen. Aktiviert der Benutzer das neue Laden der Strategieauswertung, wird der JSON-Request angestoßen, nach dem erfolgreichen Erhalt der Daten werden diese im DataService abgelegt.

Somit haben wir auch im Falle einer gescheiterten Kommunikation mit dem Backend einen korrekten (wenn auch nicht aktualisierten) Datenbestand für die Anzeige. Nachteilig fällt auf, dass wir bei dieser Implementierung die Daten aller Handelsstrategien im Speicher halten.

Die Abbildung zeigt den schematischen Ablauf der einzelnen Schritte:

- Der Benutzer aktiviert die Abfrage neuer Daten und der Request wird abgesendet.
- Im Erfolgsfall erhält der Controller vom Backend die neuen Daten.
- Diese werden dem DataService übergeben und ersetzen die bisher gespeicherten Informationen.
- Für die Anzeige werden die Daten immer aus dem lokalen Cache (im DataService) entnommen.

### Die Same-Origin-Policy

Normalerweise finden Requests aus dem JavaScript-Code mithilfe des XMLHttpRequests statt. Das funktioniert, so lange sich eine Anfrage an dieselbe Domain richtet, von der das ausführende Script geladen wurde. Ruft eine Webseite eine andere Domain auf, schlägt eine Sicherheitsbeschränkung zu: Die Same-Origin-Policy vermutet hinter Anfragen zu anderen Domains bösartige Angriffe und verhindert diese.

Die Fehlermeldung, die in der Console des Browsers (hier Google Chrome) erscheint, könnte etwa folgenden Wortlaut haben:

```
...  
XMLHttpRequest cannot load http://127.0.0.1/evotrader/strategieservice.php.  
Method JSON is not allowed by Access-Control-Allow-Methods.
```

In manchen Situationen ist dieses Verhalten sehr hinderlich, wenn eine Webseite Daten aus unterschiedlichen Quellen in den Browser integrieren soll. In anderen Szenarien wird eine Webseite nicht von derselben Adresse aufgerufen, auf der die Serverkomponente liegt. Es gibt mehrere Wege, die Same-Origin-Policy zu umgehen:

Im Browser Chrome kann man mit einem Startparameter die Richtlinie abschalten. Das kann für die Entwicklungszeit sehr hilfreich sein. Für Windows würde der Befehl in der Kommandozeile wie folgt aussehen:

```
> chrome.exe --disable-web-security
```

Im normalen Arbeitsalltag oder als Vorgaben für den generellen Einsatz einer Applikation ist dieses Vorgehen nicht akzeptabel, denn neben der Same-Origin-Policy werden ebenfalls weitere Sicherheitsprüfungen abgeschaltet.

### JSONP (JSON mit Padding)

Das Protokoll JSONP geht einen anderen Weg, der außerdem in allen Browsern funktioniert: Die Sicherheitsrichtlinien bleiben grundsätzlich in Kraft. Das Protokoll nutzt die Tatsache aus, dass beim Laden von Script-Elementen in einem HTTP-Dokument die Same-Origin-Policy nicht gefordert ist. Technisch handelt es sich um eine herkömmliche Get-Anfrage gemäß HTTP.

Ein Framework wie AngularJS simuliert mit JSONP intern eine Script-Anfrage an die gewünschte Adresse. Der Browser und die anfragende Applikation merken keinen Unterschied und AngularJS reicht die erhaltenen Daten in gewohnter Form als JSON-Objekt als Ergebnis zurück. Das interne Script-Element, das AngularJS dynamisch für die Kommunikation erstellt, könnte so aussehen:

```
<script type="text/javascript"
src="http://einedomain.de/getjson.php?jsonp=functionsname">
</script>
```

Leider muss man auf der Serverseite etwas mehr tun. Es reicht nicht aus, die Daten als JSON-Objekte zu liefern. Im Falle von JSONP sieht die Nachricht etwas anders aus: Der ursprüngliche JSON-String muss in eine Funktion verpackt werden, da der Browser ein ausführbares Script erwartet. Den Aufruf der Funktion selbst übernimmt AngularJS für uns. Wie AngularJS Kenntnis über den Funktionsnamen erhält, sehen wir weiter unten.

Für die Implementierung des Servers ist wichtig, dass wir das Ergebnis korrekt in eine Funktion verpacken und die Funktion richtig benennen. Typisch ist, dass der Client bei einer JSONP-Anfrage den Namen der Funktion als Parameter mitsendet. AngularJS erledigt dies automatisch für uns im Hintergrund.

Ein Beispiel für eine herkömmliche JSON-Nachricht könnte folgendes Aussehen haben:

```
[ {"id": "1", "name": "Peter Mustermann"},  
 {"id": "2", "name": "Hans Mueller"},  
 {"id": "3", "name": "Lisa Bauer"}]
```

Als JSONP-Nachricht müssen die Daten in eine syntaktisch korrekte JavaScript-Funktion verpackt sein:

```
JSON_CALLBACK([  
    {"id": "1", "name": "Peter Mustermann"},  
    {"id": "2", "name": "Hans Mueller"},  
    {"id": "3", "name": "Lisa Bauer"}  
]);
```

In der URL des Requests wird der Name einer Callback-Funktion mitgesendet. Ein Beispiel für einen JSONP-Request ist im nächsten Block zu sehen:

```
http://127.0.0.1/evotrader/strategieservice.php?callback=angular.callbacks.\_1
```

Den Parameter `callback` ergänzt AngularJS automatisch und nennt dem Server den Namen der erwarteten Funktion, die nach dem Erhalt im Client aufgerufen werden soll. Die letzte Ziffer nach dem Unterstrich ist eine fortlaufende Nummer, mit der AngularJS jede Anfrage eindeutig kennzeichnet. Somit stellen auch parallele Anfragen kein Problem dar.

## Implementierung des JSONP-Clients

```
$scope.readData = function()  
{  
    base_url = http://127.0.0.1/evotrader/strategieservice.php";  
    $scope.requestStarted = true;  
  
    $http( {method: "JSONP",  
            params: { callback: "JSON_CALLBACK" }, url:base_url})  
.success(function(data, status ) {  
    ...  
    DataService.strategyData2( data );  
    $scope.requestStarted = false;  
    $scope.addMessage( "Daten erhalten ...", 'success' );  
})  
.error(function(data, status ) {  
    ...  
    $scope.requestStarted = false;  
    $scope.addMessage( "Fehler ...",'error' );  
});  
}
```

Die Funktion `readData()` fragt die Auswertung der Strategiedaten beim Server an. Dazu nutzt sie den `$http`-Service, konfiguriert eine Anfrage mit dem JSONP-Protokoll und setzt die Methode, die Ziel-URL und zusätzliche Parameter.

In unserem Beispiel sind keine fachlichen Parameter notwendig. Der einzige wichtige Parameter heißt `callback` und enthält den Namen der Callback-Funktion, den AngularJS für das JSONP-Protokoll erwartet und nach dem Erhalt der Antwort aufrufen wird.

Das Beispiel definiert zwei weitere Methoden mit den Namen `success` und `error`, die in Aktion treten, wenn klar ist, ob die Anfrage erfolgreich oder fehlerhaft war. Beide Methoden definieren zwei Parameter mit den erhaltenen Daten und dem Status der Antwort. Wir erzeugen jeweils eine neue Nachricht mit der Funktion `addMessage()`, um den Benutzer über den Status zu informieren.

Im erfolgreichen Fall übergeben wir die gesendeten Daten an den `dataService()`, der diese der View wie oben beschrieben zur Verfügung stellt. Die neue Methode `strategyData2(data)` legt das erhaltene JSON-Objekt in den internen Strukturen ab.

Die Variable `$scope.requestStarted` wird vor der Anfrage auf `true` gesetzt und zeigt an, ob gerade ein Request läuft. Nach der Antwort (unabhängig davon, ob positiv oder negativ) setzen wir die Variable auf `false`. Die Variable steuert, ob der Button für die Anfrage der Strategiedaten aktivierbar ist oder nicht. Somit verhindern wir, dass der Benutzer mehrfach auf den Button klickt und mehrere Anfragen fast gleichzeitig aktiviert.

### Die Implementierung des Servers für JSONP

Manche serverseitigen Frameworks implementieren das JSONP-Protokoll. Mit der Beschreibung von oben können wir das korrekte Verhalten leicht selbst implementieren.

Im Wesentlichen müssen wir den übergebenen Parameter `callback` auswerten und das JSON-Objekt in eine syntaktisch korrekte JavaScript-Funktion verpacken. Der Name der Funktion ergibt sich direkt aus dem Inhalt des Parameters.

Es ist sinnvoll zu prüfen, ob der Parameter `callback` existiert. Sollte dieser fehlen, könnten wir als Fallback eine herkömmliche JSON-Antwort senden und so eine flexible Schnittstelle auf dem Server anbieten. Das folgende Beispiel deutet eine einfache Implementierung an:

```
<?php
header('Access-Control-Allow-Origin: *');

// Ist der Parameter callback vorhanden?
if ( isset( $_REQUEST['callback']) )
{
    // Ausgeben der JSONP-Callback-Funktion:
    echo $_REQUEST['callback'];
}
```

```
}

//Ausgeben der fachlichen Daten im JSON-Format:
echo '(
  [ { "lastdays" :
    [[5773,"11.09.2013",845332,849573,0,72992456,72992456,2678,2678,9,9,6502427]
    ,
    ...
  } ] )';

?>
```

## Lokalisierung und Internationalisierung

Das Internet macht unsere Welt zum Dorf. Und damit werden Lokalisierung und Internationalisierung für die meisten Applikationen zu wichtigen Themen. Ideal ist es, einen Webdienst für jeden Benutzer in seiner gewünschten Sprache zu präsentieren. Neben dem eigentlichen Inhalt sind für eine Webapplikation meist folgende Elemente von der Lokalisierung betroffen:

- Formatierung von Datumswerten.
- Formatierung von Beträgen inklusive Währungen.
- Anzeigen von Übersetzungen für alle Arten von Beschriftungen und Meldungen.
- Die Leserichtung von Texten ist nicht immer gleich. Im arabischen Raum wird von rechts nach links gelesen.

Das AngularJS-Team bietet Module an, die manche dieser Themen adressieren. Diese Pakete enthalten:

- Die Formate für Datumswerte in verschiedenen Längen und Ausprägungen.
- Eine Liste mit den Namen und Abkürzungen der Wochentage und Monate.
- Die Definition der Zeichen für die Trennung von Dezimalstellen und Tausendergruppen.

Zu finden ist die Datei auf der Projektseite von AngularJS für die Version 1.2.4 unter: [code.angularjs.org/1.2.4/i18n](http://code.angularjs.org/1.2.4/i18n)

Sucht man für eine explizite AngularJS-Version diese Datei, kann man über [code.angularjs.org](http://code.angularjs.org) einsteigen. Dort sind alle Versionen aufgelistet. Die Datei in der gewünschten Sprache erkennt man am Dateinamen, der das Locale und die Sprache enthält. Das Einbinden der JavaScript-Datei erfolgt nach der AngularJS-Bibliothek mit:

```
<script src="angular.js"></script>
<script src="i18n/angular-locale_de-ge.js"></script>
```

Wie formatiert man einen Betrag in der jeweiligen sprachabhängigen Formatierung? Die Formatierung ist mit AngularJS-Filter realisiert.

So gibt es einen Currency-Filter, der das eingestellte Locale und damit die Formatvorlage kennt:

```
 {{ 1000 | currency }}
```

Der Filter liefert mit dem Locale `en-us` die Ausgabe: `$1000.00` und mit dem Locale `de-DE` das Ergebnis: `Euro 1000,00`.

Für das Anzeigen von unterschiedlichen Übersetzungen von Beschriftungen gibt es in AngularJS keinen eingebauten Mechanismus. Mithilfe der Information aus dem Locale kann man sich diese Funktion selbst bauen. In der ständig wachsenden Liste von zusätzlichen Modulen und Direktiven für AngularJS wird man aber fündig.

Außerdem gibt es einen Eintrag zum Thema Internationalisierung in der offiziellen Dokumentation von AngularJS: [docs.angularjs.org/guide/i18n](https://docs.angularjs.org/guide/i18n).

### 3.5.6 Fazit zum zweiten AngularJS-Beispiel

Das zweite Beispiel geht über die Basiskonzepte von Single-Page-Web-Applikationen hinaus. Zum einen haben wir mächtige Frameworks integriert und zum anderen eine übersichtliche Struktur in der Applikation erhalten.

Google Charts ist ein schönes Beispiel für sehr ausgereifte und produktiv nutzbare Frameworks, die am Markt verfügbar sind und in der Applikationsentwicklung viel Arbeit abnehmen.

Gerade mit der konsequenten Implementierung projektbezogener Direktiven mit AngularJS eröffnen sich neue Möglichkeiten der Zusammenarbeit zwischen Webdesign und Entwicklung.

## 3.6 Spiele mit AngularJS entwickeln

Das Konzept der Single-Page-Web-Applikation taugt nicht nur für Geschäftsapplikationen, sondern es können auch Spiele damit umgesetzt werden. Spiele sind ein weites Feld und wir müssen den Fokus genauer definieren. Für sehr grafiklastige Spiele, die an die Grenzen der Leistungsfähigkeit aktueller Rechner gehen, sind JavaScript und damit AngularJS sicher nicht die besten Werkzeuge. Beispiele für dieses Genre sind 3-D-Spiele wie zum Beispiel First-Person-Shooter. Daneben gibt es einen großen Bereich von Spielen, die diese extremen Anforderungen nicht stellen und gut in Form von Web-Applikationen angeboten werden können.

### Das fertige Sudoku-Spiel ausprobieren

Das vollständige Spiel, das in diesem Kapitel erstellt wird, ist online auf der Seite von HirnSport.de verfügbar. Die Adresse lautet: [www.HirnSport.de/sudoku](http://www.HirnSport.de/sudoku)

Dieses Kapitel zeigt auch, dass AngularJS für andere Markup-Sprachen neben HTML geeignet ist. Wir nutzen für die Realisierung der View-Komponente das SVG-Format (Scalable Vector Graphics) und binden wie bisher unser Modell an.

### 3.6.1 SVG-Grundlagen

SVG ist ein Format für zweidimensionale Vektorgrafiken. Das Format wurde 2001 vom World Wide Web Consortium (W3C) als Spezifikation aus mehreren konkurrierenden Formaten zusammengeführt und vorgestellt. Ein SVG-Dokument ist in XML formuliert und kann damit formal mit einem Texteditor bearbeitet werden. Wegen der Verwendung von vielen Attributen und verschachtelten XML-Tags empfiehlt sich der Einsatz eines speziellen XML-Editors. SVG-Dateien nutzen die Dateiendung `.svg` oder `.svgz` für komprimierte Dateien.

Zurzeit ist die Version 1.1 verbreitet und wird am häufigsten eingesetzt. Die nächste Version mit der Kennzeichnung 1.2 befindet sich aktuell in der Spezifikationsphase.

Neben der Version gibt es sogenannte Profile. Ein Profil beschreibt den Funktionsumfang, den eine Implementierung von SVG bereitstellen muss. So haben die Hersteller von mobilen Browsern die Möglichkeit, einen reduzierten, aber fest definierten Funktionsumfang anzubieten. Profile werden kaum genutzt und fast alle aktuellen Browser geben an, das Profil »full« zu implementieren, obwohl trotzdem manche Funktionen fehlen.

SVG bietet eine große Fülle von Funktionen an und geht über die einfachen grafischen Elemente weit hinaus:

- Animationen können durch die »Synchronized Multimedia Integration Language« (SMIL) abgebildet werden. Dazu gehört ebenfalls die Umsetzung von einfachen Interaktionen. Zum Beispiel könnte ein Klick auf ein Element eine Animation starten, deren Ende eine weitere Animation aktiviert.
- Ein SVG-Dokument kann durch eine Skriptsprache (wie JavaScript) dynamisch verändert werden.

#### Ein erstes einfaches SVG-Beispiel

Das folgende Listing zeigt ein einfaches SVG-Beispiel. Das einleitende SVG-Element enthält die Basisangaben wie Version, Profil, Abmessungen des Zeichenbereichs und den XML-Namensraum für SVG.

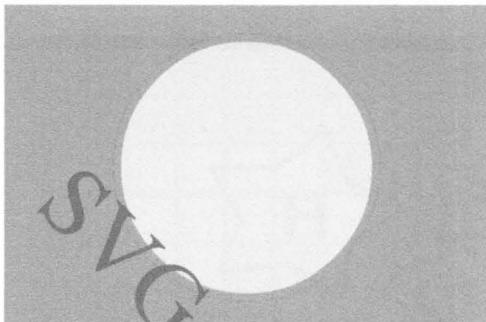
Danach füllt das Dokument den Zeichenbereich mit einem roten Rechteck und einem grünen Kreis. Der G-Tag gruppiert Elemente und wendet eine Transformation an. Alle eingebetteten Elemente werden ebenfalls der Drehung unterworfen. Im Beispiel betrifft dies nur ein Textelement:

```
<svg version="1.1"
      baseProfile="full"
      width="300" height="200"
```

```
xmlns="http://www.w3.org/2000/svg">

<rect width="100%" height="100%" fill="red" />
<circle cx="150" cy="100" r="80" style="stroke-width: 5; stroke: #009900;
fill: #00ff00;" />
<g transform="rotate(45 25 25)" >
<text x="150" y="100" font-size="60" text-anchor="middle"
fill="blue">SVG</text>
</g>
</svg>
```

Das Beispiel definiert das Aussehen der Elemente über das Style-Attribut und setzt zum Beispiel Linienbreiten (*stroke-width*), Linienfarben (*stroke*) und Füllfarben (*fill*). Bei den Möglichkeiten, das Aussehen zu definieren und komplexe grafische Elemente exakt zu positionieren, kann SVG seine Stärken ausspielen und hat mehr zu bieten als CSS und HTML. Die folgende Abbildung zeigt das Ergebnis:



**Bild 3.25:** Das Ergebnis des SVG-Beispiels im Browser.

Ein umfassender Überblick zu den Möglichkeiten von SVG würde das Kapitel bei Weitem übersteigen. Es existieren viele gute Einführungen im Internet oder auf dem Buchmarkt.

## Verbreitung von SVG

Auch wenn SVG schon seit 2001 als Standard verabschiedet ist und recht schnell danach die ersten Viewer und Browser in der Lage waren, SVG zu rendern, hat es sich im Internet bisher nur moderat verbreitet. Das hat sich erst in den letzten Monaten im Zuge von HTML5 geändert. Auch wenn SVG selbst kein Bestandteil von HTML5 ist, so implementieren es aktuell fast alle verfügbaren Browser. Selbst auf mobilen Geräten sind die Basisfunktionen von SVG nutzbar. Lediglich spezielle Funktionen wie interaktive Elemente, Animationen oder SVG-Filter fehlen auf manchen mobilen Systemen. Eine detaillierte und aktuelle Übersicht findet sich auf der Website »Can I use«: [caniuse.com/#cats=SVG](http://caniuse.com/#cats=SVG).

## Werkzeuge für das Erstellen von SVG

Fast alle kommerziellen Programme für das Erstellen von Vektorgrafiken bieten zumindest einen Export in das SVG-Format an. Besonders interessant ist das kostenlose Programm Inkscape. Die Software ist unter [inkscape.org](http://inkscape.org) für die gängigen Desktop-Betriebssysteme verfügbar.

Inkscape benutzt SVG als präferiertes Datenformat und bietet einen großen Funktionsumfang, der in vielen Punkten den kommerziellen Produkten nahekommt. Besonders beeindruckende Funktionen von Inkscape sind:

- Das Vektorisieren von Rastergrafiken.
- Die große Auswahl von Filtern, die eine Grafik in Neonoptik, als Flüssigkeit oder mit einem Leopardenfell darstellen.
- Das gitterweise Klonen von Objekten.
- Das Ausrichten von Texten und Objekten an beliebigen Pfaden.

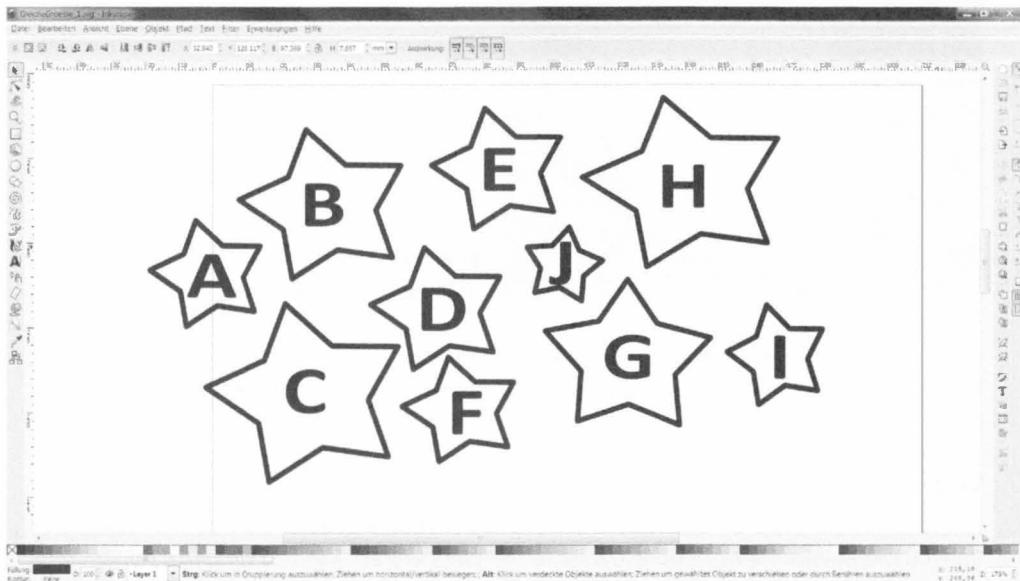


Bild 3.26: Screenshot von Inkscape.

Mehr als 26000 kostenlose Beispielgrafiken in SVG finden sich auf der Website des Open Clip Art Projects unter: [openclipart.org](http://openclipart.org).

### 3.6.2 Eine Sudoku-App mit AngularJS und SVG

Als Beispiel für ein Spiel realisieren wir eine Sudoku-Applikation. Wir konzentrieren uns auf die Darstellung der Aufgaben und den Ablauf des Spiels mit den Benutzeraktionen. Die Aufgaben und die Lösung werden bereitgestellt. Das Generieren und Lösen der Aufgaben selbst ist kein Bestandteil der Implementierung.

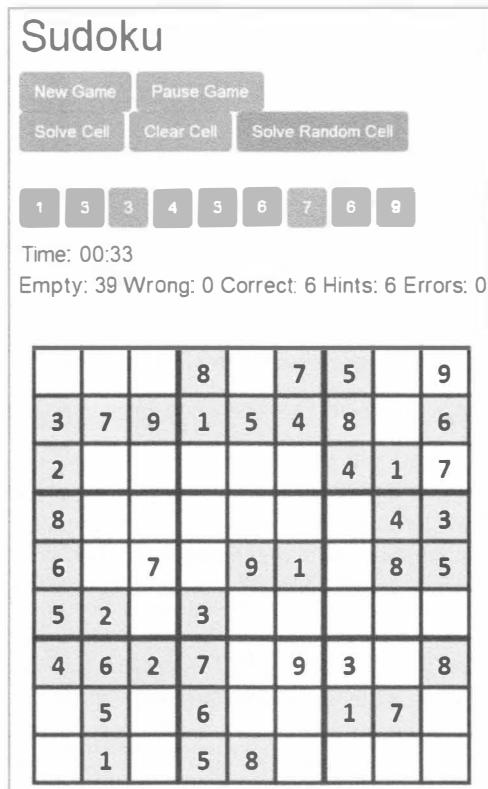


Bild 3.27: Die Sudoku-App in Aktion.

#### Die Anforderungen an das Spiel

Die Aufgaben hinterlegen wir als Zeichenketten, getrennt nach Aufgaben und Lösung – jeweils mit der Länge von 81 Zeichen. In der Aufgabe sind die vom Spieler zu füllenden Felder durch Leerzeichen entfernt. Die Zeichenkette mit der Lösung enthält alle korrekten Ziffern.

Die Applikation liest die Aufgaben ein, stellt diese dar und steuert den Ablauf des Spiels. Während der Spieler das Feld ausfüllt, läuft zusätzlich eine Uhr. Sie misst die Zeit, die der Spieler braucht. Aus dieser Dauer und dem Schwierigkeitsgrad der Aufgaben könnte man für den Spieler eine Bewertung in Form von Punkten errechnen.

Vorbelegte Felder sind grau hinterlegt und können nicht verändert werden. Zur Unterstützung des Spielers markiert die Applikation falsche Eingaben mit einem roten

Hintergrund. Falls eine Stelle zu knifflig ist und man nicht weiterkommt, kann die Applikation einen Tipp anzeigen und entweder das markierte Feld oder ein zufälliges Feld aufdecken.

### Steuerung im Spiel

Die Bedienung soll möglichst einfach sein. Der Benutzer soll gleichermaßen die Maus oder die Tastatur nutzen können. Das aktuell gewählte Feld wird farbig markiert, indem wir es etwas dunkler und ohne Markierung zeichnen.

Wir legen folgende Tasten für die Steuerung fest:

- Die Pfeiltasten navigieren über das Feld.
- Die Zifferntasten von 1 bis 9 tragen eine Ziffer in die ausgewählte Zelle des Felds ein.
- Die Taste S deckt das markierte Feld auf.
- Die Taste R deckt ein zufälliges, noch leeres Feld auf.
- Die Taste C leert das aktuelle Feld, indem eine eingegebene Ziffer entfernt wird.
- Die Taste P pausiert das Spiel und stoppt die Zeit. Die Leertaste lässt die Zeit weiterlaufen.

Die Steuerung ist zusätzlich auch mit der Maus möglich. Standardmäßig markiert der Benutzer ein Feld mit einem Klick und füllt es durch einen Klick auf den entsprechenden Ziffernbutton am oberen Rand der Applikation. Um die Eingabe auch für die Mausnutzung zu verbessern, blenden wir in der angeklickten Zelle eine Miniziffernauswahl ein.

Hat der Spieler schließlich alle Felder vollständig und korrekt gefüllt, erscheint ein Pop-up und er kann eine zufällige neue Aufgabe lösen. Eine Reihe von Aufgaben ist fest im Service hinterlegt, aus denen zufällig eine Aufgabe ausgewählt wird. Mit den Techniken zum Datentransport aus den anderen Kapiteln könnte man sich leicht Aufgaben von einem Server liefern lassen.

#### Sudoku-Regeln in Kurzform

Sudoku ist neben Kreuzworträtsel wohl eine der am meisten verbreiteten Rätselarten. Zur Vollständigkeit sollen die Regeln zumindest kurz an dieser Stelle erklärt werden:

Auf einem Sudoku-Feld befinden sich neun mal neun Zellen, die in Reihen, Spalten und neun Bereiche aus jeweils drei mal drei Zellen unterteilt sind. In jeder Reihe, Spalte und in jedem der Bereiche darf jede Ziffer von 1 bis 9 nur genau einmal vorkommen.

Zu Beginn sind auf dem Feld nur einige wenige Zellen mit den richtigen Ziffern belegt. Aus diesen lassen sich die leeren Zellen logisch erschließen.

		8				7		
	3	9				4	6	
7	5		9		4		3	8
		5		4		2		
		3			5			
	1		6		8			
4	6		8		1		9	7
	1	3				6	8	
		7			5			

Bild 3.28: Eine Sudoku-Aufgabe.

### 3.6.3 Die Umsetzung der Sudoku-App

#### Die Darstellung

Die Datei `index.html` enthält die Definition der View. Einige Besonderheiten haben sich dennoch eingeschlichen, zum Beispiel am äußersten div-Element der View:

```
<body>
<div tabindex="0" ng-controller="FieldController"
    style="width:450px;height:450px;background-
image:url(background.png);background-repeat:no-repeat;">
    ng-keydown="keypress($event)">
    ...

```

Das Attribut `tabindex` legt die Reihenfolge fest, nach der Eingabeelemente den Fokus für Tastatureingaben erhalten. Mit der Tab-Taste springt man jeweils zum nächsten UI-Element. Mit dem Wert 0 bekommt das übergeordnete DIV-Element den Fokus als Erstes und lauscht sofort auf die Tastaturnachrichten.

Mit dem Attribut `ng-keydown` definieren wir eine JavaScript-Methode für die Reaktion auf Tastaturnachrichten. Die Steuerung von Webseiten über die Tastatur gestaltet sich grundsätzlich etwas schwierig, da der Browser bestimmte Tasten für sich reserviert. Die Taste F1 öffnet typischerweise die Hilfeseite des Browsers. Und wenn der Benutzer alle fokussierbaren Felder mit der Tabulator-Taste durchblättert, landet er erfahrungsgemäß auch in den Steuerelementen des Browsers, wie der Zeile für die URL-Eingabe.

Als Nächstes erstellt der HTML-Code den Kopfbereich der Seite und einige Schaltflächen für die Steuerung des Spiels.

Außerdem bindet er die Buttons an JavaScript-Funktionen:

```
...
<button class="btn btn-danger"
  ng-click="import()">New Game</button>
<button class="btn btn-danger"
  ng-click="setState(1); ">Pause Game</button>
<br/>
<button class="btn btn-primary"
  ng-click="solveField()">Solve Field</button>
<button class="btn btn-primary"
  ng-click="clearCurrentField()">Clear Field</button>
<button class="btn btn-primary"
  ng-click="solveRandom()">Solve Random Field</button>

<br/>
<button ng-click="selectNumber(1)"
  class="btn btn-primary">1</button>
<button ng-click="selectNumber(2)"
  class="btn btn-primary">2</button>
...
<button ng-click="selectNumber(9)"
  class="btn btn-primary">9</button>
<br/>
<span style="..." >Time: {{counter | seconds2Time }}<br/>
```

Free:{{result.empty}} Wrong:{{result.wrong}} Correct:{{result.correct}} Hints:{{hints}} Errors:{{errors}}

Spannender ist der nächste Abschnitt: In die HTML-Seite ist zusätzlich ein Block mit SVG-Code eingebettet. Alternativ könnte der SVG-Abschnitt in eine separate Datei ausgelagert und referenziert sein.

```
<svg tabindex="0" width="500" height="450"
  xmlns="http://www.w3.org/2000/svg"
  style='fill-opacity:1; stroke:black; fill:black; font-weight:normal;
stroke-width:1; font-family:"Dialog"; font-style:normal; font-size:12; ' >

<g transform="matrix(1.0,0,0,1.0,12,0)">
<g id='Board'>
<g ng-repeat="row in aufgabe" ng-cloak>
<g ng-repeat="cell in row" ng-cloak>

<rect
  ng-attr-x="{{getx( $index, $parent.$index )}}"
  width="40" height="40"
```

```

    ng-attr-y="{{gety( $parent.$index )}}"
    ng-attr-style="{{getCellColor( $index,$parent.$index )}}"
  </rect>
  <text
    ng-attr-x="{{getx( $index, $parent.$index )+14}}"
    ng-attr-y="{{gety( $parent.$index )+29}}"
    style="stroke:none;font-size:24;">
{{(cell.a==0)?':cell.a;}}</text>
  <rect
    ng-attr-x="{{getx( $index, $parent.$index )}}"
    width="40" height="40"
    ng-attr-y="{{gety( $parent.$index )}}"
    style="opacity:1;fill-opacity:0;"
    ng-click="fieldClick($index,$parent.$index)" />
  </g>
</g>
</g>

<g id='DigitSelector' display='{{getLayerVisible(3)}}'
  transform="translate( {{getx( x, y )-1}} , {{gety( y )-1}} ) rotate(0)" >
  ...
</g>

<g id='PopupFinish' display='{{getLayerVisible(2)}}'
  transform="translate( {{f.x}} , {{f.y}} ) rotate(0)" >
  ...
</g>

<g id='PopupPause' display='{{getLayerVisible(1)}}'
  transform="translate( {{f.x}} , {{f.y}} ) rotate(0)" >
  ...
</g>
</g>
</svg>
```

Die SVG-Definition beginnt mit dem Tag `svg` und definiert die Ausdehnung von Grafik und Namespace für SVG analog zum Inhalt einer separaten SVG-Datei. Hinzu kommen Voreinstellungen für die Darstellungen im Attribut `style`. Falls ein Element keine speziellen Angaben enthält, kommen die Voreinstellungen zum Einsatz.

Als grafisches Format bietet SVG eine große Auswahl an Einstellungen für diverse Elemente an. Das reicht von Farben und Farbverläufen über Linienarten und Schriftgröße bis zu Schriftart und Layout von Elementen. Die folgende Tabelle stellt nur eine kleine Auswahl zusammen, die im aktuellen Beispiel vorkommt:

Genutzte SVG-Attribute und die Vorbelegung	Bedeutung der Attribute
fill-opacity:1	Definiert die Deckkraft der Füllung. Der Wert 1 bedeutet, dass die Farbe zu 100 % deckend ist. Alle anderen Werte lassen einen bestimmten Anteil des Hintergrunds durchscheinen.
stroke-width:1	Definiert die Breite von Rändern und Linien.
stroke:black	Setzt die Rand- und Linienfarbe auf Schwarz.
fill:black	Schwarz als Default für die Füllfarbe setzen.
font-family:"Dialog"	Wählt die Schriftart mit dem Namen »Dialog« aus.
font-style:normal	Schrift auf »normal« setzen. Andere mögliche Werte sind »bold« oder »italic«.
font-size:12	Die Schriftgröße auf 12 Punkte voreinstellen.

Das eingebettete SVG-Dokument ist in vier Bereiche aufgeteilt, die unterschiedliche Aufgaben erfüllen. Die Identifizierer dienen nur zur Dokumentation.

Wenn man mit JavaScript ein SVG-Element verändern möchte, stehen dieselben Möglichkeiten wie bei herkömmlichen HTML-Elementen zur Verfügung. Wir nutzen in diesem Fall ebenfalls die Bindung von Attributen mithilfe von AngularJS.

Die folgende Tabelle listet die vier Bereiche und ihre Bedeutung auf:

Beschreibung des Bereichs	Kennzeichnung im SVG-Quelltext
Stellt das Spielfeld mit den neun mal neun Zellen dar.	<code>id='Board'</code>
Der dynamische Bereich für die verkleinerte Zahlenwahl in der Zelle.	<code>id='DigitSelector'</code>
Das Pop-up für die Pause.	<code>id='PopupPause'</code>
Das Pop-up für Spielende, wenn alle Zellen korrekt ausgefüllt sind.	<code>id='PopupFinish'</code>

In der Sudoku-Applikation steuern wir die Sichtbarkeit der Elemente, indem das SVG-Attribut `visibility` durch AngularJS an eine Funktion `getLayerVisible()` gebunden

wird und den Wert dynamisch ändert. Die Funktion liefert je nach Zustand den passenden Wert:

- Der Wert `inline` macht die Gruppe sichtbar.
- Der Wert `none` versteckt die Gruppe.

Die Funktion selbst wird weiter unten genauer beschrieben.

Die Sichtbarkeit bezieht sich auf alle Elemente innerhalb der Gruppe, die mit dem G-Tag definiert sind. Die SVG-Gruppen können mit den DIV-Bereichen von HTML verglichen werden. Allerdings definiert SVG die Hierarchie der Elemente etwas anders. Es gibt keine Z-Reihenfolge, die beschreibt, wie die Elemente übereinander liegen. SVG löst die Aufgaben einfacher. Die Elemente stellen sich auf der Zeichenfläche in der Reihenfolge dar, in der sie im Dokument definiert sind. Einfacher ausgedrückt: Das zuletzt definierte Element liegt in der Darstellung ganz oben.

### Aufbau des Spielfeldes

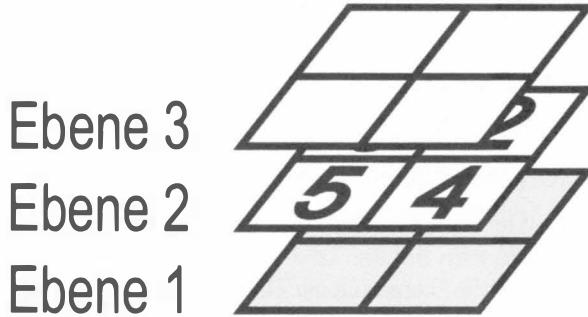
Von den vier Bereichen ist das Spiel (mit dem Identifier `Board`) am interessantesten, da es zum größten Teil dynamisch erstellt wird. Als Alternative könnte man die 81 Zellen eines Sudoku-Feldes komplett vordefinieren. Da wir aber an jede Zelle Funktionen für die dynamische Wahl der Hintergrundfarbe und der dargestellten Ziffer anhängen müssen, ist es sehr viel einfacher, SVG-Elemente über `ng-repeat`-Direktiven zu kreieren.

Jede Zelle des Feldes besteht aus drei übereinander gestapelten SVG-Elementen:

- Ein Rechteck, das die Hintergrundfarbe bestimmt.
- Ein Textelement, das die Ziffer in der Zelle darstellt
- Ein weiteres, transparentes Rechteck über dem Textelement, mit dem die Mausaktionen verbunden sind.

Für die korrekte Darstellung würden die Elemente eins und zwei ausreichen. Das letzte Rechteck erfüllt zwei Aufgaben:

- Es bewirkt, dass wir die `ng-click`-Funktion nur an das oberste Element anhängen müssen und nicht sowohl an das Hintergrundelement als auch zusätzlich an das Textelement.
- Der zweite Grund ist rein ästhetischer Natur: Das transparente Rechteck überdeckt das Textelement, ohne es zu verstecken und verhindert, dass sich der Mauszeiger über dem Schriftelement in einen Textcursor verwandelt.



**Bild 3.29:** Skizze des schematischen Aufbaus der Zellen im Spielfeld.

### Interne Repräsentation des Spielfeldes

Die Aufgaben liefert der Aufgabenservice in der Form von zwei Zeichenketten mit einer Länge von 81 Stellen. Einmal als Lösung, in der Ziffern für alle Zellen enthalten sind und einmal als Aufgaben, in der einige Felder leer sind, die der Spieler ausfüllen muss. Die Applikation holt mit der Funktion `import()` diese beiden Zeichenketten und überträgt diese in eine interne Darstellung des Spielfeldes. Ergebnis ist ein Array von 81 Zellen mit jeweils neun Zeilen und neun Spalten. Jede Zelle enthält folgende Attribute:

- **s:** Definiert den Zustand:  
Der Wert 1 bedeutet, dass die Zelle korrekt vorbelegt ist und nicht ausgefüllt werden muss. Für den Benutzer zeigen wir diese Zellen grau hinterlegt an. Der Wert 0 bedeutet, dass die Zelle in der Aufgabe leer ist und vom Spieler gefüllt werden muss.
- **a:** Dieses Attribut nimmt die Eingabe auf und ist zum Start der Aufgaben leer.
- **l:** Ist die korrekte Ziffer, die laut Lösung in diese Zelle gehört. Damit kann man entweder die Eingabe des Nutzers prüfen, oder das Feld vorausfüllen, abhängig vom Zustand des Attributes **s**.

Das Codebeispiel zeigt die interne Darstellung des Spielfeldes an einem Ausschnitt mit genau einer Zeile aus neun Zellen:

```
var aufgabe =
[
  [ { a:0, l:1, s:0 }, { a:0, l:8, s:0 }, { a:1, l:1, s:0 },
    { a:1, l:1, s:0 }, { a:1, l:1, s:0 }, { a:1, l:1, s:0 },
    { a:1, l:1, s:0 }, { a:1, l:1, s:0 }, { a:1, l:1, s:0 } ],
  ... // weitere Zeilen folgen
];
```

Die Funktion `import()` legt dieses Feld im Scope in der Variablen `aufgabe` ab.

