Christian-Albrechts-Universität zu Kiel

Institut für Informatik
Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion

Entwicklung von Web-Applikationen aus deklarativen Beschreibungen

Master Thesis

vorgelegt von
Sven Koschnicke
Matrikelnummer 715764

Oktober 2008

betreut von
Prof. Dr. Michael Hanus

Zusammenfassung

Diese Arbeit untersucht den aktuellen Stand von Frameworks zur Erstellung von Web-Applikationen (Web Application Frameworks) und stellt dabei wichtige Aspelte für die Entwicklung solcher Anwendungen heraus. Die Vorteile eines deklarativen Ansatzes bei der Entwicklung werden dargelegt und es wird schließlich ein Framework auf Basis der logisch-funktionalen Sprache Curry entwickelt, welches die Erstellung von Web-Anwendung auf einer hohen Abstraktionsebene ermöglicht. Das Framework unterstützt sämtliche Funktionalität, die in einer Web-Applikation für gewöhnlich benötigt wird. Das Grundgerüst der Anwendung kann automatisch aus einem Entity-Relationship-Diagramm erstellt werden. Danach steht eine Anwendung zur Verfügung, die dem Prinzip des Model-View-Controller-Patterns folgt und direkt einsatzfähig ist. Sie stellt eine Web-Oberfläche für die Basisoperationen auf allen definierten Entitäten zur Verfügung. Weiterhin werden Funktionen zur Verwaltung der Benutzer-Sitzungen sowie die Möglichkeit der Modellierung von Prozessen auf einer abstrakten Ebene vom Framework zur Verfügung gestellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Danksagung

Zuallererst möchte ich meinen Eltern für die großartige Unterstützung auf meinem gesamten bisherigen Lebensweg danken. Ohne sie hätte ich in meinem Studium nicht so viel erreichen können, wie ich erreicht habe. Weiterhin möchte ich meiner Freundin Steffi danken, die mir in der schwierigen Zeit während der Anfertigung dieser Arbeit, wie bisher immer, beigestanden hat. Weiterhin möchte ich Prof. Dr. Hanus für die vielen aufschlussreichen Gespräche und Hinweise danken.

Kiel, den 10. Oktober 2008 Sven Koschnicke

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe. Alle wörtlichen und sinngemäßen Zitate sind als solche gekennzeichnet.

Kiel, den 10. Oktober 2008 Sven Koschnicke

1. Motivation, Ziel und Vorgehen dieser Arbeit

Das Internet und im Speziellen das World Wide Web haben in den letzten Jahren eine Größe und Bedeutung erlangt, die nicht mehr zu übersehen ist. Durch die allgemein große Verbreitung und den dadurch einfachen Zugang zum Internet bieten immer mehr Institutionen Ihre Dienste direkt im World Wide Web an. Dabei sind diese Angebote längst nicht mehr auf statische Webseiten zur Darstellung von Informationen beschränkt. Inzwischen werden vollständige Applikationen über das World Wide Web angeboten. Die Anzahl der Web-Applikationen wächst stetig. Web-Applikationen haben gegenüber traditionellen Desktop-Anwendungen[1] die Vorteile, dass sie betriebssystemunabhängig (es wird nur ein Web-Browser benötigt) und leicht wartbar sind (der Server, auf dem die Anwendung läuft, wird vom Anbieter kontrolliert). Außerdem wird kein aufwändiger Distributionsweg benötigt, die Anwendung steht für jeden sofort zur Verfügung, der auf die entsprechende Website geht.

Durch diese hohe Beliebtheit von Web-Applikationen beschäftigen sich auch immer mehr Entwickler mit der Erstellung von immer komplexer werdenden Web-Applikationen. Die Unterschiede zu Desktop-Anwendungen erfordern hier teilweise andere Entwicklungsstrategien, es können jedoch auch viele etablierte Verfahren angewandt werden. Wie überall in der Anwendungsentwicklung wird auch bei der Web-Entwicklung versucht, durch fortschreitende Abstraktion eine schnellere Entwicklung zu ermöglichen und auch komplexere Anwendungen entwickelbar und vor allem wartbar zu machen. Dies führte von der einfachen direkten Verarbeitung von HTTP-Anfragen und zurücksenden von Webseiten zu Modellen, die die Zustandslosigkeit des HTT-Protokolles vor dem Anwender verstecken und so Web-Anwendungen ermöglichen, die sowohl aus der Sicht des Benutzers als auch aus der Sicht des Entwicklers ähnlich zu Desktop-Anwendungen sind.

Auch wenn hier schon viel geleistet wurde und dem Entwickler inzwischen mächtige Werkzeuge zur Verfügung stehen, befindet sich die Entwicklung von Web-Applikationen immer noch am Anfang und es sind noch viele Probleme zu lösen, um Web-Anwendungen auf den technischen Stand von Desktop-Anwendungen zu bringen.

[1] mit Desktop-Anwendungen sind in dieser Arbeit Rich-Client-Anwendungen gemeint, welche auf dem Rechner des Benutzers der Anwendung laufen und keine physische Trennung von Benutzerschnittstelle und restlicher Anwendung aufweisen

In der folgenden Arbeit stelle ich einige Ansätze und Werkzeuge für die Entwicklung von Web-Applikationen vor, spreche einige Probleme an, die die Web-Entwicklung allgemein und auch die vorgestellten Werkzeuge mit sich bringen und präsentiere schließlich einige Lösungen für diese Probleme, wobei die Erkenntnisse und Werkzeuge der deklarativen Programmierung genutzt werden.

Die erarbeiteten Ergebnisse setze ich zusammenfassend in einem Framework zur Entwicklung von Web-Anwendungen um.

2. Web-Applikationen

2.1. Definition

Eine Web-Applikation ist eine Client-Server-Anwendung, wobei der Client ein beliebiger Web-Browser sein kann. Der Client wird also nicht vom Entwickler der Web-Applikation entwickelt, sondern ist unabhängig von dieser. Client und Server kommunizieren über das Hypertext-Transfer-Protokoll (siehe Fielding u. a. (1999)) und der Client interpretiert hauptsächlich vom Server gesendeten Hyper-Text-Markup-Language-Code (siehe Pemberton (2002)) und Cascading-Stylesheet-Code (siehe Bos u. a. (2007)). Die gesendeten Webseiten können durch ebenfalls vom Server gesendetes und im Client ablaufendes Javascript (ein Dialekt von ECMA-Skript, siehe Cowlishaw (1999)) noch verändert werden. Weiterhin ist über Javascript eine asynchrone Kommunikation mit dem Server möglich (Ajax). Darüber hinaus ist die Einbindung von anderen Komponenten in Webseiten möglich, die aber eine entsprechende Erweiterung des Browsers benötigen und eher als eigene Client-Anwendungen zu sehen sind (beispielsweise Adobe Flash). Die Kommunikation zwischen Client und Server über das HTT-Protokoll erfolgt zustandslos, was die Entwicklung von Anwendungen, wie man sie als Desktop-Anwendungen gewohnt ist, schwieriger, jedoch keinesfalls unmöglich macht.

2.2. Heutige Bedeutung

Durch die immer schneller wachsende Verbreitung des Internets ergeben sich für Web-Anwendungen einige Vorteile gegenüber Desktop-Anwendungen. Web-Anwendungen sind leichter und schneller für Kunden nutzbar zu machen, da keine Distribution und keine Installation der Software nötig sind. Aus dem gleichen Grund ist auch eine Wartung der Software leichter, da sie zentral auf einem vom Anbieter der Software kontrollierten Server liegt. Außerdem lässt sich der Zugriff auf die Software jederzeit kontrollieren, was On-Demand-Geschäftsmodelle ohne große technische Vorkehrungen wie Digital Rights Management ermöglicht.

Aus diesen Gründen findet eine immer stärkere Verlagerung der Applikationen vom Desktop ins World Wide Web statt. Viele Anwender verwenden inzwischen ausschließlich den Webbrowser für ihre täglichen Arbeiten. Beispielsweise wissen heutzutage viele durchschnittliche Nutzer nicht mehr, dass es auch Desktop-Anwendungen zum Abrufen von eMails gibt, sondern verwenden das Web-Interface des jeweiligen eMail-Anbieters,

was natürlich den Vorteil mit sich bringt, keinen eMail-Client mehr einrichten und warten zu müssen.

Auch für Firmen sind Web-Anwendungen interessant, da Sie direkt über das Firmen-Intranet genutzt werden können und keine sonstigen Anforderungen an die Arbeitsplatzrechner stellen, wie bestimmte Betriebssystemversionen oder ähnliches.

2.3. Probleme heutiger Web-Applikationen

Obwohl das Interesse an Web-Applikationen und deren Verbreitung immer mehr zunimmt, haben die Entwicklungsmöglichkeiten von Web-Applikationen noch nicht mit denen von klassischen Desktop-Anwendungen gleichgezogen. Da das World Wide Web initial nicht als Plattform für Anwendungen gedacht war, hat sich die Anwendungsentwicklung hier hauptsächlich über kleine Skripte entwickelt, die dynamisch Webseiten oder Teile von Webseiten generierten. Diese Skripte sind meist in Skript-Sprachen wie Perl oder PHP geschrieben. Solche Skript-Sprachen sind nicht für die Erstellung komplexer Anwendungen entwickelt worden und machen die Erstellung solcher eher schwierig. Trotzdem erfreut sich vor allem die Sprache PHP wegen ihrer hohen Einsteigerfreundlichkeit sehr großer Beliebtheit.

Ein großes Problem solcher Web-Anwendungen sind die niedrige Abstraktion und die damit verbundenen Sicherheits- und Wartungsprobleme. PHP beispielsweise ermöglicht einen unkomplizierten direkten Zugriff auf Benutzereingaben aus HTML-Formularen und auf eine SQL-Datenbank. So kann es leicht passieren, dass Benutzereingaben direkt an die Datenbank weitergegeben werden und so Angriffe auf die Datenbank möglich sind. Die fehlende Typsicherheit von PHP begünstigt dies.

Zugriffe auf einen Datenbestand und Verarbeitung von Benutzereingaben sowie das Erzeugen von Webseiten, um dem Benutzer diese Eingaben zu ermöglichen sind jedoch Aufgaben, die nahezu jede Web-Applikation leisten muss. Hierfür muss nicht nur die nötige Funktionalität implementiert, sondern auch eine Struktur der Anwendung entwickelt werden, die beispielsweise einen direkten Zugriff vom Benutzerinterface auf die Datenbank möglichst erschwert.

Auch wenn jede Web-Applikation naturgemäß auf nebenläufige Ausführung ausgelegt sein sollte, ist dies in vielen Anwendungen nicht der Fall. Gerade in Web-Anwendungen, die in Skriptsprachen entwickelt werden, wird zum Beispiel selten ein Transaktions-Konzept verwendet, um eine Datenkonsistenz bei nebenläufigen Zugriffen zu garantieren. Dies liegt oft an schlechter Unterstützung dieser Konzepte durch die Skriptsprache.

Allgemein wird häufig nicht auf einen konsistenten Datenbestand geachtet. Datensätze, welche mit anderen Datensätzen verknüpft sind und von diesen benötigt werden, können einfach gelöscht werden. Der Entwickler muss selbst darauf achten, dass es nicht zu solchen Fällen kommt. Dies wird häufig jedoch vernachlässigt, da es sich nicht direkt

auf die Funktionalität der Anwendung auswirkt, sondern erst später zum Problem wird. Eine bessere Unterstützung des Entwicklers bei der Sicherstellung der Daten-Konsistenz ist daher ein Beitrag zu dauerhaft solideren Anwendungen.

3. Web-Frameworks

Wiederverwendung ist ein wichtiges Prinzip in der Softwareentwicklung. Aus diesem Grund gibt es Programmbibliotheken und Frameworks. Während Bibliotheken eine Sammlung wiederverwendbarer Funktionen darstellen, geben Frameworks Leitlinien zur gesamten Gestaltung einer Anwendung, welche auf einen großen Bereich von Anwendungen zutreffen und somit wiederverwendbar sind. Die Struktur der zu entwickelnden Anwendung wird vorgegeben, um Fehler beim Design der Anwendung oder gar ein „organisches Wachstum“ der Anwendung zu vermeiden. Außerdem stellen sie oft benötigte Funktionen (eventuell auch in Form von Bibliotheken) zur Verfügung.

3.1. Definition

Ein Framework stellt ein Gerüst für eine Software-Anwendung zur Verfügung. Im Allgemeinen findet bei der Verwendung eines Frameworks das Prinzip der Umkehrung der Steuerung („Inversion of Control“) statt. Die vom Entwickler der Anwendung geschriebenen Komponenten werden vom Framework aufgerufen. Das Framework stellt dem Entwickler also ein Gerüst zur Verfügung, in das er seine anwendungsspezifischen Funktionen einbauen kann. Die Strukturierung und die genaue Interaktion zwischen den verschiedenen Komponenten gibt das Framework vor und ermöglicht es so dem Entwickler, sich auf die Implementierung der eigentlichen Funktionalität der Anwendung zu konzentrieren.

3.2. Untersuchung bestehender Frameworks

Um nützliche Funktionen von Web-Frameworks zu identifizieren, habe ich einige dieser Frameworks untersucht, und gebe im Folgenden eine kurze Abhandlung dieser Frameworks sowie deren interessanten Funktionen. Die Beschreibung der einzelnen Frameworks erhebt keinen Anspruch auf Vollständigkeit, vielmehr sind ein paar Funktionen herausgestellt, welche ich als besonders wertvoll für die effiziente Entwicklung einer Web-Applikation halte.

3.2.1. Ruby on Rails

Das Web-Framework Ruby on Rails basiert auf der Sprache Ruby, welche eine dynamisch typisierte vollständig objektorientierte Sprache ist, die es erlaubt, sowohl imperativ als auch funktional zu programmieren. Die imperative Programmierung ist jedoch wie allgemein in der Software-Entwicklung auch in Ruby wesentlich verbreiteter und enthält Elemente, die man eher aus funktionalen Sprachen kennt, wie beispielsweise Lambda-Funktionen. Ruby on Rails verfolgt sehr stark das Prinzip der Konvention über Konfiguration, daher gibt es für nahezu alle Bestandteile des Frameworks ein Standardverhalten, welches optimalerweise das am meisten gewünschte Verhalten ist. Nur in Sonderfällen muss der Programmierer eingreifen und durch explizite Anweisungen dieses Standardverhalten abändern.

Der Beginn der Entwicklung einer neuen Web-Anwendung ist mit Ruby on Rails sehr einfach. Eines der im Framework enthaltenen Skripte erstellt automatisch die vom Framework vorgegebene Verzeichnisstruktur und alle initial benötigten Konfigurations- und Programmdateien. Da Ruby on Rails sehr stark auf Konventionen basiert, sollte die vorgegebene Struktur unbedingt eingehalten werden, um vom daraus resultierenden geringeren Konfigurationsaufwand zu profitieren.

Außerdem existieren Skripte zum Generieren der eigenen Programmkomponenten. Diese Skripte generieren automatisch eine vom Entwickler benannte Komponente in der bereits die am häufigsten benötigte Funktionalität, nämlich das Anzeigen, Anlegen, Bearbeiten und Löschen von Datensätzen, implementiert ist und direkt genutzt beziehungsweise angepasst werden kann. Das hat zum einen den Vorteil, dass dem Entwickler häufig wiederkehrende aber monotone und dadurch fehleranfällige Arbeit abgenommen wird und zum anderen kann sich der Entwickler so schnell mit den grundlegenden Verfahrensweisen des Frameworks vertraut machen, da er hierfür Code-Beispiele vorgegeben bekommt und davon lernen kann.

Ruby on Rails bietet objektrelationales Mapping an, um automatisch die Daten, welche in einer relationalen Datenbank gespeichert werden, in Objekte umzusetzen, die die Daten aus Sicht der Anwendung repräsentieren. Hierbei wird nicht nur die Repräsentation sondern auch die Abfrage der Daten abstrahiert. Daten müssen also nicht mehr direkt per SQL abgefragt werden, sondern es werden spezielle Methoden zur Verfügung gestellt, um Daten nach bestimmten Kriterien abzurufen. Dies entspricht dem *Action Record* Pattern (siehe Fowler (2002)). Hierbei wird auch stark mit Metaprogrammierung und Reflection gearbeitet, wodurch die Filter-Kriterien direkt in den Methodennamen mit einfließen können, ohne diese Methoden explizit schreiben zu müssen (zum Beispiel `Student.find_by_lastname("Koschnicke")`).

