3.6.5.5 Regelungen der Datenbank-Beschränkung

Was generell nicht eingeschränkt wurde hinsichtlich der zugehörigen Datenbank, sind die Eigenschaften und zwar mit dem Hintergedanken, dass in modernen SOA-Architekturen Datenbanksysteme kaum noch „in sich geschlossen“ sind. Ganz im Gegenteil sollen vielfach verwendbaren Services vorliegen, deren Daten auf diese Weise den unterschiedlichsten Anwendungen zugute kommen können. Das aber bedeutet, dass diese Daten über alle möglichen Verwendungen und Dateigrenzen hinaus gleich bleiben – und damit auch die ihnen zugrunde liegenden Eigenschaften.

Da die Feldanzahl eigentlich ein Eigenschafts-Attribut ist, muss deshalb auch sie über die Datenbank-Grenzen hinaus einheitlich bleiben.

Dies trifft jedoch nicht für die Tentakelzahl zu, zumindest nicht in einem ersten Anlauf. Um die Dichte der Beziehungen zu beschreiben, muss die Betrachtung entsprechend den Systemgrenzen erfolgen. Zwar sind in größeren Applikationen mehrstufige Tentakelzahlen sicher wünschenswert, um die Vernetzung in den diversen Modulen und Modulkomplexen im Auge zu behalten, für diese kleine Anwendung jedoch, die bereits Dateisysteme von 40-60 Dateien eher schlecht als recht bedienen kann, genügt es vollauf, die erste Ebene – den eigenen Dateiverbund – zu berücksichtigen.

Auch Vorgänger und Nachfolger, also Portal- und Exitdistanzen bleiben im eigenen Rahmen. Zwar können Stammdaten-Services wie Kunden oder Teile vielen verschiedenen Anwendungen dienen, doch da die „inter-modulare“ Kommunikation nicht direkt, sondern immer über definierte Schnittstellen erfolgen muss, ist der Informationsaustausch auf dieser Ebene stark reguliert und deutlich unterschiedlich von „intra-modularem Verkehr“. Da Vorgänger und Nachfolger gerade diesen „Verkehr“ des Informationstransfers abzuschätzen erlauben, macht es Sinn, auch Portal- und Exitdistanzen „mehrstufig“ zu bestimmen. Und auch hier gilt, dass die kleine Erstanwendung für solche umfangreichen Systeme gar nicht geeignet ist und deshalb eine Beschränkung auf den betrachteten Dateiverbund vertretbar ist.

Bei den verschiedenen Ansichten der Baumstruktur gibt es ebenfalls Unterschiede. Unabhängig von der zugehörigen Datenbank bleibt neben der Eigen-

schaftsanzeige auch die Darstellung aller im eigenen System gespeicherten Feld- und Dateidaten, da für die kleinen Datenbanken, die die Applikation nur handhaben kann, das Verhalten ausreichend flott blieb.

Die Auflistungen von Dateien und ihren Feldern sollte aber natürlich abfragenden Programmteilen zu erkennen geben, zu welcher Datenbank die Datei gehört. Deshalb musste die Treelist-Funktionalität ergänzt werden und zwar um eine Methode „next_parent“ in „treelist_modul.rb“, die zu einer gegebenen Auswahl aus der Baumstruktur die passende Datenbank liefert.

Nur die Treelist-Ansicht der physikalischen Datenbank („DB“) wird auf genau diejenige eingeschränkt, die auch ausgewählt ist: Dafür wurde das Partial „_db“ in „_nav.rhtml“ integriert, um alle vom aktuellen MySQL-Server betreuten Dateisysteme zur Auswahl anzubieten. Weil diese kleine Anwendung mit größeren Dateiverbünden ziemlich kämpft, wurde zuletzt eine Ampelfunktion installiert, die bei der Funktion „Datenbank aktualisieren“ anzeigt, ob es angeraten erscheint, das aufgelistete Dateisystem tatsächlich zu übernehmen.

3.6.5.6 Beurteilung der Anwendung „FirstRails“

Die kleine Applikation zeigt, dass für einen praktisch puren Stammdaten-Verbund wie bei „FirstRails“ eine Datenbank-Bewertung anhand der 4fF-Methode nicht wirklich einen Vorteil bringt gegenüber dem reinen Augenschein – dafür sind es schlicht zu wenig Dateien und zu wenig Prozesse, die diese Dateien beschreiben.