## Zeichen des Spielfeldes

Jetzt haben wir alle Informationen zusammen, um das Spielfeld zu erstellen.

```
<g id='Board'>
  <g ng-repeat="row in aufgabe" ng-cloak>
    <g ng-repeat="cell in row" ng-cloak>

      <rect ng-attr-x="{{getx( $index, $parent.$index )}}"
        width="40" height="40"
        ng-attr-y="{{gety( $parent.$index )}}"
        ng-attr-style="{{getCellColor( $index,$parent.$index)}}">
      </rect>
      <text ng-attr-x="{{getx( $index, $parent.$index )+14}}"
        ng-attr-y="{{gety( $parent.$index )+29}}"
        style="stroke:none;font-size:24;">
        {{(cell.a==0)?':':cell.a;}}
      </text>

      <rect ng-attr-x="{{getx( $index, $parent.$index )}}"
        width="40" height="40"
        ng-attr-y="{{gety( $parent.$index )}}"
        style="opacity:1;fill-opacity:0;"
        ng-click="fieldClick($index,$parent.$index)">
      />
    </g>
  </g>
</g>
```

Um das zweidimensionale Feld zu durchlaufen, sind zwei Repeat-Direktiven zu schachteln: Mit der ersten Repeat-Direktive laufen wir über die Zeilen des Feldes in der Variablen `aufgabe`. Die zweite Repeat-Direktive iteriert über alle Spalten der gewählten Zeile. Innerhalb der zweiten Schleife können wir über die lokale Variable `cell` auf jede Zelle der Datenstruktur einer Aufgabe zugreifen und einen Wert ausgeben: `cell.a`

Wir müssen beachten, dass die leeren Zellen intern mit dem Wert 0 belegt sind, der so nicht zur Anzeige kommen darf. Aus diesem Grund tragen wir mit einer Fallunterscheidung in diesem Fall nicht die Ziffer 0, sondern ein Leerzeichen ein:

```
  {{(cell.a==0)?':':cell.a;}}
```

An dieser Stelle ist ein Einwand angebracht: Sollte diese Logik nicht in eine JavaScript-Funktion ausgelagert werden? Diese Anmerkung ist berechtigt und sobald die Logik etwas komplizierter wird, sollte das der bevorzugte Weg sein.

Noch ein Tipp am Rande: Eine andere Lösungsalternative könnte ein selbst geschriebener AngularJS-Filter sein, der diese Logik kapselt. Damit können wir dasselbe Resultat erzielen.

Für einige Stellen, an denen Berechnungen von Koordinaten notwendig sind, wurden Funktionen definiert: Die Werte für die X- und Y-Koordinaten für die Positionen der Rechtecke und Textelemente beziehen wir über die Funktionen `getx()` und `gety()` aus dem Kontroller. Die Farbe für die Füllung des Hintergrundes der Zellen liefert die Funktion: `getCellColor()`.

Bei beiden Funktionen muss nicht nur der `$index` aus dem Scope der aktuellen Schleife, sondern zusätzlich der Index der umschließenden Schleife übergeben werden. Aus dem aktuellen Scope können wir mithilfe von `$parent.index` leicht auf den übergeordneten Scope zugreifen. Das funktioniert sogar über mehrere Ebenen hinweg.

An das oberste Rechteck jeder Zelle im Spielfeld hängen wir mit der `ng-click`-Direktive eine Funktion, die auf einen Mausklick reagiert. Damit ist unser Spielfeld definiert und alle Elemente sind korrekt positioniert und vorbelegt.

### Attributbindung mit Hindernissen (`ng-attr`)

Im Beispiel oben wurden manche Attribute im SVG-Dokument etwas anders an Attribute oder Funktionen gebunden: Anstatt `ng-x` wie bisher haben wir `ng-attr-x` genutzt. Das ist notwendig, da manche Browser bestimmte Attribute vorzeitig beim Einlesen interpretieren. Die Situation führt zu einer Fehlermeldung in der JavaScript-Konsole:

```
> Error: Invalid value for attribute ng-y="{{...}}"
```

Mit der Abwandlung `ng-attr` kann AngularJS die Fehlermeldung umgehen und ersetzt beim Einlesen das Attribut in eine verträgliche Schreibweise für den Browser. Das Binding funktioniert weiterhin wie erwartet. Es existiert leider keine vollständige Liste der Attribute, die diese Sonderbehandlung erfordern. Diese Sonderbehandlung gilt zum Beispiel für viele SVG-Attribute, aber ebenfalls für das `src`-Attribut im IMG-Tag.

Man ist leider gezwungen, hin und wieder einen Blick in die Konsole zu werfen. Die Fehlermeldung dokumentiert den Namen des Attributes. Mehr Informationen sind in der AngularJS-Dokumentation zu finden: [docs.angularjs.org/guide/directive](http://docs.angularjs.org/guide/directive).

### Die Logik im FieldController

Die Steuerung des Spiels findet sich vollständig im FieldController. Dieser verwaltet die Logik und den aktuellen Zustand des Spiels und des Feldes in mehreren Variablen. Die wichtigste ist die Variable `$scope.state` mit folgenden Werten:

- ➊ Wert 0: Spiel läuft gerade.
- ➋ Wert 1: Das Spiel ist angehalten und befindet sich im Pausenmodus.
- ➌ Wert 2: Alle Felder sind korrekt ausgefüllt und das Spiel ist beendet.

Auf diese Variable werden wir an mehreren Stellen im Quellcode Bezug nehmen. Das folgende Diagramm dokumentiert die Zustände und Übergänge, in denen sich das Spiel befinden kann:

Die wichtigsten Variablen stellen wir in der folgenden Tabelle zusammen:

Name der Variablen	Beschreibung und Aufgabe der Variablen
\$scope.x	Definieren die aktuelle Zelle des Spielfeldes, die der Spieler per Maus oder Tastatur fokussiert hat. Die Zelle ist optisch hervorgehoben.
\$scope.y	
\$scope.aufgabe	Diese Variable enthält interne Repräsentationen der aktuellen Aufgaben mit allen Zellen, Benutzereingaben und den Lösungsziffern für jede Zelle.
\$scope.state	Speichert den aktuellen Zustand des Spiels, wie weiter oben beschrieben.
\$scope.counter	Enthält die abgelaufene Zeit seit Start der Aufgaben in Sekunden.
Mytimeout	Speichert den Identifier des aktuell laufenden Timers. Dieser Wert ist notwendig, wenn wir den Timer vorzeitig anhalten wollen.
\$scope.hints	Dieser Wert speichert die Anzahl der Hilfestellungen (Hints) im aktuellen Spiel. Hilfestellungen sind das automatische Aufdecken von Zellen.
\$scope.errors	Analog speichert diese Variable die Anzahl der Fehleingaben, die über das gesamte Spiel erfolgten.
\$scope.result	Das Result wird in der Funktion calc() bestimmt. In der Struktur liegen die aktuellen Werte für leere Zellen, falsch und korrekt gefüllte Zellen.  var result = { empty: 0, wrong: 0, correct: 0 };
\$scope.nrfieldshow	Enthält den Zustand, ob das eingebettete kleine Ziffernfeld innerhalb einer Zelle gerade sichtbar ist.

Name der Variablen	Beschreibung und Aufgabe der Variablen
\$scope.f	<p>Der Parameter hält die Ausrichtung des Spielfeldes innerhalb des SVG-Dokuments fest, ausgehend von der linken oberen Ecke. Die Werte liegen als Koordinaten der Form X, Y vor. Die beiden Attribute Breite und Höhe definieren die Abmessung einer Zelle.</p> <pre>\$scope.f = {     x : 50, y : 40, w: 40, h :40 };</pre>

Bei der Definition des Spielfeldes weiter oben wurde die Funktion `getCellColor()` verwendet. Im Kontroller füllen wir die Funktion jetzt mit Leben:

```
$scope.colorIndex = [
  [ "fill:white;", "fill:lightgray;", "fill: #ff9999;" ],
  [ "fill:lightgreen;", "fill:gray;", "fill:red;" ] ];

$scope.getCellColor = function( x, y )
{
  var row = 0;
  var col = 0;
  var c = $scope.aufgabe[y][x];
  if (( x === $scope.x ) && (y === $scope.y ))
  {
    row = 1;
  }
  if ( c.s !== 0 )
  {
    col = 1;
  }
  if ( (c.a !== 0) && (c.a !== c.l))
  {
    col = 2;
  }
  return $scope.colorIndex[row][col];
};
```

Bei der Steuerung der Farben für eine Zelle sind folgende Informationen über den Zustand der Zelle wichtig:

- Ist die Zelle vorausgefüllt als Teil der Aufgaben?
- Ist die Zelle vom Benutzer korrekt oder falsch gefüllt?
- Ist die Zelle gerade markiert und hat den Fokus für Tastatureingaben?

Die Funktion `getCellColor()` prüft diese Situation. Die Farben sind in einem Feld abgelegt, die erste Zeile enthält die Farben für eine nicht markierte Zelle. Die zweite Spalte enthält durchweg etwas dunklere Farben für markierte Zellen.

Die Variable `$scope.state`, die den aktuellen Zustand des Spiels hält, wurde schon erwähnt. Zum Verändern dieses Zustandes spendieren wir eine eigene Funktion: `$scope.setState()`.

```
$scope.setState = function( newState )
{
    // Spiel starten und Timer starten oder fortsetzen.
    if ( newState === 0 )
    {
        mytimeout = $timeout($scope.onTimeout,1000);
    }
    // Spiel pausieren oder beenden.
    if ( ( newState === 1 ) || (newState === 2) )
    {
        $scope.stop();
        $scope.nrfieldshow = false;
    }
    $scope.state = newState;
}
```

Wir übergeben der Funktion im Parameter `newState` den neuen Zustand, in den die Applikation wechseln soll. Bei den Zustandsübergängen müssen wir beachten, dass eventuell zusätzliche Aktionen notwendig sind. Zum Beispiel muss der Timer gestartet oder angehalten werden.

### Ein- und Ausblenden von SVG-Elementen

SVG-Elemente lassen sich durch das `Display`-Attribut ein- und ausblenden. Der Wert wird über die Funktion `$scope.getLayerVisible()` gesteuert. Als Parameter übergeben wir eine Nummer, die das Element repräsentiert.

Hier als Beispiel vorgestellt ist der Bereich »DigitSelector«, der eine kleine Zahlenauswahl direkt in der ausgewählten Zelle anzeigt. An die Funktion `getLayerVisible()` wird der Wert 3 übergeben. Die Rückgabe ist entweder `none`, dann ist der DigitSelector unsichtbar, oder `inline`, was den Bereich einblendet.

```
<g id='DigitSelector'
display='{{getLayerVisible(3)}}'
transform="translate({{getx( x, y )-1}}, {{gety( y )-1}} ) rotate(0)"
style='...>
...
</g>
```

Die beiden anderen Pop-up-Bereiche für Pause (SVG-Element `PopupPause`) und das Ende des Spiels (SVG-Element `PopupFinish`) verhalten sich analog. Die folgende Funktion steuert die Sichtbarkeit der verschiedenen Bereiche:

```
$scope.getLayerVisible = function(layerNr)
{
    if ( (layerNr === 3) && ($scope.nrfieldshow))
    {
        return "inline";
    }
    if ( $scope.state === layerNr)
    {
        return "inline";
    }
    return 'none';
}
```

Falls der Wert 3 (`DigitSelector`) angefragt wird, wertet die Funktion das Kennzeichen für den DigitSelector aus: `$scope.nrfieldshow`. Für die anderen beiden Pop-ups entspricht die angefragte Layer-Nr unmittelbar dem Kennzeichen für den Zustand, in dem sich das Spiel befindet:

- ➊ Das Spiel pausiert: `$scope.state === 1` und `LayerNr === 1`
- ➋ Das Spiel ist beendet: `$scope.state === 2` und `LayerNr === 2`

## Der Timer

AngularJS bietet einen Timer-Service an. Dieser wird mit der Variablen `$timeout` über die Signatur des FieldController injiziert. In der Funktion `setState()` startet die folgende Anweisung einen neuen Timer:

```
mytimeout = $timeout($scope.onTimeout,1000);
```

Dabei wird die Funktion `$scope.onTimeout()` nach der verstrichenen Zeit automatisch aufgerufen. Wie lange die Zeitspanne in Millisekunden sein soll, steuert der zweite Parameter. Im Beispiel bedeutet der Wert 1000, dass der Timer nach einer Sekunde (entspricht 1000 Millisekunden) abläuft und feuert.

Als Rückgabe liefert die Funktion einen Identifier des Timers. Mithilfe dieses Identifiers kann der Timer wieder entfernt werden. Genau für diesen Zweck merken wir uns die `TimerId` in der Variablen `mytimeout`.

Die Funktion `onTimeout()` ist im nächsten Codeblock dargestellt:

```
var mytimeout;

$scope.counter = 0;
$scope.onTimeout = function(){
    $scope.counter++;
    mytimeout = $timeout($scope.onTimeout,1000);
```

```
}

$scope.stop = function(){
  $timeout.cancel(mytimeout);
}
```

Die Variable `$scope.counter` enthält die abgelaufene Zeit seit dem Start der Aufgabe in Sekunden. In der Funktion `onTimeout()` erhöhen wir den Zähler um eins und starten den nun ausgelaufenen Timer erneut für einen weiteren Zyklus von einer Sekunde. Ohne diesen Neustart würde der Timeout nur einmal ablaufen.

Die zweite Funktion `stop()` benutzt den Timeout-Service, um den Timer zu beenden. Das ist notwendig, wenn das Spiel endet oder der Spieler die Pausefunktion aktiviert. In diesem Fall soll die Zeit nicht weiterlaufen.

### Die Tastatursteuerung im Detail

Die Funktion für Tastatureingaben wurde im Abschnitt View schon vorgestellt, als wir diese an den umfassenden DIV-Bereich gehängt haben. Im Grunde ist die Funktion eine große Fallunterscheidung. Als Eingabe erhält sie eine Nachricht vom Browser, der wir den Keycode entnehmen können.

Wichtig ist dabei: Wir erhalten wirkliche Tastencodes. Möchte der Benutzer einen Großbuchstaben eingeben, erhalten wir zwei Meldungen:

- Für das Drücken der Shift-Taste.
- Für die Buchstabentaste selbst.

In unserem Fall kommt uns dieses Verhalten entgegen: Wir brauchen keine Unterscheidung in Groß- und Kleinbuchstaben. Die ersten beiden Fallunterscheidungen steuern das Verhalten im Pausenmodus und nach dem Ende des Spiels. Der Zustand ist in der Variablen `$scope.state` abgelegt. Der Pausenmodus kann durch die Leertaste beendet und das Spiel fortgesetzt werden, indem wir den Zustand auf den Wert 0 setzen. Ist das Spiel beendet, weisen wir alle Tasten ab.

Danach prüft die Funktion, ob der Keycode zwischen den Werten 48 und 59 liegt, was den Zifferntasten entspricht. In diesem Fall müssen wir aus dem Keycode die entsprechende Ziffer berechnen und in die aktuelle Zelle setzen.

Mit einer `switch`-Anweisung unterscheiden wir die anderen Tasten: Liegt eine Pfeiltaste vor, passt die Funktion `nextPos()` die aktuelle Zelle im Feld an, indem sie über die Parameter steuert, in welche Richtung sich die aktuelle Markierung bewegt.

Die anderen Tasten delegieren wir an entsprechende spezielle Funktionen. Hier der Sourcecode der Funktion `keypress()`:

```
$scope.keypress = function($event) {

  // Wenn das Spiel pausiert ist: Space-Taste zum fortsetzen
  if ( ($scope.state === 1) && ($event.keyCode <= 32) )
```

```
{  
    $scope.setState(0);  
    return;  
}  
// Spiel ist beendet.  
if ( $scope.state === 2 )  
{  
    return;  
}  
// Zifferntasten pruefen  
if ( ($event.keyCode >= 49) && ($event.keyCode <= 58))  
{  
    $scope.selectNumber( 1+ ($event.keyCode-49) );  
} else {  
    switch ($event.keyCode)  
{  
        case 37: // Pfeil: Links  
            $scope.nextPos(-1, 0);  
            break;  
        case 39: // Pfeil: Rechts  
            $scope.nextPos(1, 0);  
            break;  
        case 40: // Pfeil: Oben  
            $scope.nextPos(0, 1);  
            break;  
        case 38: // Pfeil: Unten  
            $scope.nextPos(0, -1);  
            break;  
        case 82: // Taste R  
            // solve random field  
            $scope.solveRandom();  
            break;  
        case 67: // Taste C  
            // clearfield random field  
            $scope.clearCurrentField();  
            break;  
        case 83: // Taste S  
            // solve current  
            $scope.solveField();  
            break;  
        case 80: // Taste P  
            $scope.setState(1);  
            break;  
    }  
}  
$scope.nrfieldshow = false;  
};
```

## Reagieren auf Mausklicks im Spielfeld

Jetzt haben wir die Steuerung über die Tastatur ausführlich behandelt. Die Maussteuerung ist natürlich weiterhin möglich. Bleibt zu klären, was passiert, wenn wir eine Zelle mit der Maus anklicken. Dafür haben wir an jede Zelle die Funktion `fieldclick()` angehängt:

```
$scope.fieldClick = function(x, y){  
    if ( $scope.state !== 0 )  
    {  
        return;  
    }  
    if ( $scope.aufgabe[y][x].s === 0 )  
    {  
        $scope.x = x;  
        $scope.y = y;  
        $scope.nrfieldshow = true;  
    }  
};
```

Die Funktion erhält als Parameter die Spalte (`x`) und die Zeile (`y`) der Zelle, die angeklickt wurde. Das ist nur sinnvoll, wenn das Spiel aktiv ist und der Zustand `$scope.state` den Wert 0 aufweist. Ist das nicht der Fall, beenden wir die Funktion sofort. Damit ignorieren wir im Pause- und Ende-Zustand die Mausaktionen.

Ansonsten prüfen wir, ob es möglich ist, die Zellen zu ändern. Dazu ermitteln wir die Zelle, die sich hinter den Koordinaten verbirgt. Wenn das Attribut für den Zustand den Wert 0 hat, ist es eine Zelle, die der Spieler ändern kann. In diesem Fall übernehmen wir die übergebenen Koordinaten in die Position im Scope des Kontrollers. Damit ist diese Zelle markiert.

Durch die Anweisung `$scope.nrfieldshow = true` schalten wir die Darstellung des Mini-Pop-ups für die Ziffernwahl in der Zelle ein. Der Benutzer kann damit unmittelbar die gewünschte Ziffer auswählen.

## Eine Ziffer entgegennehmen

Die Funktion `$scope.selectNumber()` wird immer dann aktiviert, wenn der Benutzer eine Ziffer für eine Zelle auswählt. Das kann über die Maus, per Button oder über Tastatur erfolgen:

```
$scope.selectNumber = function(nr){  
  
    var c = $scope.aufgabe[$scope.y][$scope.x];  
    // Zelle ist nicht selektierbar  
    if (c.s !== 0)  
    { return; }  
    $scope.nrfieldshow = false;
```

```

// Zelle soll gelöst werden
if ( nr === -1)
{
    if (c.a !== c.l )
    {
        nr = c.l;
        c.a = nr;
        $scope.hints++;
    }
} else {
    // Ziffer in die aktuelle Zelle setzen
    c.a = nr;
    if ( (c.a !== c.l) && (nr !== 0))
    {
        $scope.errors++;
    }
}
$scope.calc();
// Prüfen ob das Feld vollständig und korrekt ist.
if ( ($scope.result.empty === 0)
    && ($scope.result.wrong === 0) )
{
    // Ja, dann Spiel beenden.
    $scope.setState(2);
}
};


```

Im Parameter (`nr`) erhält die Funktion die Ziffer, die der Benutzer für die aktuelle Zelle gewählt hat. Zuerst beschafft sich die Funktion die aktuell fokussierte Zelle des Spielfeldes. In diese Zelle soll die Ziffer eingetragen werden.

Zusätzlich können ebenfalls zwei weitere Fälle auftreten:

Die Übergabe `-1` deckt die aktuelle Zelle automatisch auf, wenn der Benutzer die Aktion »Aktuelle Zelle lösen« oder »Zufällige Zelle lösen« gewählt hat. Die Übergabe des Wertes `0` leert das aktuelle Feld und entfernt die Ziffer darin aus der Darstellung im Rahmen der Benutzeraktion »Zelle leeren«.

Zum Schluss ruft die Funktion die Funktion `calc()` auf, die das komplette Spielfeld untersucht. Für den Fall, dass jetzt alle Zellen ausgefüllt sind und keine Fehler vorliegen, wird das Spiel in den Zustand »beendet« gesetzt. Das entspricht dem Wert `2` in der Variable `$scope.state`. Der Zustand wird über die Funktion `setState(2)` geändert.

## Die verstrichene Zeit

Über der Aufgabe geben wir die verstrichene Zeit aus. Da wir die abgelaufenen Sekunden in der Variablen `$scope.counter` speichern, steht uns diese Information zur Verfügung. Bleibt nur die Aufgabe, eine ansprechendere Formatierung zu erzeugen. Üblich ist die Darstellung im Format *Stunden:Minuten:Sekunden*, hier drängt sich ein AngularJS-Filter geradezu auf:

```
gameApp.filter('seconds2Time', function() {
    return function( input ) {
        var out = "";
        var minutes = Math.floor( ( input / 60 ) % 60 );
        var hours = Math.floor( input / (3600) );
        var seconds = Math.floor( input % 60 );
        if (seconds < 10)
        {
            seconds = "0"+seconds;
        }
        if (minutes < 10)
        {
            minutes = "0"+minutes;
        }
        if ( hours > 0)
        {
            out = ""+hours+":"
        }
        return out+minutes+":"+seconds;
    }
});
```

Der Filter erhält als Input die Zeit in Sekunden seit Beginn des Spiels. Zunächst berechnet er die Werte für Minuten und Stunden. Die restlichen Sekunden lassen sich leicht mit dem Modulo-Operator (%) ermitteln.

Mit zwei Fallunterscheidungen prüfen wir, ob die Werte für Minuten und Sekunden einstellig sind. In diesem Fall ergänzen wir die Ausgabe für beide Werte um eine führende Null. Der Wert für die Stunden soll nur erscheinen, wenn mehr als eine Stunde verstrichen ist, sonst entfällt die Angabe ersatzlos. Die Verwendung des Filters in der View ist keine Überraschung:

```
Zeit:{{counter | seconds2Time }}
```

## Weitere Funktionen

Einige weitere Funktionen des Controllers wollen wir nur im Überblick ansprechen:

Funktionsname	Beschreibung und Aufgabe der Funktion
nextPos()	Die Funktion reagiert auf die Pfeiltasten, passt die aktuell markierte Zelle an und bewegt sie um ein Offset weiter. Falls die neue Position außerhalb des Spielfeldes liegt, wird die Position korrigiert.
solveRandom()	Diese Funktion sammelt alle noch nicht gefüllten Zellen in einer Liste. Im zweiten Schritt wählt sie ein Feld zufällig aus und deckt die Ziffer als Hilfestellung für den Benutzer auf.
calc()	Die Funktion untersucht das komplette Spielfeld und zählt alle Felder, für die folgende Eigenschaften gelten:  Leere, noch nicht ausgefüllte Zellen. Falsch gefüllte Zellen. Korrekt gefüllte Zellen.  Auf Grundlage der Informationen kann man leicht entscheiden, ob die Aufgabe vollständig und korrekt gelöst ist.
import()	Importiert eine Aufgabe und startet das Spiel. Dabei werden die Aufgaben interpretiert und die interne Repräsentation für das Spielfeld erstellt.

## Der Aufgabenservice

Der Service stellt lediglich eine sehr einfache Testimplementierung bereit. Er verwaltet eine Liste von vordefinierten Aufgaben, aus denen eine zufällige Aufgabe gewählt und zurückgeliefert wird. Der Service übernimmt ebenfalls das Erstellen der internen Datenstruktur der Aufgaben. Die Aufgaben im Service liegen in einem Feld und enthalten jeweils die eigentliche Aufgabe und die Lösung:

```
var aufgabenDaten =
[
  {
    id: 1,
    aufgabe :
" 8   7   39   46 75 9 4 38   5 4 2      3 5      1 6 8   46 8 1 97 13   68   7
5  ",
    loesung : "248613759139587462756924138695148273824375916371269845462851
397513792684987436521",
```

```

    skillLevel : 3
  },
  ...
]
```

Jede Aufgabe ist als zwei Zeichenketten mit jeweils 81 Stellen abgelegt. Der erste String definiert die Aufgabe mit den vorbelegten Zellen (die Ziffern enthalten) und den leeren Zellen (enthalten Leerzeichen), die der Benutzer zu füllen hat. Die zweite Zeichenkette enthält alle Ziffern der Aufgaben und stellt die vollständige und korrekte Lösung dar.

Die einzige Funktion, die der Service nach außen bekannt gibt, ist die Funktion `getAufgabe()`. Diese wählt eine zufällige Aufgabe aus der internen Liste und transformiert sie in die interne Darstellung für das Spielfeld:

```

return {
  getAufgabe: function () {
    var r = Math.floor( Math.random()* aufgabenDaten.length );
    return prepareAufgabe(aufgabenDaten[r].aufgabe, aufgabenDaten[r].loesung
  );
  }
};
```

Die Transformation selbst erledigt die private Funktion `prepareAufgabe()`:

```

var prepareAufgabe = function (aufgabeInput, loesungInput) {
  var aufgabe = [];
  var row = [];
  var l = 0;
  var c;
  var ca, cl;
  // Alle Leerzeichen durch die Ziffer 0 ersetzen
  var aufgabeStr = aufgabeInput.replace( / /g, "0" );
  for ( l = 0; l < 81; l++ )
  {
    // Zeichen zu Ziffern umwandeln
    ca = parseInt( aufgabeStr.charAt(l) );
    cl = parseInt( loesungInput.charAt(l) );

    // Zellen erzeugen
    c = { a:ca, s:0, l:cl };
    if ( ca !== 0)
    {
      c.s = 1;
    }
    row.push( c );

    // Neue Zeile beginnen
    if (( (l+1) % 9 ) == 0 )
  }
```

```
{  
    aufgabe.push( row );  
    row = [];  
}  
}  
return aufgabe;  
}
```

Die Funktion erhält die Aufgabe in der internen Zeichenkettendarstellung (Lösung und Aufgabe als getrennte Strings). Zuerst werden in der Aufgabe alle Leerzeichen in die Ziffer Null umgewandelt. Das geht am leichtesten mithilfe eines regulären Ausdrucks:

```
aufgabeInput.replace( / /g, "0" );
```

Die Funktion iteriert anschließend alle 81 Zeichen der Aufgabe und wandelt die Zeichen der Aufgabe und der Lösung in Ziffern um. Für jede Stelle wird eine neue Zelle mit den Attributen erstellt: für den Zustand der Ziffer für die Lösung und die Ziffer, die der Eingabe entspricht. Die Darstellung wurde am Beginn des Kapitels vorgestellt.

Über die `push()`-Anweisung landen die Zellen in den Zeilen und die Zeilen wiederum in der Variablen `aufgabe`.

#### Reguläre Ausdrücke

Der reguläre Ausdruck in der `String.replace()`-Anweisung steht nicht in Anführungszeichen. Reguläre Ausdrücke sind feste Sprachbestandteile von JavaScript (so genannte »First Class Citizens«).

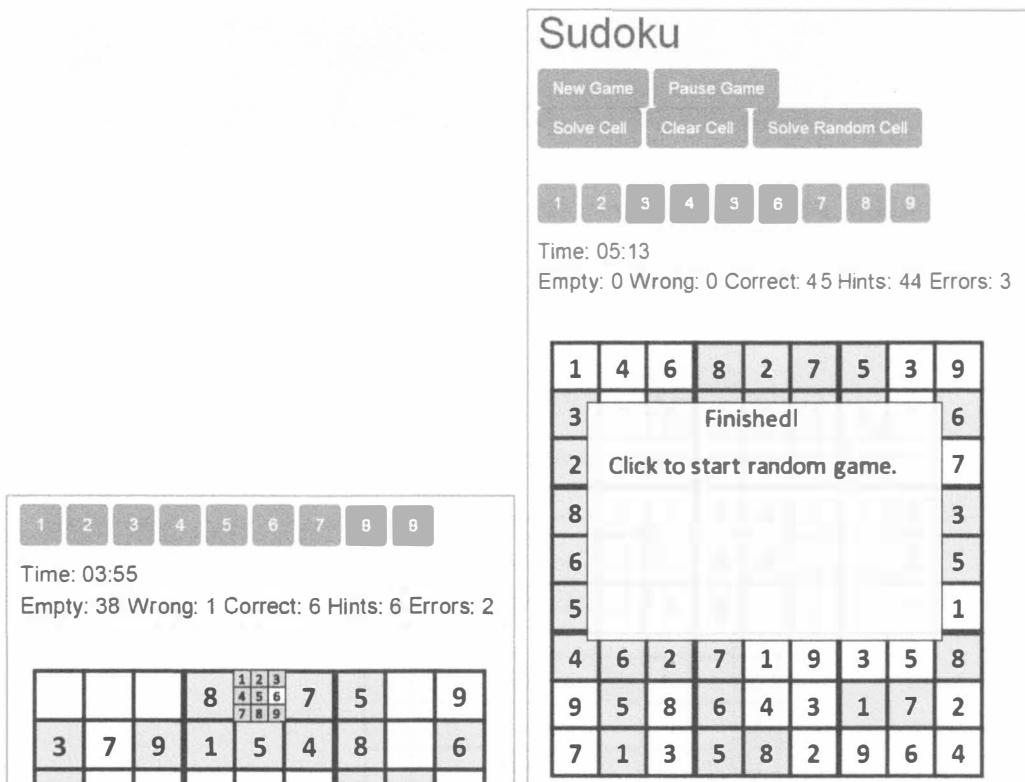


Bild 3.30: Die eingebettete Ziffernauswahl in einer Zelle und eine vollständig gelöste Aufgabe.

### Erstellen eines Sudoku-Backends

Mit den Techniken aus den beiden ersten Teilen in diesem Kapitel kann leicht eine Serverkomponente angesprochen werden, die vorbereitete Aufgaben ausliefert.

Das Erstellen neuer Sudoku-Aufgaben klammern wir an dieser Stelle aus. Es gibt einige Implementierungen als Open Source von Generatoren in ganz unterschiedlichen Programmiersprachen.

#### Kleine Anmerkung zum Erzeugen von Sudoku-Aufgaben

Die erste intuitive Implementierung eines Sudoku-Generators könnte ein vollständig gefülltes Sudoku-Feld nehmen und zufällig einige Ziffern entfernen. Je mehr Zellen leer sind, desto schwerer ist die Aufgabe zu lösen. Der Generator müsste in diesem Fall zusätzlich prüfen, ob die Aufgabe nach dem Entfernen wirklich eindeutig lösbar ist. Sonst könnte eine Konstellation entstehen, in der die Ziffern nicht logisch erschlossen werden können. Der Spieler müsste die nächsten Ziffern raten.

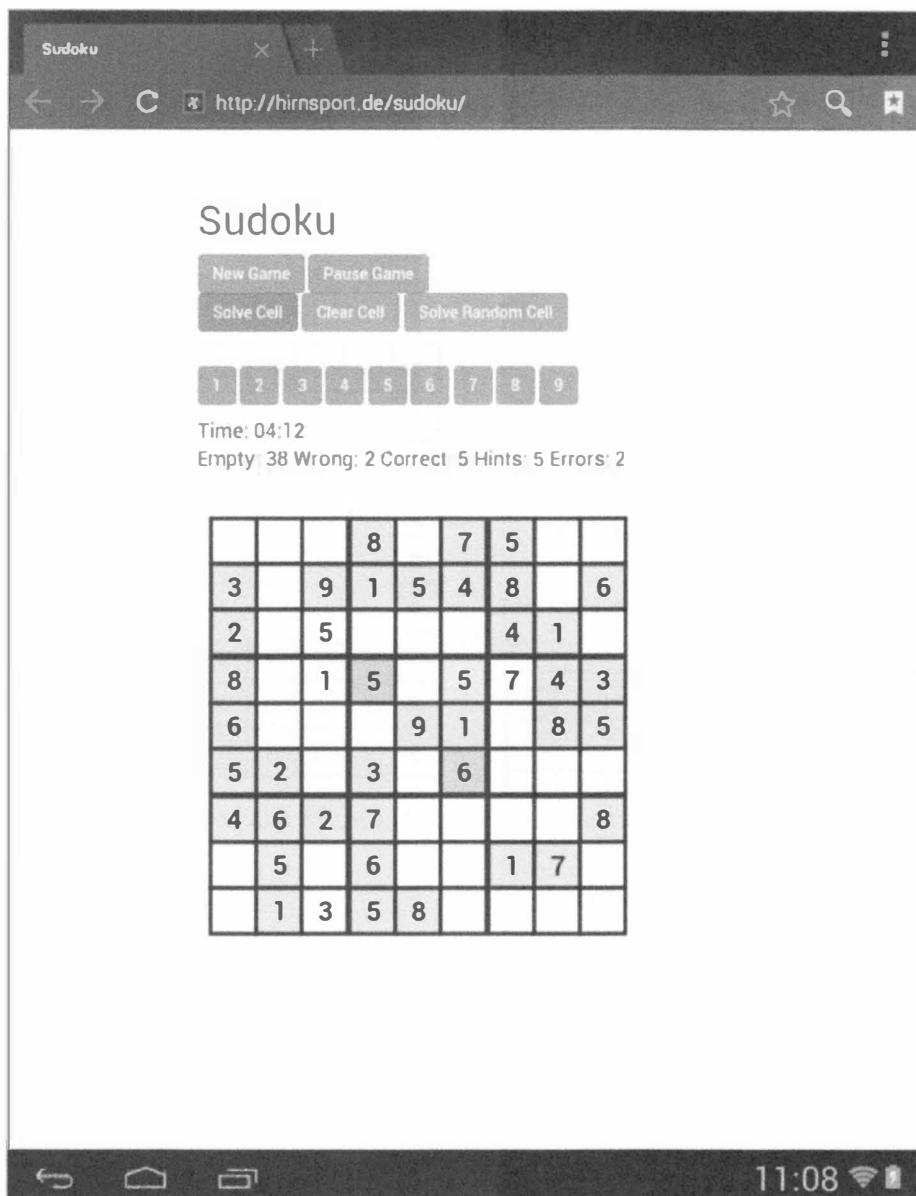


Bild 3.31: Die Sudoku-App funktioniert ebenfalls gut auf mobilen Geräten, da fast alle Plattformen SVG unterstützen.

## Ideen für die Weiterentwicklung des Spiels

Das Spiel ist noch nicht ganz fertig. Einige Erweiterungen sind offensichtlich:

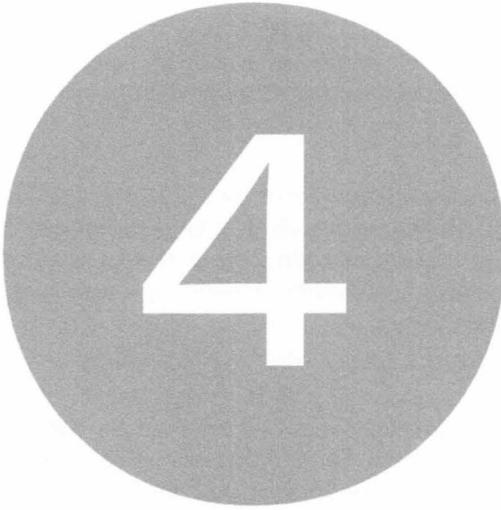
- Das Spiel könnte aus der abgelaufenen Zeit, der Anzahl an Fehlern und dem Schwierigkeitsgrad der Aufgaben einen Punktestand errechnen, der am Ende präsentiert wird. Je schneller eine schwere Aufgabe gelöst wird, umso mehr Punkte bekommt der Spieler gutgeschrieben.
- Der Zustand einer Aufgabe könnte im lokalen Speicher zwischengespeichert werden. Der Spieler könnte so später genau an der Stelle weiterknobeln, an der er ein Spiel pausiert hat.

## Fazit

Das Sudoku-Beispiel zeigt, dass AngularJS für die Realisierung sehr unterschiedlicher Arten von Applikationen geeignet ist. Die saubere Trennung von Logik und View unterstützt AngularJS auf sehr elegante Art und Weise. Insgesamt kommt man schnell an sein Ziel und kann sich meist sehr gut auf die fachliche Aufgabenstellung konzentrieren.

Das Beispiel zeigt ebenfalls, wie leicht sich UI-Elemente über AngularJS-Funktionen dynamisch erzeugen lassen und wie sich das Framework neben HTML mit anderen Markup-Sprachen versteht.





# 4

## Von der DB bis ins Web mit Meteor

### 4.1 Das Meteor-Framework

Meteor ist eine JavaScript-Plattform, basierend auf MongoDB und Node.js, die sich durch eine hohe Flexibilität, Einfachheit und neue Ansätze auszeichnet. Sie bietet klassischen Web-Frameworks die Stirn und bringt ein einheitliches Programmier-Modell und einen einheitlichen Datenbankzugriff vom Backend bis hin zum Frontend im Browser.

Das Team von Meteor hat bekannte und ausgereifte Frameworks geschickt unter einer gemeinsamen Schicht mit einem einheitlichen API vereint. Zum Einsatz kommen bekannte Namen, wie zum Beispiel: Amplify, AppCache, Backbone, Handlebars, jQuery und Underscore.

Außerdem bietet Meteor ein Umfeld, in dem man sehr schnell und effizient Applikationen entwickeln und in Aktion testen kann.

## 4.2 Die sieben Prinzipien

Meteor beschreibt sich selbst mit folgenden sieben Prinzipien:

### Data on the Wire

Es werden nur Rohdaten zwischen Server und Client ausgetauscht, nicht komplette HTML-Seiten oder HTML-Fragmente. Zudem entscheidet der Client, was er mit den Daten macht und wie er diese darstellt. Dieses Prinzip setzt sich schon seit einiger Zeit immer konsequenter in der Web-Entwicklung durch und Meteor geht diesen Weg konsequent weiter.

### One Language

Über alle Ebenen der Applikation, vom Client bis hin zur Datenbank, kommt nur eine Sprache zum Einsatz: JavaScript. Für den Browser ist das nicht verwunderlich, er versteht JavaScript als einzige native Sprache. Auf der Server-Seite erreicht Meteor das durch den Einsatz von Node.js als Basis, auf der der Meteor-Server aufsetzt. Mit der Standard-Datenbank MongoDB zieht Meteor dieses Konzept der einen Sprache bis in die Persistenzebene weiter.

### Database Everywhere

Die Datenbank steht sowohl dem Server als auch dem Client zur Verfügung. Das erreicht Meteor, indem es dieselbe API, die MongoDB auf dem Server bietet, in einfacher Form einer »MiniMongo«-Implementierung ebenfalls auf dem Client ausprägt.

### Latency Compensation

Jeder Server-Roundtrip kostet Zeit, sei die Netzwerkverbindung noch so performant. Meteor bietet Features, um diesen Zeitversatz vor dem Benutzer zu verstecken: Die Mechanismen nennen sich Vorselektierung und Model-Simulation.

Die Applikation antwortet sofort, ohne Verzögerung, und liefert ein Ergebnis. Unter Umständen ist dieses Ergebnis nur eine Näherung an die wirklichen Informationen, zum Beispiel, weil das Ergebnis von der letzten Aktion gecached wurde. Parallel werden die wirklichen Daten beim Server angefragt. Treffen diese beim Client ein, ersetzen die aktuellen Echtdaten die simulierten Daten und die Anzeige wird angepasst.

Das Vorgehen hört sich etwas abenteuerlich an, ist in vielen Anwendungsfällen aber sinnvoll. Intuitiv fühlt sich eine Applikation sehr viel lebendiger an. Die beiden Beispiele verdeutlichen Vor- und Nachteile:

- Bei einer Shopping-Website wird der Benutzer nicht verwundert sein, wenn sich die Informationen zur Lieferbarkeit eines Artikels ändern, weil die Applikation aktuellere Daten erhält.
- Bei einer Online-Banking-Applikation wird ein Benutzer wahrscheinlich nicht akzeptieren, dass der Kontostand nur ungefähr richtig ist. Bei solchen sensitiven Daten sollte man lieber warten und nur korrekte Informationen anzeigen.