Im Gegensatz zu einigen anderen Frameworks, die sich ausschließlich auf die Entwicklung der Anwendung konzentrieren, bietet Ruby on Rails auch einige Mechanismen, um den Entwickler beim Installieren der Anwendung in einer Produktivumgebung und bei der darauffolgenden Pflege während des Produktivbetriebs zu unterstützen. So gibt es eine

Art Versionierung für das Datenbank-Schema, mit der man Transitionen zwischen einzelnen Versionen des Datenbank-Schemas als kleine Ruby-Skripte, sogenannte Migrationen, definieren kann. Diese Skripte enthalten hauptsächlich spezielle auch von Framework zur Verfügung gestellte Befehle, um Datenbank-Felder hinzuzufügen und zu entfernen (ein Beispiel der sehr einfachen Anwendung von Ruby als eine domänenspezifische Sprache). Sie können aber auch beliebige andere Funktionalität implementieren, so dass man eine fehlerfreie Aktualisierung der Anwendung im Produktivbetrieb schon beim Aktualisieren der Entwicklungsversion sicherstellen kann.

Eine weitere Erleicherung zum Aktualisieren der Produktivversion ist Capistrano. Hierbei handelt es sich allerdings um eine Erweiterung für das Framework und nicht um einen fest integrierten Bestandteil. Capistrano ist ein allgemeines Werkzeug um Aufgaben auf Servern zu automatisieren und wurde für die Installation und Wartung von Ruby on Rails Anwendungen entwickelt. Mit Capistrano kann man kleine Ruby Skripte, sogenannte Rezepte, schreiben und sie von einem beliebigen Rechner aus starten. Auch hier ist es möglich, eine Aufgabe zu revidieren und den Server in seinen vorherigen Zustand zurückzuversetzen. In Kombination mit Migrationen ist so die Installation und die Aktualisierung einer Ruby on Rails Anwendung mit einem Befehl möglich. Dieses System ist darüber hinaus bereits dafür ausgelegt eine Anwendung, die auf mehreren Servern gleichzeitig läuft, mit gleichem Aufwand zu aktualisieren, als würde sie nur auf einem Server laufen.

3.2.2. Django

Django basiert auf der Skriptsprache Python. Python weist viele Ähnlichkeiten zu Ruby auf und auch das Framework Django enthält viele ähnliche Funktionen wie Ruby on Rails. So kann auch in Django eine Projektstruktur automatisch generiert werden und auch die Generierung von Anwendungsteilen für die am häufigsten benötigte Funktionalität wird unterstützt.

Eine interessante Funktion von Django ist die Verarbeitung der URLs durch das Framework. Da eine Web-Anwendung für gewöhnlich unter einer eigenen Domain oder Subdomain läuft, werden alle URLs unter der Domain vom Framework kontrolliert und es fällt dadurch die sonst übliche Einschränkung, dass sich URLs auf korrespondierende Pfade des Dateisystems des Servers beziehen, weg. Bei einer Web-Anwendung kann der URL als Teil der Benutzerschnittstelle der Anwendung gesehen werden. Daher ist es erstrebenswert, den URL vom eigentlichen Aufbau der Anwendung, der eher mit der Anwendungslogik zu tun hat, zu entkoppeln. Dies unterstützt Django durch die Möglichkeit, sogenannte URL-Patterns zu definieren. Diese Patterns bestehen aus einem regulären Ausdruck und dem Teil der Anwendung, der aufgerufen werden soll, wenn der reguläre Ausdruck auf den aufgerufenen URL passt. Gleichzeitig können dadurch Benutzereingaben, die durch einen GET-Request über den URL übergeben werden, validiert werden.

Allerdings ist diese Validierung wegen des Einsatzes von regulären Ausdrücken nur begrenzt einsetzbar. Ein Datum kann damit zum Beispiel nicht validiert werden, es kann jedoch sichergestellt werden, dass die Eingabe eine Zahl ist.

Django abstrahiert noch nicht besonders stark von dem Request-Modell des HTT-Protokolles, so wird beispielsweise vom Framework ein Request-Objekt zur Verfügung gestellt, über das man Benutzereingaben abfragt. Als Antwort und zur Anzeige einer HTML-Seite wird dann ein Response-Objekt generiert.

Angezeigte Webseiten werden aus Templates generiert, welche aus HTML-Code mit eingebetteten Code-Stücken bestehen. Templates können aus mehreren Blöcken bestehen, wobei für Blöcke eine Art Vererbung möglich ist, wie man sie aus der objektorientierten Programmierung kennt.

Listing 3.1: Django Templates

```
<html>
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block sidebar %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block content %}{% endblock %}
    </div>
</body>
</html>
```

Der hier definierte (und noch leere) Block mit dem Namen `content` kann nun durch ein anderes Template erweitert werden.

Listing 3.2: Vererbung

```
{% extends "base.html" %}

{% block title %}My amazing blog{% endblock %}

{% block content %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
```

```
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

Gleichzeitig wird hier der Block `title` überschrieben, um den Titel der Seite abzuändern.

3.2.3. Seam

Seam ist ein Framework, welches auf die Komponenten der Java Enterprise Edition aufbaut (siehe Farley (2007)). Auch wenn es sehr modular aufgebaut ist, wird es meistens in Verbindung mit dem JBoss Application Server, Java Server Faces und dem Java Persistence Framework eingesetzt. Seam abstrahiert sehr stark vom zustandslosen HTT-Protokoll, was dadurch erleichtert wird, dass auf dem Applikation-Server Objekte auch über den Zeitraum einer Anfrage hinaus existieren können. Um Objekte nicht unnötig lange im Speicher halten zu müssen, stellt Seam ein ausgefeiltes System von Kontexten zur Verfügung, denen die Objekte zugeordnet werden können und die dann die Vorhaltezeit der Objekte bestimmen. Es gibt fünf Kontexte: Application, Session, Conversation, Page und Request. Objekte im Application-Kontext existieren solange die Anwendung läuft, Session-Objekte während der gesamten Session eines Benutzers, Conversation-Objekte während einer vom Entwickler bestimmten Abfolge von Interaktionen (zum Beispiel eine Wizard-ähnliche Abfolge von einigen Seiten zur Erstellung eines Datensatzes) und Objekte im Page- beziehungsweise im Request-Kontext nur während der Anzeige einer Seite beziehungsweise während eines Requests des Browsers, was wegen asynchronen Anfragen über Javascript unterschieden werden muss (siehe Abbildung 3.1).

Darüber hinaus gibt es noch einen speziellen Kontext, den Business-Process-Kontext. Seam integriert jBPM[1], eine Implementierung zur Modellierung von Geschäftsprozessen. Der Business-Process-Kontext dient in Verbindung damit dazu, Objekte auch über die Laufzeit der Anwendung hinaus zu erhalten. Dazu werden zugehörige Daten automatisch in die Datenbank persistiert. So können in Seam Geschäftsprozesse auf einer sehr abstrakten Ebene modelliert und verändert werden, ohne dass die Änderung von Programmcode nötig wird (vorausgesetzt, es wird keine neue oder andere Funktionsweise der Anwendung benötigt).

Durch die Nutzung von Enterprise Java Beans 3.0 kann mit Seam die Datenbank vollständig aus den Klassendefinitionen der Model-Klassen erstellt werden. Gleichzeitig bietet EJB 3.0 ein sehr hochentwickeltes objekt-relationales Mapping an (siehe Burke u. Monson-Haefel (2006)).

Wie Ruby on Rails und Django können die Projektstruktur und auch Anwendungskomponenten automatisch durch einen Generator erzeugt werden.

[1] http://www.jboss.com/products/jbpm

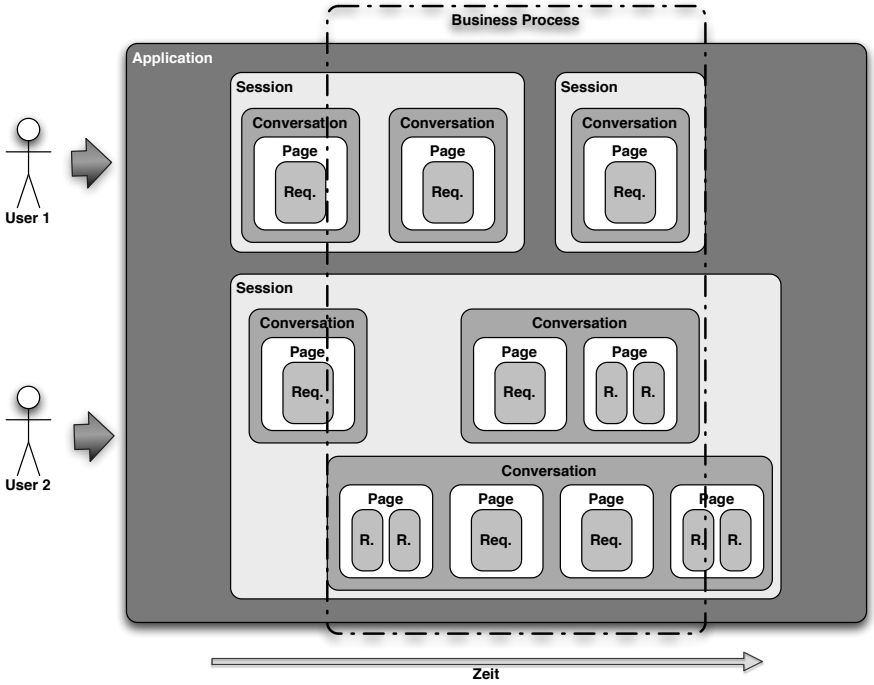

Abbildung 3.1.: Seam Kontexte und deren Lebenszeiten

3.2.4. Lift

Das Framework Lift baut auf der Sprache Scala auf. Diese wiederum erweitert Java um funktionale Komponenten wie Funktionen höherer Ordnung. Scala Programme laufen in der Java Virtual Machine und können auf vorhandene Java-Bibliotheken zugreifen. Lift läuft daher auch in einem beliebigen Java Servlet Container (wie beispielsweise Tomcat oder Jetty). Lift betont Sicherheit und Skalierbarkeit.

Eine Eigenart von Lift ist das Abweichen vom klassischen *Model View Controller* Konzept[2], in dem der Controller zuerst aufgerufen wird und dann die Views bestimmt, die angezeigt werden sollen. Lift verwendet das „View-First-Prinzip“, in dem ein Aufruf einem View zugeordnet wird, und der View dann die aufzurufenden Controller bestimmt (siehe Abbildung 3.2). Dies wird damit begründet, dass eine Webseite meistens viele verschiedene Bereiche mit verschiedenen Aufgaben beinhaltet und daher eine Seite nicht einem Controller zugeordnet werden kann. Es ist sinnvoller, den View die vielen verschiedenen Controller zusammenfassen zu lassen, anstatt einen Controller zum Hauptcontroller der Seite zu machen und diesen die anderen Controller aufrufen zu lassen.

[2]auf das *Model View Controller* Pattern wird in Abschnitt 4.4 näher eingegangen

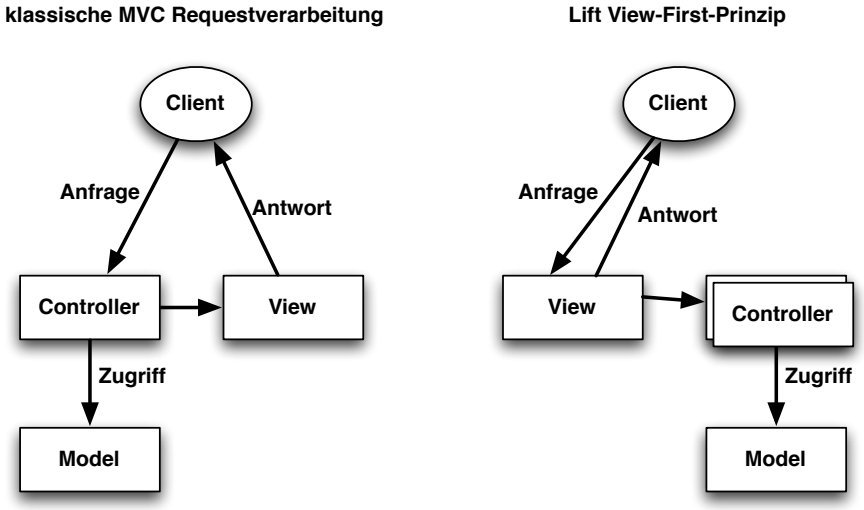

Abbildung 3.2.: Das „View-First-Prinzip“ von Lift

3.2.5. Erlyweb

Erlyweb basiert auf der funktionalen Sprache Erlang, die gerade wegen ihrer Orientierung zur Nebenläufigkeit zur Zeit wieder viel an Popularität gewinnt. Erlyweb unterstützt die automatische Generierung der Projektstruktur sowie das Generieren von Komponenten für das Erstellen, Bearbeiten und Löschen von Datensätzen. Als Struktur für die Anwendung wird auch bei Erlyweb das *Model View Controller* Pattern verwendet. Zum Speichern der Daten dient eine Abstraktionsschicht für SQL, welche sich ErlyDB nennt und die Nutzung von relationalen Datenbanken ermöglicht, ohne direkte SQL-Anfragen aus der Anwendung heraus stellen zu müssen.

Aufgrund der funktionalen und nachrichtenbasierten Natur von Erlang wird das Ergebnis eines Aufrufes durch die vom Controller zurückgegebene Nachricht bestimmt. Hier können entweder Referenzen auf andere Controller zurückgegeben werden oder eine Menge von Daten mit einer Referenz zu einem View, der dann angezeigt wird und in den die Daten eingefügt werden sollen. Hier einige Beispiele:

- `ewr`
  Weiterleitung des Browsers an den Startpunkt der Applikation.
- `{ewr, ComponentName}`
  Weiterleitung des Browser an den Startpunkt der Komponente.
- `{ewr, ComponentName, FuncName}`
  Weiterleitung des Browser an den URL der angegebenen Komponente und Funktion.

- `{data, Data}`
  Aufruf des zur aktuellen Funktion gehörenden Views und Übergabe von Daten.

3.2.6. Seaside

Seaside ist ein Web-Framework, welches auf Smalltalk aufbaut (Seaside steht für „Smalltalk Enterprise Aubergines Server with fully Integrated Development Environment“). Smalltalk ist eine objektorientierte Sprache, aus der ursprünglich auch das *Model View Controller* Konzept stammt.

In Seaside werden die Webseiten nicht durch das Einfügen von Daten in HTML-Templates erzeugt, sondern durch Smalltalk-Code beschrieben.

Listing 3.3: Smalltalk Seitenbeschreibung

```
html table:
   [html tableRow with:
       [html tableData with: [html bold: 'Name'].
        html tableData with: person name].
    html tableRow with:
       [html tableData with: [html bold: 'Age'].
        html tableData with: person age]]
```

Seaside entfernt sich etwas vom zustandslosen Modell, indem es an beliebiger Stelle sogenannte Callbacks ermöglicht. Ein Callback löst immer direkt ein Ereignis aus. So kann zum Beispiel der Klick auf einen Button direkt mit einer Aktion auf dem Server verbunden werden. Die eigentliche Umsetzung liegt hier natürlich beim Framework, aber für den Entwickler wird dadurch ein Programmieren möglich, bei dem man nicht in Requests und Client-Server-Separation denken muss.

Auch Session-Management wird über Callbacks realisiert. Ein Callback kann auch eine neue Seite Anzeigen, um die benötigten Informationen zu erhalten. So wird im folgenden Beispiel durch den Aufruf von `Colorpicker` eine neue Seite angezeigt, auf dem der Benutzer eine Farbe wählen kann. Die gewählte Farbe wird dann auf der ursprünglichen Seite angezeigt. Der Kontext des Benutzers bleibt also über mehrere Seiten erhalten.

Listing 3.4: Smalltalk Callback als Session