Was sie noch zeigt, ist, dass die bequeme Art der Baumstruktur, alles auf einmal einzuladen, nicht wirklich für große Datenmengen gedacht ist – so führte eine Datenbank mit hundert Dateien und knapp über 4.500 Felder zwar nicht zu einem Programmabsturz, aber doch zu einem erheblich verzögerten Verhalten. Dateisysteme dieser Größenordnung sollten also lieber nicht mit dieser kleinen Erstanwendung untersucht werden, während – mit Geduld – Datenbanken bis etwa 50 Dateien/500 Felder im Rails'schen Design-Modus noch behandelt werden können, wobei Apache den Eindruck erweckt, ein wenig standfester zu sein. Um die in der Anwendung verwendete Zwischenspeicherung via Klassenvariablen zu nutzen, muss dieser Server jedoch mit FCGI verwendet werden.

Auch die rigorose Art der Neuberechnung (bei Änderung wesentlicher Attribute wie Feld-Eigenschaften oder -Vorgänger einfach alle Felder neu zu kalkulieren ohne jegliche Berücksichtigung von Details oder ohne die Möglichkeit, diese Gesamtberechnung irgendwann später einmal durchzuführen) macht die Anwendung über Gebühr langsam und damit nicht wirklich professionell nutzbar.

Weiterhin ist zu bemängeln, dass die Blätter-Problematik nicht hieb- und stichfest behandelt wurde. Da Browser durch Blättern sehr einfach erlauben, vorherige Zustände desselben Datensatzes „wiederzubeleben", wäre es für professionelle Applikationen unerlässlich, eine Möglichkeit zu haben, die „Version" des Datensatzes zu erfahren und entsprechend darauf zu reagieren.

Obwohl die erste eigene Rails-Applikation also nicht wirklich für Produktiv-Umgebungen taugt, zeigt sie doch, wie mächtig Ruby und Rails sind – und wie unterschiedlich die Probleme sein können, die in so vielschichtigen Systemen wie Web-Applikationen auftauchen können. Doch auch hier gilt: „Übung macht den Meister".

3.6.6 Fazit

Am Ende dieses ersten, zwar kleinen, doch ernsthaft betriebenen Software-Projektes kann trotz aller Stolpereien eines bedenkenlos bejaht werden: Ruby und Rails sind eine sehr gute Basis für datenbank-basierende Programme, sogar unabhängig davon, ob sie auf dem Web oder nur auf dem Desktop agieren.

Zwar ist der Debugger noch verbesserungswürdig – und der interaktive Debugger gelegentlich geradezu ein Gräuel – doch das dürften Kinderkrankheiten sein genauso wie die Blätterprobleme (was script.aculo.us bereits erledigt zu haben scheint) oder die Scroll-„Anomalien" bei Ajax und die Performanz.

Wird deshalb der Zeitaufwand berücksichtigt, den eine halbwegs leistungsfähige Anwendung braucht bis zur Produktionsreife, so gibt es keinen Grund für ein weiteres Zaudern: Denn der Zug „Rails" nimmt ständig Fahrt auf und es besteht wenig Grund zur Sorge, dass diese Umgebung wieder im Dunkeln der Geschichte verschwindet. Ganz im Gegenteil ist unter Berücksichtigung der modernen Tendenzen hin zu SOA/Web 2.0 davon auszugehen, dass rasant

mehr oder minder frei verfügbare Komponenten angeboten werden, mit denen sich die eigenen Anwendungen veredeln lassen.

Besonders Software-Häuser, die auf sogenannten „Legacies“ sitzen, sollten sich mit Rails beschäftigen, denn es bietet nicht nur ein weitgespanntes Netz für alle gängigen Aufgaben der datenbank-basierenden Web-Programmierung, sondern lässt sich auch einfach erlernen – und nach Belieben aufwändig an eigene Bedürfnisse anpassen.

Und selbst wenn komplexe und zeitkritische Probleme bearbeitet werden müssen, lässt sich Rails gut empfehlen, da Ruby nicht eifersüchtig ist und jederzeit für den Fall, dass die eigenen Fähigkeiten nicht ausreichen, die Möglichkeit bietet, Teile in anderen, besser geeigneten Programmiersprachen zu erstellen und sie danach zu integrieren.

Übrigens – Anwendungsbeispiele für Ruby- und Rails-Befehle lassen sich am bequemsten über die Suchfunktion der Entwicklungsumgebung (beispielsweise von RadRails) auffinden.