## Full Stack Reactivity

Live-Daten und reaktive Programmierung sind wesentliche Elemente von Meteor. Fast alle APIs sind Event-gesteuert und reagieren bei einer Änderung der zugrunde liegenden Daten automatisch.

## Embrace the Ecosystem

Meteor steht unter der Open-Source-Lizenz des MIT, welche besagt, dass es hinsichtlich des Einsatzes des Quellcodes kaum Beschränkungen gibt. Außerdem bietet es die Möglichkeit, eigene Add-ons für Meteor zu erstellen und über einen Katalog anderen Entwicklern anzubieten. Das Prinzip besagt ebenfalls, dass sich das Meteor-Team intensiv bei dem JavaScript-Ökosystem bedient und ausgereifte Frameworks geschickt in Meteor kombiniert.

## Simplicity equals Productivity

Das Verstecken von Komplexität ist eines der Design-Prinzipien des gesamten Projektes. Viele Zusammenhänge im Programmiermodell sind über Konventionen geregelt und müssen nur in wenigen Ausnahmen geändert werden.

Die vielen eingesetzten Frameworks verstecken sich unter einer einheitlichen API-Schicht und fühlen sich deshalb sehr homogen an. Trotzdem lassen sich die Frameworks gut direkt verwenden. Damit erreicht man bei der Entwicklung mit Meteor eine hohe Produktivität und kann sich meist sehr gut um die wichtigen fachlichen Herausforderungen kümmern.

## 4.3 Starten mit Meteor

Die Software ist noch in der Entwicklung und Änderungen an der API werden zwar immer unwahrscheinlicher, können aber immer noch vorkommen.

Für die aktuellen Versionen werden die Betriebssysteme OS X und Linux vom Meteor-Team direkt unterstützt. Für Windows existiert eine inoffizielle Binary-Distribution. Bis zur finalen Version 1.0 soll es eine offizielle Windows-Unterstützung vom Projektteam selbst geben.

### 4.3.1 Installation und erste Schritte

Die folgenden Schritte sind für ein Linux-System gedacht. Um mit Meteor zu starten, muss es zuerst installiert werden. Dies erreicht man schnell und einfach über den Befehl:

```
> curl https://install.meteor.com | /bin/sh
```

Auf manchen Systemen ist der Befehl nicht vorinstalliert. In diesem Fall muss der Curl-Befehl erst installiert werden (zum Beispiel mit APT).

Meteor bietet in der Kommandozeile den Befehl `meteor`. Eine Liste aller Befehle und Parameter liefert der Aufruf:

```
> meteor --help
```

Die folgende Tabelle zeigt die wichtigsten Befehle von Meteor im Überblick:

Befehl	Beschreibung
<code>run</code>	Startet das Meteor-Projekt im aktuellen Verzeichnis. Dies ist der Default-Befehl, falls man keine weitere Angabe vorgibt.
<code>create &lt;name&gt;</code>	Erzeugt eine Applikation mit dem definierten Namen.
<code>update</code>	Aktualisiert die Meteor-Version und installiert notwendige Komponenten nach.
<code>add &lt;modulname&gt;</code>	Fügt ein neues Add-on zur Applikation hinzu.
<code>remove &lt;modulname&gt;</code>	Entfernt ein Add-on von der Applikation.
<code>list</code>	Listet alle verfügbaren Add-ons auf.
<code>bundle</code>	Packt die Applikation in ein Archiv (Tarball mit der Endung <code>tar.gz</code> ) für die Installation auf einem Server. Alle notwendigen Pakete und Tools werden mit eingepackt. Lediglich Node.js und MongoDB werden als Umgebung erwartet.
<code>mongo</code>	Verbindet sich mit einer MongoDB-Instanz, die entweder lokal existiert oder per URL spezifiziert ist.
<code>deploy</code>	Packt das gesamte Projekt in ein Archiv und deployt es auf der Infrastruktur von Meteor.com. Als Name kann eine noch freie Bezeichnung, zum Beispiel ein Projektname wie <code>myfirstapp.meteor.com</code> , genutzt werden.
<code>logs &lt;site&gt;</code>	Zeigt die Logdateien des Servers der spezifizierten Umgebung.
<code>reset</code>	Setzt den Zustand der lokalen Applikation zurück. Es werden zusätzlich alle Einträge in der lokalen MongoDB gelöscht.
<code>test-packages</code>	Führt Unit-Test der definierten Packages aus. Diese können auf einer anderen Instanz liegen. Die Ergebnisse werden im Browser angezeigt.

Zu jedem Befehl kann man über die Angabe von `--help` Detailinformationen und mögliche Parameter erhalten. Der ausgeführte Befehl bezieht sich immer auf das Meteor-Projekt, in dem man sich aktuell befindet.

Das erste Projekt mit dem Namen `myfirstapp` erstellen wir mit dem Aufruf:

```
> meteor create myfirstapp
```

Meteor kreiert ein Unterverzeichnis mit dem Namen `myfirstapp` und erzeugt darin eine einfache, aber vollständige Applikation. Dieses Vorgehen wird mit dem Begriff »Scaffolding« ([de.wikipedia.org/wiki/Scaffolding](https://de.wikipedia.org/wiki/Scaffolding)) bezeichnet und erleichtert den initialen Aufwand für ein neues Projekt. Im Verzeichnis befinden sich folgende Elemente:

- ein Ordner `.meteor` mit den Konfigurationsdateien: `package` und `release`.
- eine leere CSS-Datei: `myfirstapp.css`.
- eine HTML-Datei mit dem Namen: `myfirstapp.html`.
- und eine JavaScript-Datei mit dem Applikationscode unter `myfirstapp.js`.

#### Beispielprojekte vom Hersteller

Wollen Sie lieber vom Meteor-Team lernen? Kein Problem, der Create-Befehl kennt eine Option, die ein vollständiges Beispiel erzeugt. Auf der Meteor-Website ([www.meteor.com/examples/](https://www.meteor.com/examples/)) sind einige fertige Applikationen verfügbar:

Leaderboard: Eine Punkteliste für Wettbewerbe

To-dos: Verwaltung für Aufgaben

Wordplay: Spiel um Worte zu finden

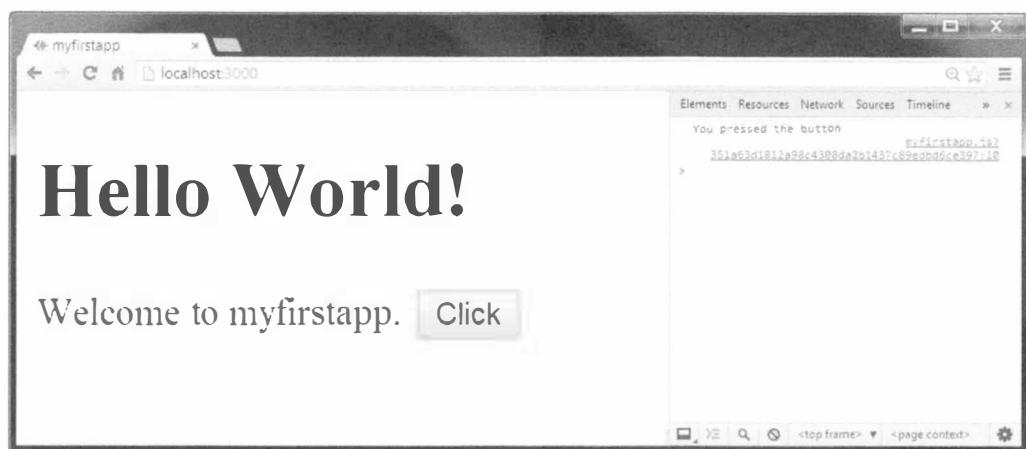
So erstellt der folgende Befehl das To-dos-Projekt im lokalen Verzeichnis:

```
> meteor create --example todos
```

Zum Starten der Applikation ist nichts weiter zu tun, als in den Ordner der Applikation zu wechseln und den Meteor-Server zu starten:

```
> meteor run
```

Als Default benutzt Meteor den Port 3000, somit ist die Applikation sofort unter der URL: <http://localhost:3000> im Browser verfügbar:



**Bild 4.1:** Eine erste (einfache) Meteor-Applikation.

Die Beispielapplikation ist nicht sonderlich nützlich und gibt lediglich eine Nachricht aus. Klickt der Benutzer auf den Button, erscheint eine Meldung auf der Konsole. Mit CTRL-C (STRG-C) lässt sich der Prozess beenden.

### Die erste Meteor-Anwendung im Detail

Was hat Meteor wirklich erzeugt? Die CSS-Datei ist leer und enthält keine Angaben zum Layout und Styling. Ein Blick in die HTML-Datei ist wesentlich interessanter:

```
<head>
  <title>myfirstapp</title>
</head>

<body>
  {{> hello}}
</body>

<template name="hello">
  <h1>Hello World!</h1>
  {{greeting}}
  <input type="button" value="Click" />
</template>
```

Es fällt auf, dass das Template ohne die Angabe von `doctype` oder ein `html`-Tag auskommt. Meteor fügt die Angaben automatisch hinzu und baut den kompletten HTML-Header mit allen notwendigen Bibliotheken und abhängigen Dateien ein.

Meteor benutzt Handlebars als Template-Engine. In der HTML-Datei äußert sich das an den dynamischen Elementen, die in doppelte geschweifte Klammern eingebettet sind: `{{ ... }}`.

Der Ausdruck: `{{> hello}}` ruft ein Sub-Template mit dem Namen `hello` auf und expandiert es an der aufrufenden Stelle. Das Template steht etwas weiter unten in der Datei und beginnt mit: `<template name="hello">`.

Innerhalb des Templates greift der Ausdruck: `{{greeting}}` direkt auf eine Information zurück, die im JavaScript-Teil definiert wird (siehe unten).

Die eigentliche Applikation, in der Datei `myfirstapp.js`, ist in einen Client- und einen Server-Teil aufgeteilt. Meteor bietet die beiden Eigenschaften `Meteor.isClient` und `Meteor.isServer` an, um im Code zu prüfen, in welcher Umgebung man sich befindet. Dieses Vorgehen ist nur für kleine Projekte sinnvoll. Wie man die Applikation in separate Dateien aufteilt, wird ein späteres Beispiel zeigen.

Im Client-Bereich wird zuerst eine Funktion definiert, die die oben beschriebene dynamische Information (`greeting`) liefert. Die Konvention für die Benennung ist von Meteor vorgegeben: `Template.<Templatename>.<Elementname>`.

Als Zweites wird ein Event für den Button erstellt. Da es nur ein Input-Element gibt, ist die Zuordnung mit 'click input' eindeutig. Alternativ ordnet Meteor die Elemente gemäß einem CSS-Selektor zu. Das heißt, man muss nicht jedes Element einzeln zuordnen, sondern kann gleich ganze Gruppen über einen geschickt gewählten Selektor verbinden.

Die Abfrage, ob die Variable `console` existiert, ist streng genommen notwendig, da nicht alle Javascript-Umgebungen eine Console anbieten.

```
if (Meteor.isClient) {
  Template.hello.greeting = function () {
    return "Welcome to app1.";
  };

  Template.hello.events({
    'click input' : function () {
      // template data, if any, is available in 'this'
      if (typeof console !== 'undefined')
        console.log("You pressed the button");
    }
  });
}

if (Meteor.isServer) {
  Meteor.startup(function () {
    // code to run on server at startup
  });
}
```

Im Server-Bereich ist nur eine Methode `Meteor.startup()` definiert, die Meteor beim Applikationsstart einmalig aufruft. Sie ist ein guter Platz, um Initialisierungen vorzunehmen.

### 4.3.2 Mehr Struktur in umfangreicheren Projekten

Im ersten Beispiel wurde schon angedeutet, dass Meteor eine Aufteilung in unterschiedliche Dateien für Client und Server anbietet. Die folgende Verzeichnisliste zeigt eine typische Struktur:

```
/myfirstapp/...
 /lib/...
 /somefolder/...
 /server/
   lib/...
     someJsFiles.js
     main.js
 /lib/...
 /client/
   someJsFiles.js
   main.js
 /public/...
   someImages.jpeg
```

Meteor definiert folgende Regeln für die Auslieferung der Dateien:

Alle Dateien unterhalb der Verzeichnisse `client` und `server` werden jeweils nur an ihre entsprechende Umgebung ausgeliefert. Die Reihenfolge ergibt sich innerhalb jeder Umgebung nach folgenden Regeln:

- ➊ die Dateien unterhalb des Verzeichnisses `lib`
- ➋ alle weiteren Dateien im jeweiligen Verzeichnis ihrer Umgebung in alphabetischer Reihenfolge
- ➌ zum Schluss die Dateien, die dem Muster `main*.js` folgen

Für statische Inhalte, wie Grafiken, Bilder oder andere Multimedia-Daten, bietet Meteor das Verzeichnis `public` an. Die Inhalte werden von Meteor nicht verändert oder mit anden Programmteilen kombiniert, sondern in ihrem Ursprungszustand an den Client ausgeliefert.

Wenn der Meteor-Server aktiv ist, erkennt er automatisch Modifikationen an den Dateien und aktualisiert sich. Das heißt, nach dem Speichern einer Änderung im Editor aktualisiert sich die Applikation von selbst und der neue Stand ist ohne weiteres Zutun im Browser zu sehen. Dadurch erhält man schnell eine Rückmeldung über die durchgeführten Änderungen.

## 4.4 Beispiel: Chat-Applikation

Um die Stärken von Meteor besser zu zeigen, bauen wir eine Chat-Applikation, in der beliebig viele Teilnehmer mit diskutieren können. Diese Art Applikationen mit der Verteilung von Daten an viele Teilnehmer wird von Meteor besonders gut unterstützt.

### 4.4.1 Schritt 1: Start und Setup

Wie im Hello-World-Beispiel am Anfang des Kapitels erzeugen wir eine neue Applikation in einem leeren Verzeichnis:

```
meteor create chat
```

Wir ändern die default-Verzeichnisstruktur in die folgende Form und legen die Dateien an:

```
client/
  client.js
  client.html
  client.css
common/
  collections.js
```

Den Server-Teil können wir ignorieren, da wir für das ganze Chat-Beispiel nur Standardfunktionen von Meteor nutzen. Lediglich die Definition der Datenstruktur für die Textmeldungen in Form einer JavaScript-Collection, welche über die MongoDB synchronisiert wird, legen wir im common-Ordner ab. Diese ist damit sowohl für den Server als auch für den Client sichtbar.

#### Definition der Collection für die Text-Nachrichten

Der Inhalt der Datei commons.js ist sehr übersichtlich. Sie definiert lediglich die MongoDB-Collection, die auf dem Server und dem Client bekannt sein soll und die Nachrichten des Chats enthält.

```
/** 
 * Common Collections and Functions
 */
Messages = new Meteor.Collection('messages');
```

#### Die Darstellung in der View

Das Template definiert die Struktur der HTML-Seite unserer Applikation:

```
<head>
  <title>Simple Chat</title>
</head>

<body>
```

```
<h1>Simple Chat</h1>
{{> welcome }}
{{> input }}
{{> messages }}
</body>

<template name="welcome">
<p>
  Welcome to the sample chat web page! <br/>
  Your name: <input type="text" id="username">
</p>
</template>

<template name="messages">
{{#each messages}}
  <strong>{{name}}:</strong> {{message}}<br>
{{/each}}
</template>

<template name="input">
<p>Message: <input type="text" id="message">
<input type="button" id="send" value="Send">
</p>
</template>
```

Im Body-Bereich werden drei Sub-Templates angesprochen, die unsere Seite ausmachen:

- Der Bereich `welcome` enthält ein Eingabefeld für den Benutzernamen, damit wir sehen, wer der Autor einer Meldung ist.
- Der `input`-Bereich bietet das Eingabefeld und einen `Send`-Button für das Absetzen neuer Nachrichten.
- Der letzte Bereich `messages` soll alle Nachrichten anzeigen. Das Template erhält die Texte in einer Collection mit dem Namen `messages` und iteriert über alle Einträge. Für jeden Eintrag entsteht durch den Handlebar-Helper `{{#each messages}}` eine neue Zeile mit der Ausgabe des Autors und dem eigentlichen Text der Nachricht.

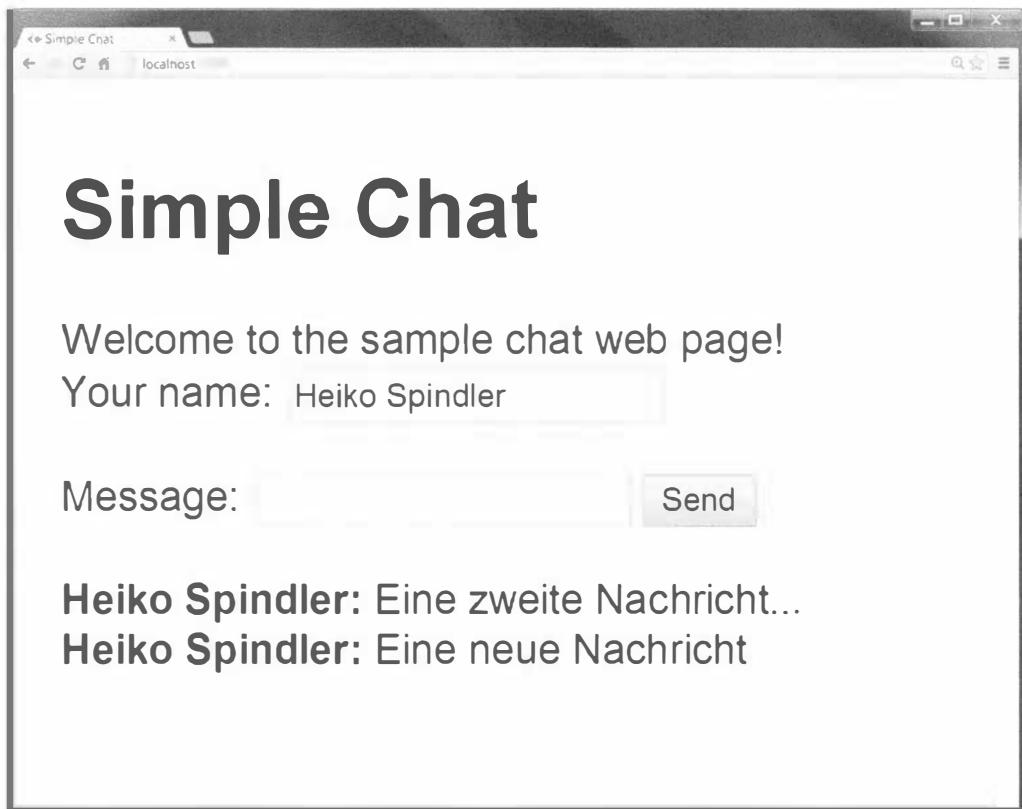


Bild 4.2: Ein erster Blick auf die Chat-Applikation.

### Der Client

Die Datei `client.js` definiert die Client-Logik der Chat-Applikation. Dazu werden Hilfsfunktionen für die Templates erstellt:

```
Template.input.events = {
  'click input#send' : function (event) {
    sendMessage( );
  }
}

var sendMessage = function()
{
  var username = $("#username").val();
  //var message = document.getElementById('message');

  if (!username) {
    username = 'Anonymous';
  }
}
```

```

var message = $("#message").val();

if (message != '') {
  Messages.insert({
    name: username,
    message: message,
    time: Date.now()
  });
  $("#message").val("");
}

Template.messages.messages = function () {
  return Messages.find({}, { sort: { time: -1 }});
}

```

Die erste Funktion definiert eine Event-Map für das Sub-Template `input` mit einem Click-Event. Die Schreibweise innerhalb der Event-Maps ist etwas ungewohnt, aber gut nachvollziehbar, da es bei der Schreibweise in CSS-Definitionen abgeschaut ist: `'click input#send'`. Die Nachricht wird an ein `input`-Element mit dem HTML-Identifier `send` geknüpft. Ein Blick in das HTML-Template offenbart, dass damit der Send-Button gemeint ist. Die Meteor-Eventmaps werden im TeamDraw-Beispiel, weiter unten in diesem Kapitel, tiefergehend vorgestellt.

Die Verarbeitung des Events ist in die lokale Funktion `sendMessage()` ausgelagert. Diese prüft, ob ein `username` definiert ist. Fehlt dieser, wird die Nachricht mit dem Absender `Anonymous` versendet. Der Zugriff auf die DOM-Elemente erfolgt über jQuery-Funktionen:

```
$("#username").val();
```

Danach holt die Funktion auf ähnliche Art den Inhalt aus dem Eingabefeld für den Chattext. Ist dieser nicht leer, fügt die `insert()`-Funktion der Collection den Text mit Benutzer und einem TimeStamp in die Liste der Nachrichten ein. Zum Schluss der Funktion leeren wir das Eingabefeld, damit der Benutzer gleich die nächste Nachricht erfassen kann.

#### Tipp: Einsatz von jQuery mit Meteor

Genau genommen muss nach dem initialen Erzeugen des Projektes jQuery als Add-on aufgenommen werden. jQuery ist in Meteor eingebaut und direkt nutzbar, könnte laut Aussagen des Entwicklerteams aber herausfallen. Mit dem expliziten Einbinden von jQuery ist man auf der sicheren Seite:

```
> meteor add jquery
```

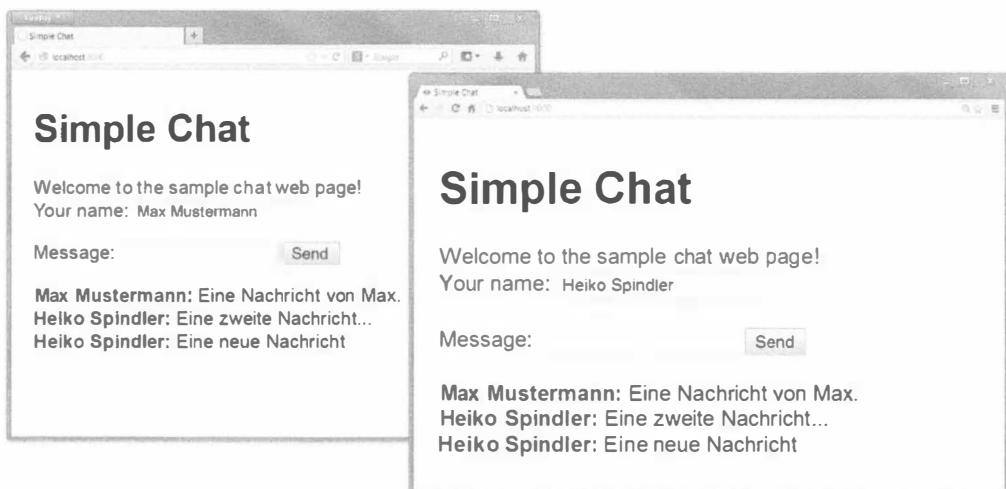
Die letzte Funktion: `Template.messages.messages` liefert die Collection der Chat-Meldungen an das `messages` Template. Die Methode `find()` der Collection erhält im ersten Argument keine Suchkriterien, sondern ein leeres Objekt `{}`. Somit enthält die

Ergebnisliste alle verfügbaren Einträge. Das zweite Argument definiert zusätzliche Eigenschaften der Ergebnisliste. In unserem Fall soll das Ergebnis nach dem Attribut time absteigend sortiert sein. Das ist sinnvoll, damit die neuesten Nachrichten als Erstes erscheinen.

Mehr als etwa 60 Zeilen HTML- und JavaScript-Code brauchen wir nicht für unsere vollständige Chat-Applikation. Der Start erfolgt wieder über die Kommandozeile mit:

```
> meteor run
```

Der Screenshot zeigt zwei unterschiedliche Benutzer, die sich mit der Chat-Applikation unterhalten. Meteor verteilt die Einträge in der Datenbank automatisch auf alle Clients, die die Web-Applikation gestartet haben. Sofort beim Start sind die letzten Einträge sichtbar.



**Bild 4.3:** Eine Unterhaltung in der Chat-Applikation mit mehreren Teilnehmern.

Leider zeigt der Screenshot das Live-Update der Inhalte nicht angemessen. Die Meldungen erscheinen automatisch ohne ein explizites Refresh (durch F5) in den Fenstern aller Teilnehmer. Es lebe die Reaktivität.

#### 4.4.2 Schritt 2: Erste Erweiterungen des Chats

Die Grundfunktionen des Chats funktionieren schon. Jetzt ist es Zeit, mehr Komfort einzubauen:

##### Tastatur-Events

Der Benutzer sollte das Senden der Nachrichten leichter ohne Maus aktivieren können, dazu lassen wir das Eingabefeld zusätzlich auf die Entertaste (Returntaste)

reagieren. Das eigentliche Senden ist schon in eine eigene Funktion ausgelagert. Deshalb reicht es aus, ein zusätzliches Event aufzunehmen.

Wir wollen Tastatur-Nachrichten im Eingabefeld mit dem Identifier `message` belauschen. Erhalten wir eine Nachricht, wird unserer Handler-Methode ein Event-Objekt übergeben. Im Attribut `which` steht dann der Tastatur-Code. Der Code 13 entspricht der Enter-Taste:

```
Template.input.events = {
  'keydown input#message' : function (event) {
    if (event.which === 13) { // 13 is the enter key
      sendMessage();
    }
  },
  'click input#send' : function (event) {
    sendMessage();
  }
}
```

### Weiteres Add-on: Twitter Bootstrap

Das Twitter-Bootstrap-Framework kennen wir aus dem letzten Kapitel. Meteor-Anwendungen lassen sich damit ebenfalls leicht verschönern, da Bootstrap direkt als Standard-Modul bereitsteht:

```
> meteor add bootstrap
```

Lästige Handarbeit entfällt, wir müssen keine Bibliotheken herunterladen und in unsere HTML-Datei einbinden.

Meteor erledigt das für uns. Danach stehen die gewohnten Bootstrap-CSS-Klassen bereit. Im AngularJS-Kapitel wurden einige Elemente schon eingehend besprochen und wer sich die Anwendung genau anschauen möchte, sei auf den Source-Code des Chat-Beispiels verwiesen.

Einen kleinen Nachteil handeln wir uns mit dem Meteor-Modul ein: Wir sind von der Bootstrap-Version abhängig, die im Meteor-Modul vorgegeben ist. In der eingesetzten Version steht Bootstrap 2.3.x bereit. Wer sein Projekt mit der neueren Version 3.x ausstatten möchte, ist entweder auf Handarbeit angewiesen oder bedient sich bei dem Add-on-Repository `Atmosphere`, dort steht die Version 3.x bereit.

### Modulare Templates durch Handlebars-Helpers

Handlebars bietet mit den Helpers einen Mechanismus, der an die AngularJS-Filter oder -Direktiven erinnert. Dazu nehmen wir uns folgende Anforderung vor: Bei jeder Textnachricht zeigen wir einen Zeitstempel mit Datum und Uhrzeit an.

In der Collection tragen wir den Zeitstempel ein, damit sich die Nachrichten leicht zeitlich sortieren lassen. Wenn wir den Zeitstempel direkt ausgeben, erhalten wir eine Zahl (Timestamp in Millisekunden), die wenig ansprechend ist. Wir brauchen eine

Möglichkeit, die Rohdaten bei oder vor der Anzeige sinnvoll zu formatieren. Genau für diese Art von Aufgabe eignen sich die Handlebars-Helper. In der `Client.js`-Datei definieren wir die Logik des Helpers mit wenigen Zeilen:

```
Handlebars.registerHelper('formatDate', function(input) {  
    return new Date(input).toUTCString();  
});
```

Die Funktion `registerHelper()` erwartet als ersten Parameter den Namen des Helpers und als zweiten Parameter eine Funktion mit der eigentlichen Logik.

Unsere Funktion soll den Timestamp als Parameter (`input`) erhalten und wandelt diesen in ein Date-Objekt um. Die Funktion `toUTCString()` liefert eine sprechende Darstellung des Datums mit Wochentag, Datum, Uhrzeit und Zeitzone als Zeichenkette.

Im Template nutzt man einen Helper durch die Angabe des Namens in geschweiften Klammern. Das nachfolgende Beispiel zeigt, wie man durch Leerzeichen getrennt die Parameter (zum Beispiel den Wert in der Variable `time`) an den Helper überträgt:

```
<template name="messages">  
  {{#each messages}}  
    <strong>{{name}}:</strong> {{message}}  
    [{{ formatDate time }}]<br>  
  {{/each}}  
</template>
```

Alternativ könnten wir das Datum in der endgültigen Darstellungsform als vollständigen String in die Collection schreiben. Das würde uns die Umwandlung und den Helper bei der Anzeige ersparen. Nachteile dieser Lösung wären:

- ➊ Die Sortierung funktioniert nicht mehr wie gewohnt.
- ➋ Die Anzeige könnte nur sehr aufwendig an andere Länder und Sprachen angepasst werden.
- ➌ Das Datum würde etwas mehr Speicherplatz belegen.

Handlebars-Helper sind eine gute Möglichkeit, generische Hilfsfunktionen zu erstellen, die die Anzeige von Daten aufbereiten. Die Helper sind nicht einem bestimmten Template zugeordnet. Verglichen mit den AngularJS-Direktiven sind die Helper sehr viel einfacher gestrickt und können sich nicht so elegant in den HTML-Code einfügen, in dem man eigene neue HTML-Tags definiert. Trotzdem sind die Helper ein sinnvolles Hilfsmittel, das für viele Aufgaben ausreicht.



**Bild 4.4:** Die Abbildung zeigt die optischen Verbesserungen: Zeitstempel und das Bootstrap Look&Feel.

#### 4.4.3 Schritt 3: Ein erstes Login

Bisher kann man beliebige Namen in das Usernamefeld schreiben. Damit ist Manipulationen Tür und Tor geöffnet. Deshalb wollen wir unsere Benutzer genauer kennen und eine Benutzer-Anmeldung einbauen.

Zum Glück bietet Meteor hierfür ein sehr ausgereiftes Add-on an. Es bietet Out-of-the-Box ein vordefiniertes UI und alle wichtigen Grundfunktionen für eine Benutzer-Authentifizierung:

- Registrieren mit optimaler Aktivierung per E-Mail
- An- und Abmelden
- Passwörter zurücksetzen und per E-Mail bestätigen

#### Das Login-Modul einbauen

Zuerst ergänzen wir das Projekt um die Login-Module. In der Kommandozeile im Projektverzeichnis führen wir folgende Anweisungen aus:

```
$ meteor add accounts-ui  
$ meteor add accounts-password
```

In der Dokumentation von Meteor wird ebenfalls das `accounts-base`-Modul erwähnt. Es ist die Grundlage der anderen Module und definiert die Datenstruktur für Benutzer-Konten und Logins. Es wird automatisch als Abhängigkeit aufgenommen, wenn wir eines der anderen Accounts-Module aufnehmen.

Zur Kontrolle können wir einen Blick in die Datei `.meteor/package` werfen. Diese enthält alle in unserem Projekt aufgenommenen Module:

```
# Meteor packages used by this project, one per line.  
#  
# 'meteor add' and 'meteor remove' will edit the file for you,  
# but you can also edit it by hand.  
  
autopublish  
insecure  
preserve-inputs  
accounts-ui  
accounts-password
```

Dieselben Informationen erhalten wir ebenfalls durch den Aufruf von:

```
>meteor list --using  
autopublish  
insecure  
preserve-inputs  
accounts-ui  
accounts-password
```

Meteor generiert einige Module automatisch in eine Beispiel-Applikation, wenn sie erzeugt wird: Insbesondere die beiden Module `autopublish` und `insecure` sind nur für prototypische Applikationen geeignet:

- ➊ Das Modul `autopublish` sorgt dafür, dass alle Daten in den MongoDB-Collections automatisch auf alle Clients verteilt werden.
- ➋ Das zweite Modul `insecure` schaltet alle Prüfungen auf Berechtigungen beim Zugriff auf Inhalte der Collections ab.

### Konfiguration des Accounts-Moduls in der Datei Client.js

Nach der Installation der Module sind einige wenige Änderungen im JavaScript-Teil notwendig. Für das Accounts-Modul gibt es zwei Konfigurationsmethoden:

- ➊ **Accounts.config(options):**  
Diese Methode konfiguriert das grundlegende Verhalten des Moduls. Dazu gehört, zum Beispiel, ob der Benutzer das Erstellen eines Accounts per E-Mail bestätigen soll oder nach wie vielen Tagen ein Login automatisch abläuft.
- ➋ **Accounts.ui.config(options):**  
Definiert das Verhalten und Aussehen des Login-UIs, zum Beispiel, welche Felder im Login-Bereich sichtbar sind und gefüllt werden müssen.

Für beide Methoden sollte man einen Blick in die Dokumentation werfen, welche zusätzlichen Parameter für die eigene Applikation nützlich sind.

Für unsere Applikation aktivieren wir einen Login-Namen und machen die E-Mail optional als Beispiel für andere Konfigurationsänderungen:

```
// Settings for Accounts module
Accounts.ui.config({
  passwordSignupFields: 'USERNAME_AND_OPTIONAL_EMAIL'
});
```

Weitere mögliche Werte für die Konfiguration zeigt die Tabelle:

Wert	Bedeutung
USERNAME_AND_EMAIL	Username und E-Mail sind Pflichtangaben und müssen beide gefüllt sein.
USERNAME_AND_OPTIONAL_EMAIL	Username ist Pflicht und E-Mail optional.
USERNAME_ONLY	Es soll nur der Loginname ohne E-Mail genutzt werden.
EMAIL_ONLY (Default)	Auf einen Loginnamen wird verzichtet und die E-Mail fungiert als Loginname.

Mit dem Modul erhalten wir einige nützliche Funktionen von Meteor bereitgestellt. Hier ist nur eine Auswahl vorgestellt:

➊ **Meteor.user()**  
liefert das aktuelle User-Objekt oder `null`, falls kein Benutzer eingeloggt ist

➋ **Meteor.userId()**  
liefert die aktuelle UserId oder `null`

➌ **Meteor.users**  
ist eine Collection aller Benutzer, die sich bei der Applikation angemeldet haben

Die Methode zum Senden der Nachrichten kann damit etwas eleganter formuliert werden. Den Usernamen erhalten wir direkt über die API des Accounts-Moduls mit `Meteor.user().username`. Als Fallback benutzen wir weiterhin den Namen `Anonymous`.

```
var sendMessage = function()
{
  var username = 'Anonymous';
  if (Meteor.user())
  {
    username = Meteor.user().username;
  }

  var message = $("#message").val();
```

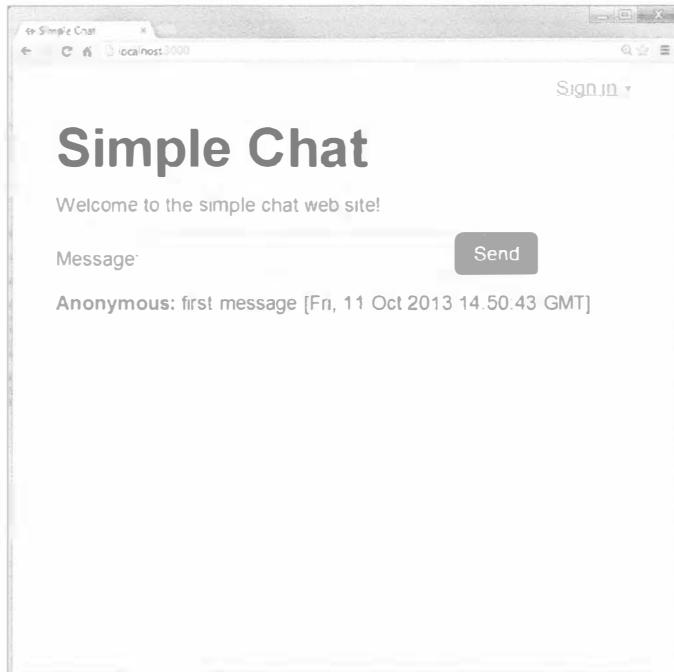
```
if (message != '') {  
    Messages.insert({  
        name: username,  
        message: message,  
        time: new Date().getTime()  
    });  
  
    $("#message").val("");  
}  
}
```

### Login-Bereich in die Webseite einfügen

Das alte Eingabefeld für den Benutzernamen entfällt. Im Template fügen wir an einer passenden Stelle die folgende Direktive ein:

```
{{login-buttons}}
```

Meteor rendert den vordefinierten Anmeldebereich. Ist kein Benutzer angemeldet, visualisiert das Modul eine Aufforderung zum Anmelden. Öffnet man den Bereich, erscheinen Felder für Login, E-Mail-Adresse und Passwort. Die Felder lassen sich über die Konfiguration steuern. Außerdem kann man sich neu registrieren, wenn man noch kein Login besitzt. Die Serie der Screenshots zeigt die wichtigsten Funktionen.



**Bild 4.5:** Aufforderung zum Anmelden in der oberen Ecke des Fensters.



Bild 4.6: Geöffneter Login-Bereich für die Anmeldung.



Bild 4.7: Geöffneter Login-Bereich für die Erzeugung eines neuen Accounts.



**Bild 4.8:** Login-Bereich für das Abmelden und Ändern des Passwortes.



**Bild 4.9:** Anfordern eines neuen Passwortes per E-Mail.

Hier ein Beispiel für den Zugriff direkt auf das User-Objekt per Collection. Die Befehle können in der Browser-Konsole innerhalb einer Meteor-Applikation getestet werden.

```
var userArray = Meteor.users.find({}).fetch();
JSON.stringify(userArray, null, 4)
"[
  {
    "_id": "DY6zt5SuP72M4z467",
    "username": "hspindler",
    "emails": [ {
      "address": "heiko.spindler@hirnsport.de",
      "verified": false
    } ]
  } ]"
```

#### 4.4.4 Schritt 4: Externe Login-Provider

Das Login-Modul von Meteor hat noch mehr zu bieten. Dabei werden externe Provider für die Benutzeridentifikation herangezogen. Beeindruckend ist ebenfalls die Liste der zusätzlichen externen Login-Provider. Als Entwickler hat man die Qual der Wahl: Von Github über Twitter, Facebook bis Google sind fast alle namhaften Anbieter vertreten und lassen sich leicht konfigurieren. Weitere Provider, die den OAuth2-Standart unterstützen, lassen sich ebenfalls aufnehmen.

Die API hierfür ist auf der Meteor-Seite dokumentiert. Meteor benutzt das Secure Remote Password Protokoll (SRP) für die Kommunikation mit den Providern. Passwörter werden nur verschlüsselt übertragen.

##### **Externen Login-Provider aufnehmen**

Als nächsten Schritt fügen wir über die Meteor-Konsole die beiden Provider, Google und Github, hinzu:

```
meteor add accounts-google
meteor add accounts-github
```

Die Basiskonfiguration für das Accounts-Modul existiert schon in der Datei `client.js`. Für die neuen Provider sind zusätzliche Einträge notwendig:

```
Accounts.ui.config({
  requestPermissions: {
    github: ['user', 'repo']
  },
  requestOfflineToken: {
    google: true
  },
  passwordSignupFields: 'USERNAME_AND_OPTIONAL_EMAIL'
});
```

Welche Felder genau notwendig sind, sollte man der Dokumentation entnehmen.

Vor dem Einsatz muss man die eigene Applikation bei dem gewünschten Provider anmelden. Am leichtesten geht dies, indem man den Provider wie in der Meteor-Dokumentation konfiguriert und erstmalig aufruft. Daraufhin erscheint ein Popup mit den Schritten zur Konfiguration des Dienstes. Diese Schritte sind je nach Provider etwas unterschiedlich.