```
changeBackgroundColor
  |color|
  color := self call: ColorPicker new.
  blog backgroundColor: color
```

4. Theoretische Überlegungen zum Framework

4.1. Ziele des Frameworks

Bei der Untersuchung der im letzten Kapitel vorgestellten Frameworks und aus der praktischen Arbeit mit Web-Applikationen ergeben sich einige Kern-Funktionalitäten, welche ein Web-Framework beinhalten sollte, um eine Anwendungsentwicklung effektiv unterstützen zu können.

Die sinnvolle Strukturierung einer Anwendung ist zu Beginn des Projektes eine nötige Aufgabe, der jedoch nicht immer genug Bedeutung zugemessen wird, da die anfängliche Anwendung klein und leicht überschaubar ist. Mit Zunahme der Komplexität wird eine Strukturierung jedoch immer notwendiger und gleichzeitig auch schwieriger. Es existieren bereits Entwurfsmuster wie das *Model View Controller* Pattern, die sich sowohl für die Strukturierung der Anwendung als auch deren Dateien und Verzeichnisstruktur nutzen lassen. Ein Framework, welches die Struktur der Anwendung vorgibt, sollte ein solches Architekturmuster umsetzen und gleichzeitig auch die Strukturierung der Dateien und Verzeichnisse übernehmen. Die Nutzung eines bekannten Architekturmusters hat den Vorteil, dass die Struktur für einen Entwickler, der zwar das Architekturmuster kennt, das Framework jedoch nicht, leicht zu durchschauen ist. Ein Generator für das initiale Erzeugen der Anwendungsstruktur, der nötigen Konfigurationsdateien des Frameworks sowie von Komponenten der Anwendung ist hier wünschenswert, da das manuelle Anlegen dieser Dinge eine unnötige, langwierige und fehleranfällige Aufgabe darstellt.

Die automatische Erstellung von Anwendungskomponenten hat einen weiteren Vorteil. Der Entwickler sieht so die Funktionsweise des Frameworks an Code-Beispielen und hat, da er den generierten Programmcode nur noch an seine Vorstellungen anpassen muss, einen viel leichteren Einstieg. Natürlich können so auch viel schneller Erweiterungen der Anwendung implementiert werden, da ein Grundgerüst automatisch erstellt wird und der Entwickler nur noch den Teil implementieren muss, der die eigentliche neue Funktionalität der Anwendung darstellt.

Ein Framework sollte aus modularen Komponenten aufgebaut sein, die sich auch durch andere Komponenten, welche die gleiche Aufgabe erfüllen, austauschen lassen. So können zum einen schon bestehende und erprobte Komponenten für das Framework genutzt

werden und zum anderen ist so eine Anpassung auf individuelle Anforderungen durch den Austausch einer Komponente leichter möglich.

Eine hohe Abstraktion sollte dem Entwickler die Einschränkung des zustandslosen HTT-Protokolles abnehmen. Das Framework sollte also die Möglichkeit bieten, Daten während des gesamten Zeitraumes, in dem ein Nutzer die Anwendung verwendet, an diesen Nutzer gebunden zu speichern und zu laden. In Verbindung hiermit ist auch die Abbildung von Prozessen interessant. Eine Folge von Interaktionen zwischen Benutzer und Anwendung kann meist in einem Prozess beschrieben werden (im einfachsten Fall ist es eine lineare Abfolge von Seitenaufrufen). Diese Prozesse auf einer hohen Abstraktionsebene modellieren zu können verbessert die Entkoppelung der einzelnen Anwendungsteile, da die Prozesse nicht mehr implizit durch den Programmcode definiert werden.

Ein weiterer Schritt zur besseren Separation der Anwendungsteile ist die Entkoppelung von URL und Anwendungskomponente, die durch den URL aufgerufen wird. Welcher URL welchen Anwendungsteil aufruft, sollte vollkommen unabhängig von der tatsächlichen Anwendungsstruktur sein. So sollte es beispielsweise möglich sein, ein URL wie `http://www.webapp.de/login/password_reminder` mit einer Anwendungskomponente zu verknüpfen, die in der Anwendungsstruktur zur Benutzerverwaltung und nicht wie von dem URL suggeriert zur Authentifizierung gehört. Dies ermöglicht auf der einen Seite benutzerfreundlichere URLs ohne auf der anderen Seite eine gute Strukturierung der Anwendung aufgeben zu müssen.

4.2. Vorteile funktional logischer Programmierung

Die Effizienz eines Frameworks hängt auch immer stark von der Wahl der zugrundeliegenden Programmiersprache und damit von den gewählten Programmierpardigmen ab. Am meisten verbreitet sind heutzutage imperative objektorientierte Sprachen, wie Java oder Ruby. Gerade im Bereich der Web-Applikationen sind durch die geschichtliche Entwicklung des World-Wide-Webs sogenannte Skript-Sprachen, welche sich dadurch auszeichnen, zur Laufzeit interpretiert zu werden, stark vertreten. Für Web-Applikationen ist eine hohe Robustheit jedoch besonders wichtig, da Fehler in Applikationen grundsätzlich von allen Personen ausgenutzt werden können, die Zugriff auf die Applikation haben. Web-Applikationen sind meist über das Internet zugreifbar, was eine maximal mögliche Zahl von Angreifern bedeutet. Aus diesem Grund sollten mögliche Fehler schon bei der Erstellung der Anwendung erkannt und verhindert werden.

Eine mögliche Fehlerquelle ist die dynamische Typisierung von Variablen oder Ausdrücken. Diese wird von den meisten Skript-Sprachen genutzt, da statische Typisierung oft eine explizite Angabe eines Typs verlangt und dadurch einen Mehraufwand bei der Entwicklung bedeutet. Typinferenz, also das Ableiten des Typs eines Ausdrucks aus seinem Kontext, ermöglicht eine statische Typisierung ohne häufige explizite Angabe des Typs. Durch statische Typisierung können mögliche Typfehler schon vom Compiler gefunden werden. Statische Typisierung hilft außerdem dem Entwickler, seine Programme

klarer zu formulieren. Durch den Zwang eines eindeutigen Typs und die Überprüfung dieser Regel durch den Compiler können ebenfalls oft Fehler und Fehlentscheidungen im Design des Programmes frühzeitig erkannt und behoben werden.

Eine weitere Eigenschaft von funktionalen Programmen ist die schon von Anfang an hohe Abstraktionsebene. Da man sich auf die Beschreibung des Ergebnisses konzentriert und nicht auf die Beschreibung des Weges dorthin ist es leichter, abstrakte Konzepte im Programmcode zu sehen und daraus zu extrahieren, was die Modularität und wiederum die Abstraktionsebene des Programmes erhöht.

Ein wichtiges Werkzeug zum Erreichen einer hohen Abstraktion sind Funktionen höherer Ordnung. Dies sind Funktionen, welche Funktionen als Argumente verarbeiten können oder selbst wieder Funktionen zurückgeben. Dies ermöglicht eine noch höhere Parametrisierung und somit Abstraktion von Funktionen.

Auch die Skalierbarkeit von Web-Applikationen ist von großer Bedeutung. Die Anzahl der Benutzer einer Web-Applikation ist meist durch Faktoren bestimmt, die nicht im Einflussbereich des Anbieters der Applikation stehen. So kann durch einen Fernsehbericht über ein Produkt, welches von einem bestimmten Online-Shop geführt wird, die Anzahl der Zugriffe auf diesen Online-Shop innerhalb von wenigen Minuten massiv ansteigen. Oder der große Nutzen einer anderen Applikation spricht sich schnell im Internet herum und die Nutzerzahl steigt sprunghaft an. Kann die Applikation in solchen Fällen nicht mitskalieren, bedeutet dies meist einen Ausfall der Applikation, da sie durch die vielen Zugriffe nicht mehr nutzbar ist. Hier bleibt dann nur eine sehr kurze Zeitspanne, um durch Aufrüsten der Infrastruktur eine höhere Leistung zu erreichen. Die Verbesserung der Skalierbarkeit im Nachhinein ist meist aussichtslos. Da wir inzwischen an die Grenze des technisch Machbaren bei der Leistung eines Prozessors gekommen sind, heisst Skalierung gleichzeitig Verteilung auf einen Verbund von Rechnern oder Prozessoren. Sprachen, die mit Seiteneffekten arbeiten, erschweren diese Verteilung. Funktionale Sprachen sind jedoch leichter verteilbar, da die rein funktionalen Teile der Anwendung ohne Probleme auf beliebigen und beliebig vielen Rechnern ausgeführt werden können. Außerdem ermöglicht die seiteneffektfreie Programmierung bessere Optimierungsmöglichkeiten durch den Compiler.

In vielen Applikationen, also auch speziell in Web-Applikationen, geht es um die Verarbeitung komplexer Daten. Dies geschieht meist durch das direkte Ansprechen von relationalen Datenbanken und die anschließende Weiterverarbeitung. Wenn es um komplexe Zusammenhänge geht, ist eine logische Sprache jedoch viel geeigneter, da sie die direkte Formulierung der gewünschten Information ermöglicht und die Implementierung der Verarbeitung der Rohinformationen, die zu der gewünschten Information führt, unnötig macht.

Allgemein lassen sich Programme in regelbasierten, also funktionalen und logischen Sprachen, sehr einfach erweitern, indem man neue Regeln hinzufügt. Das Hinzufügen von

Regeln führt aufgrund der abstrakteren Natur dieser Sprachen zu einem höheren Zuwachs an Funktionalität als beispielsweise bei imperativen objektorientierten Programmen.

4.3. Deklarativer Ansatz

Aufgrund der vielen Vorteile deklarativer Programmierung soll beim Entwurf des Web-Frameworks dieser Weg gewählt werden. Ziel ist es also, auf einer möglichst hohen Abstraktionsebene beschreiben zu können, was die Web-Applikation erzeugen soll. Dies wird beispielsweise auch bei Ruby on Rails durch den Einsatz von Ruby als domänenspezifische Sprache versucht. Mit deklarativer Programmierung kann eine Abstraktion jedoch auf einem noch höheren Level betrieben werden, da die deklarative Programmierung speziell auf hohe Abstraktion abzielt.

4.4. MVC-Schichtenarchitektur

Das *Model View Controller* Konzept hat sich in den letzten Jahren im Bereich der Web-Applikationen aber auch darüber hinaus durchgesetzt. Es unterstützt eine ausreichende Trennung der Applikationsschichten, ohne durch zu viele Schichten zu komplex für den allgemeinen Einsatz zu werden.

Die MVC-Schichtenarchitektur beschreibt eine Aufteilung der Anwendung in drei Schichten: die Model-Schicht, die Controller-Schicht und die View-Schicht.

Die Model-Schicht repräsentiert den Zugriff auf die Daten der Anwendung, die Models. Ein Model beschreibt die Geschäftsdaten einer Anwendung und beinhaltet meist einen abstrakten Datentyp, welcher speicherbare Daten repräsentiert. Models können zueinander Beziehungen haben. Die Daten und Beziehungen der Model-Schicht kann man auch als Entity-Relationship-Diagramm darstellen, hierbei entsprechen die Entitäten den Models. Models haben darüber hinaus auch Eigenschaften, die nicht unbedingt direkt gespeichert, sondern aus gespeicherten Daten hergeleitet werden. So kann ein Model, welches eine Person repräsentiert und Vorname und Name dieser Person speichert, auch eine Eigenschaft „vollständiger Name“ besitzen, welche aus den Daten für Vorname und Name vom Model berechnet wird. Dies kann schon als einfaches Beispiel für Geschäftslogik gesehen werden, welche auch in den Models enthalten ist. Aber nicht nur transiente Eigenschaften werden im Model definiert, sondern auch domänenspezifische Funktionalität wie beispielsweise die Funktion „Buchen“ bei einem Model, welches Flugreisen repräsentiert. So bilden die Models die Geschäftslogik der Anwendung ab, sind aber völlig unabhängig von der Datendarstellung oder deren Manipulation, dem Benutzerinterface.

Die View-Schicht definiert die Benutzerschnittstelle der Anwendung. Im Falle der Web-Applikation sind dies die Webseiten, mit denen der Benutzer interagiert und auf denen

Daten angezeigt und eingegeben werden. Die Views können die zum Anzeigen und Abfragen von Daten benötigte Logik enthalten (beispielsweise das Iterieren über eine Menge von Datensätzen, um diese in einer Tabelle anzuzeigen und das Darstellen jeder zweiten Zeile der Tabelle in einer anderen Farbe). Jegliche andere Logik gehört jedoch nicht in diese Schicht.

Die Controller-Schicht enthält die gesamte Zugriffslogik der Anwendung. Die in dieser Schicht befindlichen sogenannten Controller steuern sowohl den Zugriff auf die Models als auch die Views. Dieser Zugriff geht nur von den Controllern aus, das heißt es findet kein Zugriff von den Models auf die Controller oder von den Views auf die Controller statt. Die Controller sind somit für das Entgegennehmen und die Verarbeitung der Benutzereingaben sowie für die Abfrage und Manipulation der Models zuständig. In Desktop-Anwendungen ist die Trennung zwischen Controller und View oft nicht so stark ausgeprägt wie in Web-Anwendungen, bei denen eine genaue Trennung zwischen View und Controller sinnvoller ist, da hier auch eine starke Trennung zwischen Client und Server besteht. So wird der View und unter Umständen auch Teile der zum View gehördenden Darstellungslogik auf den Client übertragen und dort dargestellt beziehungsweise ausgeführt. Eine Manipulation der Daten, was in den Aufgabenbereich des Controllers fällt, findet jedoch nie auf dem Client statt (siehe Fowler (2002)).

4.5. Die logisch-funktionale Sprache Curry

Curry (Hanus u. a. (2006)) ist eine Programmiersprache, die das Programmieren nach mehreren Programmierparadigmen ermöglicht. Die beiden Hauptparadigmen bei Curry sind die funktionale und die logische Programmierung. Die Syntax von Curry ähnelt sehr stark der von Haskell. Curry ist eine statisch stark typisierte Sprache, wobei Typen sowohl explizit angegeben werden als auch, wenn möglich, durch Typinferenz bestimmt werden können. Es wird dem Programmierer also möglichst viel „Schreibarbeit“ abgenommen, ohne die Vorteile einer statischen Typisierung aufzugeben.

Neben den allgemein bekannten primitiven Typen unterstützt Curry die Definition von eigenen Datentypen. Dies wird durch das Schlüsselwort `data` gekennzeichnet. Typbezeichner beginnen standardmäßig immer mit einem Großbuchstaben. Die Definition eines Datentyps erfolgt über die Angabe von Konstruktoren, welche ebenfalls mit einem Großbuchstaben beginnen. Als Beispiel wird in der folgenden Definition der Datentyp `Tree` für einen Binärbaum beschrieben, welcher an seinen Knoten jeweils eine Ganzzahl (`Int`) enthält:

```
data IntTree = Node IntTree Int IntTree | Leaf Int
```

Ein Binärbaum besteht also immer aus entweder einem inneren Knoten (Konstruktor `Node`), welcher eine Ganzzahl und zwei Teilbäume enthält, oder einem Blatt (Konstruktor

`Leaf`), welches nur eine Ganzzahl enthält. Mit Hilfe der Konstruktoren kann man einen konkreten Baum definieren:

```
aConcreteTree =
  Node
    (Node (Leaf 4) 2 (Leaf 5))
    1
    (Node (Leaf 6) 3 (Node (Leaf 8) 7 (Leaf 9)))
```

Diese Definition entspricht der Abbildung 4.1.

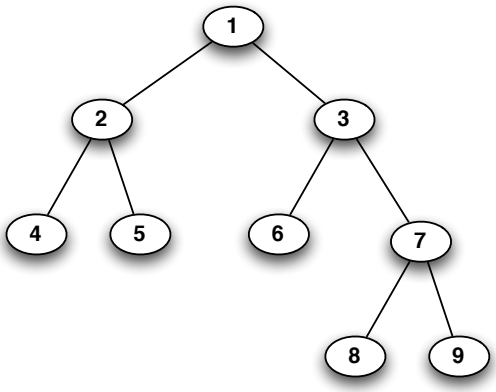

Abbildung 4.1.: Ein Binärbaum mit Ganzzahlen

Aus dem obigen Binärbaum von Ganzzahlen kann das Konzept eines allgemeinen Binärbaumes abstrahiert werden, welcher beliebige Daten enthalten kann. In Curry wird dieses Konzept durch Typvariablen realisiert:

```
data Tree a = Node (Tree a) a (Tree a) | Leaf a
```

Der Kleinbuchstabe `a` steht dabei für eine Typvariable und kann bei der Deklaration eines Baumes durch einen beliebigen Typ ersetzt werden. So entspricht `Tree Int = ...` genau der vorherigen speziellen Baumversion `IntTree = ...`.

Weiterhin ist es möglich, Typsynonyme zu definieren, um einem bestehenden Typ einen für die Anwendungsdomäne aussagekräftigeren Namen zu geben. Dies wird in Curry durch das Schlüsselwort `type` eingeleitet:

```
data Direction = North | East | South | West
type Route = [Direction]
```

Eine Route wird hier als Liste von Richtungen beschrieben. `[]` ist ein von Curry vorgegebener spezieller Konstruktor für Listen. Listen und Tupel sind wichtige Kernbestandteile der Sprache. Eine Liste enthält beliebig viele Elemente gleichen Typs, ein Tupel enthält eine vorgegebene Anzahl Elemente verschiedenen Typs in vorgegebener Reihenfolge. Die Konstruktoren für Tupel verwenden runde Klammern, so lautet der Konstruktor für ein Paar `(,)`, für ein Tripel `(,,)` und so weiter.

Tupel können dazu verwendet werden, eine Sammlung von Daten eines bestimmten Typs darzustellen.

