3.6.4 Die Sache mit Apache und FCGI

Zufrieden mit der Anwendung, denke ich wieder an die Anfänge meines Experiments zurück, an die Versuche, Rails zum Laufen zu bringen oder die Probleme mit dem Debugger, die mich nicht nur dazu bewogen hatten, von Apache abzurücken und den Ruby-eigenen WEBrick-Server zu verwenden, sondern später sogar dazu, neuere Versionen von Ruby zu ignorieren, da sie in der Konstellation auf meinem Rechner schlicht keine Möglichkeit zum interaktiven Debuggen boten.

Jetzt aber stellt sich die Frage erneut – was tut eigentlich meine Anwendung mit Apache als Server oder in der InstantRails-Umgebung?

Die Ergebnisse sind erschreckend.

Während die Anbindung an Apache durch die Aktivierung des „.htaccess"-Statements „RewriteBase /First" noch absolut problemlos ist und auf den ersten Blick ganz passabel und bei den SQL-Zugriffen sogar fühlbar flotter als WEBrick erscheint, tauchen doch Fehler auf, die ich längst ausgeräumt zu haben glaubte. Nicht nur dass die Performanz gelegentlich sehr leidet oder die Umlaute der Texte korrumpiert sind, es kommen sogar wieder Abstürze vor. So führt in der Baumansicht „DB" der Link auf angezeigte, aber noch nicht in der Datenbank hinterlegte Dateien genauso wie das Drag & Drop zu einem unerwünschten Ergebnis.

Die Kontrolle unter WEBrick zeigt dagegen, dass alles völlig „normal" verläuft.

Glücklicherweise funktioniert der „originäre" Debugger von Ruby auch mit Apache, sodass ich recht schnell herausfinde, dass WEBrick wohl großzügiger ist im Umgang mit Nil-Werten. Eine kleine Abfrage reicht denn auch schon aus, um Apache wieder zufrieden zu stellen.

Unangenehmer ist jedoch, dass Apache wohl auch die Klassenvariablen strenger behandelt als WEBrick – noch vor gar nicht lange Zeit war mir bei der Bearbeitung der Drag & Drop-Funktionalität aufgefallen, dass die Klassenvariable „@@dropdrag", die ich zwischen den diversen Instanzen der Controller hin- und herschob, bei diesen Transaktionen nicht vollständig erhalten blieb. Apache freilich scheint sie sogar ganz zu verlieren, denn obwohl sie korrekt erstellt

wurde, ist sie einfach nicht mehr da, als auf sie zugegriffen werden soll. Allmählich verstehe ich auch, warum das Laufzeitverhalten bei der Baumstruktur viel schlechter erscheint als unter Verwendung von WEBrick – Apache geht vermutlich viel rabiater mit den Ruby-Threads um, sodass auch die Klassenvariablen diese Fahrt nicht überstehen.

Das vermute ich freilich nur.

Denn als ich mir ein kleines Test-Partial bastle, das mir ganz primitiv Threads, Objekte und Klassen mit Identitäten auflistet, muss ich feststellen, dass der Haupt-Thread erhalten bleibt – und dass er sogar, im Gegensatz zu WEBrick, ständig der aktuelle Thread („Thread.current“) ist. Dass die Threadverwaltung unabhängig scheint vom Web-Server macht eigentlich auch Sinn, da dieser letztlich ein „externer Hilfsarbeiter“ ist.

Wieso mischt Apache sich dann trotzdem solchermaßen stark in die Arbeit ein?

Wie das kleine Test-Partial darüber hinaus enthüllt, geht jedoch etwas anderes verloren: die Klassen-Identität. Unter Apache erweist sich also die Verwendung eigener Klassen als genauso nutzlos zur Speicherung längerfristiger Daten wie Klassenvariable in Controllern oder Models.

Um dieses Speicherproblem in den Griff zu bekommen sehe ich mir die Session-Objekte an – schließlich müssen diese ja wohl „Server-sicher“ untergebracht sein, es sieht indessen so aus, als würde Rails solche Objekte gar nicht „in-memory“ vorhalten: Mein kleines Test-Partial kann mir keine Session-Objekte anzeigen.

Was tun, sprach Zeus, die Götter sind besoffen ...

Singletons sollen „thread-save“ sein, fällt mir bei der Sucherei auf, wobei der originäre Debugger freilich längst den Verdacht nahelegt, dass dies auch nicht die Ursache meiner Probleme sein dürfte, dass der Verlust aller Klassen viel eher etwas mit dem Verbindungsaufbau zu tun haben könnte. Doch auf einen Versuch soll's mir nicht ankommen.

Wie befürchtet, wird auch im Falle von Singletons – trotz gleichbleibendem Thread – die Klasse ausgetauscht. Und natürlich ist das bei Verwendung des WEBrick-Servers nicht der Fall.

Die Frage taucht also auf, ob Klassenvariable mit Rails überhaupt zu einer solchen „Zwischenspeicherung“ taugen. Auf dem Internet finde ich freilich nichts, was dagegen spricht. Ganz im Gegenteil kommen mir einige Texte unter, die Klassenvariable sehr wohl zum Cachen irgendwelcher temporär interessanter Ergebnisse empfehlen – und das sogar zusammen mit Apache.

Also knöpfe ich mir die Konfigurationsdatei von InstantRails vor und aktiviere die folgenden Statements zusätzlich nach dessen Vorbild:

```
LoadModule auth_anon_module modules/mod_auth_anon.so
LoadModule proxy_module modules/mod_proxy.so
```

Doch auch das ist nur eine Sackgasse, also entferne ich es vorerst wieder und versuche, dem Problem der „verlorenen Klassen“ anderweitig auf die Spur zu kommen. Leider hat selbst Google Grenzen und zwar dort, wo es darum geht, die richtigen Formulierungen zu finden, die auf die richtigen Lösungen münden – was mir in diesem Fall eher schlecht als recht gelingt.