Die folgenden Screenshots zeigen die Konfiguration bei Github:

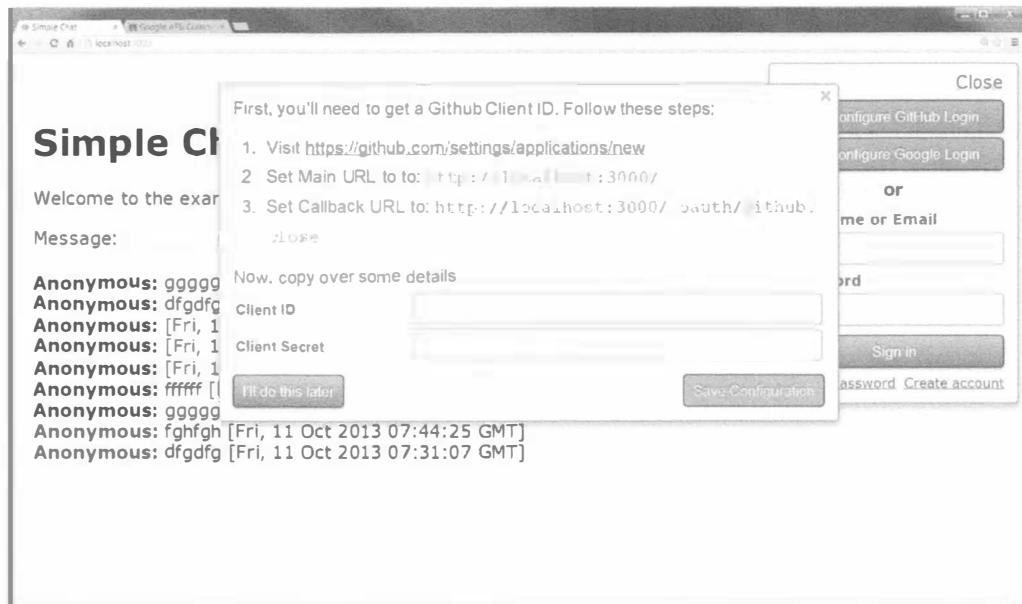


Bild 4.10: Konfigurationsdialog beim erstmaligen Anmelden mit einem Login-Provider.

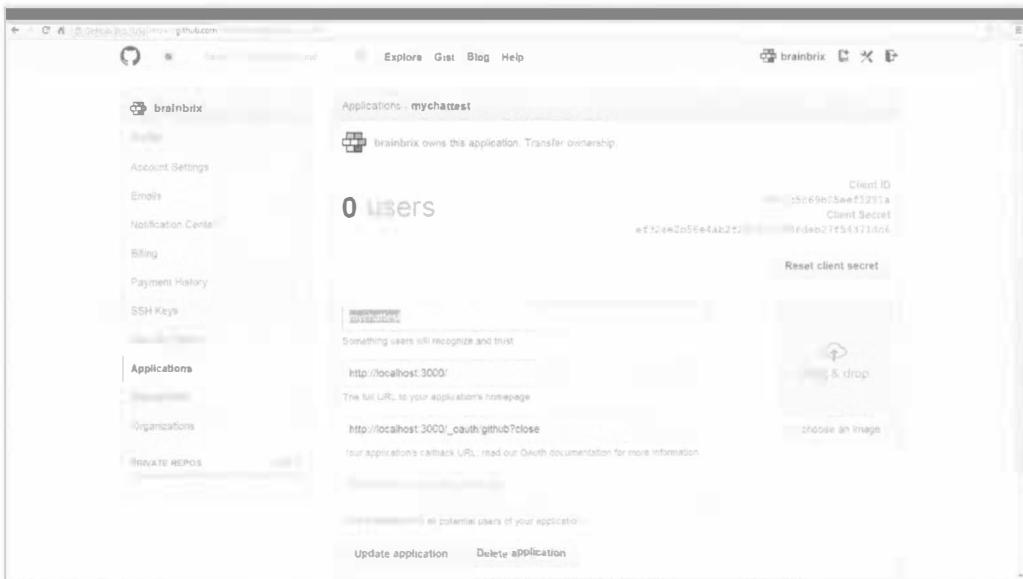


Bild 4.11: Anmeldung der Applikation bei Github (Liefert ClientID und Secret Key).

Typischerweise erhält man eine Client-Identifikation und einen Schlüssel, die die eigene Applikation identifizieren und die beim ersten Login angegeben werden müssen. Damit ist die Applikation beim Provider bekannt. Bei Github erhält man als Applikationsbetreiber ebenfalls eine Auswertung, wie viele Benutzer die Anwendung nutzen.

Nach dieser einmaligen Einstellung erscheint beim Anmelden die Provider-typische Login-Seite. Bei manchen Providern werden alle Rechte und Informationen aufgelistet, die die anfragende Applikation zu sehen bekommen möchte. Als Benutzer muss man diese Information bestätigen.

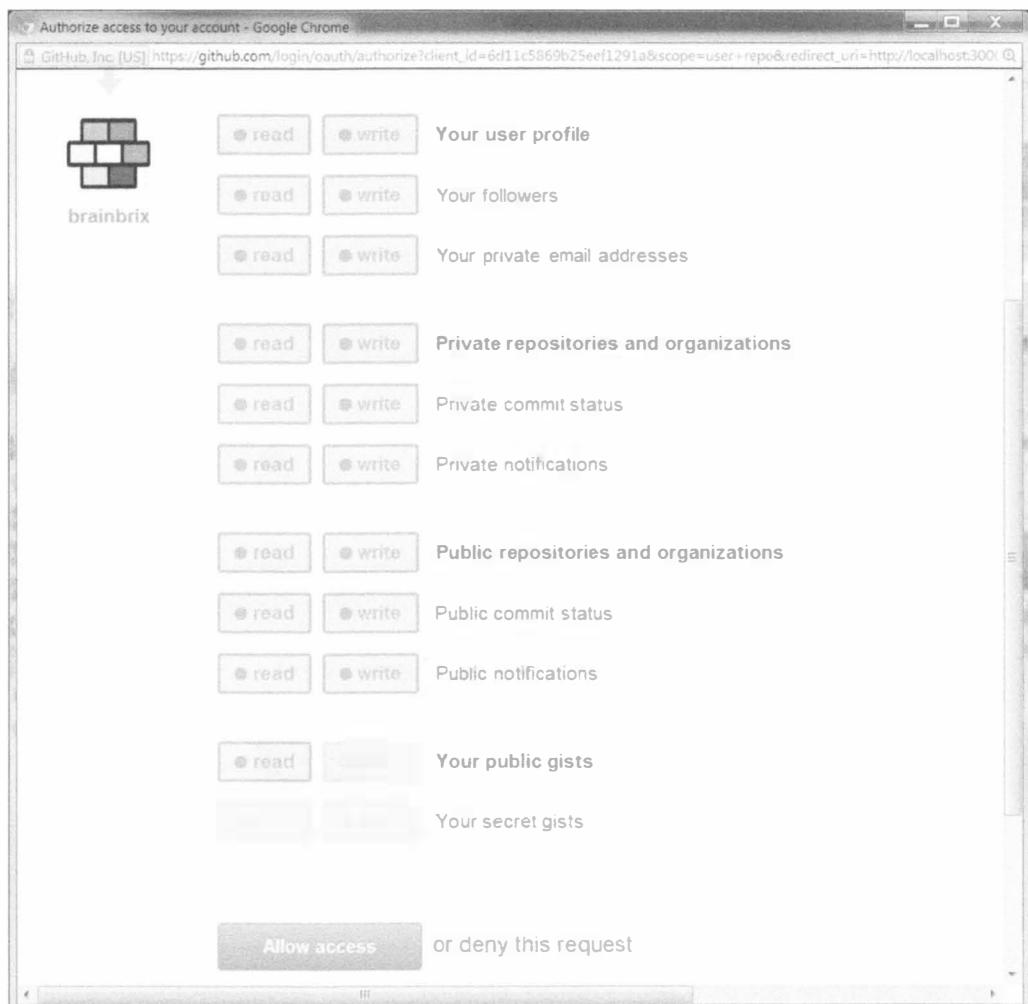
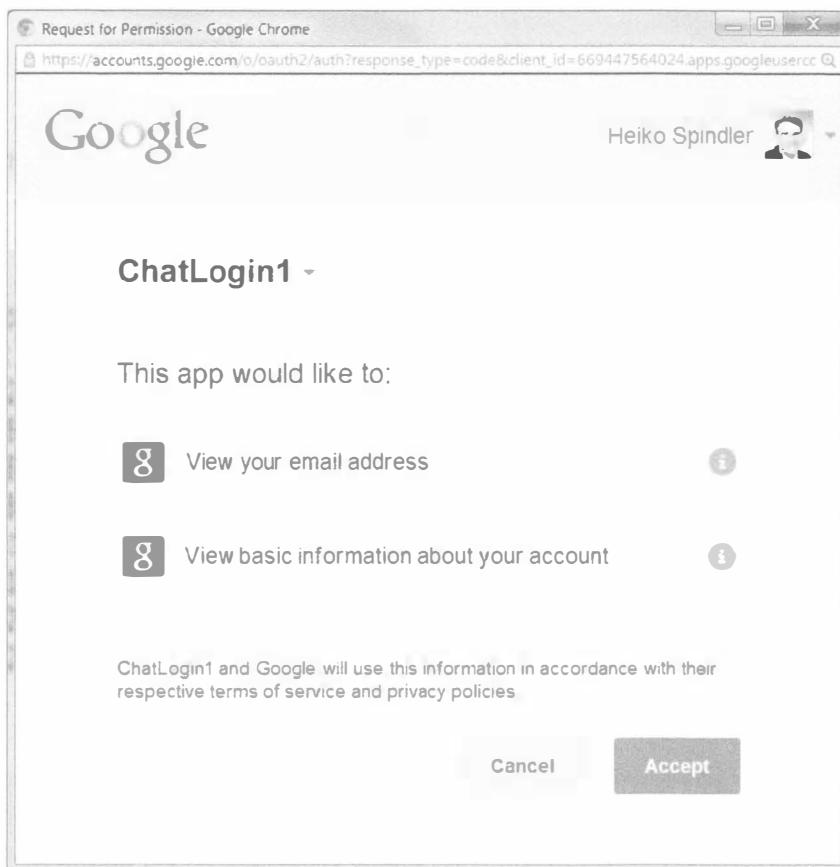


Bild 4.12: Detaillierte Berechtigungen für die Applikation auf ein GitHub-Account.



**Bild 4.13:** Bei der Anmeldung fragt der Login-Provider (Google), ob der Benutzer dem Zugriff auf Profil-Informationen zustimmen möchte.

Ist der Benutzer bei dem dem Provider entsprechenden OAuth2-Protokoll schon angemeldet und die Session ist noch aktiv, wird er in die Applikation ohne weitere Aktion oder Meldung sofort eingeloggt.

Die Darstellung der Login-Komponente zeigt nach erfolgreicher Anmeldung den Log-innamen oder eventuell die E-Mail-Adresse, je nach Konfiguration. Nicht zuletzt kann sich ein angemeldeter Benutzer über die UI auch wieder abmelden.

## 4.5 Ausflug: Persistenz mit MongoDB

MongoDB ist ein Vertreter der NoSQL-Datenbanken. Das heißt, es werden nicht die bekannte Structured Query Language (SQL) und ein relationales Modell vorausgesetzt. MongoDB ist außerdem ein Dokumentstore und speichert beliebige Strukturen.

Die Datensätze müssen nicht alle derselben Struktur genügen. Dieser Ansatz hat Vorteile, aber auch Nachteile.

MongoDB ist ein Open-Source-Projekt und steht unter der AGPL-Lizenz. Kommerzieller Support ist von der Firma MongoDB Inc. (früherer Name: 10gen) verfügbar. Mehr Informationen gibt es unter der Adresse: [www.mongodb.org](http://www.mongodb.org).

#### Der Begriff »NoSQL«

Der Begriff »NoSQL« kommt ursprünglich von der Bezeichnung »Not only SQL«, und besagt, dass neben SQL und relationalen Datenbanken neue Konzepte existieren können. Leider wird der Begriff oft missverständlich eingesetzt: SQL und relationale Ansätze wären verboten oder schlecht.

### Mapping-Begriffe

Die folgende Tabelle listet einige wesentliche Konzepte der relationalen Welt auf und stellt die Konzepte eines typischen Documentstores gegenüber.

Begriffe der relationalen Welt	Entsprechende Konzepte eines Documentstores
Tabelle (View)	Collection
Zeile (Row)	Dokument
Spalte (Column)	Attribut / Feld in einem Dokument
Index	Index
Join (Assoziation)	Es gibt keine direkte Entsprechung im NoSQL-Umfeld. Eingebettete Dokumente kommen dem Ansatz nahe, bilden aber eine fest und statische Beziehung.
Fremdschlüssel	Referenziertes Dokument

### Flexibles Datenmodell

Dokumentenorientierte Systeme legen fachliche Daten, die zusammengehören, in einem Dokument ab. Das Dokument enthält die Datenfelder gemeinsam mit der Struktur. Dabei müssen die Dokumente nicht einem exakten einheitlichen Aufbau folgen, sondern können variieren. Eine angemessene Struktur kann dem Anwendungsfall folgen und nicht einer denormalisierten Struktur, die einer relationalen Datenbank entgegen kommt.

MongoDB legt die Dokumente in Form von JSON-Dokumenten ab und passt damit sehr gut in ein JavaScript-Umfeld. Hier ein Beispieldokument, wie eine Adresse aussehen könnte:

```
{
  Name: "Mustermann",
  Vorname: "Max",
  Alter: 34,
```

```
Hobbys: ["Kino", "Volleyball", "Programmieren" ],
Adresse: {
  Wohnort: "Frankfurt",
  PLZ: 69000,
  Strasse: "Goethestrasse 17"
}
}
```

Person und Adresse sind nicht in separaten Tabellen abgelegt, sondern im selben Dokument. Der Zugriff erfolgt auf alle Daten direkt. Ein Nachladen durch zusätzliche Zugriffe oder Joins über mehrere Tabellen entfällt.

Der Nachteil, den man sich einhandelt: redundante Speicherung und weniger Effizienz im Umgang mit Speichermedien. Gerade die Kosten für Speichermedien sind in den letzten Jahren deutlich gefallen und machen diese neuen Ansätze attraktiver. Damit gewinnt die Strategie, lieber mehr zu speichern, auch wenn es (noch) nicht klar ist, ob und wie die Daten sinnvoll nutzbar sind.

Im Beispiel von oben würden die Namen der Hobbys oder die Adresse immer wieder aufs Neue in anderen Datensätzen zusätzlich abgelegt.

Es ist zwar möglich, Dokumente zu referenzieren und damit eine Trennung der Entitäten zu erreichen, für vollständig strukturierte und normalisierte Daten sollte aber eine relationale Datenbank zum Einsatz kommen.

Manche komfortable Funktion von relationalen Datenbanksystemen sucht man in NoSQL-Datenbanken vergebens:

Das Fehlen einer festgelegten Struktur führt zu mehreren Validierungen und Prüfungen, die in die Applikationsschicht delegiert werden. Sonst kann leicht Wildwuchs entstehen. Die einzelnen Collections sollten thematisch sinnvoll abgegrenzt sein und über die Zeit nicht für andere Zwecke missbraucht werden.

NoSQL-Systeme bieten meist keine Transaktionen. Die strenge Datenkonsistenz wird zugunsten einer deutlich besseren Verteilung und Skalierung der Systeme aufgegeben.

### CAP-Theorem

Wer tiefer in diese theoretische Diskussion einsteigen möchte, findet unter der Beschreibung des CAP-Theorems weitere Hintergründe:

[de.wikipedia.org/wiki/CAP-Theorem](https://de.wikipedia.org/wiki/CAP-Theorem)

### MongoDB-Collections in Meteor

Meteor integriert in der aktuellen Version MongoDB für die Datenhaltung. Ziel ist es, die Verarbeitung der serverseitigen Daten so transparent und so einfach wie möglich zu gestalten. Hierfür bringt Meteor für den Client eine Datenbank-API (mit dem Namen »Minimongo«) mit, die vergleichbare Befehle wie auf dem Server anbietet. Die Synchronisierung der Datenbankeinträge zwischen Client und Server ist verborgen.

Collections lassen sich leicht erstellen, wie es die Beispiele im Folgenden mehrfach aufzeigen werden. Eine zu persistierende Collection erhält einen eindeutigen Namen, übergeben im Konstruktor:

```
Messages = new Meteor.Collection('messages');
```

Temporäre Collections kennt Meteor ebenfalls: Ruft man den Konstruktor ohne Angabe eines Namens auf, werden die Einträge nicht automatisch gespeichert. Diese Art eignet sich gut, um einem Template Daten bereitzustellen:

```
var tempCollection = new Meteor.Collection();
```

Eine Meteor-Collection kennt die wichtigsten Datenbankfunktionen, wie `find()` bzw. `findOne()`, `insert()`, `update()` und `remove()`. Das ist trotzdem nur ein kleiner Teil der vollständigen MongoDB-Funktionen. Es ist wahrscheinlich, dass in Zukunft weitere Funktionen hinzukommen.

### Zugriffsrechte auf Daten kontrollieren

In der Grundeinstellung markiert Meteor alle Collections mit der Eigenschaft `insecure`. Das hat zur Folge, dass jeder User auf jedem Client Zugriff auf den vollständigen Datenbestand erhält. Das schließt sogar anonyme, nicht-authentifizierte und nicht-autorisierte Nutzer ein. Die Zugriffsrechte erstrecken sich auch auf die Manipulation der Datensätze und stellen damit natürlich ein enormes Sicherheitsproblem dar.

Für einen Prototyp während der Entwicklung ist das Verhalten vielleicht akzeptabel. Spätestens für eine produktive Version muss der Zugriff aber eingeschränkt werden.

Die Meteor-Collections bieten Möglichkeiten, den Zugriff einzuschränken und genau zu spezifizieren. Eine interessante Möglichkeit ist es, auf Datensatzebene durch Regeln den Zugriff explizit zu erlauben oder zu verbieten.

Die Allow-Konfiguration beschreibt dabei, wer welche Operation auf welcher Collection ausführen darf. Das Gegenstück verbietet mit der Deny-Konfiguration den Zugriff. Das folgende Beispiel zeigt den theoretischen Einsatz der Funktionen:

```
Messages.allow({
  insert: function (userId, msg) {
    // nur angemeldete Benutzer dürfen Nachrichten einfügen.
    return (userId && msg.owner == userId);
  },
  fetch: ['owner']
});

Messages.deny({
  remove: function (userId, msg) {
    //Eine gesperrte Nachricht darf nicht entfernt werden.
  }
});
```

```

    return msg.locked;
},
fetch: ['locked']
});

```

## 4.6 Beispiel: Kollektives Whiteboard

Im nächsten Beispiel erstellen wir ein kollektives Whiteboard, mit dem mehrere Benutzer gleichzeitig zeichnen können. An der Applikation lassen sich einige erweiterte Fähigkeiten von Meteor gut präsentieren.

### 4.6.1 Publish/Subscribe

In der Grundeinstellung verteilt Meteor alle Datensätze einer Collection an alle Clients (`autopublish` Flag). Das ist natürlich nicht immer sinnvoll. Deshalb gibt es in der API die Möglichkeit, einen Publish/Subscribe-Mechanismus zu nutzen.

Grundsätzlich kennen Client und Server die Collections, wie bisher. Über die Methoden `Meteor.publish()` veröffentlicht der Server genauer, welche Daten verteilt werden sollen. Der Client abonniert mit dem Befehl `Meteor.subscribe()` eine solche Veröffentlichung.

Somit kann gesteuert werden, dass nicht alle Änderungen einer Collection an alle Clients propagiert werden müssen und, dass nicht alle Felder einer Collection öffentlich sind.

Passend zum Chat-Beispiel könnte der Server eine Methode anbieten, die nur die neuesten zehn Nachrichten veröffentlicht:

```

Meteor.publish("lastMessages", function() {
  return Messages.find({}, {limit: 10, sort: {t: -1}});
});

```

Das Selektor-Objekt als zweiter Parameter in der Find-Anweisung filtert auf die zehn letzten Nachrichten und liefert diese in absteigender Reihenfolge zurück. Der Client kann diese Collection unter dem Bezeichner `lastMessages` abonnieren:

```
Meteor.subscribe("lastMessages");
```

Das Vorgehen hat zwei weitere Vorteile:

- ➊ Die Anzahl der Daten wird reduziert. Es muss nicht die vollständige Collection verteilt werden.
- ➋ Die Einträge landen bei allen Empfängern schon in der erwarteten Sortierung. Diese Aufgabe müsste sonst jeder Klient zusätzlich selbst durchführen (siehe Chat-Beispiel).

## 4.6.2 Remote Procedure Calls

Eine weitere Möglichkeit, um Berechnungen und Logik vom Client auf den Server zu verlagern, sind entfernte Methodenaufrufe, auch »Remote Procedure Calls« (RPC) genannt. Der Server bietet Methoden an, die ein Client ansprechen kann. Das Meteor-Framework versteckt viel Overhead vor dem Entwickler, indem es das Definieren und Aufrufen der Methoden fast wie lokale Aktionen erscheinen lässt.

Auf dem Server erstellt man ein Methoden-Objekt mit der Meteor-Funktion: `Meteor.methods()`. Jedes Attribut definiert eine Funktion, die automatisch für entfernte Clients aufrufbar ist:

```
Meteor.methods({
  getRandomNr: function() {
    return Math.rand();
  },
  getCurrentTime: function() {
    return new Date();
  }
});
```

Der Client benötigt keine Beschreibung der Schnittstelle oder Konfiguration. Die Methoden sind direkt erreichbar. Ein synchroner Aufruf erfolgt mit der Meteor-Funktion: `Meteor.call()`:

```
var returnValue = Meteor.call( "getCurrentTime" );
```

Synchrone Aufrufe warten so lange, bis das Ergebnis vom Server zurückkommt. Das kann natürlich negative Folgen haben. Schon wenige Sekunden Wartezeit sind meist nicht akzeptabel. Meteor bietet deshalb zusätzlich asynchrone Kommunikation an.

Der Rückgabewert entfällt bei dem asynchronen Aufruf und Meteor erwartet eine Callback-Funktion, die auf die später eintreffende Serverantwort reagiert:

```
Meteor.call( "getCurrentTime", callBackFunction );
```

Die Signatur der Callback-Funktion definiert zwei Parameter: Im Fehlerfall ist nur das Fehler-Objekt gefüllt, im Erfolgsfall nur das Result-Objekt. Meteor liefert nie beide Parameter gleichzeitig. Eine Callback-Funktion könnte so aussehen:

```
callBackFunction = function(errorValue, resultValue) {
  if (errorValue)
  {
    // Aktion im Fehlerfall
  } else
  {
    // Es ist keine Fehler aufgetreten.
    // resultValue enthält evtl. Return-Wert vom Server
  }
};
```

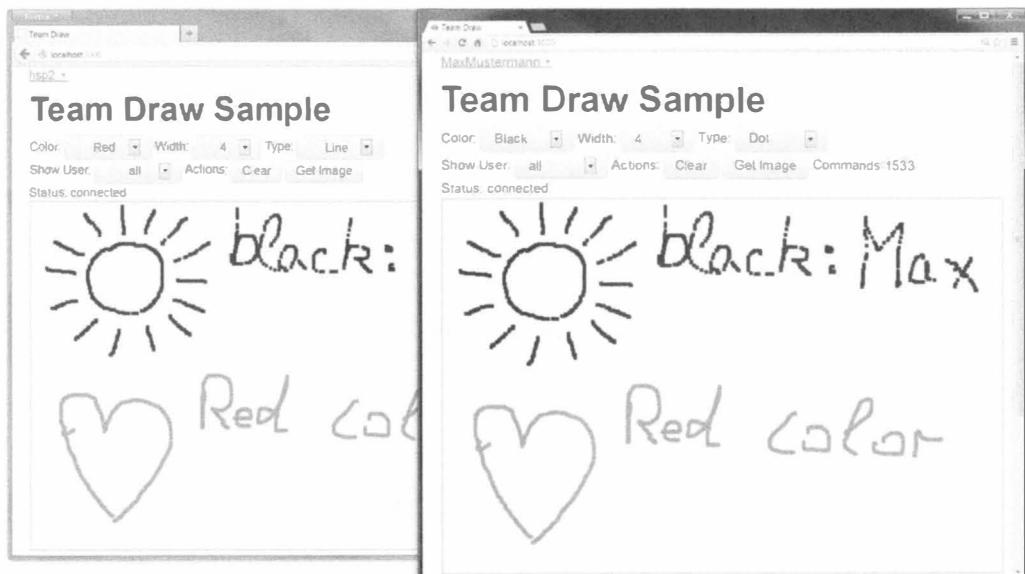
### Latency Compensation: Resultate ohne Wartezeit

Registriert man die entfernte Methode nicht nur auf dem Server, sondern parallel mit dem gleichen Namen auf dem Client, ruft Meteor beide Funktionen parallel auf. Die Client-Funktion antwortet sofort und könnte ein angenähertes oder vorläufiges Ergebnis anzeigen. Das endgültige Resultat des Servers trifft (kurze) Zeit später ein und Meteor ersetzt zum Beispiel den ersten angenäherten Wert in der Webseite. Für einen Benutzer fühlt sich die Applikation damit meist sehr aktiv und flüssig an, wenn man die Wartezeit (Latenz) der Kommunikation versteckt.

#### 4.6.3 Die Anforderungen des Whiteboards

Wir definieren einen Zeichenbereich, auf dem der Benutzer einfache Freihand-Zeichnungen erstellen kann. Es soll möglich sein, die Farbe und die Breite der Linien zu wählen. Außerdem kann man wählen, ob man einzelne Punkte zeichnet oder eine durchgezogene Linie.

Alle Benutzer der Webseite sehen denselben Arbeitsstand und können mit- und weiterzeichnen. Die eigenen künstlerischen Leistungen werden an alle anderen Teilnehmer live übertragen.



**Bild 4.14:** Mehrere Teilnehmer zeichnen gemeinsam auf der Zeichenfläche.

Alle Benutzer, die die Seite öffnen, erhalten die komplette Zeichnung mit allen Elementen angezeigt und können parallel mitmalen. Trotzdem kann man die Zeichnung jedes Benutzers isoliert betrachten, durch die Selektion eines Benutzernamens in der entsprechenden Auswahlbox.

Das Zeichenfeld kann gelöscht werden und wir wollen die Zeichnung über das Kontextmenü des Browsers lokal als Bild im Format PNG speichern können.

Die Applikation, die wir erstellen, ist sicherlich nicht direkt produktiv, trotzdem hat sie einen ernsthaften Hintergrund. Für verteilte Projektmitschriften könnte es sehr sinnvoll sein, eine solche gemeinsame Zeichenfläche für das Erstellen und Diskutieren von Ideen zu haben. Ein konkreter Anwendungsfall könnte sein, zusammen an einem Layout für eine Applikation zu arbeiten und Änderungen sofort für die anderen Teilnehmer zu visualisieren.

#### 4.6.4 Die Umsetzung

Ähnlich zum ersten Meteor-Beispiel erstellen wir ein leeres Projekt mit den Basis-Modulen:

```
> meteor create teamdraw
> cd teamdraw
> meteor add bootstrap
> meteor add accounts-ui
> meteor add accounts-base
> meteor add accounts-password
> meteor add accounts-github
```

Bevor wir uns in die Implementierung stürzen, ändern wir die Verzeichnisstruktur in folgende Form und legen initial folgende Dateien an:

```
/teamdraw
  /client
    canvaslib.js
    teamdraw.js
    teamdraw.css
    teamdraw.html
  /common
    collections.js
  /public
  /server
    teamdraw.js
```

#### Das HTML5-Canvas-Element

Die Umsetzung setzt neben Meteor auf grundlegende HTML5-Fähigkeiten. Eine ist das Canvas-Element, das alle aktuellen Browser anbieten.

Ähnlich zu den eigengebetteten SVG-Dokumenten, die wir im zweiten Kapitel untersuchten, bietet das Canvas einen Zugang zu grundlegenden Zeichenfunktionen. Es gibt einen wesentlichen Unterschied: Das Canvas kann ebenfalls per JavaScript-Befehl gestaltet werden, aber die Elemente liegen nicht in einem DOM-Baum, sondern ver-

ändern direkt den Zeichenbereich. Somit ist das einmal Gezeichnete nicht nachträglich veränderbar, es sei denn, man zeichnet etwas anderes darüber.

Das folgende Beispiel zeigt eine kleine Auswahl der Funktionen und außerdem, wie das Canvas-Element in der HTML-Seite eingebettet ist:

```
<!DOCTYPE html>
<html>
<body>
<canvas id="myCanvas" width="300" height="200" style="border:1px solid #9a9a9a;">
Sorry, HTML canvas is not supported by your browser.
</canvas>

<script>
var canvas = document.getElementById("myCanvas");
var context = canvas.getContext("2d");
context.fillStyle="#0000FF";
context.fillRect(50,50,150,75);

context.fillStyle="#FF00FF";
context.font="50px Arial";
context.fillText("Hello Canvas",10,75);

context.fillStyle="#FF0000";
context.beginPath();
context.arc(95,120,40,0,1.1*Math.PI);
context.fill();
</script>
</body>
</html>
```

Im HTML-Bereich existiert ein Canvas-Element mit dem Identifier: `myCanvas`. Im Script-Bereich holt man eine Referenz auf das Canvas-Element in der HTML-Seite über diesen Namen. Auf dieses kann nicht direkt gezeichnet werden. Hierfür ist ein Kontext notwendig, der durch Anweisung `getContext("2d")` erstellt wird. Auf diesem Kontext arbeiten typische Zeichenbefehle, wie `fillRect()`, `drawImage()` oder `arc()`. Außerdem können neben der Füllfarbe, Schriftart, Linienfarbe, Liniendicke und -art viele weitere Attribute für die Darstellung festgelegt werden. Die Abbildung zeigt das Ergebnis des Beispiels im Browser:



**Bild 4.15:** Ergebnis der einfachen Zeichnung im Browser.

Viele Bücher und Webseiten erklären die Zeichenbefehle des Canvas sehr umfassend. Im Verzeichnis sind dazu einige gute Quellen aufgeführt.

### Speicherung der Daten

Die Daten liegen, wie schon im ersten Beispiel, wieder in Meteor-Collections. Diese liegen wieder in einer separaten Datei, `commons/collections.js`, damit sie sowohl Client als auch Server bekannt sind:

```
users = new Meteor.Collection('userCollection');
commands = new Meteor.Collection('commandsCollection');
```

Die erste Collection `users` verwaltet, wie der Name schon verrät, die teilnehmenden Benutzer. Als einzige Angabe legen wir den Loginnamen ab, damit wir eine Liste der aktiven Benutzer anzeigen können. Als Alternative könnten wir sogar die Standard-Collection von Meteor für das Accounts-Modul nutzen: `Meteor.users`. Diese enthält schon den Namen als Attribut an jedem Eintrag.

Die zweite Collection `commands` nimmt die Zeichenkommandos auf. Hier sollten wir etwas genauer untersuchen, welche Inhalte gespeichert werden.

### Zeichenkommandos

Für das Speichern und Verteilen der Zeichenkommandos nutzen wir eine persistente Collection. Es kommt uns sehr entgegen, dass die Einträge in der Collection unterschiedlich sein können. Zwar nutzen wir vorerst nur zwei verschiedene Kommandos, aber es könnten später weitere hinzukommen:

Die folgende Tabelle erklärt die Attribute, die bei jedem Kommando gespeichert werden:

Attribut	Attributname	Bedeutung
Command	cmd	Das Zeichenkommando: dot oder line
Koordinaten	x, y, x2, y2	Für einen Punkt reichen x und y aus. Bei einer Linie kommen die Endpunkte x2 und y2 hinzu.

Attribut	Attributname	Bedeutung
Farbe (Color)	c	Enthält eine Zeichenkette mit den Farbwerten. Hier einige Beispiele: #000 : Schwarz #FFF: Weiß #F00: Rot #888: Grau
Breite (Width) der Linie oder des Punktes	w	Dieser numerische Wert definiert die Breite der Linie oder des Punktes. Über die UI sind die Werte 2, 4, 8, 16 oder 20 einstellbar.
Benutzer (Owner), der dieses Kommando gezeichnet hat	o	Entspricht direkt dem Loginanamen, mit dem der Benutzer angemeldet ist.
Timestamp	t	Gibt an, zu welchem Zeitpunkt (Timestamp) ein Kommando gezeichnet wurde. Gespeichert wird ein Long-Wert. Definiert die Reihenfolge der Elemente.

Die beiden folgenden Beispiele zeigen Kommandos mit sinnvollen Attributwerten, wie sie in der Collection (`commandos`) gespeichert sein könnten:

Zeichenobjekt	Beispiele der Daten in der Collection
Punkt	{ cmd: "dot", x: 10, y: 20, w: 4, c: "#000", t: 1234567, o: "HeikoSpindler" }
Linie	{ cmd: "line", x: 10, y: 20, x2: 150, y2: 220, w: 4, c: "#00F", t: 1234599, o: "HeikoSpindler" }

## Canvas-Bibliothek

Die Funktionen für das Zeichen lagern wir in eine eigene Klasse mit dem Namen Canvas aus:

```
Canvas = function ( canvasId ) {
    var self = this;
    var context;

    // Filter des aktiven Benutzers
    var name;

    // Timestamp der Elemente, die schon gezeichnet sind.
    var lastTimeStamp = 0;

    /* Position des Canvas in der HTML-Seite
       für die Anpassung der Koordinaten von Events */
    var offset_left = 0;
    var offset_top = 0;

    var createContext = function() {

        var theCanvas = document.getElementById( canvasId );
        context = theCanvas.getContext ("2d");

        // Offset des Canvas in der HTML-Seite berechnen
        for (var o = theCanvas; o ; o = o.offsetParent) {
            offset_left += (o.offsetLeft - o.scrollLeft);
            offset_top  += (o.offsetTop - o.scrollTop);
        }
        offset_left = Math.floor( offset_left );
        offset_top = Math.floor( offset_top );
    };
    createContext();

    self.clear = function() {
        context.fillStyle="#FFF";
        context.fillRect(1,1,600,400);
    };

    self.getContext = function()
    {
        return context;
    }

    self.setTimeStamp = function(val)
    {
        lastTimeStamp = val;
    }
}
```

```
}

self.setName = function(val)
{
  self.name = val;
}

...
```

Die Methode `createContext()` initialisiert das Objekt, identifiziert das HTML-Canvas-Objekt im HTML-Dokument und erzeugt den Zeichenkontext. In der Methode wird außerdem die Position (`offset`) des Zeichenbereichs im HTML-Dokument erfragt. Das ist notwendig, da wir später von den Maus-Nachrichten immer Koordinaten relativ zur linken, obere Ecke des Dokuments erhalten werden. Unsere Aufgabe ist es, aus den globalen Koordinaten relative Informationen zum Zeichenbereich auszurechnen.

Die Funktion `clear()` erklärt sich aus ihrem Namen: Sie löscht den kompletten Zeichenbereich, indem sie die Fläche weiß einfärbt. Die folgenden einfachen Set- und Get-Funktionen dienen lediglich der Kommunikation mit dem Canvas-Objekt.

Interessanter ist das Herzstück des Canvas-Objektes: Die `draw()`-Methode erhält zwei Parameter, zum einen die Collection mit allen Zeichenkommandos, die wir auf den letzten Seiten besprochen haben, zum anderen ein Kennzeichen, ob die Zeichenfläche zuvor gelöscht werden soll. Die Funktion selbst prüft, ob überhaupt in der Collection etwas zu zeichnen ist. Ist die Collection leer oder wird das Löschen explizit gefordert, löschen wir den Zeichenbereich.

In der folgenden Schleife werden alle Zeichenkommandos iteriert. Dabei wird jedes Kommando auf zwei Bedingungen geprüft:

- Ist es über den ausgewählten Benutzer sichtbar:  
Steht der Filter auf »All«, sind die Zeichenkommandos von allen Benutzern sichtbar. Wurde ein Benutzer explizit gewählt, sollen alle Zeichenelemente von anderen Nutzern weggefiltert und nicht dargestellt werden.
- Passt der Zeitstempel:  
Ist der Zeitstempel des Elementes älter (kleiner) als der letzte gezeichnete Zeitstempel, muss das Element schon früher auf der Zeichenfläche visualisiert sein und kann jetzt ignoriert werden.

Gerade die letzte Bedingung spart uns das Zeichen von sehr vielen Elementen, denn wir zeichnen nur neue Elemente, seitdem die Daten zuletzt ausgelesen wurden. In den meisten Fällen reicht das vollkommen aus.

Der Zeichenbereich muss nur in wenigen Fällen neu gezeichnet werden, zum Beispiel, wenn wir den Benutzer-Filter ändern und die Zeichnung eines anderen Benutzers sehen wollen.

Zum Schluss folgt eine Fallunterscheidung mit den einzelnen Zeichenkommandos, wie Punkte (`dot`) und Linien (`line`). Hier erfolgt das eigentliche Darstellen. Je nach Zeichenoperation sind unterschiedliche Farb- und Linienstile einzustellen:

```
...
/** 
 * Zeichnet die Kommandos.
 *
 * @param data Collection der Zeichenkommandos
 * @param doClear True: Zeichenfeld löschen
 */
self.draw = function(data, doClear) {
    var i;

    if (data.length < 1) {
        self.clear();
        return;
    }

    if ( doClear )
    {
        self.clear();
    }

    for( i = 0; i < data.length;i++)
    {
        // Nach Benutzer filtern
        if (self.name !== "all")
        {
            if (data[i].o !== self.name)
            {
                continue;
            }
        }
        // Nach Timestamp filtern
        if ( (data[i].t < lastTimeStamp) && (!doClear))
        {
            continue;
        }

        if (data[i].cmd === "dot" )
        {
            context.lineWidth= data[i].w;
            context.fillStyle= data[i].c;
            context.fillRect(data[i].x, data[i].y, data[i].w, data[i].w);
        } else if (data[i].cmd === "line" )
        {
            context.lineCap= 'round';
            context.lineWidth= data[i].w;
            context.strokeStyle=data[i].c;
            context.beginPath();
        }
    }
}
```

```

        context.moveTo(data[i].x,data[i].y);
        context.lineTo(data[i].x2,data[i].y2);
        context.stroke();
    }
}
// Letzte gespeicherte Kommando merken.
lastTimeStamp = data[i-1].t;
};

}

```

Jetzt müssen wir im Client-Code das Canvas-Objekt richtig einsetzen und mit den anderen Programmteilen verbinden.

### Die Benutzeroberfläche im Template

Werfen wir zuerst einen Blick auf das Template und die Oberfläche unserer Applikation:

```

<head> <title>Team Draw</title> </head>

<body>
  <div class="container">
    {{loginButtons align="left"}}

    <div class="jumbotron">
      <h1>Team Draw Sample</h1><br />
    </div>
    <div class="container">
      {{> drawingSurface}}
      {{> canvas}}
    </div>
    <div class="footer"><p>&copy; HirnSport.de 2013</p></div>
  </div>
</body>
...

```

Neben den bekannten Eigenschaften, wie den Loginbuttons des Accounts-Moduls und einigen DIV-Klassen des Bootstrap-Frameworks, sind nur die beiden aufgerufenen Templates `drawingSurface` und `canvas` interessant.

Das erste Template (`drawingSurface`) definiert zunächst die UI-Elemente für die Wahl der Farbe, Linienbreite und des Zeichenmodus (Punkte oder Linien). Um die Liste der aktuell teilnehmenden Benutzer kümmert sich ein Sub-Template mit dem Namen `showUsers`:

```

...
<template name="drawingSurface">
  Color:
  <select class="color btn" id="color" name="color" size="1" width="100"

```

```

<option value="#000" >Black</option>
<option value="#F00" >Red</option>
<option value="#00F" >Blue</option>
<option value="#0F0" >Green</option>
<option value="#999" >Gray</option>
<option value="#FFF" >White</option>
</select>
Width:
<select class='linewidth btn' id="linewidth" name="linewidth" size="1" width="80" style="width: 80px">
<option value="2" >2</option>
<option value="4" >4</option>
<option value="8" >8</option>
<option value="16" >16</option>
<option value="20" >20</option>
</select>
Type:
<select class="type btn" id="type" name="type" size="1" width="100" style="width: 100px">
<option value="dot" >Dot</option>
<option value="line" >Line</option>
</select> <br>
{{> showUsers}}
Actions:
<input type="button" class="clearButton btn" value="Clear"/>
<span >Status: {{ status }} </span>
</template>
...