```
data Bread = Wheat | White | Rye
data Vegetable = Tomatoe | Pepper | Cucumber
data Meat = Beef | Chicken

type Burger = (Bread, Vegetable, Meat, Bread)
```

Außerdem können Sie benutzt werden, wenn sichergestellt sein soll, dass eine bestimmte Anzahl von Daten benötigt wird.

```
type PeriodOfTime = (ClockTime, ClockTime)
```

Als funktionale Sprache unterstützt Curry Funktionen höherer Ordnung, das heißt Funktionen können in Curry Funktionen als Argumente nehmen oder Funktionen zurückgeben. Damit einher geht die Möglichkeit, anonyme Funktionen, auch Lambda-Funktionen genannt, zu definieren.

```
-- Anwendung von f auf 2
argumentTwo f = f 2

-- Übergabe der Funktion * (Multiplikation)
multiplyWithTwo = argumentTwo (*)

 -- Übergabe Lambda-Funktion
addThreeAndMultiply = argumentTwo (\x -> (*) (3 + x))

multiplyWithTwo 5       -- ergibt 10
addThreeAndMultiply 5   -- ergibt 25
```

Die Funktion `argumentTwo` nimmt eine beliebige Funktion und wendet diese auf die Zahl Zwei an. `multiplayWithTwo` übergibt `argumentTwo` die Funktion `*`, die geklammert werden muss, da sie als Infix-Funktion definiert ist. Es ergibt sich somit eine Funktion, welche noch eine weitere Ganzzahl nimmt und diese mit Zwei multipliziert.

Schließlich übergibt `addThreeAndMultiply` der Funktion `argumentTwo` eine Lambda-Funktion, welche ein Argument `x` nimmt, auf dieses Drei aufaddiert und eine Funktion liefert, die das Ergebnis mit einer zu übergebenden Ganzzahl multipliziert. Der Aufruf `addThreeAndMultiply 5` ergibt also `(*) (3 + 2) 5`. Durch Funktionen höherer Ordnung lassen sich viele Konzepte weiter abstrahieren. Beispielsweise kann das Durchgehen einer Liste und die Anwendung einer Funktion auf jedes Listenelement mit Funktionen höherer Ordnung leicht beschrieben werden, da die Funktion, welche auf die Listenelemente anzuwenden ist, einfach als Argument übergeben und so nicht genauer spezifiziert werden muss.

Das zweite Hauptprogrammierparadigma von Curry ist die logische Programmierung. Curry ermöglicht hierzu die Definition von freien Variablen, deren Ergebnis berechnet wird, so dass der Ausdruck, in dem die freie Variable vorkommt, reduzierbar ist. Weiterhin können Bedingungen definiert werden, in denen freie Variable so belegt werden, dass die Bedingung lösbar ist. Durch logische Programmierung kann, anstatt eines konkreten Algorithmus zur Berechnung einer Lösung, eine Reihe von Fakten und eine Beschreibung der gewünschten Lösung definiert werden. Die eigentliche Berechnung geschieht automatisch. Es muss also kein konkreter Lösungsweg angegeben werden, das Finden der Lösung erfolgt durch die Beschreibung des Problems.

Diese zahlreichen Möglichkeiten, eine hohe Abstraktion zu erreichen, machen Curry zu einer guten Wahl für die Entwicklung eines Web-Frameworks mit den in Abschnitt 4.1 genannten Zielen.

Für eine ausführliche Dokumentation sind der Curry Language Report (Hanus u. a. (2006)) und ein Curry-Tutorial (Antoy u. Hanus (2007)) verfügbar.

4.6. Persistierung von Daten mit Curry

Über die logischen Aspekte von Curry kann man Daten als Fakten ausdrücken. Aus diesen Daten können dann mit Hilfe von Bedingungen Informationen gewonnen werden, die gerade von der Anwendung benötigt werden. Mit der Bibliothek `Database` wird es möglich, diese Fakten persistent in Dateien zu speichern und zu laden (Fischer (2005)). Die Definition der zu speichernden Datenstrukturen sowie die Implementierung der Zugriffsoperationen auf diese Datenstrukturen muss jedoch immer noch von Hand geschehen, was ein aufwändiger und fehleranfälliger Prozess ist, vor allem wenn es um komplexe Anwendungen geht. Zur Beschreibung von Datenstrukturen und deren Beziehungen untereinander werden für gewöhnlich Entity-Relationship-Diagramme verwendet. Die in Curry enthaltenen Werkzeuge sowie die ERD-Bibliothek ermöglichen eine abstrakte Beschreibung von Datenstrukturen durch Entity-Relationship-Diagramme oder eine für diesen Zweck geschaffenen Curry-Datenstruktur und die anschließende Generierung der beschriebenen Datenstrukturen sowie deren Standard-Zugriffsoperatoren (siehe Brassel u. a. (2008)). Die so generierten Module können einfach in die Anwendung eingebunden werden und

zur Speicherung aller Anwendungsdaten genutzt werden. Die Anwendung arbeitet dabei nur mit Curry-Datentypen, deren Definition direkt aus dem ER-Diagramm generiert wurde. Die Abfrage von Daten erfolgt über die Angabe eines Ziels im Sinne der Logikprogrammierung. Die Anwendung ist somit vollständig von der darunterliegenden Implementierung zur tatsächlichen Speicherung der Daten entkoppelt. Auf die Funktionsweise der ERD-Bibliothek gehe ich in Abschnitt 5.2.1 genauer ein.

4.7. Erzeugen von webbasierten Benutzerschnittstellen mit Curry

Die Benutzerschnittstelle einer Web-Anwendung besteht aus einer Reihe von HTML-Seiten, welche von der Web-Anwendung dynamisch mit Inhalten gefüllt werden, sowie Formularen für Benutzereingaben. In vielen Programmiersprachen und auch in vielen Web-Frameworks werden diese HTML-Seiten durch die Nutzung von Templates generiert. Ein Template ist dabei in der Regel eine HTML-Seite mit eingebetteten Steuerbefehlen, die entweder in der gleichen Sprache wie die restliche Anwendung oder in einer Template-Sprache geschrieben werden können. Zur Generierung der Webseite werden die Steuerbefehle im Template interpretiert und durch deren Ergebnis ersetzt. Da weder die Struktur des Templates geprüft wird, noch klar ist, was für ein Ergebnis die eingebetteten Steuerbefehle haben werden, ist die Sicherstellung, dass der generierte HTML-Code auch valide ist, sehr schwierig und wird meistens gar nicht erst durchgeführt. Der Programmierer muss somit selbst darauf achten, dass es nicht zu einer Generierung fehlerhaften HTML-Codes kommen kann.

Curry bietet eine Bibliothek `HTML`, welche Datenstrukturen zur Beschreibung von Webseiten und deren Generierung aus diesen Datenstrukturen anbietet. Dynamische Webseiten, das heißt Seiten, die teilweise erst bei deren Aufruf mit Daten gefüllt werden, können mit Hilfe der HTML-Bibliothek erst als Curry Datenstruktur generiert werden und zu deren Ausgabe in HTML-Code umgewandelt werden. Da mit der vorgegebenen Datenstruktur ausschließlich wohlgeformte HTML-Seiten beschrieben werden können, ist es über diesen Weg unmöglich, HTML-Code zu erzeugen, der nicht wohlgeformt ist.

Durch die Einschränkungen des HTT-Protokolles sind die Entgegennahme und das Reagieren auf Benutzereingaben schwieriger zu behandeln als bei traditionellen Desktop-Anwendungen, die auf Schnittstellen des Betriebssystems zugreifen können, welche direkt für Benutzerinteraktion entworfen worden sind. Da das World-Wide-Web aber nicht für die Benutzerinteraktion entworfen wurde, ist es für Web-Anwendungen umständlicher, mit Benutzereingaben umzugehen. Ein Web-Framework sollte diese Komponente für den Entwickler möglichst stark vereinfachen und abstrahieren. Curry enthält für diesen Zweck bereits zwei sehr mächtige Bibliotheken.

Mit der `HTML`-Bibliothek ist es möglich, HTML-Formulare zu generieren und Buttons in diesen Formularen an Handler zu knüpfen, die bei einem Klick durch den Benutzer auf den Button aufgerufen werden (siehe auch Hanus (2001)). Beispielsweise lässt sich ein Formular, welches den Benutzer auffordert etwas in ein Textfeld einzugeben und bei dem nach dem Klick auf einen Button eine Handler-Funktion `clickHandler` aufgerufen wird, welche den eingebenen Text anzeigt, durch folgenden Code mit Hilfe der `HTML`-Bibliothek implementieren:

Listing 4.1: Definition von Formularen zur Benutzerinteraktion in Curry

```
benutzerdialog = return $ Form "Eingabe"
        [htxt "Bitte geben Sie etwas ein:", textfield inputRef "",
         button "Fertig" clickHandler]
  where
    inputRef free

    clickHandler env = return $ Form "Antwort"
      [h1 [htxt ("Ihre Eingabe war: "++(env inputRef))]]
```

Ein großer Vorteil dieser Methode ist, dass das Formular zum Eingeben von Daten und die Funktionen zum Verarbeiten dieser Daten zusammen definiert werden und auch explizit durch die Button-Handler verknüpft werden, obwohl diese beiden Schritte rein technisch durch das HTT-Protokoll getrennt werden.

Trotzdem müssen die einzelnen Formularfelder immernoch von Hand definiert werden und Funktionen geschrieben werden, die die einzelnen Benutzereingaben wieder zu in der Anwendung verwertbaren Datenstrukturen verarbeiten. Meistens bilden die verschiedenen Formularfelder eines Formulares jedoch zusammen einen Datensatz, wobei jeder Teil des Datensatzes eine bestimmte Form haben soll. Beispielsweise besteht ein Formular zur Eingabe eines Datums aus Feldern für Tag, Monat und Jahr. Es darf kein Feld leer sein und sie müssen Zahlen enthalten, wobei die Zahl im Falle des Tages zwischen Eins und Einunddreißig und im Falle des Monats zwischen Eins und Zwölf liegen muss. Die drei Zahlen zusammen müssen ein gültiges Datum bilden. Unter Verwendung der `HTML`-Bibliothek müssen diese Validierungen als Handler-Funktionen implementiert werden und das Formular muss erneut angezeigt werden, falls eine Validierung fehlschlägt (mit den vorher schon eingegeben Werten in den entsprechenden Feldern). Eine große Erleichterung für diese Fälle bietet die `WUI`-Bibliothek von Curry (siehe **?**). Mit dieser ist es möglich, ein Formular als Abbildung einer Datenstruktur der Anwendung zu definieren, die bestimmten Bedingungen genügen müssen. Dann wird garantiert, dass nur ein valider Datensatz zurückgeliefert wird. Solange die Benutzereingaben nicht valide sind, wird der Benutzer um Korrektur gebeten. Ein Datumsformular würde beispielsweise wie folgt aussehen:

Listing 4.2: Formular zur Eingabe eines korrekten Datums

```
wDate = wTriple (wSelect show [1..31])
                (wSelect show [1..12])
```