CGI untauglich für „in-memory“

Endlich jedoch stoße ich genau auf die richtige Stelle: In den Rails-Kommentaren von „caching.rb“ im Ordner „action_controller“ wird CGI als untauglich für „in-memory“-Speicherung beschrieben, weil es ständig neue Ruby-Prozesse verwendet.

Ok, dieser Hinweis passt prächtig zu meinen „verlorenen Klassen“, weshalb ich also den Umstieg auf FCGI anpeile, um festzustellen, ob das meinen Source-Code rettet. Im Wiki finde ich denn auch eine geeignete Anleitung. Da ich freilich wenig Lust (und Kompetenz) dazu habe, hier im Entwicklerteam mitzumachen und mir auch die Befehlszeilen-Notation sehr spanisch vorkommt, versuche ich, über die spezifischere Windows-Seite der Sache beizukommen. So organisiere ich mir über deren Link die Datei „mod_fastcgi.dll“ und verschiebe sie wie vorgeschrieben in den Apache-Ordner „modules“. Auch das total veraltete Ruby-Tool „Ruby For Apache“ besorge ich mir und obwohl es auf einen Fehler läuft, den ich mit „ignore“ übergehe, scheint es für meine Zwecke zu ge-

nügen. Wie beschrieben ändere ich noch die beiden Konfigurationsdateien von Apache und Rails ab: In „http.conf“ wird nur

```
LoadModule fastcgi_module modules/mod_fastcgi.dll
```

ergänzt und in der Alias-Definition der dortige CGI-Handler gegen

```
AddHandler fastcgi-script .fcgi
```

ausgetauscht. In „.htaccess“ muss ebenfalls CGI durch FCGI ersetzt werden:

```
RewriteRule ^(.*)$ dispatch.fcgi [QSA,L]
```

Und siehe da – mein Test-Partial meldet mir tatsächlich, dass meine Klassenvariablen nun gerettet werden.

Leider trübt auch hier ein Wehmutstropfen das Erfolgsgefühl, denn wie im Rails-Kommentar bereits angedeutet, funktioniert alles ausschließlich dann reibunglos, wenn Apache nur auf die „selbst-gemachten“ Ruby-Prozesse zugreifen kann. Habe ich aber weitere Prozesse, beispielsweise von einem WEBrick-Server, im Angebot, dann verhält sich meine Applikation gelegentlich sehr sprunghaft, und nur mein Test-Partial kann mir dann noch sagen, woran es liegt: an den unterschiedlichen Prozessen mit ihren unterschiedlichen „in-memory“-Inhalten, die meine Anwendung nach undurchschaubarem Muster heranzieht.

Dieses Verhalten ist sicher mit ein Grund, warum InstantRails noch einen Mongrel-Server vor Apache geschaltet hat, wobei mir das Internet zu verstehen gibt, dass Mongrel wohl im Moment der angesagte Ruby-Server ist – WEBrick ist offenbar keine Konkurrenz dazu.

Meine Version von InstantRails – Version 1.7, das auf Ruby 1.8.6 und Rails 1.2.3 basiert – erweist sich aber auch als nicht wirklich hilfreich. Um die erforderlichen MySQL-Datenbanken anzulegen, bietet es zwar den mitgelieferten phpMyAdmin an, es erkennt auch sofort – durch einfaches Kopieren der Anwendung in den Ordner „rails-apps“ – dass eine neue Applikation zu berücksichtigen ist, doch ansonsten scheinen die neueren Versionen nicht so ganz mit dem zu harmonieren, was die älteren taten oder erlaubten.

Dass „database.yml“ hinsichtlich des Passwortes korrigiert werden muss, und auch, dass die Datei „current.txt“, die den Namen der letzten ausgewählte Datenbank enthält, auf die neue Umgebung angepasst werden müssen, ist dabei nicht wirklich das Problem.

Das Problem ist, dass ich bisher meine Applikation einfach über

```
http://localhost:3000/application
http://127.0.0.1:3000/application
```

aufgerufen hatte. Das führt unter InstantRails nun aber zu einem „no route found to match“-Fehler. Zwar lässt sich die Anwendung dann doch über

```
http://127.0.0.1:3000/dateis
```

aktivieren, läuft jedoch bei manchen Aktionen wiederum auf einen Routing-Fehler auf.

Und als ich dann noch versuche, das InstantRails-eigene Ruby durch eine ältere Version auszutauschen, läuft gar nichts mehr. Nicht einmal nach einer Neu-Installation von InstantRails kann ich es mehr dazu bewegen, meine Applikation zu mögen. Diesmal sind es tatsächlich ständig wechselnde Threads, die mir meine Klassenvariablen zerstören.

Weil ich dieses Verhalten freilich auf die neueren Versionen zurückführe, belasse ich es dabei, denn Rails 1.2.x weist umfangreiche Veränderungen – wie eine modifzierte Session-Cache-Behandlung – auf und lässt damit Befürchtungen aufkeimen, dass die Migration einiges an Arbeit kosten kann.

Das aber ist „biz as usual“ und kommt in allen Programmiersprachen und Umgebungen immer wieder vor, völlig unabhängig von Ruby oder Rails, weshalb ich beschließe, dass meine Einarbeitung in RoR nun als beendet angesehen werden darf.