```

Das Sub-Template `showUsers` ist ausgelagert und kümmert sich nur um die Darstellung der Liste der angemeldeten Mitzeichner. Es definiert ebenfalls eine Auswahlliste. Der erste Eintrag mit dem Wert `all` ist statisch und soll die Zeichnungen aller Benutzer gleichzeitig visualisieren. Die restlichen Einträge kommen dynamisch in die Liste.

Eine Template-Funktion wird uns alle Benutzer als Collection liefern. Für jeden Benutzer erstellen wir einen Option-Tag in der Auswahlliste und tragen den Benutzernamen als Wert und für die Anzeige ein. Falls die Webseite neu gezeichnet wird, sollten wir die letzte Auswahl in der Liste wiederherstellen. Dazu gibt es mehrere Möglichkeiten:

Wir rendern das Attribut `selected` in dem Option-Tag. Das könnten wir mit einem If-Konstrukt von Handlebars erreichen oder, wie es im Beispiel unten realisiert ist, durch einen Handlebar-Helper. Dieser vergleicht die Variable `selectedName` (aus einer Hilfsfunktion des Templates, siehe Beschreibung des Clients) mit dem Element `name` aus der Iteration über alle User-Instanzen.

```

...
<template name="showUsers">
```

```

Show User:
<select class="showUser btn" id="showUser" name="showUser" size="1"
width="100" style="width: 100px">
  <option value="all" >all</option>
  {{#each users}}
    <option {{selected selectedName name}} value="{{name}}>
  {{name}}</option>
  {{/each}}
</select>
</template>
...

```

Das letzte Template unserer Applikation ist recht übersichtlich und enthält schließlich das Canvas-Element selbst. Wichtig ist natürlich, dass der Identifier (Attribut `id`) zu dem Aufruf aus dem JavaScript-File passt.

```

...
<template name="canvas">
  <div id='canvas'>
    <canvas width="600" height="400" id="mycanvas" style="border:2px solid
#d3d3d3;">
    </canvas>
  </div>
</template>

```

## Der Client-Code

Zuerst definieren wir einige Variablen. Das Canvas-Objekt, das unsere Zeichenfunktionen kapselt, erhält den Namen `canvas`. `Lastx` und `lasty` speichern die letzten Zeichenkoordinaten, die für Linien notwendig sind. `drawing` ist eine boolesche Variable, in der wir uns merken, ob der Benutzer gerade zeichnet, indem er die linke Maustaste gedrückt hat. In den drei Variablen `selectedColor`, `selectedWidth` und `selectedCmd` merken wir uns die aktuellen Einstellungen für die wichtigsten Auswahlfelder und belegen diese Werte sinnvoll vor:

```

var canvas;
var selectedColor = "000";
var selectedWidth = 2;
var selectedCmd = "dot";
var lastx = 0;
var lasty = 0;
var drawing = false;
...

```

Im Client werden zunächst die Funktionen zur Unterstützung der Templates definiert. Dazu zählt ebenfalls der selbst definierte Handelbar-Helper, der für das korrekte Select-Attribut in der Benutzer-Auswahl sorgt:

```
Template.showUsers.users = function () {
{
  return users.find({}, {sort: { name: 1}}).fetch();
};

Template.showUsers.selectedName = function () {
{
  return Session.get("selectedName");
};

Handlebars.registerHelper('selected', function(foo, bar)
{
  return foo == bar ? 'selected' : '';
});
```

Die Accounts-Konfiguration kennen wir schon aus dem letzten Beispiel:

```
// Settings for Accounts module
Accounts.ui.config({
  requestPermissions: {
    github: ['user', 'repo']
  },
  passwordSignupFields: 'USERNAME_AND_OPTIONAL_EMAIL'
});
```

Meteor bietet sowohl für den Server als auch für den Client eine `Startup`-Funktion. Diese eignet sich sehr gut für Initialisierungen. Wir erzeugen hier das Canvas-Objekt und definieren mit der Funktion `Deps.autorun()` die automatische Aktualisierung der Zeichnung, falls sich der Inhalt der Collection mit den Zeichenkommandos ändern sollte:

```
Meteor.startup( function() {
  canvas = new Canvas();

  Deps.autorun( function() {
    redraw(false);
  });
});
```

Das eigentliche Zeichen findet in der Funktion `draw()` statt. Die Zeichenkommandos werden ausgelesen und die komplette Collection (`data`) an das Canvas-Objekt übergeben.

```
var redraw = function(toClear)
{
  var data = commands.find({}).fetch();
  if (canvas) {
    canvas.draw(data, toClear);
  }
}
```

Wichtig ist, dass wir die Aktualisierungen der Daten vom Server erhalten. Dazu müssen wir uns auf die veröffentlichten Collections des Servers anmelden, durch den Befehl `Meteor.subscribe()`:

```
Deps.autorun( function () {
  Meteor.subscribe('commandsSubscription');
  Meteor.subscribe('usersSubscription');
});
```

Wenn sich ein neuer Benutzer einloggt und anfängt zu malen, ist dieses Ereignis für die anderen Clients von Interesse. Wir verpacken die Abfrage nach einem neuen Benutzer in die Methode `Deps.autorun()`. Damit wird die Logik von Meteor aufgerufen, wenn sich an den grundlegenden Informationen (in `Meteor.user()`) etwas ändert.

Falls wir jetzt einen angemeldeten Benutzer haben (`Meteor.user()` ist wahr), muss er sich gerade neu angemeldet haben. In diesem Fall holen wir uns den Namen des Benutzers mit der Methode `getUserName()` und delegieren das Anlegen des Benutzers an eine Server-Methode mit dem Namen `createUserIfNotExist`. Diese Methode wird asynchron aufgerufen. Wir wissen nicht, wann ein Ergebnis zurückkommt. Deshalb definieren wir eine CallBack-Funktion, die das Ergebnis zu einem späteren Zeitpunkt verarbeitet.

Im Else-Teil der Verzweigung löschen wir die Auswahl des sichtbaren Benutzers, da wir jetzt keinen angemeldeten Loginnamen mehr haben. Ein Benutzer hat sich offensichtlich gerade abgemeldet. Damit werden die Zeichnungen aller Nutzer angezeigt:

```
Deps.autorun(function()
{
  if (Meteor.user()) {
    Meteor.call( "createUserIfNotExist", getUserName(),
    _onCreateUserCallback);
  } else {
    Session.set("selectedName", "");
    Session.set("name", "");
  }
});
```

Die Callback-Funktion setzt im Erfolgsfall den Namen des angemeldeten Benutzers in der Session. Wir erhalten den Namen von der Server-Funktion wieder als Parameter `username` zurück. Am Ende der Funktion aktualisieren wir den kompletten Zeichenbereich:

```
_onCreateUserCallback = function(error, username) {
  Session.set("selectedName", username);
  Session.set("name", username);
  canvas.setName(username);
  redraw( true );
};
```

## Grundlagen zu Meteor-Eventmaps

Ein großer Teil des Clientcodes umfasst die Reaktion auf Events. Im ersten Meteor-Beispiel wurde die EventMap schon eingeführt: Eventmaps bieten eine elegante Methode, die Handler-Funktionen für Benutzeraktionen zu verwalten. Technisch gesehen ist eine Eventmap ein JavaScript-Objekt, dessen Attribute für Events und die jeweiligen Handler stehen. Die Eventmap kann für ein komplettes Template alle Nachrichten behandeln und so mehr Übersicht erreichen.

Jedes Attribut besteht aus den folgenden Elementen:

- Der Event-Typ definiert, auf welche Nachrichten geachtet werden soll. Alle anderen sind zu ignorieren.
- Der Selector wählt die Elemente im DOM-Baum der HTML-Seite aus, an denen die Nachricht auftreten könnte, zum Beispiel für den Identifier.
- Die Handler-Methode enthält das Applikationsverhalten, wie auf die Nachricht zu reagieren ist.

Hier einige Beispiele für EventMaps:

```
Template.templatename.subtemplate( {
  // Behandelt Click auf alle Elemente:
  'click': function (event) { ... },

  // Achtet auf einen Click an Elemente mit Selector
  // Class-Attribut '.accept'
  'click .accept': function (event) { ... },
});
```

Die wichtigsten Attribute eines Events zeigt die folgende Tabelle:

Attributname	Bedeutung
type	Der Typ des Events als String. Beispiele sind: click, mousedown oder keypress.
target	DOM-Element, an dem die Nachricht angestoßen wurde.
currentTarget	Das aktuelle DOM-Element, das die Nachricht behandelt, gemäß dem Selektor in der Eventmap.
which	Für Nachrichten mit Maustasten enthält das Attribut die exakte Nummer der Taste (1=Linke Taste, 2=Mittlere Taste, 3=Rechte Taste). Für Tastatur-Nachrichten enthält das Attribut den Keycode der Taste.

Überblick der wichtigsten Eventtypen:

Nachrichtentyp	Beschreibung
click, dblclick	Dieser Typ wird ausgelöst bei einem Klick (bzw. Doppelklick) auf ein DOM-Element. Falls ein Klick-Event auf einen Link abgefangen werden soll, so kann mit der Methode <code>event.preventDefault()</code> das Standardverhalten des Links verhindert werden.
focus, blur	Nachrichten dieser Typen werden versendet, wenn UI-Elemente den Eingabefokus erhalten (focus) oder verlieren (blur). Auch andere HTML-Elemente können durch das Setzen des Attributs <code>tabindex</code> den Fokus erhalten. Grundsätzlich ist zu beachten, dass sich die Browser leider unterschiedlich verhalten.
change	Die Selektion einer Liste oder einer Checkbox wurde geändert. Der Nachrichtentyp gilt nicht für Änderungen in Textfeldern.
mouseenter, mouseleave	Der Mauszeiger betritt oder verlässt ein HTML-Element.
mousedown, mouseup	Eine Maustaste wurde über einem HTML-Element gedrückt oder losgelassen.
keydown, keypress, keyup	Der Benutzer hat eine Taste gedrückt oder losgelassen. Die Nachricht <code>keypress</code> umfasst die beiden Nachrichten <code>keydown</code> und <code>keyup</code> für dieselbe Taste.
tap	Für Touch-Geräte, wie Tablets oder Smartphones, ist diese Nachricht ein Ersatz für ein Klick-Ereignis.

In der TeamDraw-Applikation müssen wir auf mehr Ereignisse reagieren als im Chat-Beispiel. Dadurch ist die Eventverarbeitung umfangreicher. Dabei können wir zwei Klassen von Nachrichten unterscheiden:

- ➊ Nachrichten von einfachen UI-Elementen, wie Buttons und Auswahllisten
- ➋ Nachrichten von Mausaktionen im Zeichenbereich

Für jeden Bereich legen wir eine eigene Eventmap an, da jeder Bereich einem anderen Template gehört. Diese Strukturierung dient dem besseren Überblick. Es könnten auch alle Events in einer gemeinsamen Eventmap liegen. Die erste Eventmap behandelt die Nachrichten von UI-Elementen:

```
// Eventmap for UI-Elements:  
Template.drawingSurface.events({  
  'click input.clearButton': function (event)  
  {  
    Meteor.call('clear', function() {  
      canvas.clear();  
    });  
    redraw( true );  
  },  
  'change #color': function (event)  
  {  
    Session.set( "color", event.target.value);  
  },  
  'change #linewidth': function (event)  
  {  
    Session.set( "width", event.target.value);  
  },  
  'change #type': function (event)  
  {  
    Session.set( "selectedCmd", event.target.value);  
  },  
  'change #showUser': function (event)  
  {  
    var showUser = event.target.value;  
    canvas.setTimeStamp(0);  
    canvas.setName( showUser );  
    redraw( true );  
    Session.set("selectedName", showUser );  
  }  
});
```

Die zweite Eventmap fokussiert sich auf die Mausnachrichten für unseren Zeichenbereich. Dabei spielen die globalen Variablen `lastx` und `lasty` eine entscheidende Rolle: Sie speichern für das Linienkommando die Position des letzten Linienabschnitts. Diesen brauchen wir, um die Liniensegmente miteinander zu verbinden. Vorher aktualisieren wir die Position des Canvas-Bereichs in der HTML-Seite, indem wir `canvas.offset` aktualisieren. Das ist notwendig, da wir von dem Event die Koordinaten relativ zum Seitenursprung erhalten.

Bei einer Mouseclick-Nachricht legen wir immer diese letzten Koordinaten an. Falls wir aktuell Linien zeichnen, so haben wir uns damit auf jeden Fall schon mal den Startpunkt gemerkt. Zeichnen wir gerade Punkte, spielen diese letzten Koordinaten keine Rolle. Wir legen sofort ein neues Zeichenobjekt an. Dazu nutzen wir die Hilfsfunktion: `insertCommand()`.

Bewegt der Benutzer die Maus über den Zeichenbereich, erhalten wir ein Mousemove-Event, das wir im zweiten Eventmap-Attribut behandeln. Für uns ist es nur von Interesse, wenn die linke Taste gedrückt ist. Andernfalls ignorieren wir die

Nachricht. In diesem Fall erzeugen wir ein neues Kommando mit den x- und y-Werten. Von welchem Typ das Kommando sein wird, entscheidet sich später.

Zeichnen wir im Moment Linien, setzen wir uns zusätzlich das Linienende in den Variablen mit x2 und y2 und merken uns wieder die aktuellen Koordinaten, falls die Linie in Zukunft weitergehen sollte. Jetzt können wir das Zeichenobjekt in die Collection speichern, indem wir der Methode `insertCommand()` das ausgewählte Kommando und das erzeugte Zeichenobjekt übergeben.

```
// Eventmap for Canvas-Element:  
Template.canvas.events({  
  'mousedown': function (event)  
  {  
    drawing = true;  
    canvas.offset = $('#canvas').offset();  
    lastx = (event.pageX - canvas.offset.left);  
    lasty = (event.pageY - canvas.offset.top);  
    if ( selectedCmd === "dot" ) {  
      insertCommand( "dot", {  
        x: lastx,  
        y: lasty  
      });  
    } else if ( selectedCmd === "text" ) {  
      var str = prompt( "Define text to insert:" );  
      insertCommand( "text", {  
        x: lastx,  
        y: lasty,  
        text: str  
      });  
    }  
  },  
  'mouseup': function( event)  
  {  
    drawing = false;  
  },  
  'mousemove': function (event)  
  {  
    if ( drawing )  
      // Funktioniert leider im Firefox nicht:  
      // if (event.which === 1 )  
    {  
      var newCmd = {  
        x : (event.pageX - canvas.offset.left),  
        y : (event.pageY - canvas.offset.top) };  
  
      if (selectedCmd === "line")  
      {
```

```

        newCmd['x2'] = lastx;
        newCmd['y2'] = lasty;

        lastx = newCmd.x;
        lasty = newCmd.y;
    }
    insertCommand( selectedCmd, newCmd );
}
}
});

```

Wer einen Blick in die Dokumentation des Event-Objektes wirft, findet die Variable `which`, die für die Mausevents den aktuell gedrückten Button enthält. Mit diesem Wert könnten wir auf die Verarbeitung des Mouseup-Events und die Variable `drawing` komplett verzichten. Leider funktioniert dieser Wert in aktuellen Firefox-Versionen nicht und macht einen Work-Around notwendig.

Die Hilfsfunktion `getUserName()` liefert uns den Namen des eingeloggten Benutzers. Je nach Login-Provider liegt der Name in einer anderen Datenstruktur. Bietet der Provider ein Profil an, muss der Name als Attribut des Profils extrahiert werden. Im anderen Fall ist er als `Username` direkt im User-Objekt abgelegt:

```

var getUserName = function()
{
    if (Meteor.user())
    {
        var user = Meteor.user();
        if (user.profile) {
            return user.profile.name;
        } else {
            return user.username;
        }
    }
    return "";
}

```

Höchste Zeit, dass wir uns die Hilfsmethode zum Speichern der Kommandos anschauen. Die Funktion selbst ist übersichtlich. Falls kein angemeldeter Benutzer vorliegt, wird kein Kommando gespeichert.

Liegt ein Benutzer vor, wird das übergebene Zeichenkommando um die Standardangaben, wie Timestamp, gewählte Farbe, Breite und Benutzername ergänzt und in die Collection eingefügt.

```

var insertCommand = function( command, obj )
{
    if ( getUserName() === "" )
        return;

```

```

obj.cmd = command;
obj.t = (new Date()).getTime();
obj.c = selectedColor;
obj.w = selectedWidth;
obj.o = getUserName();
commands.insert(obj);
}

```

### Eventmaps und Touch-Event

Einen Nachteil hat die Eventmap: Zurzeit werden Touch-Events kaum berücksichtigt. Lediglich das Antippen (Tap-Event) existiert. Für das Zeichnen fehlen uns leider die Events: touchstart, touchmove und touchend. Diese müssten wir in der aktuellen Version, an Meteor vorbei, direkt an die Elemente hängen und verarbeiten.

### Der Server

Der Server übernimmt mehrere Aufgaben. Er veröffentlicht die Collections und verteilt die Änderungen der Daten an alle Clients.

Für die Zeichenobjekte ist es sinnvoll, die Collection nach dem Timestamp absteigend zu sortieren, damit die Clients im Normalfall nur die neuen, für sie noch nicht gezeichneten Elemente, visualisieren müssen.

```

Meteor.publish('commandsSubscription', function () {
  return commands.find({}, {sort: {t: 1}});
});

Meteor.publish('usersSubscription', function () {
  return users.find({});
});

```

Letztlich deutet das Beispiel eine sinnvolle Logik auf dem Server nur an, da nicht viel mehr als die Sortierung passiert. Der Mechanismus ist trotzdem ein mächtiges Werkzeug. Der Server bietet weitere Funktionen an, deren Aufrufe wir im Client-Code angesprochen hatten:

Die Methode `clear()` liegt auf dem Server, da ein Client im Meteor-Framework nur Einträge mit einem exakten Identifier (`id`) löschen kann und nicht alle Einträge auf einmal. Die Methode `remove()` entfernt alle Einträge aus der Collection und löscht damit alle Zeichnungen von allen Benutzern.

Die zweite Methode, die wir von den Clients aus aufrufen, ist die Methode: `createUserIfNotExists()`. Diese prüft, ob ein Benutzername nicht existiert und legt diesen dann erst an.

```

Meteor.methods({
  'clear': function () {
    commands.remove({});
  }
});

```

```
});

Meteor.methods({
  // add new user if not exist
  createUserIfNotExist: function(username) {
    var user = users.findOne({name: username});
    if (!user) {
      users.insert({name: username });
    }
    return username;
  }
});
```

Die Funktion `_ensureIndex({ "t": 1 })` fügt einen Index einer Collection hinzu. Der Zugriff erfolgt über diesen Weg optimiert. Für uns ist der Zugriff über den Timestamp entscheidend. Die Funktion ist vom Meteor-Team als experimentell eingestuft und versteht aktuell nur eine Index-Spalte.

In Zukunft wird sich diese Funktion sehr wahrscheinlich ändern, da eine umfassende Index-API erstellt werden soll, die alle Features von MongoDB in Meteor bereitstellt.

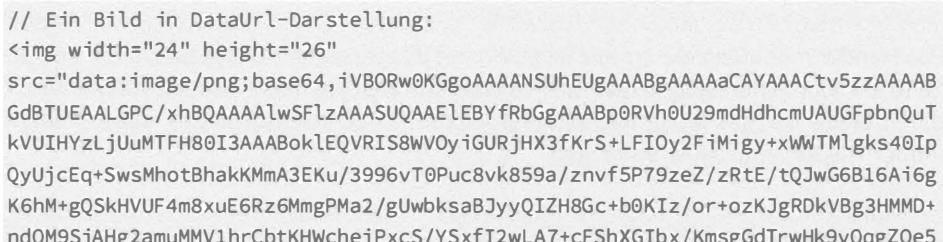
```
Meteor.startup( function() {
  commands._ensureIndex( { "t": 1 } );
});
```

## 4.6.5 Wir erweitern das Whiteboard

### Das Whiteboard als Image

Bisher existiert das kollaborativ erstellte Kunstwerk nur in der Applikation zur Laufzeit. Um es als Bild für die Nachwelt zu erhalten, müssten wir einen Screenshot vom Bildschirm erstellen.

Wir können mit JavaScript direkt ein Bild aus unserem Canvas erstellen. Hierfür existiert die Methode: `getDataUrl()`, die die Zeichnung in eine 64-bit-codierte URL umwandelt. Als Voreinstellung liefert die Methode die Bilder im Format PNG. Auf Wunsch erstellt sie ebenfalls ein Jpeg-Bild, wenn man statt des Wertes `image/png` eine anderes Wunschformat (zum Beispiel `image/jpeg` oder `image/gif`) übergibt. Hier folgt ein Beispiel für die Darstellung eines Bildes:

```
// Ein Bild in DataUrl-Darstellung:

src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAABgAAAAACtv5zzAAAABGdBTUEAALGPC/xhBQAAAAlwSFzAAASUQAAElEBYfRbGgAAABp0RVh0U29mdHdhcmUAUGFpbnQuTkVUIHYzLjUuMTFH80I3AAABokLEQVRIS8WVOyIGURjHX3fKrs+LFI0y2Fimigy+xWTMlgks40IpQyUjcEq+SwsMhotBhakKMmA3EKu/3996vT0Puc8vk859a/znvf5P79zeZ/zRtE/tQJwg6B16Ai6gK6hM+gQSkrHVUF4m8xuE6Rz6MmgPMa2/gUwbksaBJyyQIZH8Gc+b0KIz/or+ozKJgRDkVBg3HMMD+ndQM9SiAHg2amuMMV1hrCbtKHcheiPxcS/YSxfI2wLA7+cFShXGIbx/KmsgGdTrwHk9vQqgZ0e5"/>
```

```
AQ0aYATYdzF8xx0DLU7pmL0WRtMxhUuCF9SAyx5ZvakmTBeJXzLWiwrUiwsAw+gAu9kUXZq8UwkI
aNOcI7TX0X/RZkUx/vjINo2jSCY796hjrSRsH3PqlmkdRJS5jh8rGwcMUVQCbSjxPMznte2ac0A4
VXhq4Upz5lFPQZA6Jbl1qkt8QcATsDbLr0EbIUAbVkcZkMA3oi+QwydAa+ZYOP9H0qkve80ZkfAj
RFwi7gZiMXYB9VakjPm3gjosiaUcfw7hboI9Pk9FuavG4Bfa07+UD5KSaOYgAAAABJRUErkJgg
g==" />
```



Bild 4.16: Ergebnisbild der Base64-Codierung.

Über die folgende Hilfsfunktion erstellen wir aus unserem Canvas-Context ein neues Bild und liefern es zurück.

```
// Converts canvas to an image
function convertCanvasToImage(context2d)
{
    var image = new Image();
    image.src = context2d.canvas.toDataURL("image/png");
    image.width=300;
    image.height=200;
    return image;
}
```

In der Oberfläche erstellen wir einen Button, der die gerade definierte Methode anstößt und das Ergebnisbild einem DIV-Container im unteren Rand zuweist. Über die rechte Maustaste und das Kontext-Menü des Browsers können wir die Zeichnung mit den Standard-Browser-Funktionen im Dateisystem lokal speichern.

In der Eventmap behandeln wir dafür eine neue Klick-Benachrichtigung des Buttons »toImage«:

```
...
'click input.toImage': function(event)
{
    var image = convertCanvasToImage(canvas.getContext());
    $("#pngHolder").empty();
    $("#pngHolder").append( image );
},
...
```

Die Handlermethode erzeugt das Bild, und mit jQuery legen wir das Bild in der HTML-Seite im Container `#pngHolder` ab. Der Befehl `empty()` leert den Container zuvor, damit bleibt nur das letzte Bild sichtbar. Ohne das Leeren würde der `append`-Befehl immer wieder neue Bilder anhängen.

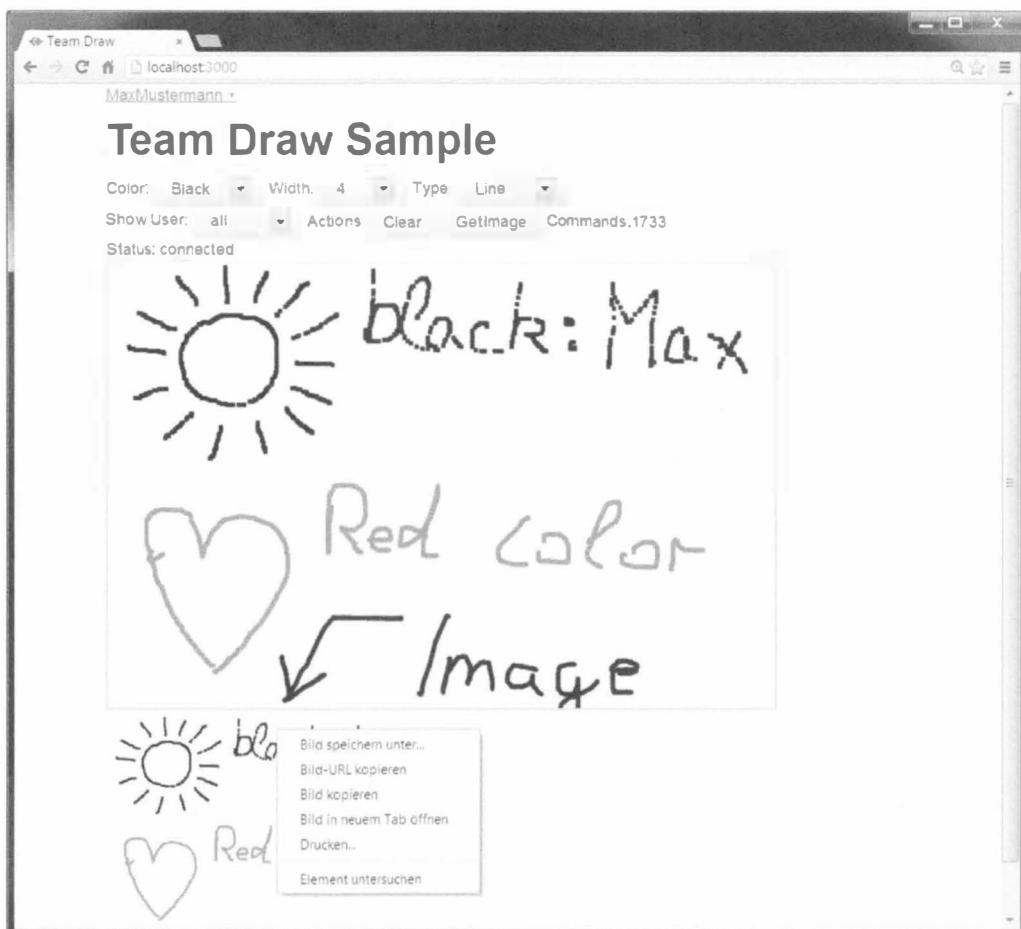


Bild 4.17: Der Inhalt der Zeichenfläche als Bild am unteren Rand.

### Ein neues Zeichen-Kommando: Text

Die beiden aktuell benutzten Zeichenkommandos für Punkte und Linien sind sich sehr ähnlich. Bei den Linien kommen lediglich zwei Werte hinzu. Was ist zu tun, wenn wir unsere Zeichenmöglichkeiten erweitern wollen?

Als Beispiel ergänzen wir Text als neue Operation. Für die Eingabe der Zeichenkette nutzen wir den `prompt`-Befehl, der in allen Browsern verfügbar ist. Das ist nicht sonderlich komfortabel, macht die Implementierung aber sehr einfach. Es erscheint ein Eingabefeld, in das der Benutzer die Textnachricht einträgt.

Die View ist schnell ergänzt: In der HTML-Seite nehmen wir Text als neuen Eintrag in die Auswahlliste der Zeichenkommandos im `drawingSurface`-Template auf:

```
...
Type:
```

```

<select class="type btn" id="type" name="type" size="1" width="100"
style="width: 100px">
  <option value="dot" >Dot</option>
  <option value="line" >Line</option>
  <option value="text" >Text</option>
</select> <br>
...

```

Um einen Text anzulegen, erweitern wir das Mousedown-Ereignis und prüfen, ob Text als Kommando ausgewählt ist. Im Else-Teil der Verzweigung steht das ursprüngliche Verhalten der Nachrichtenverarbeitung:

```

Template.canvas.events({
  'mousedown': function (event)
  {
    if (selectedCmd === "text")
    {
      var str = prompt( "Define text to insert:");
      insertCommand( "text", {
        x: (event.pageX - canvas.offset.left),
        y: (event.pageY - canvas.offset.top),
        text: str
      });
    } else
    {
      // Ursprüngliches Verhalten bei nicht Text-Kommandos:
      lastx = (event.pageX - canvas.offset.left);
      lasty = (event.pageY - canvas.offset.top);
    }
  },
...

```

Für ein Text-Kommando erfragen wir mit dem `prompt` den eigentlichen Text. Mit den Koordinaten des Events stehen alle Informationen für das Text-Objekt bereit. Das Einfügen in die Collection funktioniert analog zu den bekannten Zeichenobjekten. Hier ein Beispiel für ein vollständiges Text-Kommando:

```

{ cmd: "text",
  x: 10, y: 20,
  text: "Diesen Text anzeigen.",
  w: 4, c: "#00F",
  t: 1234599, o: "hsp1"
}

```

Das eigentliche Zeichnen des Textes erfolgt, wie für die anderen Zeichenoperationen auch, in unserem Canvas-Object. In der Fallunterscheidung für die Zeichenobjekte in der Methode `draw()` nehmen wir `text` als neues Kommando auf:

```
...  
} else if (data[i].cmd === "text" )  
{  
    context.font="20px Arial";  
    context.strokeStyle=data[i].c;  
    context.fillStyle=data[i].c;  
    context.fillText( data[i].text, data[i].x, data[i].y );  
}  
...  
...
```

Neben dem Setzen der Zeichenattribute erledigt der Befehl `fillText()` die eigentliche Darstellung des Textes. Der folgende Screenshot zeigt die Eingabe des Textes und die Darstellung in Aktion:



Bild 4.18: Texteingabe und ein Text in der Zeichenfläche.

### Ausgewählte Themen

Die Meteor-Funktionen `Session.set()` und `Session.get()` sind etwas verwirrend benannt. Wer andere Web-Frameworks kennt, erwartet eine Benutzersession, die Informationen nicht nur über mehrere Requests, sondern ebenfalls über einen Refresh der Seite hält. Die Meteor-Funktionen unterscheiden sich von diesem Konzept. Beim Neuladen der Seite (zum Beispiel mit F5) gehen alle Session-Werte verloren und die

Applikation wird neu initialisiert. Das ist im ersten Moment verwirrend und meist nicht gewollt.

Abhilfe schafft die Erweiterung Amplify. Mit ihr ist es leicht, die Meteor-Session um das erwartete Verhalten zu erweitern. Leider muss man etwas tiefer in die Trickkiste greifen. Wir brauchen die Amplify- und die Underscore-Erweiterung, die wir wie üblich installieren:

```
$ meteor add amplify  
$ meteor add underscore
```

Mit diesen beiden Frameworks können wir das Meteor-Session-Objekt um die gewünschte Langlebigkeit erweitern. Amplify überträgt alle Werte, die im Session-Objekt liegen, automatisch in einen LocalStorage. Underscore dient als Hilfsmittel, um die Objekte zu erweitern und alle Schlüssel zu durchlaufen.

```
SessionAmplify = _.extend({}, Session,  
{  
  keys: _.object(_.map(amplify.store()), function (value, key)  
  {  
    return [key, JSON.stringify(value)];  
  }),  
  set: function (key, value)  
  {  
    Session.set.apply(this, arguments);  
    amplify.store(key, value);  
  }  
});
```

Wenn die Methode `set()` aufgerufen wird, legt sie den neuen Schlüssel und den zugehörigen Wert in die herkömmliche Meteor-Session und zusätzlich in den Amplify-Speicher.

In unserer Applikation ändern sich alle Zugriffe auf die drei Variablen `selectedColor`, `selectedCmd` und `selectedWidth` in Aktionen auf Attribute eines Objektes vom Typ `SessionAmplify`. Die globalen Variablen sind nicht mehr notwendig. Ein Beispiel aus der Eventbehandlung für Farbauswahl sieht so aus:

```
...  
'change #color': function (event)  
{  
  SessionAmplify.set( "color", event.target.value);  
},  
...
```

Der vollständige Code mit der Amplify-Erweiterung findet sich im Quellcode des Beispiels »TeamDraw3«.

### Live HTML: Erweckt die Seite zum Leben

Im herkömmlichen Programmiermodell muss der Entwickler bei einer Änderung der Daten selbst dafür sorgen, dass sich die Darstellung in der View entsprechend aktualisiert. Typischerweise würde dafür ein Refresh angestoßen, der die HTML-Seite neu lädt. In extremen Fällen müsste in regelmäßigen kurzen Zeitabständen ein Timer den Refresh auslösen (Polling).

Reaktive Ansätze funktionieren für diesen Fall sehr viel eleganter. Ähnlich zum Databinding, wie wir es im AngularJS-Kapitel gesehen haben, bietet Meteor mit Live HTML automatische View-Updates. Das folgende Beispiel zeigt den Einsatz:

```
var fragment = Meteor.render(  
  function () {  
    var count = Session.get("countCommands") || "0";  
    return "Commands:" + count + "";  
  });  
$("#cmdPlaceHolder").append(fragment);
```

`Meteor.render()` erwartet eine anonyme Funktion. Diese legt den Inhalt des HTML-Fragments fest und greift dabei auf Informationen in einem Meteor-Session-Objekt mit `Session.get()` zu. Die Rückgabe der `Meteor.render()`-Funktion kann danach mit einem typischen jQuery `append()` im HTML-Dokument abgelegt werden. Damit ist kein üblicher statischer HTML-Block erstellt worden, sondern ein reaktives Element. Immer wenn sich in Zukunft der Inhalt der Session-Variable ändert, aktualisiert sich das Fragment automatisch, ohne weiteres Zutun.

Immer wenn wir das folgende Statement ausführen, aktualisiert sich die Anzeige. In unserem Beispiel aktualisieren wir die Anzeige der Gesamtzahl der Zeichenkommandos in der Funktion `redraw()`:

```
Session.set("countCommands", data.length);
```

Im HTML-Code der Seite selbst fehlt der Platzhalter für den aktualisierten Inhalt. Das ändert sich, indem wir folgende Zeile nach den Actions-Buttons aufnehmen:

```
<span id="cmdPlaceHolder"></span>
```

### Ideen für weitere Funktionen im TeamDraw-Beispiel

Hier einige Ideen als Anregung für die Erweiterung des TeamDraw-Beispiels:

- Ein Benutzer kann nur seine eigene Zeichnung löschen, nicht die komplette Collection mit allen Zeichenkommandos von allen Benutzern.
- Wir können die Applikation um eine Undo-Funktion erweitern. Die Aktion löscht jeweils das letzte eingefügte Zeichenelement des Benutzers aus der Collection. Wichtig ist, dass danach die komplette Zeichnung neu dargestellt wird.

- Weitere Zeichenfunktionen könnten die Applikation aufwerten. Es bieten sich Rechtecke, Kreise oder andere Formen an. Dazu müsste man die Maus-Nachrichten (Mousedown und Mousemove) um das Zeichnen der Formen erweitern. Erst beim endgültigen Loslassen der Maustaste darf das neue Element angelegt werden. Das Abspeichern der neuen Zeichenkommandos in der MongoDB-Collection erfolgt analog zu vorgestellten Beispielen und sollte recht einfach realisierbar sein.

## 4.6.6 Deployment & Packaging

Ist die eigene Applikation fertig, möchte man vielleicht die Welt teilhaben lassen. Das Meteor-Team macht diesen Schritt der Bereitstellung (Deployment) sehr leicht, indem es eine eigene kostenlose Hosting-Infrastruktur für den Testeinsatz bietet. Die Applikation ist auf einer Subdomain von [meteor.com](http://meteor.com) erreichbar. Dadurch ist die Anwendung sehr schnell für einen Nutzerkreis verfügbar und bietet eine tolle Möglichkeit für ein erstes Feedback. Das Veröffentlichen ist denkbar einfach und mit einer Zeile auf der Konsole durchführbar:

```
> meteor deploy teamdraw.meteor.com
```

Für eine produktive Applikation ist es sinnvoller, ein eigenes System zu betreiben. Hierfür müssen auf der Umgebung zunächst jeweils eine Instanz von Node.js und MongoDB laufen. Mit dem Befehl `bundle` packt Meteor alle zum Betrieb der Applikation notwendigen Ressourcen in ein Tarball-Verzeichnis zusammen, welches in der eigenen Umgebung installiert werden kann:

```
> meteor bundle teamdraw.tgz
```

## 4.7 Abschließende Anmerkungen zu Meteor

### Update der Meteor-Version

Die beständige Weiterentwicklung, in der sich Meteor aktuell befindet, hat leider auch einen Nachteil: Es ist häufig notwendig, die eigenen Projekte an die neuen Meteor-Versionen anzupassen.

Meteor bietet hierfür eine Unterstützung an. Es ist ratsam, die Ausführung vor einem endgültigen Update zu testen, mit der Anweisung:

```
>meteor --release X.Y.Z
```

Meteor aktualisiert das Projekt in diesem Fall nicht permanent, sondern führt die Applikation einmalig mit der im Parameter (x.y.z) spezifizierten Version aus.

Funktioniert die Anwendung und sind eventuell notwendige Anpassungen für neue Meteor-Funktionen oder API-Änderungen durchgeführt, kann man das Projekt komplett umstellen:

```
> meteor update
```

Diese Umstellung auf die aktuelle Version bleibt permanent in der Projektkonfiguration erhalten. Hilfreich ist in diesem Zusammenhang eine Funktion von Meteor, um die Inhalte der Datenbank neu zu initialisieren:

```
> meteor reset
```

#### 4.7.1 Meteor-Erweiterungen: Atmosphere

Welche produktive Dynamik in einer aktiven Community steckt, zeigt die riesige Auswahl von Node.js-Erweiterungen. Meteor baut auf Node.js auf, definiert aber ein eigenes Package-Format mit dem Namen »Smart-Packages«. Es ist möglich, jede für Node.js zur Verfügung stehende Bibliothek in Meteor zu nutzen. Hierzu muss die Bibliothek neu gepackt und eine Smart- bzw. Package-JSON-Konfigurationsdatei für Meteor erstellt werden.

Meteorite ist eine Erweiterung der Kommandozeile für Meteor und bietet die Möglichkeit, eigene Packages aus einem GitHub-Verzeichnis in Meteor zu installieren. Passend dazu gibt es unter dem Namen »Atmosphere« ein Verzeichnis mit Erweiterungen, die von der Meteor-Community erstellt wurden. Die Liste wächst beständig an und bietet viele interessante Ergänzungen. Die meisten Erweiterungen integrieren bekannte JavaScript-Frameworks für den Einsatz in eigenen Meteor-Applikationen und sparen so viel eigene Handarbeit.

Besonders nützliche Atmosphere-Packages schaffen es sogar direkt, in den Meteor-Kern aufgenommen zu werden und das Team hat angekündigt, zukünftig den Zugriff auf Atmosphere direkt in Meteor einzubauen, ohne den Umweg über Meteorite gehen zu müssen.

The screenshot shows a web browser window with the URL <https://atmosphere.meteor.com>. The page title is "Atmosphere Beer. Wings. Smart Packages". A search bar at the top left contains the placeholder "Find a package...". To the right of the search bar are two buttons: "Packages" and "WTF". Below the search bar, there's a section titled "Smart Packages" with a small icon. The main content area lists ten packages, each with a name, description, version, and last updated time.

Packagename	Description	Version	Last updated
stale-session	Stale session and session timeout handling for meteorjs	v1.0.3	5 minutes ago
template-animation-helper	Allows to animate templates by adding a <code>animate</code> class with css transition	v0.3.5	2 hours ago
simple-schema	A simple schema validation object with reactivity. Used by collection2 and	v0.2.20	10 hours ago
headers	Access HTTP headers on both server and client, and client IP	v0.0.11	12 hours ago
moot	Moot for Meteor, forums and commenting re- imagined	v0.0.1	15 hours ago
tiny-scrollbar	Tiny Scrollbar can be used for scrolling content - custom element scrollbar	v0.0.1	18 hours ago
easy-search	Searching made simple	v0.1.4	20 hours ago
jquery-transit	jQuery Transit CSS transformations and transitions plugin packaged for Meteor	v0.1.0	a day ago
factory		v0.1.5	

At the bottom of the page, there's a note: "Built for the Meteor community by the Meteor community" and "Atmosphere v2 coming soon!"