```
            wInt
        `withCondition` correctDate
```

Die Validerungsfunktion `correctDate` muss dabei nur noch prüfen, ob es sich bei der Eingabe um ein gültiges Datum handelt. Dass die Eingaben Zahlen sind und in den entsprechenden Wertebereichen liegen, wird durch die Definition des Formulares bereits sichergestellt. Funktionen zum Umwandeln der abstrakten Datentypen in WUI-Definitionen sind zwar in den meisten Fällen sehr einfach zu definieren, müssen aber trotzdem von Hand geschrieben werden. Ein Framework sollte dem Entwickler diese Arbeit abnehmen.

5. Struktur des Frameworks

Im folgenden arbeite ich eine logische Struktur für das zu erstellende Framework heraus, benenne die einzelnen Teile des Frameworks und beschreibe deren Funkion. Das entwickelte Framework soll den Namen „Spicey“ tragen.

5.1. Vorteile einer festen Struktur

Die klare und sinnvolle Strukturierung einer Anwendung ist für deren langfristigen Erfolg von großer Bedeutung. Fehlt eine gut ausgearbeitete Struktur, gerät die Anwendung ab einer gewissen Komplexität außer Kontroller und der Aufwand für Wartung und Weiterentwicklung steigt sehr schnell.

Eine fehlende oder nicht ausreichend ausgearbeitete Struktur begünstigt das Entstehen von unnötigen oder falschen Abhängigkeiten zwischen Teilen der Anwendung, also die Vermischung von Aufgabenbereichen. Dies erschwert zum einen das Austauschen aber auch schon das Ändern von Teilbereichen, da eine Änderung oder ein Austausch zwangsläufig auch die davon abhängigen Teile beeinflusst und so entweder Fehler entstehen oder eine nachträgliche und somit wesentlich arbeitsintensivere Entkopplung geschehen muss. Zum anderen behindert eine Vermischung von Aufgabenbereichen auch die nachträgliche Abstraktion von Funktionalität, da Funktionen, die abstrahiert werden könnten, bei einer Vermischung von Aufgaben nicht so leicht identifiziert werden können, weil sie über mehrere Stellen in der Anwendung verteilt sind.

Eine gut gewählte und gut umgesetzte Strukturierung einer Anwendung kann oft auch das Treffen von falschen Designentscheidungen bei der Weiterentwicklung der Anwendung verhindern, da der Entwickler bereits durch die Struktur und den Ort, wo er sich gerade in der Anwendung befindet, besser entscheiden kann, ob eine bestimmte Funktionalität direkt dort implementiert werden sollte oder ob sie in eine andere Komponente gehört. Auch das Implementieren der gleichen Funktionalität an verschiedenen Stellen aufgrund von Unkenntnis über das Vorhandensein einer Funktion wird durch eine höhere Übersichtlichkeit erschwert.

Viele Strukturen von Anwendungen wurden bereits in bekannten Architekturmustern („Design Patterns“) festgehalten. Nutzt eine Anwendung eines dieser Muster zur Strukturierung, ist es für einen Entwickler, welchem das Muster bekannt ist, leichter, sich in der Anwendung zurechtzufinden. Genau so verhält es sich, wenn ein Entwickler, der bereits das verwendete Framework kennt, auf eine für ihn unbekannte aber mit dem

Framework entwickelte Anwendung trifft. Dieser Vorteil kommt besonders bei Anwendungen, die über längere Zeit gepflegt werden und deren Entwicklungsteam wechselt, zum Tragen.

5.2. Aufbau von Spicey

Grundsätzlich kann man eine Anwendung in drei logische Schichten unterteilen: die Präsentationsschicht, die Domänenschicht und die Datenquelle. Die Präsentationsschicht beinhaltet die Benutzerschnittstelle (oder auch Programmschnittstelle, falls der Zugriff nicht durch Menschen sondern durch andere Programme erfolgt). Sie ist für die Darstellung von Daten und die Interpretation von Eingaben und Umwandlung dieser in Aktionen, die auf der Domänenschicht oder Datenquelle ausgeführt werden, verantwortlich. Die Domänenschicht enthält jegliche Logik der Problemdomäne, also die Geschäftslogik. Unter Datenquellen fallen jegliche Teile der Anwendung, welche zur Kommunikation mit Systemen verantwortlich sind, die der Anwendung Daten zur Verfügung stellen. Meistens enthält die Anwendung nur ein solches System: die Datenbank. Es können jedoch auch Schnittstellen zu Web-Services oder ähnlichem sein.

Eine Trennung dieser Schichten sollte aus bereits genannten Gründen immer angestrebt werden und daher sollte ein Framework dem Entwickler auch Hilfestellung bei der Umsetzung dieser Trennung geben.

Eine Möglichkeit für eine solche Trennung ist das *Presentation-Abstraction-Control* Pattern (siehe Buschmann u. a. (2007)). Hier wird die Anwendung in sogenannte Agenten aufgeteilt, welche die verschiedenen Aufgaben der Anwendung erfüllen. Jeder Agent verfügt über drei Schichten für Datenbeschaffung, Verarbeitung und Darstellung, welche speziell für die Aufgaben des Agenten implementiert werden. Die Agenten bilden eine Hierarchie, wobei der Agent auf der obersten Ebene den Zugriff auf die eigentliche Datenbank, welche nicht zum System der Agenten gehört, regelt. Die restlichen Agenten können dann in einer Baumstruktur unter diesem Agenten angeordnet sein, je nach zu erfüllender Aufgabe. Alle Agenten sind klar voneinander getrennt und auch innerhalb der Agenten findet eine klare Trennung der Schichten statt. Der Vorteil dieses Systems ist die hohe Flexibilität. Ein so aufgebautes System kann leicht viele verschiedene Anforderungen verschiedenster Nutzer erfüllen, indem eigene Agenten für die einzelnen Anforderungen implementiert werden. Außerdem können verschiedene Agenten leicht auf verschiedenen physischen Rechnern laufen. Der Nachteil des *Presentation-Abstraction-Control* Patterns ist die hohe Komplexität und viel unnötiger Implementierungsaufwand, sofern die Anwendung nicht besonders viele, sehr unterschiedliche Anforderungen erfüllen muss. Daher eignet sich dieses Pattern für einen Großteil der Anwendungen nicht und wird daher nicht für das Spicey-Framework genutzt.

Eine weitere Möglichkeit stellt das *Model View Controller* Pattern dar. Es ermöglicht eine hinreichende Trennung der logischen Schichten (siehe Abschnitt 4.4) und aufgrund der weitreichenden Akzeptanz und Bekanntheit des *Model View Controller* Patterns,

dessen Einfachheit sowie dessen gute Eignung für Web-Applikationen wird es auch in Spicey konsequent umgesetzt. Das *Model View Controller* Pattern gibt der Anwendung genug Struktur, um von den in Abschnitt 5.1 genannten Vorteilen zu profitieren, ist aber dennoch einfach genug, um die Verwendbarkeit des Frameworks nicht auf bestimmte Anwendungsgebiete einzuschränken (Fowler (2002)). Das heißt eine Anwendung, welche das Spicey-Framework nutzt, wird grundlegend in drei Bereiche aufgeteilt: die Model-Schicht, die Controller-Schicht und die View-Schicht. Dies sollte jedoch keinesfalls mit den drei logischen Schichten Präsentation, Domäne und Datenquelle verwechselt werden. Im MVC-Pattern gehören View und Controller in die Präsentationsschicht und die Modelschicht zur Domäne. Die Datenquelle ist für gewöhnlich an die Modelschicht durch eine weitere Schicht angebunden.

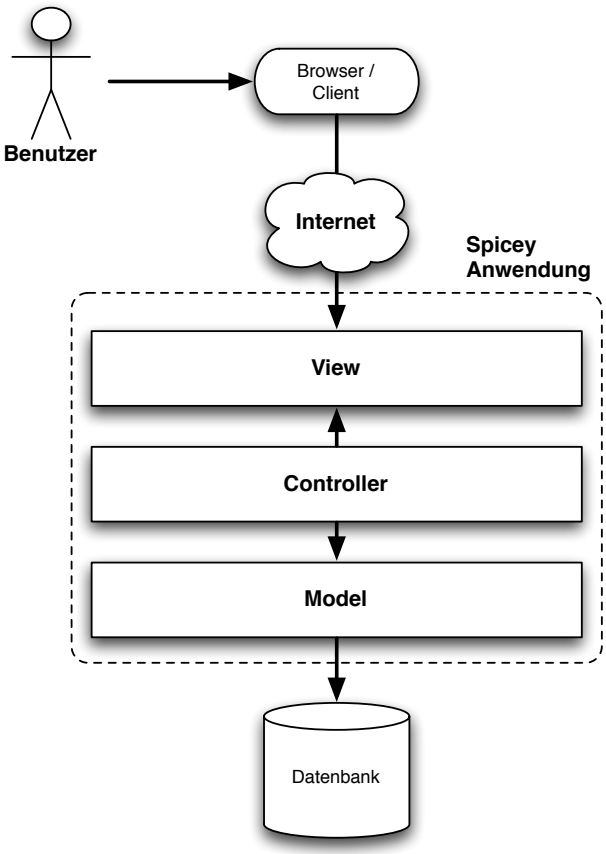

Abbildung 5.1.: Der allgemeine Aufbau einer mit Spicey entwickelten Anwendung

5.2.1. Model-Schicht

Für die Model-Schicht von Spicey werden die schon von Curry mitgelieferte ERD-Bibliothek und der zugehörige Generator genutzt (siehe Brassel u. a. (2008)). So können die Models in einem Entity-Relationship-Diagramm beschrieben werden, wobei die Models vom Generator aus den definierten Entitäten erzeugt werden. Der Code der Models beinhaltet Funktionen zum Persistieren von durch das Model beschriebenen Datensätzen. Das Abfragen von bereits persistierten Datensätzen wird durch vordefinierte Funktionen, die die im ER-Diagramm festgelegten Beziehungen zwischen den Models beschreiben, erleichtert.

Beispielsweise wären die Models für ein einfaches Weblog-System die Einträge in das Blog (`Entry`), Kommentare zu diesen Einträgen (`Comment`) und Tags, welche die Blog-Einträge kategorisieren (`Tag`). Ein Eintrag kann von beliebig vielen Kommentaren kommentiert werden, ein Kommentar bezieht sich immer auf genau einen Eintrag. Ein Eintrag kann beliebig viele Tags besitzen und ein Tag kann beliebig vielen Einträgen zugeordnet werden. All das kann in einem ER-Diagramm beschrieben werden.

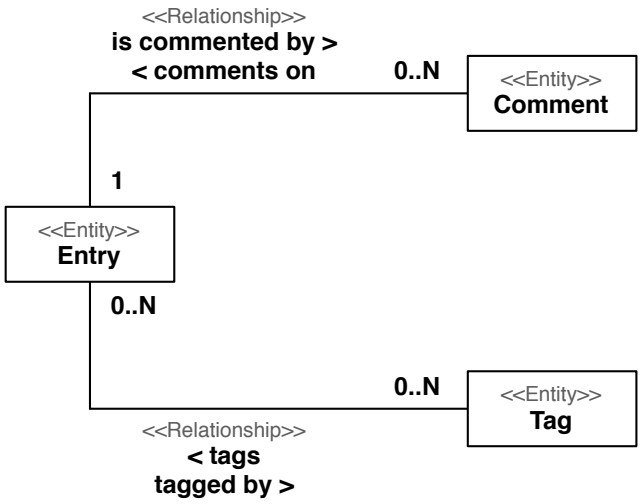

Abbildung 5.2.: ER-Diagramm für ein einfaches Weblog (UML Notation)

In dem obigen Diagramm sind die Attribute der Entitäten nicht zu sehen, sie können jedoch im Diagrammeditor festgelegt werden, was für die Generierung natürlich zwingend notwendig ist. Diese grafische Repräsentation wird zunächst in eine Term-Notation transformiert[1]. Wenn kein grafischer Editor verwendet werden soll, kann auch direkt die

[1] die Transformation findet auf der XML-Repräsentation des Diagramms statt, siehe Brassel u. a. (2008)

Term-Notation zur Definition des ER-Modells verwendet werden. Die Term-Notation des obigen Diagramms sieht wie folgt aus:

Listing 5.1: Term-Notation des ER-Diagrammes

```
(ERD "Blog"
[
 (Entity "Entry"
        [
         (Attribute "Title" (StringDom Nothing) Unique False),
         (Attribute "Text" (StringDom Nothing) NoKey False),
         (Attribute "Author" (StringDom Nothing) NoKey False),
         (Attribute "Date" (DateDom Nothing) NoKey False)
        ]
 ),
 (Entity "Comment"
        [
         (Attribute "Text" (StringDom Nothing) NoKey False),
         (Attribute "Author" (StringDom Nothing) NoKey False),
         (Attribute "Date" (DateDom Nothing) NoKey False)
        ]
 ),
 (Entity "Tag"
  [
         (Attribute "Name" (StringDom Nothing) Unique False)
        ]
 )
]
[
 (Relationship "Commenting"
  [
   REnd "Entry" "commentsOn" (Exactly 1),
   REnd "Comment" "isCommentedBy" (Range 0 Nothing)
  ]
 ),
 (Relationship "Tagging"
  [
   REnd "Entry" "tags" (Range 0 Nothing),
   REnd "Tag" "tagged" (Range 0 Nothing)
  ]
 )
]
)
```

Generiert wird hieraus schließlich ein neues Modul, welches die Models enthält. Darin werden die Datentypen für Einträge, Kommentare und Tags definiert:

```
data Entry
     = Entry Key String String String CalendarTime
```

```
--               Title  Text   Author Date
data Comment
     = Comment Key String String CalendarTime Key
--                 Text   Author Date         EntryKey
data Tag
     = Tag Key String
--             Name
```

Hierbei werden allerdings die Konstruktoren von dem Modul nicht exportiert, was das Erstellen eines Datensatzes ohne Verbindung zur Datenbank unmöglich macht. Die eindeutige Identifizierung des Datensatzes gewährleistet der Key, welcher ebenfalls nicht direkt manipulierbar ist. Hier wird das Entwurfsmuster *Constrained Constructor* genutzt (siehe Antoy u. Hanus (2002)). So wird sichergestellt, dass jeder Datensatz mit den zugehörigen Funktionen erstellt und manipuliert wird:

```
--- Inserts a new Entry entity.
newEntry :: String -> String -> String -> CalendarTime -> Transaction Entry

--- Updates an existing Entry entity.
updateEntry :: Entry -> Transaction ()

--- Deletes an existing Entry entity.
deleteEntry :: Entry -> Transaction ()

--- Gets a Entry entity stored in the database with the given key.
getEntry :: EntryKey -> Transaction Entry
```

Die Funktionen nutzen die `Transaction`-Monade, womit das Konzept einer Datenbank-Transaktion umgesetzt wird. Transaktionen können Datenbankoperationen zusammenfassen, die dann atomar ausgeführt werden, wodurch sichergestellt wird, dass keine Daten während der Ausführung der Transaktion von anderen Zugriffen auf die Datenbank geändert werden. Außerdem werden alle Operation einer Transaktion rückgängig gemacht, falls eine der Operationen fehlschlägt, wodurch die Datenbank immer in einem konsistenten Zustand bleibt.

Für alle im ER-Diagramm angegebenen Beziehungen werden Prädikate erstellt, die diese Beziehungen repräsentieren und mit deren Hilfe Abfragen formuliert werden. Die folgende Funktion nutzt die drei Prädikate `entry`, `tag` und `tagging`, um eine Abfrage für die Liste von Tags, welche dem übergebenen Eintrag zugeordnet sind, anzufertigen:

```
getTaggingTags :: Entry -> Query ([Tag])
getTaggingTags eTag = queryAll (\t -> let
```

