Hochschule **RheinMain**
University of Applied Sciences
Wiesbaden Rüsselsheim

Beschleunigung Web-basierter Anwendungen durch verbesserte SW-Architektur

Maria Mazur

Analyse & Vorgehensweise

Admin on Rails (AoR) ist eine in *Ruby on Rails* (Rails) geschriebene Webbasierte Anwendung.
Um die Anhaltspunkte für die Performance-Optimierung zu finden, muss man die Grundprinzipien von Rails sich vor Augen halten.

- don't repeat yourself (DRY)
- Konvention vor Konfiguration

Des Weiteren sollte man schauen, wie sich die Umsetzung der Prinzipien in der Anwendungsarchitektur und der Datenbankstruktur widerspiegelt.
Die historische Entwicklung der Anwendung sollte man auch nicht außer Acht lassen. Die erste Version von AoR wurde in Rails 3 realisiert. Ihm Rahmen des aktuellen Projekts erfolgte die Umstellung auf Rails 5.
Als Gegenstand für die Performance Analyse wurden die *Index Views* ausgewählt. Wie der Name verrät, ist der Index View direkt mit einer der sieben **RESTful** Aktionen assoziiert. Typischerweise stellt eine Indexaktion eine Darstellung einer Sammelressource bereit.

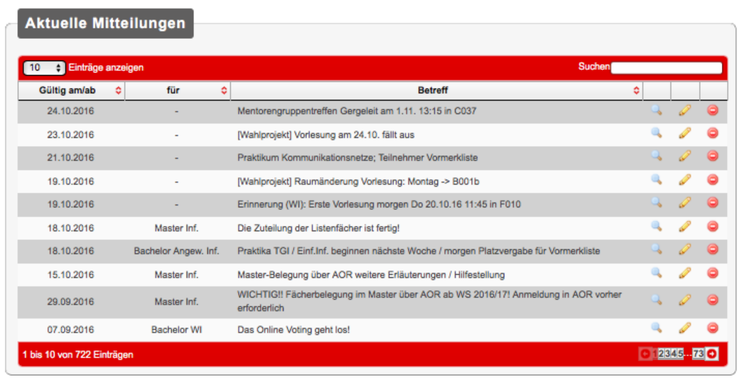


Abbildung 1: Index-View Beispiel

Tabellenansichten im AoR

Eine typische Tabelle in der AoR Anwendung verfügt neben der reinen Informationswiedergabe über die folgenden Funktionalitäten:

- Suche
- Filterung
- Sortierung
- Pagination

All dies wurde mithilfe von Datatables, einem jQuery plug-in, umgesetzt. Bis jetzt wurden die Datatables `clientseitig` umgesetzt, was den initialen Entwicklungsaufwand begrenzte und zu dem Zeitpunkt der Umstellung durch eine geringere Anzahl der Datensätze gerechtfertigt war.
Bei der `clientseitigen` Verarbeitung wurden bei der initialen Seitenaufbau alle Daten vom Server geladen, die anschließende Suche und Navigation erfolgten sehr schnell.
Solches Verhalten, kombiniert mit ständig wachsenden Datenmengen, ist aus der UX Seite kritisch.
Die Umstellung der Datatables auf `serverseitige` Verarbeitung wurde exemplarisch getestet. Dabei sollte eine Lastverteilung erfolgen und die Daten bei jeder Benutzeraktion mit z.B. Limit 10 geladen werden.

Vorteile
Ladezeitabsenkung um bis zu 60 %
Verbesserung der UX
Revision und Verschlankung der JavaScript Dateien
Einschränkungen
Pro Index View eine zusätzliche Konfigurationsdatei
Keine komplexen Index Aktionen
Decorator Pattern steht im Wege

Tabelle 1: Senkung der Ladezeit

N + 1 Problem

N + 1-Query tritt auf, wenn es eine Assoziation zu einer angeforderte Resource existiert, die zu N zusätzlichen separaten Abfragen führt.
In dem aufgeführten Beispiel ist die `announcement` Anfrage gefolgt von N `channel` Anfragen. Dies kann verhindert werden, wenn in der Index Aktion `Announcement.all` zu `Announcement.includes(:channel)` verändert wird.
Dies hat zur Folge eine einzige DB Anfrage mit einem `JOIN`, was in verbesserten Antwortzeiten resultiert.
Der Entwickler von Rails Framework, David Heinemeier Hansson, äußerte eine kontroverse Meinung zu dem Thema. Laut seiner Aussage hat die `lazy loading` Strategie vom ActiveRecord ihre Vorteile, wenn man sie mit einer geeigneten Caching Strategie, wie z.B. Russian Doll Caching kombiniert. Die individuellen Anfragen an die Datenbank sind einfach und können individuell gecached werden, was ein Vorteil gegenüber der komplizierten `JOIN` Anfragen haben kann.
Im AoR Fall ist es dennoch sinnvoller, die Menge der Abfragen zu reduzieren und sich für kürzere Queries zu entscheiden.

```
Announcement Load (1.9ms)  SELECT  `announcements`.* FROM `announcements`
INNER JOIN `channels` ON `channels`.`id` = `announcements`.`channel_id` WHERE `
announcements`.`type` = 'SimpleAnnouncement' ORDER BY announcements.effective ASC LIMIT 10 OFFSET 0

 Channel Load (0.5ms)  SELECT  `channels`.* FROM `channels` WHERE `channels`.`id` = 558717449 LIMIT 1
Rendered shared/internal/_show.html.haml (4.8ms)
Rendered shared/internal/_edit.html.haml (4.4ms)
Rendered shared/internal/_delete.html.haml (5.4ms)
 CACHE (0.0ms)  SELECT  `channels`.* FROM `channels` WHERE `channels`.`id` = 558717449 LIMIT 1
Rendered shared/internal/_show.html.haml (0.8ms)
Rendered shared/internal/_edit.html.haml (0.7ms)
Rendered shared/internal/_delete.html.haml (1.3ms)
 CACHE (0.0ms)  SELECT  `channels`.* FROM `channels` WHERE `channels`.`id` = 558717449 LIMIT 1
Rendered shared/internal/_show.html.haml (0.9ms)
Rendered shared/internal/_edit.html.haml (0.8ms)
Rendered shared/internal/_delete.html.haml (1.3ms)
```