Bild 4.19: Übersicht der Atmosphere-Packages.



Bild 4.20: Detailansicht und Beschreibung eines Atmosphere-Packages.

## 4.7.2 Bewertung von Meteor

Das Team macht bemerkenswerte Fortschritte und entwickelt das Framework beständig weiter. Bis zu einer stabilen und für die Produktion tauglichen Version liegt aber noch etwas Arbeit vor den Entwicklern von Meteor. Trotzdem besticht das Framework mit herausragenden Fähigkeiten und macht die Entwicklung für viele Szenarien einfacher, gerade wenn die eigene Applikation von der Verteilung der Daten, dem Live-HTML oder der MongoDB-Anbindung profitiert.

Zurzeit liegt Meteor in der Version 0.7.x vor. Das Entwicklungsteam hält sich mit Aussagen für den Erscheinungstermin einer Version 1.0 bedeckt, kommuniziert aber einen Zeithorizont zwischen einem und zwölf Monaten. Bei dem aktuellen Entwicklungsfortschritt ist mit einer finalen und ausgereiften Version im Verlauf des aktuellen Jahres zu rechnen.

Es gibt schon zahlreiche Websites mit Tipps und Tricks rund um die Plattform. Die Dokumentation auf der Meteor-Website kann sich sehen lassen und gibt eine gute Hilfestellung beim Einstieg und bei der täglichen Arbeit.



## Mobile Single-Page-Web-Apps

### 5.1 Mobile Anwendungen werden zum Standard

Der Markt klassischer PCs und Notebooks stagniert. Der Boom von Smartphones und Tablets ist ungebrochen und immer mehr Modelle erblicken das Licht der Welt und die strahlenden Augen neuer Eigentümer. Die Anzahl verfügbarer Apps steigt stetig weiter, nicht nur für die beiden Platzhirsche Android und iOS. Neue Plattformen wie Windows Phone, Firefox OS und Tizen drängen auf den Markt und wollen ein Stück vom Kuchen haben.

Das Umfeld wird zunehmend heterogener. Das führt bei der Entwicklung für mobile Plattformen zu besonderen Herausforderungen. Welche Strategie ist langfristig die richtige?

- Entweder muss man sich für eine Zielplattform entscheiden.
- Oder man muss seine Applikation gleich für mehrere Plattformen parallel anbieten.

Beide Strategien haben deutliche Nachteile. Ein alternativer dritter Weg ist, sich auf HTML5 als Plattform zu konzentrieren. Damit handelt man sich den Nachteil ein, nicht immer alle nativen Fähigkeiten der Geräte ausreizen zu können. Dafür gewinnt man eine viel größere Basis möglicher Zielgeräte, die aus einem Entwicklungsstand bedient werden könnten. Für viele Einsatzzwecke ist diese Strategie sehr sinnvoll.

Unterstützt wird dieser Gedanke zusätzlich von der Werkzeugseite: jQuery Mobile ([jquerymobile.com](http://jquerymobile.com)) und andere Frameworks bieten sich als Hilfsmittel für diese Strategie förmlich an. Dieses Kapitel zeigt, wie man mit jQuery mobile Applikationen erstellt. Außerdem gehen wir auf die Stärken und Schwächen des Frameworks ein.

Als konkrete Aufgabe bauen wir eine Rechercheapplikation für Zitate und Weisheiten. Mit einer einfachen Benutzeroberfläche kann der Nutzer in einem großen Bestand von Aphorismen nach Stichworten und Autoren suchen. Diese Applikation folgt ebenfalls dem Ansatz der Single-Page-Web-Applikationen.

#### Die fertige Beispiel-Applikation ausprobieren

Die fertige Applikation ist online auf der Seite von Zita.de verfügbar. Die Adresse lautet: [zita.de/mobile/zitate.html](http://zita.de/mobile/zitate.html)

### Mobile Design: Guide für mobile Applikationen vom W3C

Die Displaygrößen und Auflösungen für Smartphones und Tablets nehmen deutlich zu. Trotzdem sollte eine Website, die für klassische Desktopgeräte oder Notebooks erstellt wurde, an mobile Geräte angepasst werden. Zum einen muss auf eine größere Bandbreite von Bildschirmen Rücksicht genommen werden, zum anderen weicht die Bedienung für Touchgeräte deutlich von klassischen Konzepten ab. So müssen zum Beispiel Bedienelemente wie Schaltflächen deutlich größer sein.

Vom W3C gibt es einen Guide mit vielen Tipps und Richtlinien für mobile Applikationen. Der Guide ist sehr zu empfehlen und geht auf viele unterschiedliche Aspekte ein. Als Hilfsmittel gibt es ein Überblicksdokument (»Flip Cards«), in dem die wichtigsten Punkte in einer Art Präsentation zusammengefasst sind: [www.w3.org/2007/02/mwbp\\_flip\\_cards.html](http://www.w3.org/2007/02/mwbp_flip_cards.html)

#### 5.1.1 Einleitung zu jQuery Mobile

jQuery Mobile fokussiert sich ganz auf die Entwicklung von mobilen Applikationen und Websites. Hierfür bringt es spezielle Komponenten mit. Die Grundidee ist einfach: Eine HTML-Seite wird mithilfe von CSS und JavaScript für die Darstellung auf Smartphones und Tablets optimiert und das Aussehen möglichst an eine herkömmliche App angelehnt.

Welche wesentlichen Fähigkeiten hat jQuery Mobile zu bieten?

- jQuery Mobile funktioniert auf einer breiten Palette von mobilen Geräten und Browsern.
- Das Framework ist auf Touch-Events optimiert.
- Es bietet spezielle UI-Elemente und Verhalten wie zum Beispiel Übergänge und Animationen beim Seitenwechsel.
- Die Größe und damit die Ladezeiten sind für mobile Szenarien optimiert.

- Skins sind sehr leicht auszutauschen.
- Es reduziert den Zugriff über das Netzwerk, indem Seiten zusammengefasst oder intelligent im Hintergrund nachgeladen werden.

jQuery Mobile ist kein Ersatz für, sondern eine Ergänzung von jQuery. Die Grundbibliothek muss ebenfalls in das Projekt aufgenommen werden und die Funktionen stehen unverändert bereit.

jQuery Mobile adaptiert das Aussehen und Verhalten der eingesetzten Plattform weitestgehend. In einigen speziellen Situationen kommt jQuery Mobile aber nicht an wirkliche, nativ entwickelte Apps heran.

HTML, JavaScript und CSS sind die Basis für alle Seiten und UI-Elemente. Die meisten Funktionen sind so implementiert, dass eine jQuery-Mobile-Applikation auch ohne aktiviertes JavaScript zumindest funktioniert. Eventuell sind in diesem Fall Animationen oder andere UI-Funktionen nicht vollständig verfügbar. Diese Eigenschaft preist das Team unter dem Schlagwort »Progressive Enhancement« an.

jQuery Mobile bietet sich sogar als Werkzeug für die Implementierung von Klick-Dummies an. So könnte eine schnelle erste Version ohne Funktionalität mit jQuery Mobile erstellt werden. Auf dieser Grundlage diskutieren alle Beteiligten das endgültige Aussehen und die erwünschten Funktionen, bevor die Implementierung startet.

Es gibt offenbar gute Gründe, einen Blick auf die Entwicklung mit jQuery Mobile zu werfen. Für welche Aufgaben ist jQuery Mobile nicht die beste Wahl?

- Wenn eine Applikation die Fähigkeiten von mobilen Geräten ausreizen muss oder spezielle Features notwendig sind.
- Bei der hardwarenahen oder nativen Entwicklung, zum Beispiel von 3-D-Spielen.

Der Schwerpunkt von jQuery Mobile liegt in der Entwicklung einfacher, UI-lastiger Business-Applikationen, zum Beispiel der Darstellung von Rechercheergebnissen oder Katalogen und Informationssystemen. Für das Erfassen von Informationen über die Tastatur sind Smartphones ebenfalls weniger gut geeignet. Daran kann auch Frameworks jQuery Mobile nichts ändern.

jQuery Mobile wird konsequent und aktiv weiterentwickelt. Aktuell wird die Version 1.4.x als stabil eingestuft. Das Team arbeitet sehr konsequent an der Weiterentwicklung und neue Versionen erscheinen regelmäßig.

### Eine erste Seite mit jQuery Mobile

In eine jQuery-Mobile-Webseite müssen folgende Verweise aufgenommen werden, sofern man die Bibliotheken nicht selbst in seinem Projekt ablegen möchte:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1">
<title>dialog demo</title>
<link rel="stylesheet"
      href="http://code.jquery.com/mobile/1.3.2/jquery.mobile-1.3.2.min.css">
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script src="http://code.jquery.com/mobile/1.3.2/jquery.mobile-
1.3.2.min.js"></script>
</head>
```

Eine einfache jQuery-Mobile-Seite hat folgenden Aufbau:

```
<body>

<div data-role="page" id="page1">
  <div data-role="header">
    <h1>jQuery Mobile Example</h1>
  </div>
  <div data-role="content">
    Etwas Inhalt als statischer Text.<br> Und noch etwas mehr.
    <a href="#dialogPage" data-rel="dialog" data-role="button">Open
    dialog</a>
  </div>
  <div data-role="footer">
    <h2></h2>
  </div>
</div>

<div data-role="page" id="dialogPage">
  <div data-role="header">
    <h2>Dialog</h2>
  </div>
  <div data-role="content">
    <p>Ich bin ein Dialog.</p>
  </div>
</div>

</body>
</html>
```

Eine Besonderheit fällt sofort ins Auge: Es ist vorgesehen, mehrere Seiten in einem HTML-Dokument abzulegen. Die Motivation dahinter besteht darin, nicht jede Seite einzeln über das Netz abzurufen, sondern zwischen den Seiten schnell umzuschalten. Natürlich funktioniert das Muster nur, wenn die Seiten keine dynamischen Inhalte erzeugen und von ihrem Umfang her überschaubar sind.

jQuery Mobile fügt keine neuen Tags oder Elemente ein, sondern nutzt eigene Attribute (mit dem Präfix: `data-`) zur Auszeichnung der HTML-Elemente (meist DIV-Tags).

Diese mit HTML5 eingeführte Möglichkeit nennt sich »custom data attributes«. Jede Applikation kann eigene data-*Attributnamen*-Attribute definieren und eine eigene Semantik hinterlegen. So wird HTML offener für Erweiterungen, ohne zu viel neue Syntax zu definieren oder andere Attribute zu missbrauchen.

jQuery Mobile nutzt diese Attribute sehr intensiv. Alleine im Beispiel oben ist fast jedes DIV-Element zusätzlich gekennzeichnet – als Seite, Header, Kontenbereich oder Footer. Dieses Prinzip zieht sich über fast alle HTML-Elemente inklusive Links und Buttons. Der nächste Abschnitt stellt die wichtigsten Deklarationen für das Data-Attribut vor.

In einem Browser rendert sich die Seite wie in den folgenden Screenshots. Dabei zeigt sich, wie jQuery Mobile das Aussehen in Richtung einer mobilen App trimmt. Das funktioniert ebenfalls in einem Desktop-Browser.

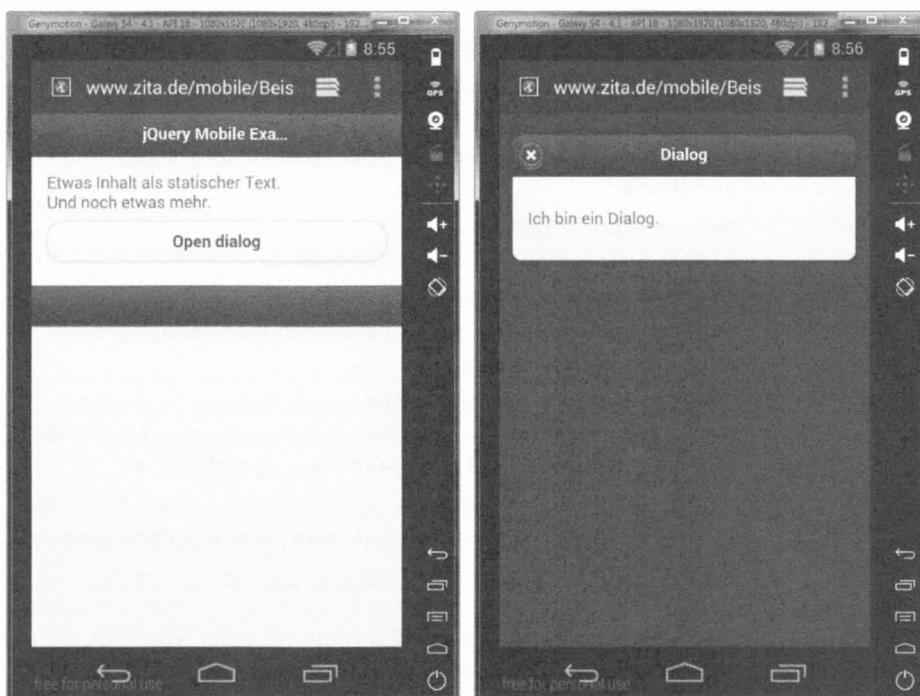


Bild 5.1: Eine einfache Beispieleseite und ein einfacher Dialog im Android-Emulator

### Die wichtigsten Role-Elemente im Überblick

Um den Rahmen nicht zu sprengen, kann das Kapitel nur einen Ausschnitt der wichtigsten Elemente vorstellen. Für eine vollständige Dokumentation mit allen Optionen ist die jQuery-Mobile-Website eine gute Quelle. Dort ist das Verhalten auch gleich live zu sehen.

Das wichtigste Element ist die Rolle (Attribut `data-role`), die im obigen Beispiel intensiv genutzt wurde. Sie legt fest, welche Rolle ein Standard-HTML-Element in der mobilen Seite spielt. Das DIV-Element hat in HTML sehr allgemeine Aufgaben: einen Bereich kennzeichnen und als Container für andere Elemente bereit stehen. jQuery Mobile nutzt die Rolle sehr viel spezifischer. Die Tabelle listet die wichtigsten Rollen auf:

<b>Wert des Role-Attributes</b>	<b>Bedeutung</b>
page	Kennzeichnet eine Seite. Im Rahmen von jQuery Mobile können in einem HTML-Dokument mehrere Seiten und Dialoge definiert sein. Jede Seite muss über ein ID-Attribut eindeutig identifiziert werden.
content	Innerhalb der Seite (page) existiert ein Content, in dem der weitere Inhalt liegt.
header	Optionaler Bereich innerhalb einer Seite, der den Kopfbereich einer Seite oder eines Dialoges kennzeichnet.
footer	Kennzeichnet den Fußbereich (footer) innerhalb einer Seite oder eines Dialoges.
dialog	Stellt den Inhalt nicht als Seite, sondern als Dialog dar. Header und Footer können analog zu einer Seite enthalten sein.
navbar	Stellt einen Navigationsbereich meist im Footer dar, in dem Listenelemente (LI-Tags) als Links zu anderen Bereichen innerhalb der Applikation dienen.
button	Meist werden hiermit HTML-Links als Buttons umdefiniert.

Eine umfassende Beschreibung aller Rollen findet sich natürlich ebenfalls in der Dokumentation von jQuery Mobile: [api.jquerymobile.com/data-attribute](http://api.jquerymobile.com/data-attribute).

## Aufbau einer Seite

Den typischen Aufbau einer Seite hat das einführende Beispiel gezeigt. Dieser Aufbau gilt analog für Dialoge:

```
<div data-role="dialog">
  <div data-role="header">...</div>
  <div data-role="content">...</div>
  <div data-role="footer">...</div>
</div>
```

Leider müssen der Header oder Footer, selbst wenn sie für zwei Seiten identisch sind, in jeder Seite jeweils erneut deklariert werden. Das fällt leider schon bei wenigen Seiten negativ auf und eine Art Template-Mechanismus vermisst man schmerzlich. Es ist keine Pflicht, alle Seiten einer Applikation in eine HTML-Datei zu stecken, die Aufteilung auf unabhängige Dateien funktioniert gleichfalls.

Die Übergänge zwischen den Seiten erfolgen entweder mit Standard-HTML-Links oder anderen UI-Elementen wie Buttons oder Listeneinträgen. Um den Anschein einer mobilen App zu wahren, können Navigationselemente leicht mit der Auszeichnung: `data-role="button"` als Buttons visualisiert werden:

```
<a href="#home" data-role="button" data-rel="back" data-icon="back">Back</a>
```

Als Alternative funktioniert analog ein herkömmlicher Link, die Rolle `button` beeinflusst in erster Linie das Aussehen. Auf den meist kleinen mobilen Displays ist es oft einfacher, einen Button mit den Fingern zu treffen als einen Link, bei dem oft nicht ersichtlich wird, wo die Begrenzungen verlaufen.

Beim Seitenwechsel erledigt das Framework einige Aufgaben im Hintergrund:

- ➊ Abspielen von animierten Übergängen (Transitionen) zwischen Seiten mithilfe von CSS3-Animationen.
- ➋ Die Browser-Historie wird eventuell aktualisiert, sodass der Zurück-Button des Browsers wie gewohnt arbeitet.
- ➌ Deep-Links funktionieren, damit ein Benutzer eine Seite als Favorit speichern und später direkt anspringen kann.
- ➍ Der Header einer Seite erhält über das Attribut `data-add-back-btn="true"` automatisch einen Zurück-Button, was der Standardnavigation in iOS entspricht. Der Button erscheint nur, wenn es eine Vorgängerseite gibt.

### Automatisches Laden von Seiten mit Ajax

Wie schon angedeutet, sind vollständige Links auf Seiten in anderen HTML-Dokumenten möglich. Liegt die referenzierte Datei auf derselben Domain, so lädt jQuery Mobile das Zieldokument automatisch im Hintergrund vor und erlaubt so das schnelle Umschalten der Seiten. Der Inhalt wird unbemerkt in den DOM-Baum eingehängt, als ob dieser schon in einem gemeinsamen Dokument enthalten wäre. Der Link muss dafür in keiner Art konfiguriert werden:

```
<a href="external.html">Go to external page</a>
```

So können Ladezeiten vor dem Nutzer vollständig versteckt werden und die Arbeit mit der Seite fühlt sich wesentlich besser an.

### Transitionen zwischen Seiten

Um dem Verhalten eines Smartphones nahezukommen, animiert jQuery Mobile den Übergang zwischen einzelnen Seiten oder Dialogen. Diese können für die gesamte Applikation oder für jede Transition einzeln festgelegt werden.

Da die Übergänge mit CSS-Animationen implementiert sind, müssten sie auf den meisten modernen Browsern optimiert ablaufen. Typische Effekte sind das Rein- und Rausgleiten oder Umklappen von Seiten. Auf der Projektseite lassen sich alle Übergänge live testen. Die folgende Liste enthält die Namen einiger Transitionen:

- fade
- slide
- slideup
- slidedown
- slidefade
- pop
- fade
- flip
- flow
- turn

Es gibt mindestens drei Wege, um das Verhalten der Übergänge zu definieren:

Über das Attribut `data-transition` an jedem Link, der auf eine andere Seite, einen anderen Dialog oder externe Adresse verweist, kann der Übergang deklarativ festgelegt werden. Hier ein Beispiel für das Setzen der Transition:

```
<a href="external.html" data-transition="flip">
```

Wird ein Übergang programmatisch ausgelöst, erlaubt der Befehl `changePage()` das Übermitteln eines Konfigurationsobjektes mit zusätzlichen Angaben. Darunter befindet sich auch das Attribut `transition`:

```
$.mobile.changePage("#home", { transition: "fade" } );
```

Die beiden genannten Alternativen gelten jeweils für genau einen spezifizierten Übergang. Es ist möglich, das Verhalten global für alle Transitionen einer Applikation zu bestimmen:

```
$.mobile.defaultPageTransition = 'flip';
```

### Schaltflächen (Buttons)

Insgesamt sind die UI-Elemente im Vergleich zu einer herkömmlichen Webseite größer dargestellt, um diese auf kleinen Displays besser erkennen und treffen zu können. Schaltflächen können über folgende Wege definiert werden:

- Als HTML-Button-Element.
- Als Input-Element mit einem der Werte `button`, `submit`, `reset` oder `image` im Attribut `type`.
- Als `data-role="button"` in einem Link-Tag.

- Einen Link-Tag in einem Header- oder Footer-Bereich rendert jQuery Mobile automatisch als Button. Die Angabe des `data-role`-Attributes ist nicht notwendig.

Das Aussehen von Schaltflächen kann durch einige Angaben konfiguriert werden und jQuery Mobile macht es sehr einfach, Icons zusätzlich oder anstelle der Beschriftung zu nutzen:

```
<a href="#" data-role="button" data-corners="false">Rechteckiger Knopf</a>
<a href="#" data-role="button"
  data-inline="true">Inline button</a>
<a href="#" data-role="button" data-inline="true"
  data-icon="refresh">Knopf mit Icon</a>
Button (notext):
<a href="#" data-role="button" data-inline="true" data-icon="refresh" data-
iconpos="notext">Icon-Knopf</a>
```

Mit dem Attribut `data-inline="true"` nimmt sich ein UI-Element nur so viel Platz, wie es braucht und erlaubt andere Elemente in derselben Zeile.

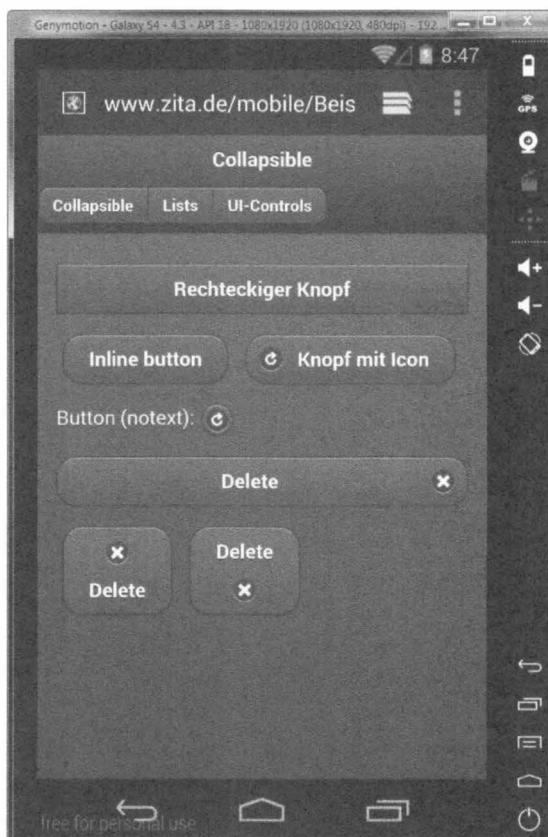


Bild 5.2: Beispiele verschiedener Schaltflächen (Buttons) mit jQuery Mobile.

### Schaltflächen können variabel gestaltet werden.

jQuery Mobile bringt für einige Standardsituationen vordefinierte Icons mit. Ab Version 1.4 liegen diese als SVG vor und werden auf hochauflösenden Displays besser dargestellt:



Bild 5.3: Eine Auswahl der vordefinierten Icons.

Die Position des Icons kann individuell mit dem Attribut `data-iconpos` ausgerichtet werden:

```
<a href="index.html" data-role="button"
  data-icon="delete" data-iconpos="right">Delete</a>
<a href="index.html" data-role="button" data-inline="true"
  data-icon="delete" data-iconpos="top">Delete</a>
<a href="index.html" data-role="button" data-inline="true"
  data-icon="delete" data-iconpos="bottom">Delete</a>
```

### Eigene Icons festlegen

Man muss sich nicht auf die vordefinierten Icons beschränken, es gibt einen einfachen zusätzlichen Weg, um neue Icons zu definieren. Im Attribut `data-icon` benutzt man dafür einen eigenen Namen:

```
<a href="#settings" data-role="button" data-icon="app-setting" >Open
Settings</a>
```

Das Framework erwartet die Definition einer CSS-Klasse, die das Icon enthält:

```
.ui-icon-app-setting {
  background-image: url("app-setting.png");
}
```

Das funktioniert analog ebenfalls mit CSS-Sprite-Icons, bei denen mehrere Abbildungen in einer Grafikdatei liegen und man über das Offset festlegt, welcher Bereich jeweils für ein bestimmtes Icon steht.

Schaltflächen lassen sich in Gruppen zusammenfassen und werden als eine Einheit mit abgerundeten Ecken visualisiert:

```
<div data-role="controlgroup" data-direction="horizontal">
  <a href="#" data-role="button">Button 1</a>
```

```
<a href="#" data-role="button">Button 1</a>
</div>
```

## Auswahllisten

In klassischen Webapplikationen spielen Tabellen eine große Rolle, um Daten zu visualisieren. Auf mobilen Geräten ist meist die Breite der Anzeige sehr begrenzt. Somit muss man sich bei umfangreichen Informationen größtenteils auf eine listenartige Darstellung verlassen. Listen sind mächtige Werkzeuge von jQuery Mobile und bieten einige interessante Funktionen:

- ➊ Listen bieten automatisches lokales Filtern: Über der Liste erscheint ein Textfeld und bei einer Eingabe verschwinden die Elemente, die nicht auf den Filter passen.
- ➋ Die Einträge lassen sich gruppieren und können hierarchisch in andere Listen eingebettet werden.
- ➌ Interaktive Einträge können Aktionen auslösen und zum Beispiel auf eine andere Seite springen.
- ➍ Die Einträge visualisieren zusätzliche Icons und numerische Angaben.

Um eine Liste zu definieren, erweitert man ein HTML-Listentag (UL Tag) mit der Kennzeichnung über das Attribut: `data-role="listview"`. Die Filterfunktion erhält man über das Attribut: `data-filter="true"`. Das Eingabefeld erscheint automatisch ohne weiteres Zutun und die Filterung ist aktiv:

```
<ul data-role="listview" data-filter="true">
    <li data-role="list-divider">JavaScript Frameworks
        <li>Jasmine
        <li>Knockout
        <li>Angular
        <li>jQuery
    <li data-role="list-divider">Java Frameworks
        <li>Hibernate
            <span class="ui-li-count">4</span>
        <li>Spring
        <li>Java Server Faces
        <li>Greece
    <li data-role="list-divider">PHP Frameworks
        <li>Zend
        <li>Symfony
</ul>
```

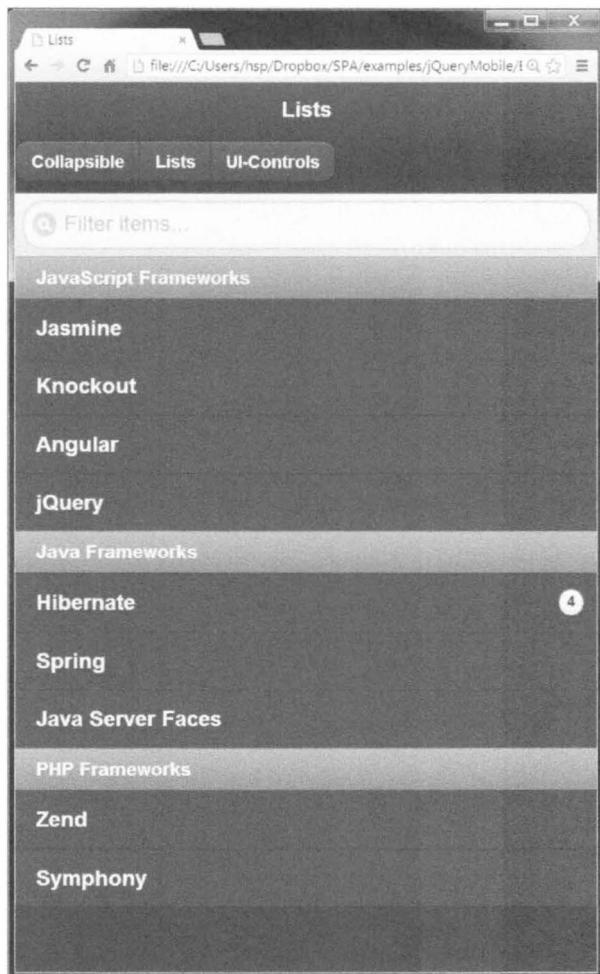


Bild 5.4: Beispiel einer Liste mit Filter und Gruppen.

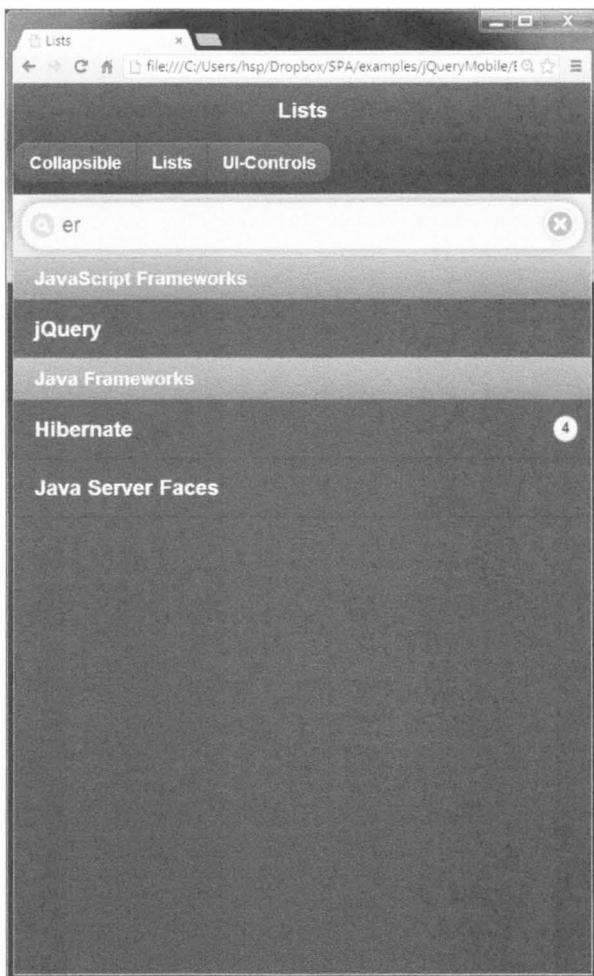


Bild 5.5: Beispiel einer Liste mit aktiviertem Filter.

## Collapsibles

Das Collapsible entspricht einem Accordion, das in früheren Kapiteln schon hilfreich war. Es enthält einige Bereiche mit UI-Elementen. Klappt man einen Bereich auf, so schließen sich die anderen. Damit verhindert man in vielen Situationen übermäßiges Scrolling und nutzt den knappen Platz auf dem Display besser aus. Das Collapsible wird uns in der Beispielapplikation nützlich sein. Im Code sehen wir ein Set mit zwei Bereichen:

```
<div data-role="collapsible-set">
  <div data-role="collapsible">
    <h2>Bereich 1</h2>
    <p>Inhalt des Bereichs 1.</p>
  </div>
```

```
<div data-role="collapsible" data-collapsed="false" >
  <h3> Bereich 1</h3>
  <p>Inhalt des Bereichs 2.</p>
</div>
</div>
```

Ein Collapsible muss nicht im Set stehen, sondern kann alleine existieren. Der zweite Bereich wird initial als offen vorausgewählt: `data-collapsed="false"`.

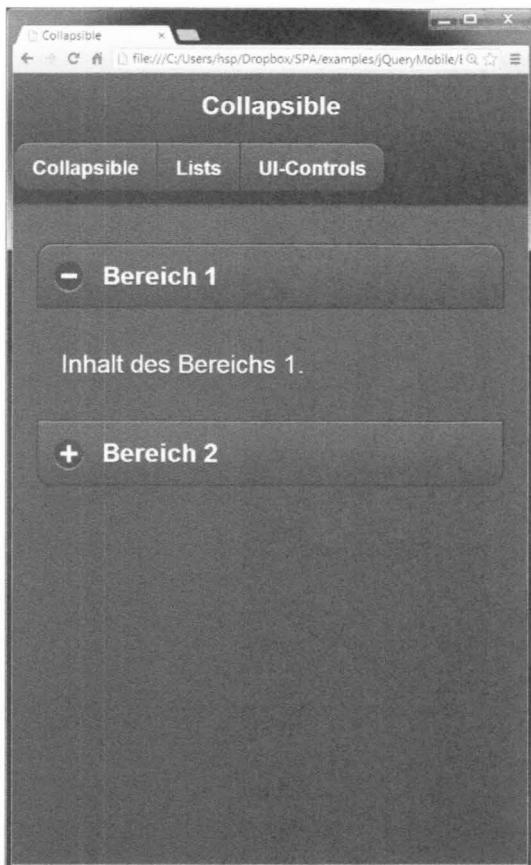


Bild 5.6: Beispiel eines Collapsibles.

### Weitere UI-Controls

Natürlich umfasst jQuery Mobile die typischen Eingabeelemente wie Textfelder, Slider, Radio-Buttons und Auswahllisten. Alle sind für das Look and Feel sowie die Bedienung für Touch-Geräte und kleine Bildschirme optimiert. So passt sich die Zuordnung der Beschriftungen automatisch der Bildschirmabmessung an und die Labels stehen entweder über oder vor den Kontrollelementen.

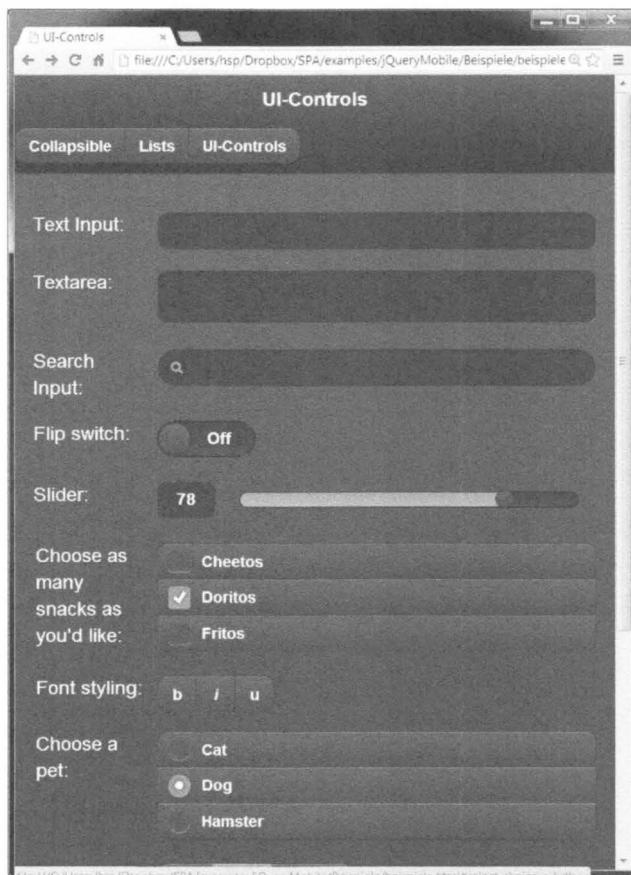


Bild 5.7: Beispiele der typischen Kontrollelemente.

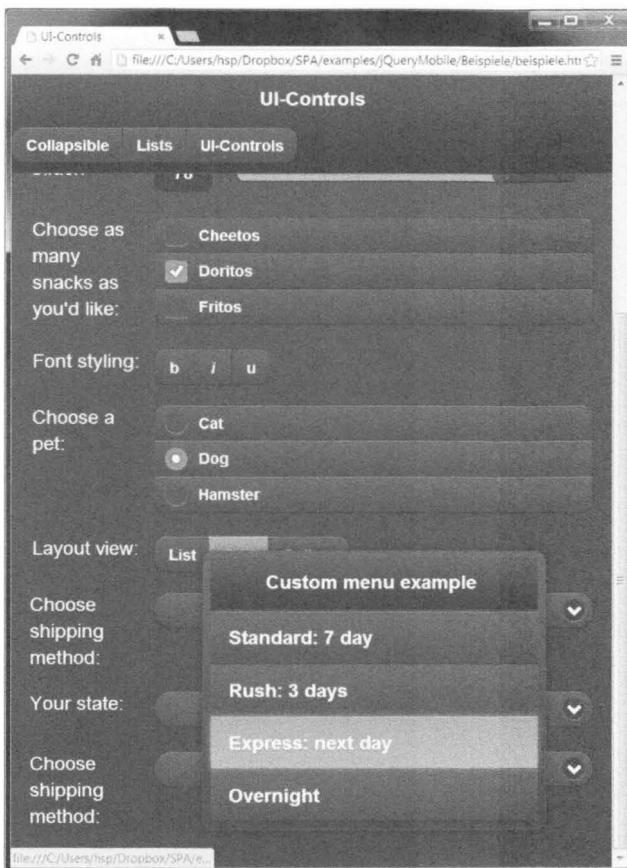


Bild 5.8: jQuery Mobile bietet eine große Auswahl an Auswahllisten.

### Life-Cycle von jQuery Mobile

Beim initialen Start und bei jedem Wechseln der Seite durchläuft jQuery Mobile eine bestimmte Reihenfolge von Events. Jede Applikation hat die Möglichkeit, sich in diesen Ablauf einzuhängen und notwendige Initialisierungen oder Prüfungen durchzuführen. An dieser Stelle können wir nur einen groben Überblick über das ausgefeilte Nachrichtenkonzept geben:

Beim Start der Applikation kann eine Applikation auf die `mobileinit`-Nachricht lauschen und frühzeitig Vorbereitungen treffen. Das Einhängen in die Nachricht erfolgt mithilfe der jQuery-Funktion `bind()`:

```
$(document).bind("mobileinit", function(){
    //Durchführen der Initialisierungen...
});
```

Die Nachricht erfolgt unmittelbar nach dem Laden von jQuery Mobile. Um korrekt reagieren zu können, muss die Bindung möglichst frühzeitig existieren. Aus diesem Grund sollte man sich an folgende Reihenfolge beim Laden der Scripte halten:

```
<script src="jquery.js"></script>
<script src="mein-code.js"></script>
<script src="jquery-mobile.js"></script>
```

Auf die Konfigurationswerte könnte man auf folgende Art zugreifen und globale Werte setzen:

```
// Alternative 1
$(document).bind("mobileinit", function(){
  $.extend( $.mobile , {
    config_variabel: config_value;
  });
});

// Alternative 2
$(document).bind("mobileinit", function(){
  $.mobile.config_variabel= config_value;
});
```

Damit ist das Eventhandling einer jQuery-Mobile-Applikation noch lange nicht abgeschlossen. Bei jedem Wechsel einer Seite zu einer anderen feuert das Framework eine ganze Kaskade von Nachrichten ab.

Im einfachen Fall wird eine Seite (als die erste Seite einer Website) geladen, initialisiert und schließlich im Browser geöffnet. Dabei laufen folgende Events der Reihe nach ab:

- pagebeforechange
- pagebeforecreate
- pagecreate
- pageinit
- pagebeforeshow
- pageshow
- pagechange

Etwas komplizierter wird der Ablauf beim Übergang von einer Seite auf eine andere. jQuery Mobile erzeugt auch für die erste (zu schließende) Seite eine Reihe von Nachrichten, die eine Applikationen für Aufräumarbeiten nutzen kann.

Der genaue Ablauf der Nachrichten und die Zuordnung zur jeweiligen Seite ist hier zu sehen:

- `pagebeforechange`
- `pagebeforecreate` (Seite 2)
- `pagecreate` (Seite 2)
- `pageinit` (Seite 2)
- `pagebeforehide` (Seite 1)
- `pagebeforeshow` (Seite 2)
- `pagehide` (Seite 1)
- `pageshow` (Seite 2)
- `pagechange`

Um sich in einen solchen Ablauf einzuklinken, könnte man folgenden Code nutzen:

```
$('#Seite2').on('pageshow', function (event, ui) {  
    // Logik wird vor der Anzeige ausgeführt.  
});  
$('#Seite1').on('pagehide', function (event, ui) {  
    // Logik wird bei dem Verlassen ausgeführt.  
});
```

Die Seiten werden über die HTML-Identifier `Seite1` und `Seite2` angesprochen. Beim Umschalten von der ersten auf die zweite Seite würde erst die `pagehide`-Nachricht gesendet und danach die `pageshow`-Nachricht. Mit den beiden optionalen Parametern `event` und `ui` hat die Funktion Zugriff auf detaillierte Informationen, so zum Beispiel auf den Namen der Seite, die gerade verlassen wird. Der Codeabschnitt selbst könnte in der weiter oben vorgestellten Behandlung für die `mobileinit`-Nachricht stehen.

### Theme Roller: der einfache Weg zum neuen Aussehen

Ein besonderes Feature von jQuery Mobile ist das leicht austauschbare Aussehen der Skins fast aller UI-Elemente. Dafür gibt es einen eigenen Webdienst, den Theme Roller. Von der Farbpalette aus zieht man die Farben direkt auf die UI-Elemente und legt so fest, welche Darstellung für Hintergrund, Header, Footer und Schrift zum Einsatz kommt. Dabei lassen sich gleich mehrere Themes parallel festlegen, gekennzeichnet durch Buchstaben von A bis Z. Im Screenshot sieht man drei Themes in der Bearbeitung nebeneinander.

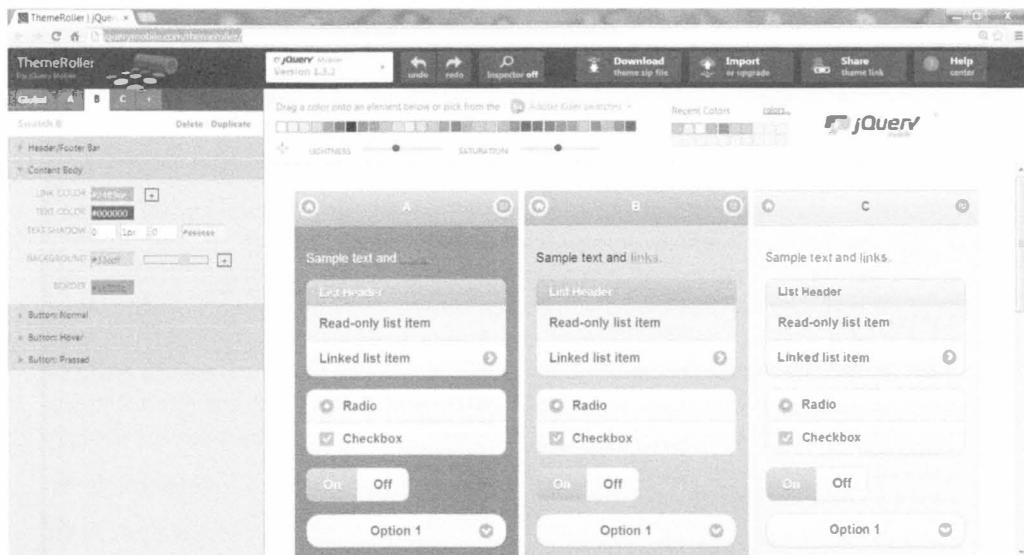


Bild 5.9: Skins definieren per Drag & Drop im Theme Roller.

Per Download liefert der Theme Roller eine fertige CSS-Datei für das eigene Projekt. Über die Buchstaben wählt man für Seiten oder einzelne UI-Elemente das entsprechende Theme aus: `data-theme="a"`. Viel einfacher als mit dem Theme Roller kann das nicht funktionieren.

Natürlich stehen nicht alle Freiheiten zur Verfügung, die der CSS-Standard im Theme Roller bietet. Wer die Hintergrundfarbe der Eingabefelder speziell festlegen möchte, muss sich tiefer in die CSS-Klassen von jQuery Mobile einarbeiten und selbst Hand anlegen.

## 5.2 Eine mobile Zitate-Datenbank mit jQuery Mobile

Als Nächstes erstellen wir eine Beispielapplikation, um einige typische Funktionen im Kontext zu untersuchen. Der fachliche Hintergrund ist eine Datenbank für Weisheiten und Zitate, auf die wir per Smartphone zugreifen wollen.

Die Basis stellt das Projekt [www.Zita.de](http://www.Zita.de) dar. Die Website existiert seit Ende 1999 als kostenlose Datenbank für Weisheiten. Die Finanzierung erfolgt über Werbung. Im Bestand befinden sich etwa 14.000 Zitate und Aphorismen.

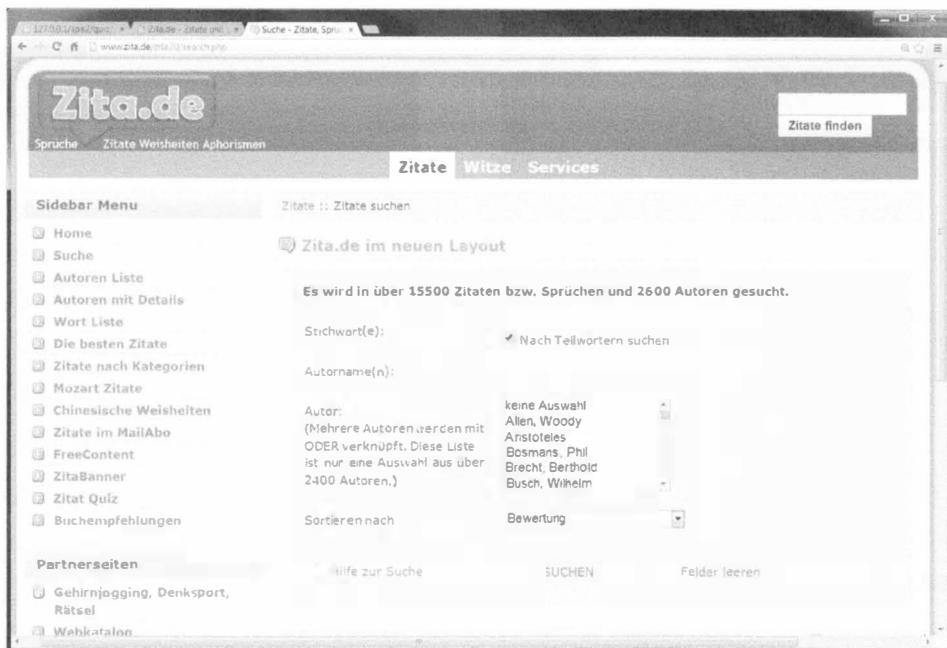


Bild 5.10: Die Suche nach Zitaten bei Zita.de.

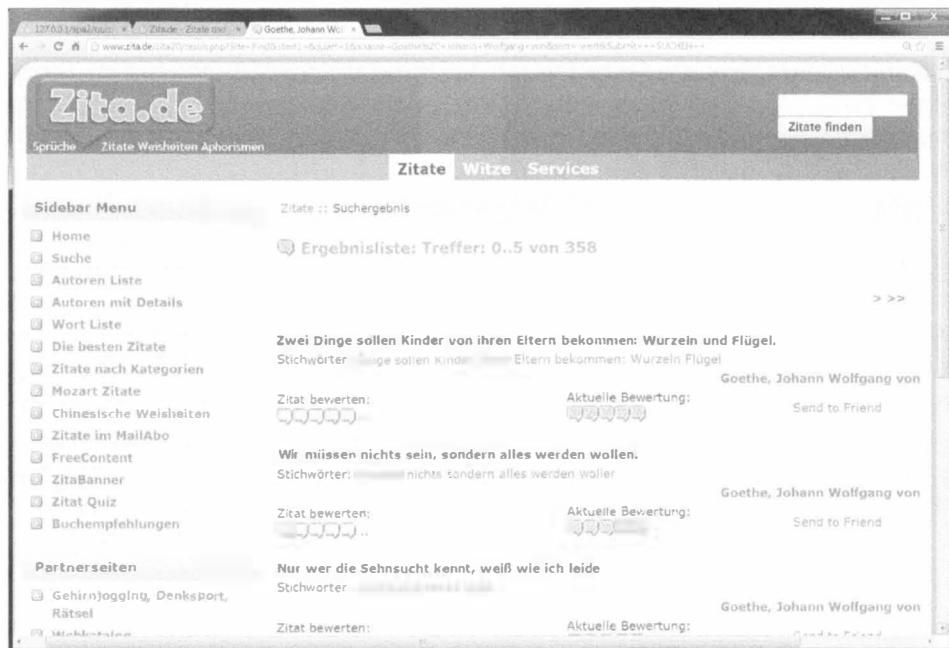


Bild 5.11: Die Ergebnisliste in Aktion.

## Die Funktionen der Applikation

Die Hauptfunktion der mobilen Seite soll das Recherchieren nach Zitaten sein. Dabei kann man über Autorennamen und Stichworte suchen. In den Fokus der Applikation stellen wir eine leichte Bedienung, damit der Einsatz auf einem Smartphone möglich ist und verzichten bewusst auf komplexe Funktionen.

Die Startseite bietet neben einem zufälligen Zitat die Suche nach Autorennamen und Stichwörtern. Beide Felder werden von einer Autovervollständigung unterstützt. Gibt der Benutzer mehr als zwei Zeichen ein, sucht die Applikation im Hintergrund nach passenden Einträgen und listet diese unter dem Suchfeld auf. Der Benutzer muss nicht den vollständigen Namen für einen Autor exakt eingeben, in den meisten Fällen reichen die ersten Buchstaben aus. Ein Klick auf einen gefundenen Eintrag löst die eigentliche Suche nach den Zitaten des Autors oder des Stichwortes aus. Die Ergebnisseite listet die gefundenen Zitate auf. Die wichtigsten Funktionen im Überblick:

- Zufallszitat
- Autorenauswahl
- Stichwortauswahl
- Ergebnisliste

Wie bei den früheren Applikationen zeigen die folgenden Abbildungen die grobe Visualisierung der geplanten Masken:



**Bild 5.12:** Auf der Startseite sind verschiedene Bereiche sichtbar: Zufallszitate und Suche nach Autoren.



Bild 5.13: Der Bereich für die Suche nach Stichworten ist geöffnet.



Bild 5.14: Die beiden Masken zeigen den Info-Dialog und die Ergebnisliste.

Das Beispiel ist in mehrere Schritte eingeteilt und wir werden nach und nach Funktionen hinzufügen:

- Zuerst besprechen wir die Backend-Services für das Abfragen der Zitate.
- Danach erstellen wir das Grundgerüst mit der Anzeige des Zufallszitats auf der Startseite.
- Im dritten Schritt bauen wir die Auswahllisten für Stichworte und Autorennamen und erstellen dann die eigentliche Suche und die Ergebnisseite.
- Zum Schluss bauen wir einige Erweiterungen wie die Anzeige der Anzahl in den Auswahllisten und einen »Loading Indicator« ein.

### 5.2.1 Schritt 1: Backend und Anbindung

Das Backend von *Zita.de* ist klassisch mit PHP und MySQL realisiert. Für den Zugriff auf den Datenbestand von *Zita.de* gibt es vier Funktionen. Alle erwarten die Parameter in der URL einer HTTP-Get-Anfrage und liefern JSON-Objekte zurück.

#### Ein zufälliges Zitat abfragen

Der einfachste Weg, Zitate zu erhalten, besteht darin, das Backend nach einem zufälligen Zitat zu fragen. Parameter sind für diese Abfrage nicht notwendig:

[www.zita.de/mobile/randomquote.php](http://www.zita.de/mobile/randomquote.php)

Die Antwort enthält ein zufälliges Zitat mit weiteren Informationen. Jedes Zitat enthält folgende Attribute:

- Eine eindeutige Nummer (`id`)
- Den eigentlichen Text
- Den Namen des Autors (`name`)
- Die Identifikation des Autors (`autorid`)
- Eine numerische Bewertung im Feld `wert`.

Für den Autor sind die beiden Angaben redundant. Über den Identifier könnte der Name des Autors abgefragt werden. Die Ausprägungen der Bewertung erstrecken sich von »weniger interessant« (1) bis »sehr gut« (5). Das Beispiel zeigt Zitate als Rückgabe des Services:

```
{ "id":32680,
  "text": "Wer nimmt sich denn Zeit, nach dem eigenen Ich zu fragen.",
  "autorid":136,
  "name":"Nietzsche, Friedrich",
  "wert":3 }
```

### Zugriff auf die Vorschläge für Stichworte

Dieser Aufruf liefert eine Vorschlagsliste mit vollständigen Stichwörtern zurück, wenn man den Service mit einem Wortanfang anfragt. In der Beispielanfrage ist im Parameter (`sname`) das Teilwort »Autofahr« enthalten:

[www.zita.de/mobile/ajaxwortlookup.php?sname=autofahr](http://www.zita.de/mobile/ajaxwortlookup.php?sname=autofahr)

Daraufhin liefert der Service alle Stichworte, die mit diesem Teilwort beginnen und in mindestens einem Zitat vorkommen. Die Antwort umfasst eine Liste mit den vollständigen Worten und der jeweiligen Anzahl, in wie vielen Zitaten das gesuchte Wort auftaucht:

```
{ "worte": [  
    { "wort": "AUTFAHRER", "anzahl":3 },  
    { "wort": "AUTFAHREN", "anzahl":2 }]  
}
```

### Zugriff auf die Vorschläge für Autoren

Der Zugriff auf die Liste der Autoren erfolgt ähnlich. Die Anfrage erwartet im Parameter `sname` die ersten Buchstaben eines Autorennamens. In der Beispiel-URL ist »goe« enthalten:

[www.zita.de/mobile/ajaxautorlookup.php?sname=goe](http://www.zita.de/mobile/ajaxautorlookup.php?sname=goe)

Der Service antwortet mit einer Liste der Autoren, mit der eindeutigen Identifikation, dem vollständigen Namen. Das letzte Attribut (`anzahl`) enthält die Anzahl der Zitate, die diesem Autor zugeordnet sind:

```
{ "autoren": [  
    { "id": 399, "name": "Goes, Albrecht", "anzahl":2 },  
    { "id": 103, "name": "Goetz, Curt", "anzahl":27 },  
    { "id": 102, "name": "Goethe, Johann Wolfgang von", "anzahl":318 },  
    { "id": 15849, "name": "Goethes Mutter", "anzahl":1 } ]  
}
```

### Gezieltes Suchen nach Zitaten mit einem Autor oder Stichwort

Bleibt als Letztes die spannendste Abfrage: Wie können wir gezielt nach Zitaten mit einem konkreten Autor oder Stichwort suchen?

Der Service (`quotes.php`) bietet eine ganze Reihe von Parametern, um die Suche zu steuern. Es müssen nicht alle Werte gefüllt sein:

Parameter	Bedeutung
stext1	Das Stichwort, nach dem gesucht werden soll.
rstart	Startindex der Ergebnisliste. Gedacht als Möglichkeit für das Blättern innerhalb der Ergebnisliste.
rshow	Die maximalen Einträge in der zurückzuliefernden Ergebnisliste. Die gesamte Trefferzahl könnte sehr viel höher sein.
spart	Bei der Suche nach Stichworten definiert dieser Parameter, ob nach Wortteilen gesucht werden soll: 1: Suche nach Teilworten erlauben. 0: Das Suchwort muss exakt passen.
sort	Die Sortierung der Zitate ist für unsere Anwendung nur nach der Bewertung sinnvoll. Durch die Angabe »-wert« erhalten wir die Trefferliste absteigend sortiert: Die am besten bewerteten Zitate stehen am Beginn. Bleibt dieser Parameter leer, werden die Zitate in zufälliger Reihenfolge geliefert.
sauthor	Der exakte Identifier eines Autors, nach dem gesucht werden soll.

Die Antwort enthält eine Liste der gefundenen Zitate im Attribut quotes. Das Format der einzelnen Zitate selbst entspricht der Struktur, die wir schon bei der Antwort des Services für die Zufallszitate gesehen haben.

In der vorliegenden Form stehen schon alle Informationen, die wir für die Anzeige brauchen, zur Verfügung. Es ist keine weitere Anfrage notwendig.

Die Antwort liefert so viele Zitate, wie über den Parameter rshow angefordert wurden. Im abschließenden Wert anzahl steht die Gesamtzahl der Einträge, die für die Suchkriterien im Datenbestand existieren. Die Antwort selbst enthält nur den über rshow und rstart angeforderten Bereich der Ergebnisliste.

[www.zita.de/mobile/quotes.php?stext1=gott&rstart=0&spart=1&rshow=5](http://www.zita.de/mobile/quotes.php?stext1=gott&rstart=0&spart=1&rshow=5)

Das Beispiel zeigt einen Auszug für eine Antwort mit zwei Zitaten:

```
{
  "quotes": [
    {
      "id": 82,
      "text": "Das eine ist der Gottheit selbst verwehrt: das, was getan ist, ungeschehen zu machen.",
      "autorid": 246,
      "name": "Aristoteles",
      "wert": 3
    },
    {
      "id": 108,
      "text": "Was nicht ist, kann nicht geschehen.",
      "autorid": 246,
      "name": "Aristoteles",
      "wert": 3
    }
  ]
}
```

```

    "text": "Wenn Gott gewollt hätte, dass wir uns waschen, hätte er das
Parfum nie zugelassen.",
    "autorid":173,
    "name":"Bonaparte, Napoléon",
    "wert":3 ,
    ...
], "anzahl":311
}

```

### Leere Ergebnislisten

Für die drei zuletzt vorgestellten Services könnte die Ergebnisliste leer sein. Dann enthält die Antwort trotzdem ein gültiges JSON-Dokument. Das Beispiel zeigt für diesen Fall eine leere Liste von Zitaten:

```
{
  "quotes": [],
  "anzahl":0
}
```

Damit haben wir alle Services vorgestellt, die wir für die Zitate-Applikation brauchen.

## 5.2.2 Schritt 2: Grundgerüst und zufällige Zitate

Wir brauchen für den Start eine HTML-Seite, in die wir die Funktionen einhängen können.

### Die Oberfläche der Zitate-Applikation

Der HTML-Header entspricht dem Beispiel vom Anfang des Kapitels inklusive der Script-Links auf jQuery und jQuery Mobile. Hinzu kommt ein neues Stylesheet-Dokument (`zitaTheme.css`), das mit dem Theme Roller von jQuery Mobile erstellt wurde. Es definiert die Farben für UI-Elemente und Hintergründe gemäß dem Styleguide von [Zita.de](#). Die Applikationslogik wandert in die referenzierte JavaScript-Datei `zitaMobile.js`.

```

<!DOCTYPE HTML>
<HTML>
<head>
  <meta charset="UTF-8">
  <title>Zita.de Mobil</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <meta name="apple-mobile-web-app-capable" content="yes">
  <meta name="mobile-web-app-capable" content="yes">
  <link rel="stylesheet" href="zitaTheme.css"/>
  <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
  <script src="http://code.jquery.com/mobile/1.3.2/jquery.mobile-
1.3.2.min.js"></script>

```

```
<script src="zitaMobile.js"></script>
</head>
<body>
```

Die Startseite ist im folgenden DIV-Bereich definiert. Die Seite enthält den Identifier `home` und wird als Seite mit der Rolle `page` ausgezeichnet.

Für alle Elemente soll global das Theme A zum Einsatz kommen:

```
<div data-role="page" id="home" data-theme="a">

    <div data-role="header" data-theme="a" data-position="fixed" data-
    id="fixedHeader" data-tap-toggle="false" >
        <h1>Zita.de - Zitate und Weisheiten</h1>

        <div data-role="controlgroup" data-type="horizontal">
            <a href="#home" data-role="button" data-icon="home">Start</a>
            <a href="#infoDialog" data-role="button">Info</a>
        </div>
    </div>
```

Der folgende Block in der Seite ist als Header ausgezeichnet. Die Attribute `data-position="fixed"` und `data-tap-toggle="false"` fixieren den Header am oberen Rand der Seite. Leider funktioniert das bei älteren, mobilen Browsern nur eingeschränkt. jQuery Mobile versucht in diesen Fällen, den Header per JavaScript immer wieder neu zu positionieren. Das beansprucht vom Benutzer intensiv scrollt.

Der Header enthält die Überschrift und einen Button mit einem Link, der auf den Informationsdialog (`#infoDialog`) verweist. Das Logo ist mit den beiden CSS-Klassen `ui-btn-right` und `logo` ausgezeichnet. Die erste Angabe positioniert das Logo rechtsbündig in dem Header. Die Klasse `logo` nutzen wir später, um eine Klick-Nachricht an das Logo zu binden.

Der eigentliche Inhalt der Seite liegt im DIV-Tag mit der Rolle `content`. Wesentliches Element ist das Accordion (`collapsible-set`) mit den einzelnen Reitern, die als `collapsible` markiert sind. In jedem Reiter liegt ein H3-Element, das vom Accordion automatisch zur Überschrift des Reiters wird.

Den ersten Reiter mit dem Zufallszitat öffnen wir als Voreinstellung mit der Angabe `data-collapsed="false"`. Inhalt ist der Container für den Zitat-Text (`randomquote`) und eine Schaltfläche (`nextQuote`), um das nächste Zitat zu holen:

```
<div data-role="content">
    <div data-role="collapsible-set">
        <div data-role="collapsible" data-collapsed="false" >
            <h3>Zufallszitate</h3>
```

```

<h3 id="randomquote">
    <!-- Zitat eingefügen -->
</h3>
<a href="" id="nextQuote" data-role="button">Neues Zitate</a>
</div>

<div data-role="collapsible" >
    <h3>Suche nach Autor:</h3>
    <input type="search" id="autorSuchfeld" placeholder="Autornname" >
    <ul id="autorliste" data-role="listview" data-inset="true">
        <!-- Autoren werden hier eingefügt -->
    </ul>
</div>

<div data-role="collapsible">
    <h3>Suche nach Stichwort:</h3>
    <input type="search" id="wortSuchfeld" placeholder="Stichwort">
    <ul id="wortliste" data-role="listview" data-inset="true">
        <!-- Stichworte werden hier eingefügt -->
    </ul>
</div>
</div>
</div>

```

Die beiden weiteren Reiter sind sich ähnlich. Sie enthalten jeweils ein Eingabefeld und eine leere Liste, die später die Vorschläge für Stichworte und Autorennamen aufnimmt (siehe Kommentare im Sourcecode).

Der letzte Abschnitt im Dokument beherbergt einen statischen Dialog (`data-role="dialog"`), der einige allgemeine Informationen über die Applikation präsentiert:

```

<div data-role="dialog" id="infoDialog" data-theme="a">

    <div data-role="header" data-theme="a">
        <h2>Information</h2>
    </div>

    <div data-role="content" style="text-align: center">
        <br/>
        Ein Service von <a href="http://www.zita.de">www.Zita.de</a>. <br/>
        &copy; 2013 Zita.de
    </div>
</div>

</body>
</HTML>

```

Die Seite ist damit vollständig und kann im Browser betrachtet werden.

## Die erste Logik

Als Nächstes fügen wir die erste Logik hinzu und erwecken die Seite damit zum Leben. Die Logik befindet sich in der Datei `zitaMobile.js` und besteht aus wenigen einfachen Funktionen. Dabei kommt die jQuery-Grundfunktion `$()` zum Einsatz und bindet einige Funktionen an Klick-Ereignisse:

Der Knopf mit dem Identifier `nextQuote` wird an ein Klick-Event gebunden und soll bei der Aktivierung die Funktion `getRandomQuote()` rufen:

```
$(document).on("click", "#nextQuote", function (event, ui) {  
    getRandomQuote();  
});
```

Der folgende Eventhandler ist eine Notlösung. Das Logo befindet sich im Header-Bereich der Seite und soll eigentlich auf die Projektseite von [www.Zita.de](http://www.Zita.de) verweisen. Leider stellt jQuery Mobile alle Links, die im Header vorkommen, als Buttons dar. Damit das Logo unverändert erscheint, verzichten wir auf den statischen HTML-Link und hängen als Ersatz einen Eventhandler an das Bild, der bei Aktivierung zur gewünschten Zieladresse springt:

```
$(document).on("click", ".logo", function (event, ui) {  
    window.open("http://www.zita.de", "system");  
});
```

Das Event `pageinit` tritt einmalig auf, wenn jQuery Mobile die Seite initialisiert. Wir nutzen hier alternativ die Möglichkeit, einen Eventhandler an die Seite zu hängen. Dieser Eventhandler lauscht auf die `pageshow`-Nachricht, die immer dann auftritt, wenn die Seite angezeigt wird. Somit aktualisiert die Funktion das Zufallszitat, wenn wir wieder auf die Seite navigieren.

```
$(document).on("pageinit", "#home", function (event) {  
    $('#home').on('pageshow', function (event, ui) {  
        getRandomQuote();  
    });  
});
```

Abschließend betrachten wir die Funktion `getRandomQuote()` für das Abfragen des neuen Zitats. Hierzu nutzen wir die jQuery-Funktion `getJSON()` und versorgen sie mit der Adresse. Wir brauchen für die Abfragen keinen fachlichen Parameter, aber wir müssen das gewünschte Format (`jsonp`) setzen.

Die Funktion `getJSON()` erstellt ein Ergebnisobjekt, da die Anfrage asynchron ist. An diesem Ergebnisobjekt definieren wir mit dem Aufruf `done()` eine Callback-Funktion, die aktiv wird, wenn die Antwort auf die Anfrage vorliegt. Diese Callback-Funktion verarbeitet die erhaltenen Daten aus den Parametern (`data`). Die Struktur der Nachricht wurde oben schon vorgestellt.

Der Text des Zitates (`data.text`) ist immer gefüllt. Beim Namen des Autors (`data.name`) können wir uns nicht sicher sein und hängen diesen nur an das Zitat, wenn ein Autorname existiert.

Das Einfügen des Textes in den Platzhalter mit dem Identifier `randomquote` ist eine Aufgabe für jQuery und wird von der Funktion `html(text)` erledigt:

```
var getRandomQuote = function () {
    $.getJSON("http://www.zita.de/mobile/randomquote.php", {
        format: "json"
    }).done(
        function (data) {
            var text = data.text;
            if ( data.name ) {
                text = text + "<br>(" + data.name + ")"
            }
            $("#randomquote").html(text);
        });
}
```

### 5.2.3 Schritt 3: Vorschläge für Autoren und Stichworte

In der View sind die Bereiche für die vorgeschlagenen Autorennamen und Stichworte noch unbelebt. Um diese zu füllen, definieren wir Funktionen, die auf Eingaben in die beiden Felder `#autorSuchfeld` und `#wortSuchfeld` lauschen.

Wenn man sich nicht sicher ist, ob das Binden eines Events mehr als einmal passieren könnte, hilft folgendes Muster:

```
$("#autorSuchfeld").
    off("input").
    on("input", function (e) { ... })
```

Die Funktion `off()` entfernt das Event und mit `on()` wird es erneut angehängt. jQuery selbst führt keine Prüfung durch und würde die Behandlungsfunktion letztlich so oft ausführen, wie sie gebunden wurde.

Die Inhalte der beiden Funktionen sind sich sehr ähnlich. Zuerst holen diese die aktuelle Eingabe. Die Suche wird erst wirklich aktiv, wenn der Benutzer mehr als ein Zeichen eingefügt hat und die Abfrage spezifisch genug ist. Kaum ein Mensch möchte eine Vorschlagsliste aller 500 Stichwörter sehen, die mit dem Buchstaben »S« beginnen, schon gar nicht auf einem kleinen Display.

Sind genügend Zeichen vorhanden, tritt erneut die `getJSON()`-Funktion in Aktion. Im Gegensatz zur Abfrage der Zufallszitate müssen wir jetzt neben dem Format einen fachlichen Parameter senden. Im Falle der Autorennamen heißt der Parameter `sname`, im Falle der Stichworte `stext1`.

```

$("#autorSuchfeld").
off("input").
on("input", function (e) {
    var text = $(this).val();
    if (text.length < 2) {
        $("#authorlist").html("");
        $("#authorlist").listview("refresh");
    } else {

        $.getJSON("http://www.zita.de/mobile/ajaxautorlookup.php", {
            format: "jsonp",
            sname: text
        }).done(createDoneFunction("#autorliste"));
    }
});

$("#wortSuchfeld").off("input").on("input", function (e) {
    var text = $(this).val();
    if (text.length < 2) {
        $("#wortliste").html("");
        $("#wortliste").listview("refresh");
    } else {
        $.getJSON("http://www.zita.de/mobile/ajaxwortlookup.php", {
            format: "jsonp",
            stext1: text
        }).done(createDoneFunction("#wortliste"));
    }
});

```

Das Ergebnis behandeln wir für beide Fälle fast gleich. Es ist sinnvoll, eine gemeinsame Bearbeitungsfunktion zu definieren. Diese erstellen wir über einen Umweg mithilfe der Generatorfunktion `createDoneFunktion()`.

Diese Generatorfunktion erhält als Parameter lediglich einen Identifier der HTML-Liste, die die jeweilige Ergebnisliste später anzeigt. Die beiden Listen heißen `autorliste` und `wortliste`. Zur Verdeutlichung zeigt der HTML-Auszug ein Beispiel von oben:

```

...
<ul id="autorliste" data-role="listview" data-inset="true">
    <!-- Autoren werden hier eingefügt --&gt;
&lt;/ul&gt;
...
</pre>

```

Mit dieser Information erzeugt die Generatorfunktion eine passende Callback-Funktion. Die erzeugte Funktion selbst erhält als Parameter das gelieferte JSON-Objekt vom Backend im Parameter `data`.

Der folgende Code zeigt die Signaturen der Generatorfunktion und der Ergebnisfunktion ohne den Inhalt im Rumpf:

```
var createDoneFunction = function (resultListName) {
    var retFunction = function (data) {
        ...
    }
    return retFunction;
}
```

Als Nächstes betrachten wir den Inhalt der erzeugten Funktion selbst: Wenn die Request-Antwort eintrifft, leert die Funktion alle Ergebnislisten in der Anwendung und holt dann die Einträge aus den erhaltenen Daten. Da die Funktion sowohl auf Autoren als auch auf Stichworte reagieren soll, müssen wir unterscheiden, welche Daten vorliegen.

Umfasst die Ergebnisliste mehr als 25 Einträge, zeigen wir eine Meldung, dass die Liste zu lang ist. Der Wert 25 ist relativ willkürlich gewählt.

Ist die Liste kürzer, iteriert die Each-Funktion über die Liste und erzeugt für jeden Eintrag einen neuen Link mit dem Identifier `searchLink`. Jeder Link erhält die jeweiligen Parameter für Autor und Stichwort. Natürlich ist immer nur ein Parameter sinnvoll belegt, je nachdem, welche Liste vorliegt. Für die Autoren nutzen wir als Parameter den technischen Schlüssel (`item.id`), für die Stichworte nutzen wir das Wort selbst (`item.wort`). Für die Anzeige nutzen wir den Namen des Autors oder das Stichwort und verpacken diese in einem H3-Tag. Die jQuery-Funktion `append()` fügt das Ergebnis dem Link hinzu.

Ist die Liste vollständig, hängen wir einen Klick-Handler an jeden Eintrag. Das könnte man schon in der vorgestellten Schleife mit erledigen, eleganter geht es mit einem DOM-Selektor, der den Identifier `searchLink` nutzt.

Die Handler-Funktion, die wir an alle gerade erzeugten Einträge hängen, liest die Parameter des aktvierten Links aus und legt diese in globalen Variablen ab. Zum Schluss springt der Handler auf die Ergebnisseite. Die eigentliche Suche nach den Zitaten erfolgt erst bei der Anzeige der Ergebnisseite selbst.

Damit die erzeugte Liste in der Seite erscheint, senden wir der Liste eine Refresh-Nachricht: `listview('refresh')` und erzwingen so eine Aktualisierung durch jQuery Mobile.

Ganz am Ende der Generatorfunktion `createDoneFunktion()` liefern wir die erzeugte Funktion als Ergebnis.

```
var createDoneFunction = function (resultListName) {
    var retFunction = function (data) {
        $("#wortliste").empty();
        $("#autorliste").empty();
        $('#quotelist').empty();
```

```
var list = data.autoren || data.worte;

if (list.length > 25) {
    var span1 = $("<span>Zu viele Treffer: " + list.length + "</span>");
    span1.appendTo(resultListName);
} else {

    $.each(list, function (i, item) {

        var link = $("<a></a>");
        link.attr("id", "searchLink");
        link.attr("data-paramid", item.id);
        link.attr("data-paramword", item.wort);

        var label = item.name || item.wort;
        link.append("<h3>" + label + "</h3>");

        var li = $("<li></li>");
        li.append(link);
        li.appendTo(resultListName);
    });

    // OnClick-Handler setzen für alle Links:
    $('a[id="searchLink"]').on("click", function (event) {
        aid = $(this).attr("data-paramid");
        word = $(this).attr("data-paramword");

        $.mobile.changePage("#detail" );
    });
}

// Liste aktualisieren.
(resultListName).listview('refresh');
}

return retFunction;
}
```

Interessant ist, dass jQuery mehrere Append-Funktionen anbietet. Genutzt werden beide Varianten im Listing oben:

- ➊ `domZielElement.append( insertMe )`: Fügt dem Objekt, auf dem wir diese Funktion aufrufen, neue Elemente (`insertMe`) als Inhalt hinzu.
- ➋ `domElement.appendTo( zielElement )`: Fügt das übergebene Objekt selbst als Inhalt dem Element `zielElement` hinzu, das im Argument steht.

## Wechseln der Seite

Für das Wechseln der aktiven Seite bietet sich die Funktion `changePage()` an. Der Übergang zur Zielseite kann sehr genau über Parameter gesteuert werden, wenn er von den Voreinstellungen abweichen soll. Die Parameter erhält die Funktion in einem Übergabeobjekt. Der Beispielcode stellt einige Attribute vor:

```
$.mobile.changePage("#detail", {
    allowSamePageTransition: true,
    transition: 'flip',
    showLoadMsg: false,
    reloadPage: false
});
```

Neben der Transitionsart kann das Neuladen der Seite erzwungen werden. Das ist gerade dann sinnvoll, wenn eine dynamische Seite angesprungen wird.

### 5.2.4 Schritt 4: Ergebnisliste

Wählt ein Benutzer in der Vorschlagsliste ein Element aus, merken wir uns bisher nur die Suchkriterien. Die eigentliche Suche erfolgt beim Anzeigen der Ergebnisliste. Die Suche selbst implementiert die Funktion `search()`.

Ähnlich wie die Abfrage für die Vorschlagslisten setzt die Funktion einen Request an den Zitate-Service ab und füllt zuvor die Parameter mit den Werten aus den globalen Variablen. Wichtig sind in diesem Fall die Werte für die Identifikation des Autors und das Stichwort. Wir geben immer beide Werte an, obwohl nur einer von beiden sinnvoll gefüllt ist. Der Backend-Service ignoriert nicht gefüllte Werte.

Die Verarbeitung der Antwort folgt dem bekannten Muster: Die Callback-Funktion bereitet das Ergebnis auf, sobald der Service antwortet. Darin leeren wir die Ergebnisliste (`quotelist`) und setzen die Anzahl der gefundenen Zitate. Dieser Wert stellt die Anzahl der Treffer im gesamten Datenbestand dar. Wie viele Zitate geliefert wurden, steuerte der Parameter `rshow`. Initial sind maximal sieben Zitate sichtbar.

Danach erstellen wir wieder mit der `each()`-Funktion die Ergebnisliste. Gibt es zum Zitat einen Autor, so wird dieser in Klammern unter dem Zitat angehängt. Abschließend fügen wir die fertige Liste mit `appendTo()` in den DOM-Baum der Seite ein.

```
var search = function () {

    var autorId = aid || "";
    $.getJSON("http://www.zita.de/mobile/quotes.php?rstart=0&spart=1", {
        format: "jsonp",
        sauthor: autorId,
        stext1: word,
        rshow: 7
    }).done(
        function (data) {
```

```
$('#quotelist').empty();
$('#count').html("") + data.anzahl;

$.each(data.quotes, function (i, item) {
    var li = $("<h4></h4>");
    li.append(item.text);
    if (item.name)
    {
        li.append("<br> (" + item.name + ")")
    }
    li.appendTo("#quotelist");
});
});
}
```

Somit enthält die Applikation alle Grundfunktionen von der Suche bis zur Anzeige in der Ergebnisliste.

## 5.2.5 Schritt 5: Feinarbeiten und Erweiterungen

### Mehr Zitate nachladen

Die Ergebnisseite zeigt bisher nur den Anfang mit den ersten Zitaten an. Das hat seinen Grund: Wir wollen Anfragen und Antworten kurz halten. Es ist keine vollständige Liste aller Zitate aus dem Datenbestand gefordert, sondern ein schnelles passendes Ergebnis. Falls es viele Treffer gibt, implementieren viele Webseiten die Möglichkeiten, zu blättern und seitenweise neue Treffer zu zeigen (Paging).

Nachteil dieses Musters ist, dass viele UI-Elemente nötig sind: Zum einen brauchen wir Buttons oder Links für das Blättern selbst und zum anderen muss die aktuelle Position in der Gesamtliste sichtbar sein. Gerade für kleine Bildschirme scheint dies wenig geeignet.

Für mobile Apps empfiehlt sich ein anderes Verhalten: Scrollt der Benutzer an das Ende der Liste, lädt die Applikation automatisch neue Elemente nach. In unserem Fall aktivieren wir dieses Nachladen nicht automatisch, sondern geben dem Benutzer die Kontrolle und bauen einen Button an das Ende der Liste. Der Benutzer kann damit weitere Zitate explizit anfordern.

Die Suchfunktion und die Anzeige bleiben fast unverändert. Eine zusätzliche globale Variable (`visibleHits`) merkt sich, wie viele Zitate bisher angezeigt wurden. Der erwähnte Button erhöht die Anzahl der gewünschten Ergebnisse. In der Antwort vom Zitate-Service erhalten wir die Gesamtzahl der möglichen Treffer, die auf unsere Suchkriterien passen. Somit können wir vergleichen, ob es sinnvoll ist, den Button für mehr Zitate überhaupt zu aktivieren.

Die zweite Änderung betrifft die Parameter der Anfrage. Hier fügen wir mit der Variablen `visibleHits` ein, wie viele Einträge die Ergebnisliste umfassen soll:

```
var search = function () {

    var autorId = aid || 0;
    if (visibleHits === 0) {
        visibleHits = 7;
    }
    $.getJSON("http://www.zita.de/mobile/quotes.php?rstart=0&spart=1", {
        format: "jsonp",
        sauthor: autorId,
        stext1: word,
        rshow: visibleHits
    }).done(
        function (data) {

            // Aufbau der Ergebnisliste ist unverändert
            ...

            if (data.anzahl > visibleHits) {
                $("#moreButton").removeClass('ui-disabled');
            } else {
                $("#moreButton").addClass('ui-disabled');
            }
        });
}
```

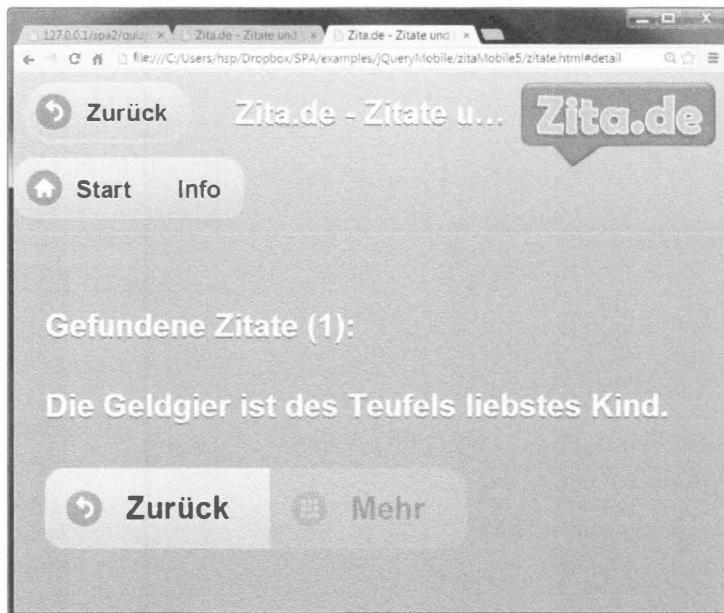
Auf der Ergebnisseite fügen wir neben dem Zurück-Knopf am Ende einen neuen Button (`moreButton`) für das Nachladen hinzu:

```
<a id="moreButton" data-role="button" data-icon="grid">Mehr</a>
```

Die Ereignisbehandlung der neuen Schaltfläche ist nicht kompliziert. Die Funktion erhöht die Variable `visibleHits`, die wir in der Anfrage mit senden, um 15 Treffer und löst erneut die Suche aus.