```
ekey free
tkey free
 in
   ((entry ekey eTag)
   <> ((tag tkey t)
   <> (tagging ekey tkey))))
```

Über die Prädikate wird festgelegt, welche Daten abgefragt werden sollen. Das Prädikat `entry` stellt sicher, dass es sich bei `eTag` um einen Datensatz vom Typ `Entry` handelt, wobei der Schlüssel `ekey` des Entry-Datensatzes als freie Variable deklariert wurde und somit nicht festgelegt ist. Das Prädikat `tag` garantiert das gleiche für Datensätze vom Typ `Tag` und das Prädikat `tagging` definiert eine Verknüpfung zwischen den beiden Datensätzen, welche über ihre Schlüssel spezifiziert werden. Zusammen bedeutet die Abfrage also „hole alle `t`, wobei `t` ein Tag ist, eine Verknüpfung zwischen dem Schlüssel von `t` und einem Schlüssel `ekey` besteht und der Datensatz, der zum Schlüssel `ekey` gehört, ein Entry ist“. Diese Abfrage kann dann eigenständig oder als Teil einer Transaktion ausgeführt werden, um die Liste der Tags zu bekommen.

Die durch die Prädikate definierte Abfrage wird in einer `Query` Datenstruktur zusammengefasst. Diese Abfrage kann dann in einer Transaktion, auch in Kombination mit weiteren Abfragen, ausgeführt werden.

Das generierte Grundgerüst der Models kann vom Entwickler um noch benötigte Funktionen erweitert werden. Beispielsweise können die grundlegenden Prädikate zu komplexeren kombiniert werden, um oft benutzte Abfragen zu realisieren oder komplexe Beziehungen zu modellieren. Weiterhin können speziellere Funktionen zum Erstellen und Manipulieren der Daten hinzugefügt werden. Oft besteht eine Anwendung jedoch hauptsächlich aus dem Anlegen und Manipulieren von Datensätzen. In einem solchen Fall enthalten die generierten Models bereits die gesamte benötigte Funktionalität.

5.2.2. Controller-Schicht

Die Controller-Schicht hat in der Web-Anwendung die Aufgabe, Eingaben zu interpretieren und entsprechende Aktionen in der Model-Schicht auszulösen und dann die Daten für eine Antwort auf die Eingaben zu sammeln. Anschließend wird die Kontrolle an die View-Schicht abgegeben, welche sich um die Darstellung der Daten kümmert. Prinzipiell gibt es zwei Arten, eine Controller-Schicht aufzubauen, entweder als *Page Controller* oder als *Front Controller* (siehe Fowler (2002)).

Beim *Page Controller* gibt es für jede Webseite der Anwendung einen Controller. Manchmal ist es sogar sinnvoll, für verschiedene Aktionen auf einer Seite jeweils einen Controller zu haben, zum Beispiel wenn auf einer Seite mehrere Buttons zu klicken sind. Der Vorteil dieser Aufteilung ist die klare Zuordnung zwischen Webseiten der Anwendung und dahinterliegendem Code. Außerdem erhält man eine geringere Komplexität

der Controller-Schicht als bei einem *Front Controller*. Dies ist gerade bei Anwendungen von Vorteil, welche nur eine sehr dünne Controller-Schicht benötigen, die hauptsächlich die Anfragen nur weiterleitet. Durch die Verbindung zwischen aufgerufener Seite und zuständigem Controller bestimmt der Webserver, welcher Controller angesprochen wird.

Unter Verwendung des *Front Controller* Patterns gibt es nur einen Controller, welcher alle Anfragen annimmt und je nach Anfrage entscheidet, welcher Teil der Anwendung zuständig ist. Dadurch ist der Entwickler nicht so stark vom verwendeten Webserver abhängig. Der große Vorteil ist hier, dass Funktionalität, die bei jeder Anfrage, unabhängig von dessen Art, angewendet werden muss, direkt an den einen Controller hängen kann. So wird Duplizierung von Code vermieden. Natürlich kann auch beim *Page Controller* Funktionalität ausgelagert und überall verwendet werden, jedoch müssen zumindest die Aufrufe müssen in jeden Controller eingebaut werden, was zu Fehlern und Sicherheitslücken führen kann, wenn dies in einem Controller vergessen wird. Der Nachteil des einen Controllers für alle Anfragen ist das Verlorengehen der direkten Beziehung zwischen Controller und Webseite, welche eine gute Hilfe zur Strukturierung der Anwendung darstellt.

Die größere Komplexität des *Front Controller* Patterns ist bei der Entwicklung mit einem Framework nicht von Bedeutung, da das meiste vom Framework erledigt wird und dem Entwickler dadurch diese Komplexität wieder abgenommen wird. Das *Front Controller* Pattern hat aber den entscheidenden Vorteil, der Anwendung mehr Kontrolle beim Verarbeiten der Anfragen zu geben, was es für die Verwendung in einem Framework sehr attraktiv macht. Den Nachteil der klareren Strukturierungsmöglichkeit beim *Page Controller* kann dadurch ausgeglichen werden, indem unter den *Front Controller* standardmäßig für jede Seite oder jeden Anwendungsbereich ein Controller definiert wird, der dann von dem einen Controller aufgerufen wird. So erhält man die Flexibilität eines *Front Controller* und die leicht zu erfassende Struktur eines *Page Controller*.

5.2.3. View-Schicht

Die View-Schicht dient der Erzeugung einer Benutzerschnittstelle zur Anwendung, durch die sowohl Daten dargestellt als auch Eingaben durch den Benutzer vorgenommen werden können.

Auch für die View-Schicht gibt es zwei grundsätzliche Möglichkeiten der Realisierung. Die erste Möglichkeit ist der Einsatz von sogenannten Templates, was der Umsetzung des *Template View* Patterns entspricht (siehe Fowler (2002)). Das Template ist eine HTML-Seite, in die Steuerbefehle eingebettet werden, die vor der Anzeige durch dynamischen Inhalt (den Daten des Controllers) ersetzt werden. Dieses Vorgehen hat den Vorteil, dass das Verfahren, wie Seiten entstehen, leicht zu verstehen ist und dass die Templates mit gängigen grafischen HTML-Editoren erstellt werden können. So können auch Webdesigner ohne Kenntnisse über die Web-Applikation das Benutzerinterface erstellen. Allerdings

sollte die Gestaltung durch den Webdesigner auch allein über Stylesheets möglich sein, die bestimmen, wie der von der Anwendung generierte HTML-Code dargestellt werden soll, solange der generierte HTML-Code semantisch sinnvoll ist.

Die Nachteile dieser Methode sind jedoch zum einen, dass nicht sichergestellt wird, dass durch das Ersetzen der Steuerbefehle in den Templates ein Ergebnis erzeugt wird, welches noch einer validen Struktur entspricht. Zum anderen ist es schwer, die Steuerbefehle auf reine Markierungen zu beschränken. Es wird Fälle geben, bei denen ein Teil der Seite nur unter einer bestimmten Bedingung angezeigt werden soll. Hierfür muss es ein Steuerzeichen geben, welches als Konditionalausdruck benutzt werden kann. Dabei ist die Gefahr sehr groß, dass Logik in das Template eingebettet wird. Die Lösung dafür wäre, die Bedingung in den eigentlichen Programmcode auszulagern und ein leeres Ergebnis zu liefern, wenn die Bedingung zur Anzeige nicht zutrifft. Dies funktioniert jedoch nicht mehr, wenn Elemente der Seite basierend auf einer Bedingung verschiedenes Aussehen bekommen sollen. Wird hier die Bedingung in den Programmcode ausgelagert, zieht das auch die Auslagerung eines Teils des Aussehens des Benutzerinterface in den Programmcode nach sich. Ein ähnliches Problem ergibt sich mit ebenfalls oft benötigten Iteratoren, um zum Beispiel eine Liste von Datensätzen anzuzeigen. Templates erschweren also die saubere Trennung von Logik und Präsentation und sind darüberhinaus fehleranfällig.

Die Alternative zum *Template View* ist das *Transform View* Pattern (Fowler (2002)). Beim *Transform View* werden die vom Controller zum Anzeigen gelieferten Daten durchgegangen und für jedes Element eine Transformation in die passende Darstellung durchgeführt. Ein sehr bekanntes und verbreitetes Beispiel für ein Verfahren einer solchen Transformation ist die Umwandlung von XML mittels XSLT (siehe Ray (2003)). Bei einer Transformation ist die Trennung zwischen Präsentation und Programmlogik wesentlich leichter und es ist ebenfalls einfacher sicherzustellen, dass die erzeugten HTML-Seiten eine valide Struktur haben. Außerdem eignen sich funktionale Sprachen sehr gut für Transformationen (XSLT ist ebenfalls eine funktionale Sprache). Daher wird das *Transform View* Pattern auch in der HTML-Bibliothek von Curry umgesetzt und kommt somit auch in Spicey zum Einsatz. Der Nachteil dieses Verfahrens ist allerdings, dass Designer ohne Programmierkenntnisse weniger Einfluss auf die Benutzerinterfaces haben, als es beim *Template View* der Fall wäre. Dies ergibt sich daraus, dass der HTML-Code komplett generiert wird und ein Designer die Darstellung nur durch Stylesheets bestimmen kann, was jedoch, die Generierung von semantisch sinnvollem HTML-Code vorausgesetzt, kein Problem darstellen sollte.

Eine weitere Variationsmöglichkeit ist die Entscheidung zwischen einem einzigen oder einem zweiteiligen Umwandlungsprozess. Oft ändern sich von Seite zu Seite nur Teile der angezeigten Seite und es existiert eine Art Rahmen, der immer gleich bleibt (beispielsweise ein Logo und ein Navigationsmenü). Dies führt bei einem einzigen Transformationsschritt zu dupliziertem Code, da bei jeder Seite der Rahmen generiert werden muss. Eine Lösung hierfür ist, die Erzeugung der HTML-Seite in zwei Teile aufzuteilen. Zum einen die Transformation der Daten in den Inhalt der Seite und zum anderen das Hinzufügen

des Rahmens, was an einer zentralen Stelle geschehen kann und somit Duplizierung von Code vermeidet.

Die HTML-Bibliothek von Curry setzt das *Transform View* Pattern durch Bereitstellung einer Datenstruktur um, mit der HTML-Seiten beschrieben werden können. Eine HTML-Seite wird dabei über den Konstruktor `HtmlPage` für einfache Seiten oder `HtmlForm` für Seiten mit Elementen zur Eingabe von Benutzerdaten definiert. Diese beinhalten eine Liste von HTML-Ausdrücken `HtmlExp`, welche ebenfalls wieder HTML-Ausdrücke beinhalten können, um so die Baum-Struktur von HTML-Seiten abzubilden. Da es hiermit wirklich nur möglich ist, eine Baumstruktur zu konstruieren, kann nicht wohlgeformtes HTML gar nicht erst entstehen.

6. Implementierung

6.1. Allgemeine Implementierung der Schichten

Wie in den vorangegangenen Kapiteln bereits dargelegt wurde, ist eine klare Strukturierung der Anwendung eine entscheidende Erleichterung in der Entwicklung. Das Framework unterstützt diese Strukturierung auf der obersten Ebene durch Einteilung in Schichten.

Durch die Verwendung des *Model View Controller* Patterns ergeben sich drei zu implentierende Schichten. Die Models werden durch die ERD-Werkzeuge von Curry generiert. Dabei entsteht ein Modul mit allen Datenstrukturen und Funktionen zum Zugriff auf diese. Geschäftslogik kann hier vom Entwickler hinzugefügt werden.

Die Controller-Schicht wird durch Spicey generiert. Die Generierung erfolgt wie bei den Models auch auf Basis des Entity-Relationship-Diagrams, es müssen also keine weiteren Definitionen durch den Entwickler getroffen werden, als die initiale Erstellung des Entity-Relationship-Diagrammes. In den generierten Controllern sind alle Funktionen vorhanden, um die grundlegenden Operationen Anzeigen, Anlegen, Editieren und Löschen von allen im Entity-Relationship-Diagramm definierten Daten zu realisieren.

Die View-Schicht wird zusammen mit der Controller-Schicht ebenfalls von Spicey generiert. Alle Seiten und Formulare, welche für die oben genannten Operationen benötigt werden, sind in den generierten Views enthalten.

Um die Unterscheidung der einzelnen Schichten für den Entwickler zu erleichtern, spiegelt sich die Schichten-Aufteilung auch in der generierten Verzeichnisstruktur wider. So werden die Models in das Verzeichnis `models`, die Controller entsprechend im Verzeichnis `controller` und die Views in `views` abgelegt.

Die generierte Anwendung unterstützt das Anzeigen, Anlegen, Editieren und Löschen aller Models. Neben dem direkten Bearbeiten von Datensätzen werden hierbei auch immer mögliche, im Entity-Relationship-Diagramm angegebene, Verknüpfungen beachtet. Für die Benutzeroberfläche ist hier zwischen Zu-Eins-Beziehungen und Zu-N-Beziehungen aus Sicht des bearbeiteten Models zu unterscheiden. Bei Zu-Eins-Beziehungen muss genau ein fremdes Model für die Verknüpfung ausgewählt werden, bei Zu-N-Beziehungen können mehrere fremde Models ausgewählt werden.

In den generierten Controllern werden dem View Daten zu allen möglichen fremden Models für jede Verknüpfung zur Verfügung gestellt und nach einer Bearbeitung die

eigentliche Verknüpfung durchgeführt. Auch hier wird zwischen Zu-Eins-Beziehungen, bei denen genau ein Model verknüpft werden muss, und Zu-N-Beziehungen, wo der Controller das Hinzufügen und Entfernen von beliebig vielen Verknüpfungen unterstützen muss, unterschieden.

6.2. Generierung einer Projektstruktur

Der erste Schritt bei der Erstellung einer Web-Anwendung unter Verwendung des Spicey-Frameworks ist die Generierung der Projektstruktur. Zur Projektstruktur zählen alle Verzeichnisse und Dateien, welche später von der Anwendung benötigt werden. Dazu dient das Shell-Skript `spiceup.sh`, welches in dem Verzeichnis aufgerufen werden muss, in dem das Projekt erstellt werden soll. Das Shell-Skript ermittelt das Verzeichnis, aus dem es aufgerufen wurde sowie das Verzeichnis, in dem es selbst liegt und ruft mit diesen Parametern ein kleines Curry-Programm, den Projektgenerator, auf. Der Projektgenerator enthält eine Beschreibung der Projektstruktur, nach der er das Projekt generiert:

Listing 6.1: Beschreibung der Projektstruktur

```
structure =
  Directory "." [
    Directory "scripts" [
      ResourceFile "deploy.sh",
      ResourceFile "run.sh",
      ResourceFile "compile.sh" ],
    Directory "system" [
      ResourceFile "main.curry",
      ResourceFile "Spicey.curry",
      ResourceFile "Routes.curry",
      ResourceFile "Session.curry",
      ResourceFile "Processes.curry" ],
    Directory "views" [
      ResourceFile "SpiceySystemView.curry",
      GeneratedFromERD (createViewsForTerm),
      GeneratedFromERD (createHtmlHelpersForTerm) ],
    Directory "controllers" [
      ResourceFile "SpiceySystemController.curry",
      GeneratedFromERD (createControllersForTerm) ],
    Directory "models" [
      GeneratedFromERD (createModelsForTerm),
      ResourceFile "ERD.curry",
      ResourceFile "ERDGeneric.curry" ],
    Directory "config" [
      GeneratedFromERD (createRoutesForTerm) ],
    Directory "public" [
      Directory "css" [
        ResourceFile "style.css"
```

```
      ],
      Directory "images" [
        ResourceFile "spicey-logo.png",
        ResourceFile "text.png",
        ResourceFile "time.png",
        ResourceFile "number.png",
        ResourceFile "foreign.png"
      ]
    ]
  ]
```

Die Beschreibung ist eine Baumstruktur und enthält drei Arten von Komponenten: `Directory`, `ResourceFile` und `GeneratedFromERD`. `Directory` modelliert ein Verzeichnis, welches vom Generator erstellt wird, sofern es noch nicht existiert. `ResourceFile` ist eine Datei, welche unverändert aus dem Resourcen-Verzeichnis des Generators an die entsprechende Stelle im Projekt kopiert wird. Dies sind vom Framework benötigte Bibliotheksfunktionen sowie einige Shell-Skripte zur Erleichterung häufig auftretender Aufgaben. Die letzte Komponente schließlich ist `GeneratedFromERD`, welche eine oder mehrere aus dem Entity-Relationship-Diagramm erstellte Dateien beschreibt. Es wird die angegebene Generator-Funktion vom Projekt-Generator aufgerufen, welche dann die Dateien erzeugt. Auf diese Generierung gehe ich in den folgenden Abschnitten ein.

6.3. Scaffolding

Als Scaffolding wird die Möglichkeit bezeichnet, aus einer abstrakten Beschreibung, hier aus dem Entity-Relationship-Diagramm, automatisch ein Gerüst für eine komplette Anwendung generieren zu können. Die Entwicklung eines Programmes, welches selbst wieder ein Programm erzeugt, wird Metaprogrammierung genannt.

Zur besseren Erläuterung der Generierung verwende ich das vorher schon angesprochene Beispiel einer einfachen Weblog-Anwendung.

6.3.1. Metaprogrammierung mit der AbstractCurry-Bibliothek

In Curry wird Metaprogrammierung durch die AbstractCurry-Bibliothek ermöglicht. Diese definiert einen Datentyp für die deklarative Beschreibung eines beliebigen Curry-Programmes. Das Programm generiert eine deklarative Beschreibung, welche dann von einer Funktion der AbstractCurry-Bibliothek in Programmcode transformiert werden kann. Durch die Beschreibung des Programmes über den definierten Datentyp wird die Wahrscheinlichkeit, ungültige Programme zu generieren, im Vergleich zu der direkten Erzeugung von Programmcode drastisch reduziert. Außerdem entspricht die Generierung

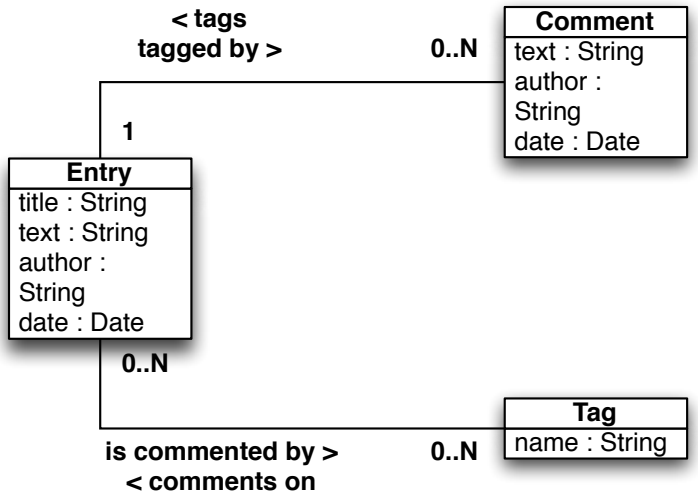

Abbildung 6.1.: ER-Diagramm-Beispiel für das Scaffolding

eines Programmes der Transformation einer Datenstruktur in eine andere, was in einer funktionalen Sprache einfacher zu implementieren ist.

6.3.2. Generierung der Grundoperationen

Die Grundoperationen auf Daten, welche in den meisten Applikationen benötigt werden, sind das Anlegen von neuen, das Editieren von bestehenden, das Löschen von und das Anzeigen angelegter Datensätze. Um diese Funktionalität automatisch aus dem Entity-Relationship-Diagramm generieren zu können, müssen Komponenten in allen drei Schichten erstellt werden.

Eine Generierung der Models wird bereits von der ERD-Bibliothek abgedeckt. Diese enthalten die für die Grundoperationen benötigten Funktionen. Hierauf wurde in Abschnitt 5.2.1 bereits eingegangen.

Generierung der Views

Für das Benutzerinterface müssen pro Model zwei grundlegende Views erzeugt werden. Ein View zum reinen Anzeigen der Daten und ein View mit einem Formular zum Anlegen und Editieren der Daten. Hierzu wird pro Model ein Modul im `views`-Verzeichnis des Projektes erstellt. Die Module sind nach den Namen der Models benannt, an die noch ein „`View`“ angehängt wird. In den Views werden die für die Grundoperationen benötigten

HTML-Seiten unter Zuhilfenahme der Definitionen in der HTML- und WUI-Bibliothek beschrieben.

Unter der Annahme, dass E dem Namen des Models entspricht, werden folgende View-Funktionen erzeugt:

wE legt die Darstellung des Formulars fest. Hier werden die Typen der Formularfelder spezifiziert (für die automatische Validierung durch die WUI-Bibliothek) und es wird festgelegt, dass jedes Formularfeld ein Label bekommen soll.

```
wEntry :: [Tag] -> WuiSpec ((String,String,String,CalendarTime,[Tag]))
wEntry tagList =
  withRendering
    (w5Tuple
      wString wString wString wDateType
      (wMultiCheckSelect (\tag -> [(htxt (tagName tag))]) tagList)
    )
    (renderLabels entryLabelList)
```

Da ein `Entry` beliebig viele zugeordnete `Tags` haben kann, werden neben den Daten des `Entry` auch immer die Liste zugeordneten `Tags` zusammen mit den `Entry`-Daten im Tupel sowie zusätzlich eine Liste mit `Tags`, die zugeordnet werden können, übergeben.

Allgemein werden bei n:m-Beziehungen alle verknüpften Models im Datentupel als Liste mitgeführt. Bei Beziehungen, die in eine Richtung genau ein Model beinhalten (1:n, n:1 und 1:1), wird anstatt einer Liste nur ein Model im Tupel mitgeführt.

tuple2E aktualisiert ein Model mit den Daten aus einem abgeschickten Formular.