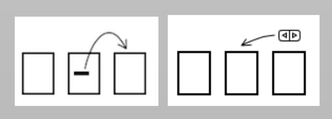
Abbildung 2: Beispiel: N + 1 Problem

Geschachtelte Partials

In diesem speziellen Fall liegt der Antwortszeitengpass an einer Stelle, die aus der konzeptionellen Seite korrekt und sehr stark nach DRY Prinzipien umgesetzt wurde. In jedem Index View soll es pro Datensatz je nach Anwendungsfall die Links zu den Show, Edit oder Delete Views geben, die alle wie von Rails vorgeschrieben dem gleichen Schema folgen. Zum Beispiel verweist ein `app/views/shared/_show_delete.html.haml` Partial auf zwei weitere Partials, die wiederum auf je eine Datei mit polymorphischen Pfad zeigen: `shared/internal/show.html.haml` und `shared/internal/delete.html.haml`
Obwohl sehr elegant gelöst, bringt in diesem speziellen Fall das `unDRYing` einen deutlichen Zeitgewinn, da Rails `fixed overhead` für jedes Partial mitkalkuliert.

Turbolinks-Technik

Turbolinks beschleunigt die Navigation der Web-Anwendung und bietet alle Leistungsvorteile einer `single page application` ohne die zusätzliche Komplexität der clientseitigen JavaScript Verarbeitung.
Rails bietet die Turbolinks-Unterstützung, die `AJAX` verwendet, um das Seiten-Rendering in den Anwendungen zu beschleunigen.
Turbolinks fügt zu allen <a>-Tags auf der Seite einen Click()- Handler. Unterstützt der Browser `PushState`, macht Turbolinks eine `AJAX`-Anfrage für die Seite, analysiert die Antwort und ersetzt den gesamten <body>-Teil der Seite durch den <body>-Teil der Antwort. Die Semantik der URLs wird dabei beibehalten.
Turbolinks initialisiert `restoration visit` bei der Vor-/Zurück-Navigation im Browser automatisch.



Da Turbolinks das normale Ladeverhalten der Seite überschreibt, soll bei der Verwendung von Javascript oder Coffee Scripten auf die richtige Initialisierung geachtet werden: `$(document).ready` soll durch `$(document).on turbolinks:load` ersetzt werden

Zusammenfassung & Fazit

Bei Ruby on Rails handelt es sich um ein relativ junges Framework, dass erst im Juli 2004 zum ersten Mal der Öffentlichkeit vorgestellt wurde. Nach der Begeisterung der Anfangsphase sank die Popularität des Frameworks seit 2008 kontinuierlich, bis 2015 der Trend sich wendete. Laut TIOBE Programming Community Index hat Ruby on Rails 2016 die höchste Popularität erreicht.
Die Geschwindigkeit der Framework Entwicklung fordert aber eine häufige Anpassung der Applikationen. Am Beispiel AoR ist deutlich zu sehen, dass die Entscheidungen, die zur Zeiten von Rails 3 und Rails 4 als optimal galten, heute revidiert werden müssen.
AoR bietet auch ein perfektes Beispiel für eine sich iterativ entwickelnde Anwendung, wo die genauen Anforderungen sich erst bei der Benutzung herauskristaillisieren.
Für die nachträgliche Laufzeitoptimierung eignen sich deshalb die vorgestellten Strategien, wie serverseitige Verarbeitung der Index Views, Reduktion der Verschachtelungsebenen bei Partials und der Einsatz von Turbolinks.

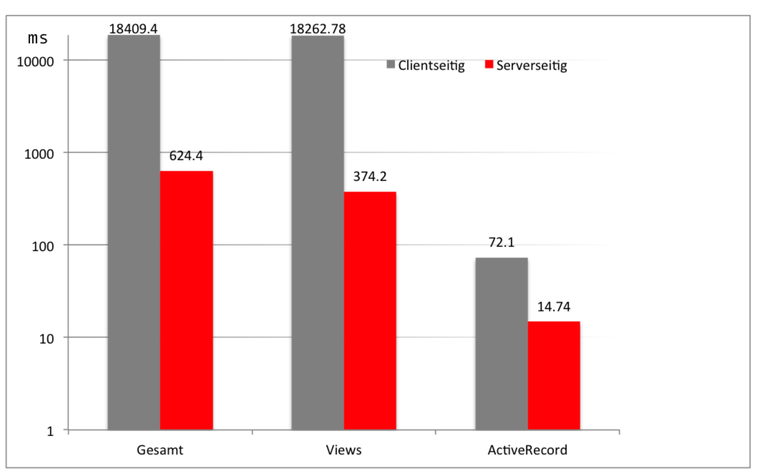


Abbildung 3: Vergleich der Ladezezeiten

Department of Computer Science
University of Applied Sciences Wiesbaden
Wiesbaden, Germany