```
$(document).on("click", "#moreButton", function (event, ui) {
    visibleHits = visibleHits + 15;
    search();
});
```

Ein Nachteil dieser einfachen Implementierung soll nicht verschwiegen werden: Bei jedem Request wird die Ergebnisliste immer länger, da wir fortgesetzt die komplette Liste (von Beginn) bis zur aktuellen Position abfragen und anzeigen. Eine elegantere Lösung könnte nur die zusätzlichen Zitate laden und an die schon vorhandenen Einträge anhängen, die sich nicht geändert haben.



**Bild 5.15:** Die Ergebnisliste mit dem deaktivierten Knopf für mehr Zitate.

### Loading Indikator und globale Konfiguration

Die Übertragungsraten mobiler Geräte steigen, trotzdem sind die Verbindungen weniger stabil. Es ist eine gute Strategie, den Nutzern Feedback über den laufenden Datentransfer zu geben. Eine einfache Lösung bauen wir in die Zitate-Applikation ein: Diese zeigt eine animierte Grafik an, während die Datenübertragung läuft.

Die animierte Grafik legen wir in die Header-Bereiche der beiden Seiten unterhalb der anderen Einträge ab. Es handelt sich um eine einfache animierte Grafik:

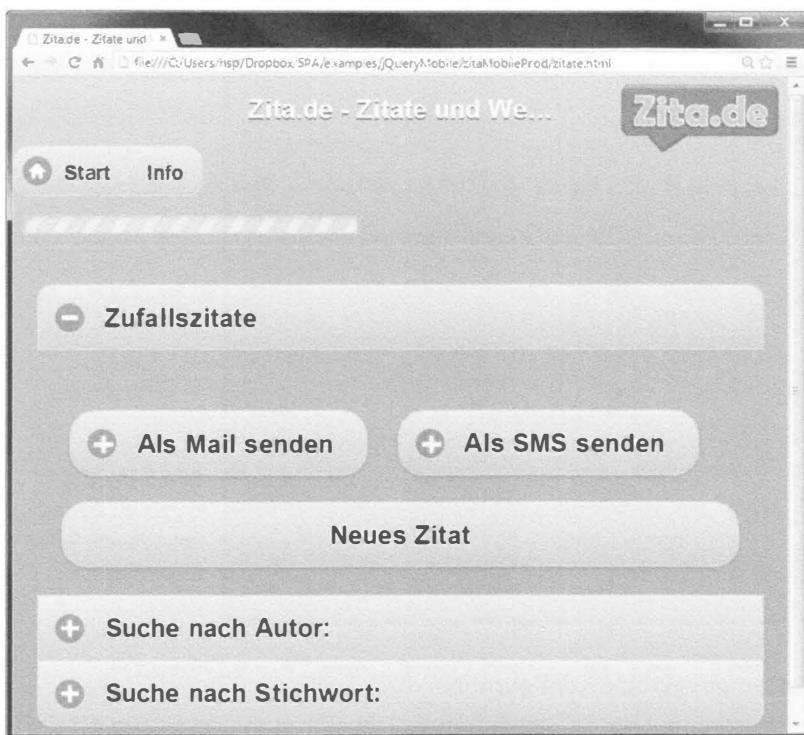


Bild 5.16: Animierte Grafik für den Loading-Indicator.

```

```

Die Bilder in den beiden Header-Bereichen erhalten die gleiche CSS-Klasse, über die sie gemeinsam angesprochen werden können.

Die erste Implementierungsidee würde die Grafiken manuell vor jedem JSON-Request an- und am Ende wieder abschalten, indem wir die Bilder anzeigen und verstecken. Grundsätzlich ist das so implementierbar. Dieses Verhalten müssten wir an jeder Stelle einbauen, an der eine solche Kommunikation erfolgt.

Zum Glück nimmt uns jQuery Mobile diese Arbeit ab und erlaubt, das Verhalten global an einer Stelle einmalig zu konfigurieren. In einem `ajaxSetup`-Objekt definieren wir die beiden Funktionen: `beforeSend()` und `complete()`, in denen wir das gewünschte Ein- und Ausblenden des Indikators einheitlich durchführen:

```
$ .ajaxSetup({
  beforeSend: function () {
    $(".loadingIndicator").show();
  },
  complete: function () {
    $(".loadingIndicator").hide();
  }
})
```

```
});  
$(".loadingIndicator").hide();
```

Das Anzeigen und Verstecken realisieren die jQuery-Funktionen `show()` und `hide()`. Initial verstecken wir die Grafik. Diese Konfiguration selbst liegt in der Handler-Funktion, die auf das `pageinit`-Ereignis der Startseite reagiert.

Dies ist ebenfalls eine gute Stelle, um globale Einstellungen zu definieren. Für alle Seitenübergänge legen wir zentral die Art der Transitionen fest:

```
$.mobile.defaultPageTransition = 'flip';
```

jQuery Mobile bietet eine ganze Reihe weiterer globaler Konfigurationsmöglichkeiten an: [api.jquerymobile.com/global-config](http://api.jquerymobile.com/global-config)

### Anzahl der Treffer in der Vorschlagsliste

Um den Nutzern möglichst früh aufzuzeigen, wie viele Treffer eine Suche erwarten lässt, können wir schon in den Vorschlagslisten die Anzahl anzeigen. Der Backend-Service versorgt uns mit den Werten. jQuery Mobile bietet eine ansprechende Möglichkeit, diese Zahlen in den Auswahllementen zu visualisieren: Sie erscheinen rechtsbündig mit einem abgerundeten Rand.

Für die Umsetzung fügen wir in der Generatorfunktion `createDoneFunktion()` in jeden Link-Eintrag der Auswahlliste ein Span-Element ein und formatieren es mit der CSS-Klasse `ui-li-count`. Die eigentliche Anzahl ist schon in der Request-Antwort sowohl für Autoren als auch für Stichworte im Feld `item.anzahl` enthalten:

```
...  
    if (item.anzahl) {  
        var span = $("<span></span>");  
        span.attr("class", "ui-li-count");  
        span.append(item.anzahl);  
        link.append(span);  
    }  
...  
...
```

Zum besseren Verständnis zeigt das folgende Listing den statischen HTML-Code, um eine Zahl in einer Auswahlliste zu ergänzen:

```
<ul data-role="listview">  
    <li><a href="inbox.html">  
        Inbox<span class="ui-li-count">12</span>  
    </a></li>  
    <li><a href="outbox.html">Outbox  
        <span class="ui-li-count">2</span>  
    </a></li>  
</ul>
```

Die folgenden Screenshots zeigen die gerade eingebauten Funktionen in Aktion:

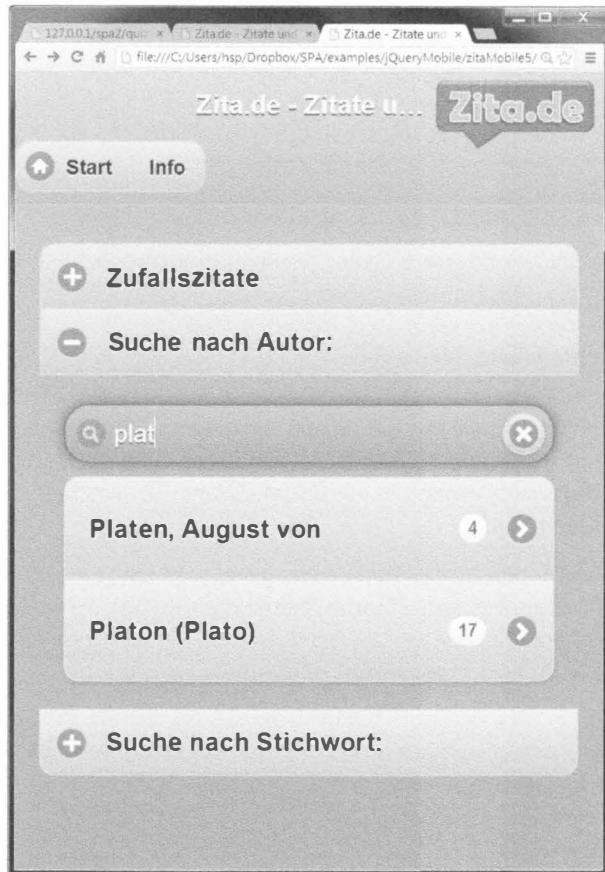


Bild 5.17: Anzahl der zu erwartenden Treffer in den Auswahllisten.

#### Daten zwischen Seiten austauschen:

In manchen Situationen ist es nötig, Daten zwischen den Seiten auszutauschen. Im Zitate-Beispiel wollen wir auf der Detailseite wissen, welcher Autor (Identifier) und welches Stichwort gewählt sind. Es gibt mehrere Wege, diese Informationen zu transportieren:

Der einfachste Weg ist der, globale Variablen zu definieren. In einem Teil der Applikation setzen wir den Wert, wenn der Benutzer in der Autorenliste seinen Favoriten anklickt. Die Applikation wechselt auf die Detailseite und liest die Variable wieder aus. Leider funktioniert das Vorgehen nur, wenn kein Page-Refresh auftritt. In diesem Fall würde der Browser die komplette Applikation und ebenso alle globalen Variablen neu initialisieren.

Auch wenn jQuery Mobile selbst keinen Request beim Wechsel der Seiten durchführt, so könnte der Benutzer jederzeit einen Reload manuell erzwingen.

Ein anderes bewährtes Muster ist es, die Informationen über die URL zu transportieren – so, wie häufig URL-Parameter an den Server übertragen werden. Das funktioniert analog dem entsprechenden Vorgang in einer Single-Page-Webapplikation. Diese Strategie hat ebenfalls den Vorteil, dass die Anwendung damit bookmarkfähig ist und der Zustand beim neuen Laden aus den Parametern komplett restauriert werden könnte. Das Interpretieren der URL-Zeile muss leider selbst vorgenommen werden, ist nicht immer einfach und abhängig davon, wie viele Parameter mit welchen Typen vorkommen.

Einen eleganteren Weg eröffnet der LocalStorage, den wir schon in einem früheren Kapitel untersucht haben. Mit diesem können wir den Zustand der Applikation ablegen und von Seite zu Seite navigieren. Bei einem späteren Reload funktioniert die Applikation ebenfalls. Diese Strategie erfordert einen modernen Browser, der dieses Feature unterstützt. In fast allen Browsern für aktuelle Smartphones ist das Feature verfügbar.

### Zitate versenden per E-Mail und SMS

Eine schöne Erweiterung für unsere Applikation wäre das Versenden der Zitate an Freunde und Bekannte. Wenn die Website auf einem Handy läuft, ist SMS als Versandoption ebenfalls möglich.

Da wir eine Website erstellen, können wir spezielle Link-Typen nutzen, die schon im HTML-Standard existieren. Genau genommen setzen wir in der URI des Links einfach ein anderes Protokoll, das der Browser bei einem Klick interpretiert und die passende Standardapplikation öffnet. Im Fall des Mail-Links öffnet der Browser die voreingestellte Mail-Applikation, im Fall des SMS-Links sollte eine SMS-Applikation erscheinen, wenn diese existiert. Existiert die entsprechende Applikation nicht, zum Beispiel auf einem Notebook, wird der Link ignoriert. Moderne Smartphone-Betriebssysteme bringen eine eigene Funktion für das Teilen über verschiedene Kanäle (E-Mail, SMS und andere Applikationen) mit. Diese ist Teil des Systems und funktioniert mit den angezeigten Zitattexten. Untersuchen wir zunächst die Syntax für den Mail-Link:

```
mailto:<email_destination>[?parameters]
```

Neben der Zieladresse sind die üblichen Parameter die nachfolgend aufgeführt:

- **cc** und **bcc** mit den Adressen, die eine Kopie oder Blindkopie erhalten.
- **subject** enthält die Überschrift der Mail.
- **body** definiert den eigentlichen Inhalt.

Alle Parameter müssen im URL-Format als Key-Value-Paare (key=value) vorliegen und als URI encodiert sein.

Für SMS-Versand bleibt neben dem Adressaten (als Telefonnummer) nur der Parameter `body` für den Inhalt der Nachricht übrig.

Leider ist die Welt nicht ganz so einfach, wie sie sein könnte. Für die SMS-URL gibt es keinen eindeutigen Weg, um herauszufinden, ob der korrekte Protokollname `sms://` oder `smsto://` lautet:

```
sms[to]://[<destination number>][?parameters].
```

Auch für die SMS ist die Zielnummer, an die eine Meldung gesendet werden soll, optional. Der Browser öffnet die SMS-Applikation des Betriebssystems. Der Benutzer kann den Inhalt der SMS verändern und ergänzen. Er hat den gewohnten Zugriff auf seine Kontakte und den eigentlichen Versand der Nachricht. Nicht auszudenken, wenn eine Applikation eigenständig SMS oder E-Mails versenden könnte.

Das folgende Beispiel zeigt einen HTML-Abschnitt mit einem Mail- und SMS-Link für eine einfache Kontaktaufnahme:

```
<!-- Versenden einer Email -->
<a href="mailto:hs@zita.de?subject=Anfrage&
body=Inhalt%20der%20Mail.">
Mail us
</a>

<!-- Versenden einer SMS -->
<a href="sms://?body=Inhalt%20der%20SMS.">
Invite a friend by SMS
</a>
```

### Einbau in die Zitate-Applikation

Bauen wir zunächst die Funktion zum Versenden in die Anzeige der Zufallszitate ein. Der einfache Teil ist der Einbau der beiden neuen Knöpfe in der Startseite:

```
...
<a href="" id="sendMail" data-role="button"
  data-inline="true" data-icon="mail" >Als Mail senden</a>
  <a href="" id="sendSMS" data-role="button"
  data-inline="true" data-icon="mail" >Als SMS senden</a>
...
```

Das Link-Ziel im Attribut `href` bleibt zunächst leer, wir füllen es erst, wenn das Zitat selbst sichtbar wird. Der folgende Abschnitt zeigt den neuen vollständigen Code für die Funktion `getRandomQuote()`:

```
var getRandomQuote = function () {
  $.getJSON("http://www.zita.de/mobile/randomquote.php", {
    format: "jsonp"
  }).done(
    function (data) {
```

```
var text = data.text;
var msgText = data.text;
if ( data.name )
{
    text = text +"<br>(" + data.name + ")"
    msgText = msgText +"(" + data.name + ")"
}
var mailUrl =
"mailto:?subject=Zitate%20von%20Zita.de&body=Inhalt%20der%20Mail%0A%0A"+transform( msgText );
var smsUrl = "sms:?body=Inhalt%20der%20Mail%0A%0A"+transform( msgText
);
$( "#sendMail" ).attr("href", mailUrl );
$( "#sendSMS" ).attr("href", smsUrl );

$( "#randomquote" ).html(text);
});
```

Analog zur lokalen Variablen `text` bauen wird eine zweite Variable mit dem Zitate-Text für den Mail- oder SMS-Inhalt zusammen. Zumindest in einem Zeilenumbruch unterscheiden sich beiden Darstellungen.

Spannend sind die beiden Variablen, die die beiden URLs für den SMS- und den Mail-Aufruf enthalten: `mailUrl` und `smsUrl`. Diese werden wie oben vorgestellt zusammengebaut. Der Inhalt des Zitates wird mit einer Methode `transform(msgText)` um die Darstellung der Umlaute aufbereitet.

Danach übertragen wir die Variableninhalte in die beiden Links, genauer gesagt, in die `href`-Attribute der Links. Bleibt als Letztes noch, die Transformfunktion für die Aufbereitung der Umlaute zu besprechen.

### Problem: Zeichendarstellung der deutschen Umlaute

Die deutschen Umlaute liefern die Backend-Services in einer für HTML codierten Form: Die Zeichenkette `&Auml;` entspricht dem deutschen »Ä«. Eigentlich sollten die Mail- und SMS-Applikationen mit dieser codierten Form zureckkommen. Mehrere Tests mit aktuellen Geräten und Browsern haben aufgezeigt, dass es in vielen Fällen besser funktioniert, die Umlaute direkt in die URL ohne spezielle Umwandlung zu übertragen. Lediglich ein Motorola Xoom 2 mit Android 4.0.4 konnte die Zeichen nicht korrekt interpretieren und ignorierte deren Inhalt.

Eine alternative und sehr konservative Lösung, die in jedem Fall funktioniert, ist das Umwandeln der deutschen Umlaute in normale Zeichen des Alphabets: Ein »Ä« wird zu der Zeichenkette »Ae«. Diese Strategie funktioniert immer – inklusive älterer Geräte und Betriebssysteme. Der Quellcode für diese Implementierung ist im Beispielprojekt enthalten.

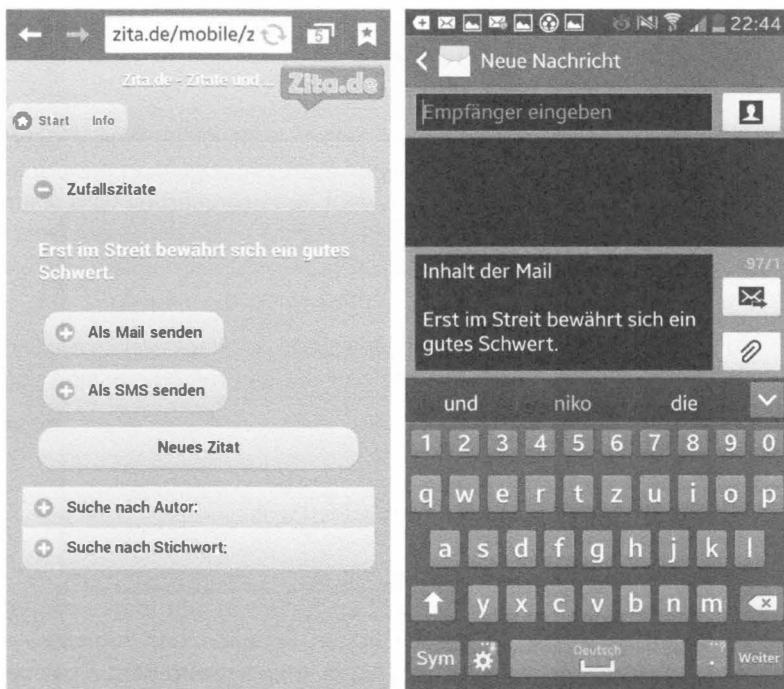
Für unsere Applikation nutzen wir diese Variante und transformieren die Darstellung der Umlaute in die allgemein übliche Form:

```
var transform = function(text) {  
    text = text.replaceAll( "&uuml;", "ü" );  
    text = text.replaceAll( "&Uuml;", "Ü" );  
    text = text.replaceAll( "&auml;", "ä" );  
    text = text.replaceAll( "&Auml;", "Ä" );  
    text = text.replaceAll( "&ouml;", "ö" );  
    text = text.replaceAll( "&Ouml;", "Ö" );  
    text = text.replaceAll( "?", "%63f" );  
    text = text.replaceAll( "&szlig;", "ß" );  
    return text;  
}
```

Leider bietet JavaScript die gerade genutzte `replaceAll()`-Funktion für Zeichenketten nicht an. Es gibt nur ein primitives `String.replace(search, replacement)`, das nur das erste Vorkommen ersetzt. Mithilfe des String-Prototyps können wir uns leicht eine eigene Implementierung kreieren:

```
String.prototype.replaceAll = function(search, replacement) {  
    var target = this;  
    return target.split(search).join(replacement);  
};
```

Wird die Funktion auf einem String angewendet, haben wir über die Variable `this` Zugriff auf den Inhalt. Die Funktion zerteilt die Zeichenkette mit der `split()`-Funktion anhand des Teils, der zu ersetzen ist. Das Ergebnis ist ein Array mit allen Teilen, die übrig bleiben. Diese setzen wir mit dem `join()`-Befehl wieder zusammen und füllen die Lücken mit dem einzufügenden Teilstring. Wir könnten die Funktionalität ebenfalls elegant mit regulären Ausdrücken implementieren.



**Bild 5.18:** Nach der Auswahl »Als SMS versenden« öffnet sich die SMS-Applikation mit dem Text des Zitats.

### Weitere Ideen für die Zitate-Applikation

Im Moment bietet die App eine einfache Suche, entweder nach Autor oder nach einem Stichwort. Sinnvoll wäre eine erweiterte Suche, die mehreren Kriterien verknüpft. So könnte man nach den Zitaten eines Autors suchen, in denen ein bestimmtes Stichwort vorkommt.

Wenn man sich die JSON-Objekte des Zitate-Services genauer anschaut, fällt auf, dass zu jedem Zitat eine Bewertung existiert, die Ziffern zwischen 1 und 5 liefert. In der Ergebnisliste könnte man diese Werte in einer ansprechenden Form (zum Beispiel als Sterne) mit anzeigen.

## 5.3 Fazit zu jQuery Mobile

jQuery ist für die Entwicklung von mobilen Webapplikation sehr zu empfehlen. Das Framework nimmt viel Arbeit ab und hilft bei der Konzentration auf das Wesentliche. Das Aussehen und das Verhalten kommen in vielen Bereichen dem nativen Verhalten nahe. Für die meisten UI-Elemente fallen kaum Unterschiede auf.

Leider orientiert sich jQuery Mobile von seinem Look and Feel her etwas stärker an iOS, was aus der Perspektive von Android-Geräten an einigen (zum Glück wenigen) Stellen auffällt.

Insgesamt ist es für einfache und mittelkomplexe Seiten recht leicht, eine jQuery-Mobile-Applikation zu erstellen. Bis zu einem gewissen Punkt, solange man keine speziellen Fähigkeiten eines Gerätes nutzt oder das Layout zu anspruchsvoll wird, kann die Entwicklung komplett ohne mobiles Gerät erfolgen. Jede Seite ist in einem Desktop-Browser nutz- und testbar. Ab wann es wirklich notwendig ist, auf eine native Entwicklung zuzugreifen, hängt von der konkreten Anforderung ab.

Die Dokumentation des Projektes ist mit vielen Beispielen sehr gut gelungen und erleichtert den Einstieg. Vorkenntnisse bei der Implementierung mit jQuery sind von Vorteil, aber keine zwingende Voraussetzung.

## 5.4 Testen von mobilen Applikationen

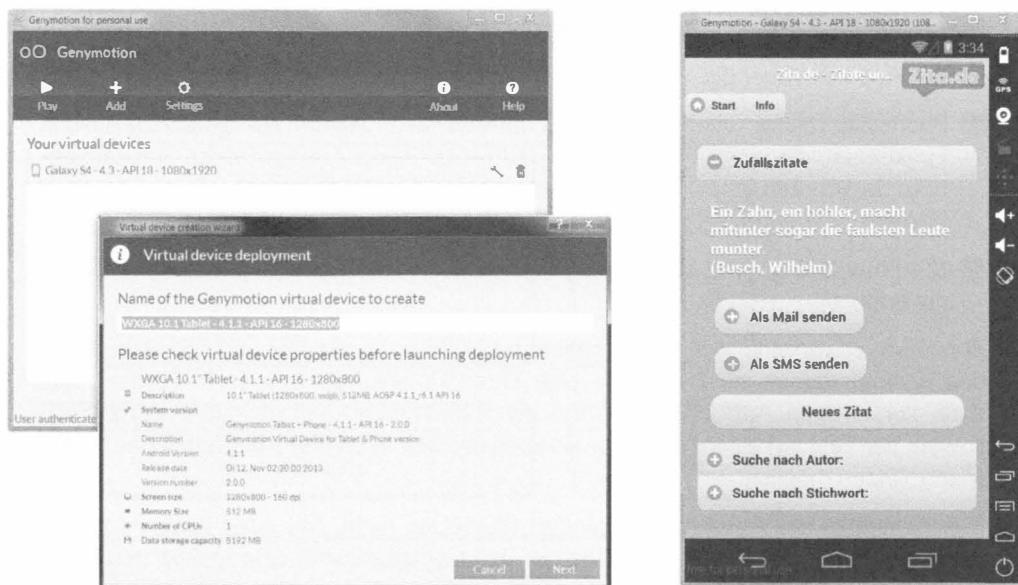
Grundsätzlich kann man seine jQuery-Mobile-Seite in jedem Desktop-Browser ausprobieren. Das grobe Verhalten ist somit sehr leicht zu testen. Es gibt immer wieder kleine Unterschiede im Resultat auf den diversen Geräten, Browsern und Betriebssystemen; so kommt man um einen abschließenden Test auf den angestrebten Zielgeräten kaum herum.

### Emulatoren

Alternativ gibt es einige Emulatoren für mobile Plattformen, um das Ergebnis zu testen. Auch wenn diese Emulatoren nicht in jedem Fall das wirkliche Aussehen und Verhalten widerspiegeln, sind sie eine gute Option.

In den letzten Monaten sind einige neue Emulatoren für Android auf dem Markt erschienen. Damit kann man das Verhalten einer Webseite oder App im Emulator ohne die eventuell lästige Übertragung auf ein Endgerät ausprobieren. Ein großer Vorteil der Emulatoren ist, dass sich sehr leicht unterschiedliche Abmessungen und Bildschirmorientierungen prüfen lassen.

Ein Emulator wird unter dem Namen Genymotion angeboten. Der Einsatz ist für den persönlichen Bedarf kostenfrei. Der Anbieter hält für jedes Gerät eine virtuelle Maschine bereit, die nach der Konfiguration des gewünschten Zielgerätes eventuell erst geladen werden muss. Hierfür ist ein Benutzeraccount beim Hersteller notwendig. Unterstützt werden die Android-Versionen ab 2.3 aufwärts. Der Emulator ist unter dem Link [www.genymotion.com](http://www.genymotion.com) zu finden.



**Bild 5.19:** Android-Emulator Genymotion mit Optionen und Auswahl von Geräten.

Im Android SDK befindet sich ein vollständiger Android-Emulator, der vom Entwicklungsteam immer - wie die Android-Plattform selbst - auf dem aktuellen Stand gehalten wird. So hat man die Chance, eine neue Android-Version schon beim Erscheinen zu testen. Leider fällt der Emulator durch seine extreme Langsamkeit negativ auf. Es gibt einige Tricks und Zusatzprogramme, um den Emulator etwas zu beschleunigen, diese Maßnahmen lösen das Problem aber nur teilweise.

## 5.5 Zur nativen App mit PhoneGap

Jede jQuery-Mobile-App ist letztlich eine Webanwendung, auch wenn das Aussehen in Richtung einer nativen App getrimmt ist. Trotzdem gibt es einen Weg in Richtung nativer Applikation: Das Framework PhoneGap verpackt eine Webapplikation in eine native App für eine spezifische Zielplattform. Zur Auswahl stehen zurzeit mindestens folgende Plattformen:

- Apple iOS
- Google Android
- HP webOS
- Microsoft Windows Phone
- BlackBerry (ab Version 4.6)

Durch dieses Vorgehen gewinnt man folgende Vorteile:

- Die App kann über die Marktplätze der Plattformen vertrieben werden.
- PhoneGap bietet über eine JavaScript-API den Zugriff auf gerätespezifische Funktionen, die weit über die SMS- und E-Mail-Links aus dem Beispiel hinausgehen. So könnten Applikationen direkt auf die Kontaktdaten zugreifen und selbst Kamera, GPS und Sensoren sind nutzbar.
- Die Entwicklung kann als Webapplikation zumindest teilweise plattformunabhängig erfolgen.

PhoneGap erwartet eine Webapplikation mit HTML-, JavaScript- und CSS-Artefakten und erzeugt daraus ein Projekt für eine Entwicklungsumgebung (z. B. Eclipse), das dann zur fertigen App kompiliert und gepackt werden kann. Damit sind natürlich sowohl der Test auf einem Emulator als auch das Signieren und Veröffentlichen in den Marktplätzen möglich.

Das Aussehen der Applikation verändert PhoneGap nicht. Mit einer jQuery-Mobile-Webseite kommt man sehr viel näher an das Verhalten einer Handy-App heran, als wenn man mit anderen Frameworks eine klassische Webapplikation erstellt und sie mit PhoneGap verpackt.

### Namensverwirrungen

Leider gibt es immer wieder Verwirrungen im Hinblick auf den Namen: PhoneGap wurde von Adobe 2011 vom ursprünglichen Hersteller Nitobi gekauft. Mittlerweile ist die JavaScript-Schnittstelle für gerätespezifische Funktionen in das Apache-Projekt *Apache Cordova* überführt worden.

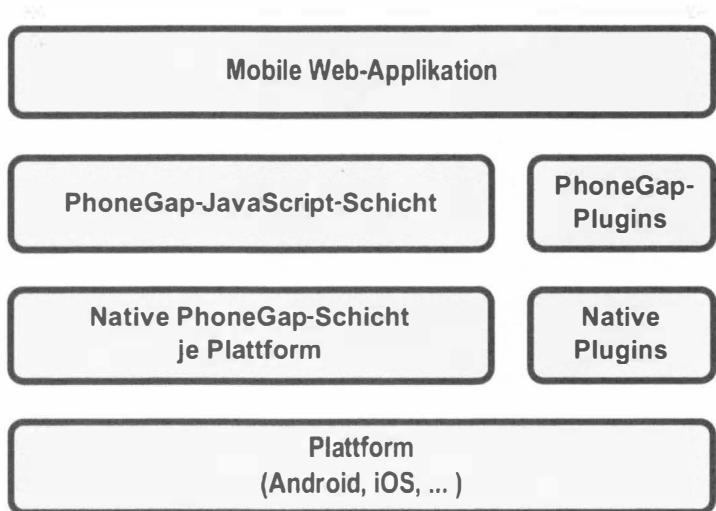
### Wie funktioniert PhoneGap?

PhoneGap verändert die Applikation kaum, sondern stellt einen eingebetteten Browser bereit, der die Webapplikation ausführt. Dabei versteckt PhoneGap den Browser so weit wie möglich und stellt ihn bildschirmfüllend dar. Zusätzlich überträgt PhoneGap Nachrichten des Gerätes an die eingebettete Applikation und geht ebenfalls den umgekehrten Weg: Benutzeraktionen an das ausführende Betriebssystem zu übertragen.

PhoneGap teilt sich in zwei unterschiedliche Schichten auf:

- Eine JavaScript-Schicht, die durch Apache Cordova abgebildet wird und eine plattform- und geräteunabhängige Schnittstelle zu den typischen Smartphonefunktionen bietet.
- Eine native Schicht, die die gerätespezifischen Funktionen kapselt und für die darüber liegende JavaScript-Schicht zur Verfügung stellt. Diese native Schicht ist für jede Zielplattform unterschiedlich.

PhoneGap erlaubt es, in beiden Schichten eigene Plugins zu erstellen und so flexibel auf Besonderheiten von Geräten zu reagieren, worin eines der Erfolgsrezepte des Projekts besteht.



**Bild 5.20:** Die Schichten einer PhoneGap-Applikation.

Das Zitate-Beispiel mit PhoneGap in eine native App zu verwandeln, würde leider den Umfang des Kapitels deutlich sprengen. Es gibt mittlerweile viele gute Tutorials für die Entwicklung mit PhoneGap.





## Anhang

### Ein Einsatzbeispiel für dynamisches JavaScript

Der folgende Anwendungsfall zeigt, wie der Einsatz der Funktion `eval()` das Leben einfacher macht. Seine Eleganz sucht man in vielen anderen Programmiersprachen vergeblich:

Eine Applikation soll flexibel um neue Funktionen erweiterbar sein und diese Erweiterungen sollen "on-the-fly" übergeben werden. Zum Beispiel soll eine Berechnungsformel mit übergeben werden, die zuvor nicht bekannt war.

Den Client könnte der Server mithilfe von HTTP-Anfragen ansprechen und direkt den neuen Quellcode in der Antwort erhalten. Der Client führt den neuen Programmteil mit der `eval()`-Funktion aus. Das folgende Beispiel zeigt, wie leicht diese Funktion auf dem Client implementierbar ist.

```
function evalRequest(url) {  
    var xmlhttpReq = new XMLHttpRequest();  
    xmlhttpReq.onreadystatechange = function()  
    {  
        if (xmlhttpReq.readyState==4 && xmlhttpReq.status==200)  
        {  
            // Direktes Ausführen des erhaltenen Codes.  
            eval(xmlhttpReq.responseText);  
        }  
    }  
    xmlhttpReq.open("GET", url, true);  
    xmlhttpReq.send(null);  
}
```

Die Funktion erstellt eine Get-Anfrage als XMLHttpRequest-Objekt. Zuerst wird eine Callback-Funktion definiert, die die Antwort des Servers entgegennimmt: `onreadystatechange()`. Der Rumpf der Methode prüft zunächst, ob die Anfrage erfolgreich war, indem die Werte der Attribute `readyState` und `status` geprüft werden. Im Contentbereich liegt die Antwort des Servers mit dem auszuführenden Code im Attribut `responseText` vor. Dieser kann direkt an die `eval`-Funktion zur Ausführung übergeben werden. Am Ende der Funktion `evalRequest` wird die Kommunikation mit dem Server mit den Befehlen `open()` und `send()` angestoßen.

Die Funktion sollte nicht direkt in produktiven Systemen eingesetzt werden. Es gibt keine weiteren Prüfungen auf Zugriffsberechtigungen oder ob der erhaltene Code für diesen Einsatzzweck vorgesehen ist.

# Stichwortverzeichnis

## A

- Accordion 129
- AJAX (Asynchronous JavaScript and XML) 15
- Amplify 228
- AngularJS 61
- AngularJS-Direktiven 68, 125
- Apache Cordova 282
- Atmosphere 231

## C

- CAP-Theorem 200
- Cloaking 20
- Collapsible 247
- Content Delivery Network (CDN) 25

## D

- Datumsauswahl 131
- Deep Links 19
- Dependency-Injection (DI) 67
- Dialogfenster 116

## E

- ECMA-262, Edition 5 27
- EcmaScript 6 56
- Emulatoren 280
- Error-Interceptoren 112
- eval() 31
- Exceptions 48

## F

- Filter 65
- Firefox OS 13
- Funktionsausdrücke 44
- Funktionskonstruktor 43

## G

- Google (JavaScript Style Guide) 47
- Google Apps for Business 15
- Google Charts 115, 120

## H

- Handlebars 178
- Handlebars Helper 186
- History API 19
- HTML5 Canvas 205

## I

- Identitätsoperator (====) 32
- Immediately Invoked Function Expression (IIFE) 45
- InstanceOf 37
- Internationalisierung 142

## J

- jQuery 184, 224
- jQuery Mobile 236
- JSF (Java Server Faces) 15
- JSON 53, 260
- JSONP (JSON mit Padding) 139

## L

- Latency Compensation 204
- Life-HTML 229
- LocalStorage 96
- LocalStorageModul 99
- Login Modul 188
- Login-Provider 194
- Lokalisierung 142

## M

- Meldungsfenster 118

- MessageBox 119  
Meteor-Collection 201  
Meteor-Eventmaps 217  
Meteorite 231  
Mobile Design  
    Guide 236  
MongoDB 173, 199  
MVC-Pattern 63
- N**
- Node.js 173  
NoSQL-Datenbanken 198
- O**
- Objektorientierte Programmierung 33  
Objektorientierung 56  
Open-Closed-Prinzip 35
- P**
- Parameter 46  
Partial Page Rendering (PPR) 15  
Properties 39  
Property Descriptoren 39  
Prototyp-Konzept 36  
Publish/Subscribe-Mechanismus 202
- R**
- Refactoring 21  
Reguläre Ausdrücke 168  
REST 105  
Restangular 109
- S**
- Same-Origin-Policy 138  
Schwache Typisierung 31  
Scopes 42  
Secure Remote Password Protokoll (SRP)  
    194
- SEO (Search Engine Optimization) 19  
Separation of Concerns 66  
Serviceorientierten Architekturen (SOA)  
    104  
Services 66  
SessionStorage 96  
Skins 237  
Slim 106  
S-O-L-I-D-Prinzipien 35  
Strict Mode 50  
Structured Query Language (SQL) 198  
Sudoku-Regeln 148  
SVG 144
- T**
- Tastatursteuerung 161  
Theme Roller 253  
Tizen 13  
Touch-Event 222  
Transitionen 242  
Twitter Bootstrap 70, 114  
typeof 29
- U**
- Umlaute 277
- V**
- Vektorgrafiken 146  
Vererbung 36  
Vergleichsoperator (==) 32
- W**
- WebGL 19
- X**
- XML (Extensible Markup Language) 53

# Single-Page-Web-Apps

Single-Page-Web-Apps sind eine neue Art, Web-Anwendungen zu bauen. Im Gegensatz zu klassischen Webseiten führen Single-Page-Web-Apps keinen Seitenwechsel mehr durch – die Oberfläche wird über dynamischen Austausch der HTML-Elemente auf einer einzigen Seite mit JavaScript verändert.

Die Implementierung erfolgt mit den Technologien HTML5, CSS3 und JavaScript. Die HTML-Seiten werden zum größten Teil dynamisch im Browser erzeugt. Die Daten werden meist über JSON oder XML mit einem Backend ausgetauscht.

## Bessere Verteilung

Eine Single-Page-Web-App ist über eine URL im Browser universell erreichbar. Eine Installation ist nicht notwendig. Im Unternehmensumfeld reduziert diese Eigenschaft enorm die Kosten für Installation und Verteilung.

## Einheitliche Plattform

Viele Benutzer wollen Dienste auf unterschiedlichen Geräten nutzen. Der Zugriff muss vom heimischen PC genauso gut funktionieren wie vom Tablet oder Smartphone aus. Eine separate Entwicklung für jede Zielplattform ist teuer. Das Web wird die übergreifende Plattform für alle Betriebssysteme und Gerätearten. Dank HTML und JavaScript laufen Single-Page-Web-Apps auf allen wichtigen mobilen Betriebssystemen wie Android, Windows Phone und iOS.

## Offline-Fähigkeiten

Mit den neuen Fähigkeiten von HTML5, wie dem LocalStorage, gibt es zum ersten Mal eine Möglichkeit, effiziente Cache-Strategien und Offline-Fähigkeiten zu etablieren.

## Geringe Einstiegshürden

Es ist sehr leicht, mit den verwendeten Technologien zu starten – als Entwicklungsumgebung reicht ein guter Texteditor aus. Zum Ausführen reicht ein Browser, der mit den entsprechenden Plug-ins sogar Debugging bereitstellt. Alle diese Komponenten sind für den Einsteiger kostenlos verfügbar.

## Aus dem Inhalt:

- JavaScript für Entwickler
- JSON – JavaScript Object Notation
- Gerüstet für die Zukunft: EcmaScript 6
- Apps mit AngularJS einfacher entwickeln
- Spiele als Single-Page-Web-App
- Das Meteor-Framework
- Private Chat-Anwendung entwickeln
- Google-Anmeldedienst für eigene Apps nutzen
- Kollektives Whiteboard im Browser
- Single-Page-Web-Apps für Smartphones und Tablets
- Mobile Apps auf dem PC entwickeln und testen
- Mobile Datenbank mit jQuery Mobile entwickeln
- Von der Web-App zur nativen App mit PhoneGap

## Über den Autor:

Heiko Spindler ist Softwarearchitekt und Autor für Themen rund um die Softwareentwicklung. Er spricht regelmäßig auf Konferenzen und schreibt Fachartikel. Als Dozent unterrichtet er seit 2008 an der Fachhochschule Gießen-Friedberg im Studiengang Wirtschaftsinformatik.

Zusätzlich beschäftigt er sich seit vielen Jahren mit Kreativitätstechniken, Gehirnjogging, Mind Mapping und Gedächtnistraining. Heiko Spindler betreibt die Websites HirnSport.de und DenkTipps.de und ist Buchautor zum Thema Gehirnjogging.



9 783645 603102

30,- EUR [D] / 30,90 EUR [A]

ISBN 978-3-645-60310-2

Besuchen Sie  
unsere Website  
[www.franzis.de](http://www.franzis.de)

FRANZIS