```
tuple2Entry ::
  Entry -> (String,String,String,CalendarTime,[Tag]) -> (Entry,[Tag])
tuple2Entry entryToUpdate (title,text,author,date,tags) =
  (setEntryTitle
    (setEntryText
      (setEntryAuthor
        (setEntryDate entryToUpdate date)
      author)
    text)
  title, tags)
```

E**2Tuple** erzeugt ein Tupel von Daten zur Anzeige in einem Formular aus einem Model und gegebenenfalls mit diesem Model verknüpften Models.

```
entry2Tuple :: (Entry,[Tag]) -> (String,String,String,CalendarTime,[Tag])
entry2Tuple (entry,tags) =
  (entryTitle entry,
   entryText entry,
   entryAuthor entry,
   entryDate entry,
   tags)
```

tuple2*E* und ***E*2Tuple** werden als Konvertierungsfunktionen der Daten von der Model-Repräsentation in die Formular-Darstellung des Views und umgekehrt benötigt. So kann die Controller-Schicht immer mit der Model-Repräsentation gearbeitet werden. Eine Ausnahme bildet die Erstellung von neuen Datensätzen, da hier nach Eingabe der Daten noch keine Model-Repräsentation existiert, wird das Tupel mit den Daten aus dem Formular durch den Controller in einem neuen Datensatz gespeichert.

w*E*Type ist ein Adapter, welcher unter Verwendung der Funktionen tuple2*E* und *E*2Tuple sowie w*E* ein Model in ein Formular umwandelt, mit dem es möglich ist, die Daten des Models direkt zu ändern.

```
wEntryType :: Entry -> [Tag] -> [Tag] -> WuiSpec ((Entry,[Tag]))
wEntryType entry tags tagList =
  transformWSpec (tuple2Entry entry,entry2Tuple) (wEntry tagList)
```

Die WUI-Bibliothek unterstützt auch eine einfachere Definition des Formulares zum Editieren eines Datensatzes, bei der man die Umwandlungsfunktionen nicht explizit in beide Richtungen angeben muss. Dies ist wegen der Struktur der Models als *Opaque Type* (siehe Antoy u. Hanus (2002)) jedoch nicht möglich. Außerdem unterstützt eine explizite Angabe die Entkopplung zwischen Model und View und gibt dem Entwickler mehr Anpassungsmöglichkeiten. Weiterhin ist es so möglich, das Model nicht über Preisgabe von systeminternen Daten, wie einer ID, an das Formular zu knüpfen.

Darüber hinaus sind die eigentlichen Views enthalten:

create*E*View beschreibt das Formular zum Anlegen eines neuen Datensatzes, wobei der Funktion für jedes Feld ein Vorgabewert übergeben werden kann.

```
createEntryView ::
  String -> String -> String -> CalendarTime -> [Tag] -> [Tag] ->
  ((String,String,String,CalendarTime,[Tag]) ->
  IO ([HtmlExp])) -> [HtmlExp]
createEntryView
  defaultTitle defaultText defaultAuthor defaultDate
  defaultTags possibleTags controller =
  let
```

```
    (hexp,handler) =
      wui2html (wEntry possibleTags)
      (defaultTitle,defaultText,defaultAuthor,defaultDate,defaultTags)
      (nextControllerForData controller)
  in
    [
      (h1 [(htxt "new Entry")]),
      hexp,
      (wuiHandler2button "create" handler)
    ]
```

blank*E*View verwendet create*E*View zur Definition eines Formulars mit leeren Feldern, was der Standardfall beim Anzeigen eines Formulars zum Neuanlegen eines Datensatzes ist und daher von dieser Funktion abgedeckt wird.

```
blankEntryView ::
  [Tag] -> ((String,String,String,CalendarTime,[Tag]) ->
  IO ([HtmlExp])) -> [HtmlExp]
blankEntryView possibleTags controller =
  createEntryView " " " " " " " (CalendarTime 1 1 2000 0 0 0 0) []
    possibleTags controller
```

edit*E*View beschreibt schließlich das Formular zum Editieren eines bestehenden Datensatzes, wobei das Model und eventuell verknüpfte Models übergeben werden. Das übergebene Model wird durch das Formular direkt editiert.

```
editEntryView :: (Entry,[Tag]) -> [Tag] -> ((Entry,[Tag]) ->
  IO ([HtmlExp])) -> [HtmlExp]
editEntryView (entry,tags) possibleTags controller =
  let
    (hexp,handler) =
      wui2html (wEntryType entry tags possibleTags)
        (entry,tags) (nextControllerForData controller)
  in
    [
      (h1 [(htxt "edit Entry")]),
      hexp,
      (wuiHandler2button "change" handler)
    ]
```

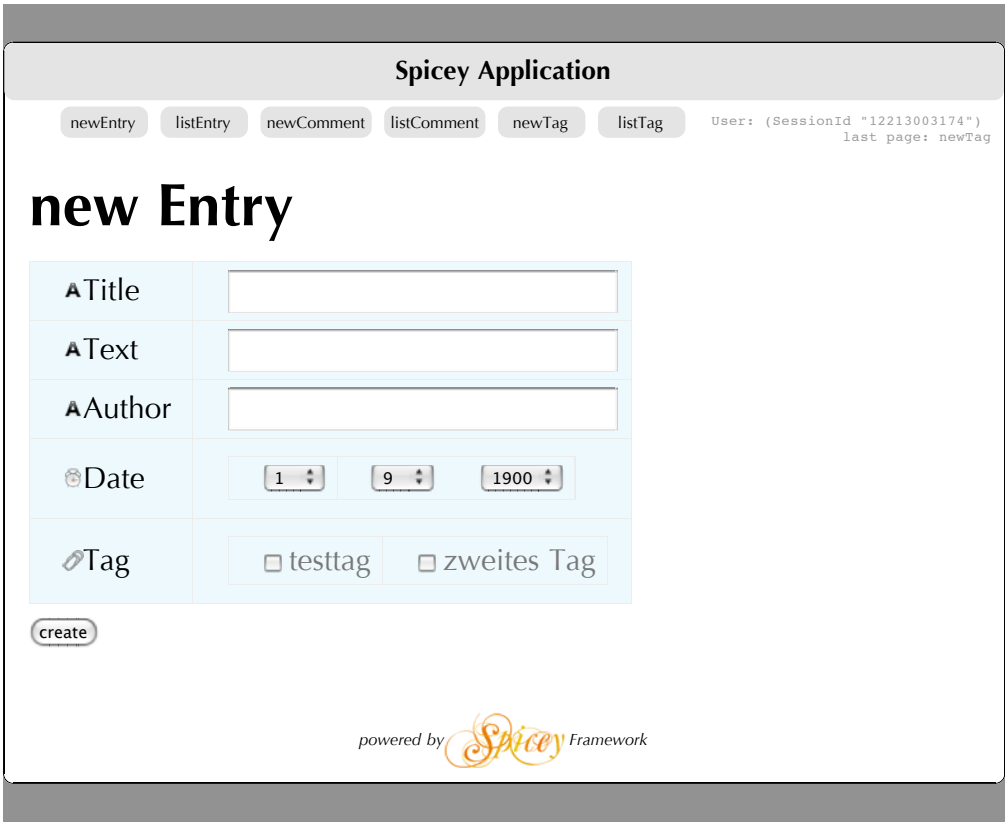

Abbildung 6.2.: Formular zum Anlegen eines neuen Eintrags

list*E*View zeigt eine Liste von Models in einer Tabelle an. Zu jedem Datensatz wird ein Button zum Editieren und ein Button zum Löschen des Datensatzes angezeigt.

```
listEntryView ::
  [Entry] -> (Entry -> IO ([HtmlExp])) ->
  (Entry -> IO ([HtmlExp])) -> [HtmlExp]
listEntryView
  entrys editEntryController deleteEntryController =
  [(h1 [(htxt "EntryList")]),
   (table ([(take 4 entryLabelList)] ++
    (listEntrys entrys))
   )
  ]
where
  listEntrys :: [Entry] -> [[[HtmlExp]]]
  listEntrys [] = []
  listEntrys (entry:entryList) =
```

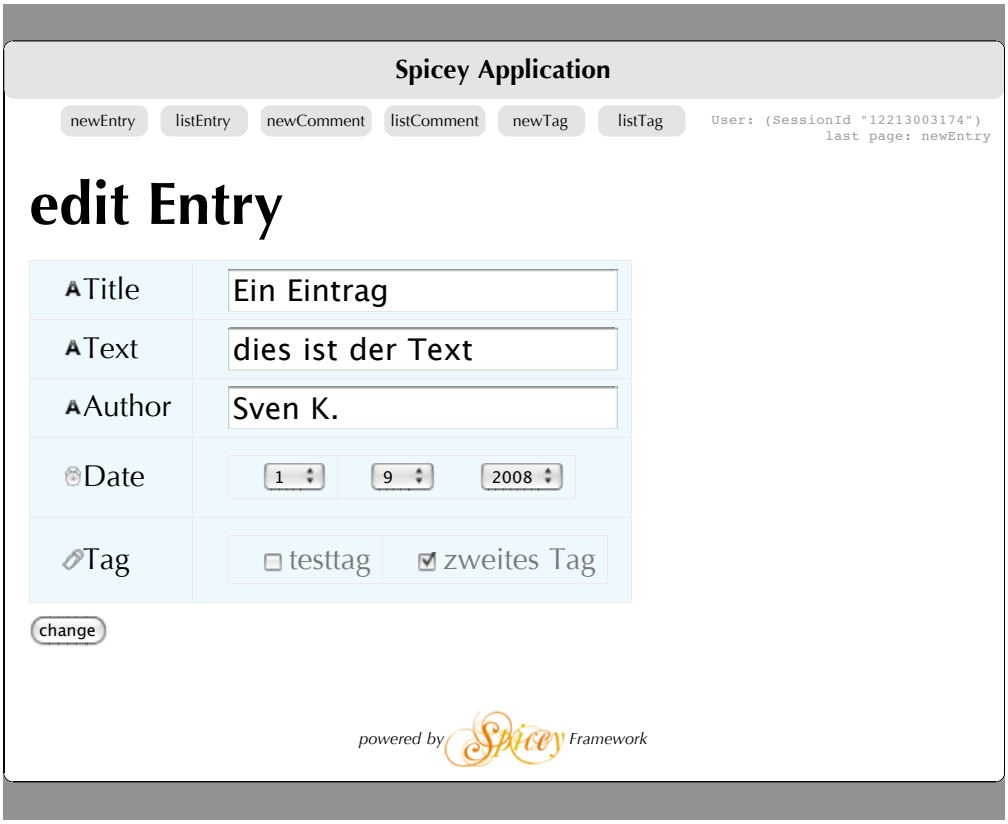

Abbildung 6.3.: Formular zum Editieren eines neuen Eintrags

```
(
  ((entryToListView entry) ++
  [[(button "edit" (nextControllerWithoutRefs
    (editEntryController entry))),
  (button "delete" (nextControllerWithoutRefs
    (deleteEntryController entry)))]]
) : (listEntrys entryList))
```

Des Weiteren wird noch ein Modul mit Hilfskomponenten für alle Views erstellt, auf die die einzelnen Views zurückgreifen können. Dieses Modul trägt den Namen des Entity-Relationship-Schemas mit angehängtem „`EntitiesToHtml`“. Darin werden für alle Models E jeweils drei Darstellungsformen für deren Daten generiert:

EtoListView erzeugt eine Zeilendarstellung der Daten aus dem Model, die zur Anzeige der Datensätzen im list$E$View verwendet wird.

```
entryToListView :: Entry -> [[HtmlExp]]
```

Abbildung 6.4.: Auflistung aller Einträge

```
entryToListView entry = [
  [(stringToHtml (entryTitle entry))],
  [(stringToHtml (entryText entry))],
  [(stringToHtml (entryAuthor entry))],
  [(calendarTimeToHtml (entryDate entry))]
]
```

***E*toShortView** erzeugt eine Kurzdarstellung des Models, die bei der Anzeige der Verknüpften Models verwendet wird. Diese Darstellung beinhaltet lediglich den Wert des ersten eindeutigen Feldes aus dem Entity-Relationship-Diagramm.

```
entryToShortView :: Entry -> String
entryToShortView entry = entryTitle entry
```

***E*toDetailsView** erzeugt eine Tabelle mit allen Daten des Models und dessen Feldnamen.

```
entryToDetailsView :: Entry -> [HtmlExp]
entryToDetailsView entry = [
  (table (map (\(label,value) -> [label,value])
```

```
    (zip entryLabelList (entryToListView entry))
  ))
]
```

*E***LabelList** ist eine Liste der Bezeichnungen für alle Datenfelder des Models, wie sie im Entity-Relationship-Model angegeben wurden. Diese Liste wird sowohl in den Formularen des entsprechenden Models als auch in den Tabellenauflistungen als Feldbezeichner verwendet. Die Bezeichnungen sind bereits mit CSS-Klassen angereichert, auf die ich später in Abschnitt 6.3.3 genauer eingehe. Bei einer konsequenten Nutzung dieser Liste von Bezeichnungen in allen Teilen der Anwendung lassen sich die Bezeichnungen jederzeit leicht in dieser Funktion zentral durch den Entwickler ändern.

```
entryLabelList :: [[HtmlExp]]
entryLabelList = [
  [(textstyle "label label_for_type_string" "Title")],
  [(textstyle "label label_for_type_string" "Text")],
  [(textstyle "label label_for_type_string" "Author")],
  [(textstyle "label label_for_type_calendarTime" "Date")],
  [(textstyle "label label_for_type_relation" "Tag")]
]
```

Diese Funktionen wurden in ein eigenes Modul ausgelagert, da eventuell nicht nur das Modul des zugehörigen Models darauf zugreifen muss, sondern auch Module von Models, die das Model als Verknüpfung haben. Weiterhin hat der Entwickler hier ein zentrales Modul, in dem er die Darstellung aller Models sowie deren Feldbezeichner anpassen kann.

Generierung der Controller

Zur Steuerung des Verhaltens der Views wird für jedes Model ein Controller-Modul generiert. Die Module sind nach dem Modelnamen mit angehängtem „`Controller`“ benannt und werden in das Verzeichnis `controllers` des Projektes generiert. Für jede der Grundoperationen gibt es Controller-Funktionen im zugehörigen Modul: für das Anzeigen und Löschen von Models jeweils eine Funktion, für das Anlegen und Editieren eines Models jeweils zwei Funktionen.

Zwei Controller-Funktionen zum Anlegen und Editieren sind notwendig, da das Anlegen und Editieren ein zweistufiger Prozess ist. Das Formular, welches für das Anlegen oder Editieren eines Models angezeigt wird, muss vorher vom Controller mit Daten gefüllt werden. Beim Anlegen müssen nur die Daten der Models, die möglicherweise mit dem neu angelegten Model verknüpft werden können, durch den Controller an den View übergeben werden. Beim Editieren müssen zusätzlich noch die Daten des zu bearbeitenden Models geladen und übergeben werden.

```
editEntryController :: Entry -> IO ([HtmlExp])
editEntryController entryToEdit = do
  allTags <- getAllTags
  taggingTags <- getTaggingTags entryToEdit
  return (editEntryView
    (entryToEdit,taggingTags)
    allTags
    updateEntryController)
```

Wurden nun Daten durch den Benutzer im Formular eingegeben oder verändert und wird das Formular dann abgeschickt, wird die zweite Controller-Funktion aufgerufen. Diese speichert das neu angelegte Model beziehungsweise die Änderungen an dem editierten Model und aktualisiert die Verknüpfungen des Models anhand der vom Benutzer angegebenen Daten. Eine Validierung der Benutzereingaben ist hier nicht mehr notwendig, da die WUI-Bibliothek sicherstellt, dass nur gültige Werte vom Benutzer eingegeben werden können. Die ERD-Bibliothek stellt weiterhin beim Speichern des Models sicher, dass keine Konsistenzbedingungen des Entity-Relationship-Model verletzt werden. Sollte festgestellt werden, dass durch das Speichern eine Konsistenzbedingung verletzt würde, so werden die Änderungen beziehungsweise der neue Datensatz nicht gespeichert und stattdessen eine Fehlermeldung ausgegeben. Ist die Speicherung erfolgreich, wird die Controller-Funktion zum Anzeigen der Liste aller Datensätze des entsprechenden Models aufgerufen.

```
updateEntryController :: (Entry,[Tag]) -> IO ([HtmlExp])
updateEntryController (entry,tagsTagging) = do
  oldTaggingTags <- getTaggingTags entry
  transResult <- runT ((updateEntry entry) |>>
    ((removeTagging oldTaggingTags entry) |>>
    (addTagging tagsTagging entry)))
  either
    (\_ -> listEntryController [])
    (\error -> displayError (showTError error) [])
    transResult
```

Die Funktion zum Anzeigen aller Datensätze lädt alle entsprechenden Models aus der Datenbank und übergibt sie dem list*E*View. Außerdem werden noch die Controller-Funktionen zum Editieren und Löschen eines Models übergeben, welche mit den entsprechenden Buttons im View verknüpft werden. Die zu verknüfenden Funktionen werden also nicht direkt im View angegeben, da die Festlegung der aufzurufenden Funktionen und der damit folgenden Views zum Verhalten der Benutzeroberfläche zählt und somit in den Aufgabenbereich des Controllers fällt.

```
listEntryController :: [String] -> IO ([HtmlExp])
listEntryController _ = do
  entrys <- getDynamicSolutions (\e ->
    let
      key free
    in (entry key e)
  )
  return
    (listEntryView entrys
      editEntryController deleteEntryController)
```

Schließlich existiert noch eine Controller-Funktion, welche ein Model aus der Datenbank löscht. Diese hat keinen eigenen View und kann direkt mit einem Button im View verknüpft werden. Nach erfolgreicher Löschung ruft diese Controller-Funktion die Funktion zum Anzeigen aller noch verbleibenden Datensätze auf. Sollte es beim Löschen zu einem Fehler kommen, wird dieser Fehler angezeigt.

```
deleteEntryController :: Entry -> IO ([HtmlExp])
deleteEntryController entry = do
  oldTaggingTags <- getTaggingTags entry
  transResult <- runT (
    (removeTagging oldTaggingTags entry) |>>
    (deleteEntry entry))
  either
    (\_ -> listEntryController [])
    (\error -> displayError (showTError error) [])
    transResult
```

Je nach Verknüpfungen zu anderen Models hat das jeweilige Controllermodul noch einige Hilfsfunktionen zum Abfragen der verknüpften und aller Models sowie zum Aktualisieren der Verknüpfungen. Wenn R der Name der im Entity-Relationship-Model angegebenen Verknüpfung ist, sind dies **getAll**E**s** sowie bei einer Multiplizität von Eins in Bezug auf das Verknüpfte Model (z.B. ein Kommentar bezieht sich auf genau einen Blog-Eintrag) **get**RE, die das verknüpfte Model liefert. Bei einer Multiplizität größer Eins (einem Blog-Eintrag sind beliebig viele Tags zugeordnet) analog dann **get**RE**s**, welche eine Liste der verknüpften Models zurückgibt. Außerdem gibt es bei n:m-Beziehungen die Funktionen **remove**R und **add**R, um beliebig viele Verknüpfungen auf einmal (durch Entfernen aller Verknüpfungen und anschließendes Hinzufügen der neu gewählten Verknüpfungen in einer Transaktion) zu aktualisieren.

6.3.3. Nutzung der Informationen aus dem Entity-Relationship-Modell in allen Teilen der Anwendung

Wie bereits erwähnt, wird die gesamte Anwendung aus dem durch den Entwickler vorgegebenen Entity-Relationship-Modell generiert. Da Curry eine statisch stark typisierte Sprache ist, werden neben den Entity-Namen, den Namen deren Attribute und den Verknüpfungen der Entitäten untereinander auch die Typen der Attribute in die generierte Anwendung übertragen. Diese ziehen sich durch den gesamten Curry-Code. Damit die Typinformationen auch beim Übergang in die Präsentationsschicht und somit in eine HTML-Darstellung erhalten bleiben, werden die Typinformationen in Form von CSS-Klassen in die View-Definitionen generiert.

```
commentLabelList :: [[HtmlExp]]
commentLabelList = [
  [(textstyle "label label_for_type_string" "Text")],
  [(textstyle "label label_for_type_string" "Author")],
  [(textstyle "label label_for_type_calendarTime" "Date")],
  [(textstyle "label label_for_type_relation" "Entry")]
]
```

Neben der allgemeinen Klasse `label`, welche die Ausgabe als Feldbezeichnung markiert wird auch jeweils eine typspezifische Klasse hinzugefügt. Diese kann genutzt werden, um dem Benutzer einen visuellen Hinweis darauf zu geben, welcher Typ bei einer Eingabe erwartet wird und verbessert so die Benutzerfreundlichkeit der Formulare und Datentabellen.

Wie man in Abbildung 6.5 sieht, wird im Formular zum Eingeben eines Blog-Kommentars der Typ der einzelnen Felder durch entsprechende Icons vor den Feldnamen verdeutlicht.

6.3.4. Generierte Formulare und Anpassungsmöglichkeiten

Der generierte Code für die View-Schicht ist, wie der restliche generierte Code auch, schon stark auf Modularisierung und Wiederverwendbarkeit ausgelegt. Er erfüllt damit nicht nur den Zweck, den Anforderungen der sehr einfachen generierten Anwendung zum Anlegen, Anzeigen, Editieren und Löschen von Datensätzen zu genügen, sondern kann darüber hinaus auch für die weitere Entwicklung genutzt werden. Die Funktionen im Modul *R*EntitiesToHtml sind Transformationen aller Models in eine HTML-Repräsentation, die immer dann genutzt werden kann, wenn ein Model in der Benutzeroberfläche dargestellt werden muss.

Die verschiedenen Varianten der Darstellungen erlauben eine Nutzung egal ob die Model-Daten im Vordergrund stehen (*E*ToDetailsView) oder nur knapp angezeigt werden sollen

Abbildung 6.5.: Screenshot: Formular zum Editieren eines Blog-Kommentars

(*E*ToShortView). Bei einer konsequenten Nutzung dieser Darstellungen in der gesamten Anwendung kann weiterhin durch die Anpassung einer Transformation an einer zentralen Stelle die Anzeige in der gesamten Benutzeroberfläche abgeändert werden. Außerdem bleibt die Darstellung der Daten in der gesamten Anwendung konsistent, was die Wiedererkennung von Elementen der Benutzeroberfläche fördert und so die Bedienung erleichtert.

Wird eine noch feinere Kontrolle über die Darstellung gewünscht, ist es ebenfalls noch möglich, die Darstellung jedes einzelnen Datums zu ändern. Auch dies ist an einer zentralen Stelle in der Anwendung möglich. Die Formulare nutzen die Transformationsfunktionen `stringToHtml`, `intToHtml` und `calendarTimeToHtml` für die primitiven Typen der Models. Diese Funktionen sind im Modul `Spicey` definiert. Durch deren Anpassung ist eine Änderung der Darstellung auch dieser Daten in der gesamten Anwendung möglich.

6.4. Layout

Durch Nutzung einer Kombination des *Page Controller* und *Front Controller* Patterns (siehe Abschnitt 5.2.2) ist es möglich, die durch beliebige Controller-Aufrufe generierten

Webseiten um weitere Elemente anzureichern. So gibt es für gewöhnlich einige Elemente einer Website, die sich selten von einer zur anderen Seite ändern, beispielsweise ein Kopfbereich mit Logo oder ein Navigationsmenü. Für diese Elemente ist es sinnvoll, an zentraler Stelle zum eigentlichen Inhalt der Seite hinzugefügt zu werden. Im Spicey-Framework wird hierfür im Modul `Spicey` die Funktion `addLayout` aufgerufen, welche eine Liste von HTML-Ausdrücken (Datentyp `HtmlExp` aus der HTML-Bibliothek) als Parameter nimmt (die vom Controller generierte Seite) und eine Liste von HTML-Ausdrücken zurückgibt (die beliebig angereicherte Seite). Die Standard-Funktion, welche nach der Generierung der Anwendung definiert ist, fügt eine Überschrift und einen Seitenfuß mit dem Spicey-Logo sowie das aus den Routes generierte Navigationsmenü (siehe Abschnitt 6.5.3) zu jeder Seite hinzu:

```
addLayout :: [HtmlExp] -> [HtmlExp]
addLayout viewblock =
  [blockstyle "header" [h1 [htxt "Spicey Application"]],
   routeMenu] ++
   viewblock ++
  [blockstyle "footer"
    [par [htxt "powered by",
          image "images/spicey-logo.png" "Spicey",
          htxt "Framework"]]]
```

6.5. Routes

Durch die Nutzung eines zentralen Eintrittspunktes von Anfragen in die Anwendung, wie es das *Front Controller* Pattern vorschlägt, erhält die Anwendung die volle Kontrolle über die Weiterleitung der Anfragen und kann somit bestimmen, welcher Teil der Anwendung welche Anfrage bearbeitet. Zur Unterscheidung der Anfragen sind die im URL enthaltenen Parameter (beziehungsweise der Pfad in der URL, falls ein Umschreiben der URL durch den Webserver stattfindet) von entscheidender Bedeutung. Diese Parameter können auch als Parameter eines HTTP-GET-Requests bezeichnet werden. Bei eines HTTP-POST-Request, wie er beim Übermitteln von Formulardaten in der Regel genutzt wird, stehen die POST-Daten im Vordergrund. Hier wurde aber beim vorherigen Aufruf des Formulars bereits entschieden, welcher Teil der Anwendung verantwortlich ist.

Die Zuordnung der Verantwortlichkeiten auf Basis des URL übernehmen in Spicey die sogenannten Routes[1].

[1] Die Bezeichnung „Routes“ stammt aus dem Ruby on Rails Framework und wird dort für die gleiche Funktionalität verwendet, weshalb dieser Begriff einfach übernommen wurde.

6.5.1. Datentyp zur Festlegung der Aufrufweiterleitung

Zur Definition der Routes wurde ein eigener Datentyp definiert:

```
routes :: [(String, UrlMatch, ControllerFunctionReference)]
```

Routes sind demnach eine Liste von Tripeln. Der erste Wert ist eine textuelle Beschreibung der Route und gibt dem Entwickler einerseits eine Möglichkeit, den Zweck der Route zu dokumentieren, andererseits wird der dort angegebene Text für die Links im automatisch erstellten Navigationsmenü verwendet. Der zweite Teil ist eine Definition des URL und der dritte Teil ist eine Referenz auf den verantwortlichen Controller. Der Typ `UrlMatch` ist wie folgt definiert:

```
data UrlMatch
     = Exact String
     | Matcher (String -> Bool)
     | Always
```

Es gibt also drei Möglichkeiten einen für den Aufruf des angegbenen Controllers passenden URL zu spezifizieren. Entweder man gibt den Wert des enthaltenen GET-Parameters direkt an (`Exact`), dann wird der zugehörige Controller aufgerufen, wenn der Parameter dem angegebenen Wert entspricht. Es kann auch eine Funktion angegeben werden, die den Parameter auf beliebige Eigenschaften untersucht und `true` zurückgibt, wenn der Controller aufgerufen werden soll, `false` sonst (`Matcher`). Oder der angegebene Controller soll unabhängig von den Parametern immer aufgerufen werden, in diesem Fall kann `Always` angegeben werden.

Zu beachten ist hier noch, dass nur der erste Parameter des URL berücksichtigt wird. Der URL wird an den durch `&` markierten Stellen in Parameter zerlegt[2] und der erste dieser Parameter wird zur Ermittlung der zuständigen Controller-Funktion verwendet. Die restlichen Parameter werden an die Controller-Funktion übergeben.

Wie die letzte Möglichkeit `Always` nahelegt, ist die Reihenfolge der Paare in der Liste ebenfalls von Bedeutung. Die Liste wird von vorne nach hinten durchgegangen und es wird die Anfrage an den ersten Controller weitergegeben, für den die angegebene Bedingung zutrifft. Dahinter in der Liste stehende Bedingungen werden gar nicht erst untersucht, es wird also immer höchstens ein Controller aufgerufen. In Anbetracht dessen ist es sinnvoll, am Ende der Liste einen `Always`-Eintrag zu haben, womit dann festgelegt werden kann, welcher Controller zuständig ist, wenn der im URL enthaltene Parameter keinem der angegebenen Kriterien entspricht. Falls die Liste komplett durchlaufen wird, ohne einen passenden Controller zu finden, wird ein Fehler ausgegeben.

[2] siehe das RFC zu Uniform Resource Locators, Berners-Lee u. a. (1994)

Für die durch das Scaffolding generierte Funktionalität werden auch entsprechende Routes generiert. Um den initialen Funktionsumfang der Anwendung nach der Generierung zu nutzen, muss der Entwickler also keine Anpassungen an den Routes vornehmen. Die generierten Routes zu dem Weblog-Beispiel sehen wie folgt aus:

```
routes :: [(String,UrlMatch,ControllerFunctionReference)]
routes = [
  ("new Entry",Exact "newEntry",NewEntryController),
  ("list Entry",Exact "listEntry",ListEntryController),
  ("new Comment",Exact "newComment",NewCommentController),
  ("list Comment",Exact "listComment",ListCommentController),
  ("new Tag",Exact "newTag",NewTagController),
  ("list Tag",Exact "listTag",ListTagController),
  ("default",Always,ListEntryController)
]
```

Die Seiten zum Anzeigen aller Models und dem Anlegen neuer Datensätze sind hier direkt erreichbar. Da zum Editieren und Löschen immer ein Datensatz angegeben werden muss, können diese Funktionen nicht direkt über den URL, sondern nur aus dem List-View aufgerufen werden. Falls kein oder ein unbekannter Parameter im URL angegeben wird, wird eine Liste von Datensätzen für das erste im Entity-Relationship-Diagramm angegebene Model angezeigt.

Statt in den Routes direkt die passende Controller-Funktion anzugeben, wie es durch die Behandlung von Funktionen als „Bürger erster Klasse“ in Curry durchaus möglich wäre, werden hier Referenzen auf die Controller-Funktionen angegeben, die durch die Funktion `getController` in die eigentliche Controller-Funktion aufgelöst werden. Die Controller-Referenzen werden zusammen mit den Routes im Modul `RoutesData` definiert und sehen für das Weblog-Beispiel wie folgt aus:

```
data ControllerFunctionReference
     = NewEntryController
     | ListEntryController
     | NewCommentController
     | ListCommentController
     | NewTagController
     | ListTagController
```

Der Vorteil der Nutzung einer Datenstruktur zur Definition der Referenzen gegenüber der Nutzung von einfachen Strings ist, dass bereits der Compiler überprüfen kann, ob alle in der Anwendung verwendeten Referenzen gültig sind. So können beispielsweise Tippfehler bereits zur Compilezeit erkannt werden, welche bei der Nutzung von Strings erst zur Laufzeit der Anwendung zu einem Fehler führen würden.

Die Nutzung von Controller-Referenzen an Stelle der direkten Controller-Funktionen hat den Grund, dass auch in den Views auf die Controller verwiesen werden können muss, um Links zu anderen Teilen der Anwendung generieren zu können. Hierfür existiert die Funktion `linkToController`, die aus einer Controller-Referenz einen URL erzeugt, welcher zum Aufrufen des entsprechenden Controllers genutzt werden kann. Die Funktion kann natürlich nur URLs für Controller-Funktionen erzeugen, für die eine Route mit Spezifikation der URL per `Exact` definiert ist. Würde nun nicht der Umweg über eine Controller-Referenz gewählt werden, sondern die Controller-Funktion direkt angegeben, so ergäbe sich eine Abhängigkeit der View-Module von den Controller-Modulen (über das Routes-Modul) und da die Controller-Module auch von den View-Modulen abhängig sind (da sie diese aufrufen), entstünde ein Abhängigkeitskreis zwischen View- und Controller-Modulen. Solche zyklischen Abhängigkeiten werden vom Curry-Compiler jedoch nicht unterstützt. Durch die indirekte Angabe der Controller-Funktion über eine Controller-Referenz wird dieser Zykel aufgelöst, und die Abhängigkeiten sind, wie in Abbildung 6.6 dargestellt, ohne Zykel.

6.5.2. Ablauf einer Anfrage

Zur Betrachtung der Verarbeitung einer Anfrage in Spicey muss zuerst zwischen GET- und POST-Anfragen unterschieden werden. Während es sich bei GET-Anfragen meist um einfache Seitenaufrufe handelt, resultieren POST-Anfragen aus dem Abschicken eines Formulares. Die Verarbeitung von POST-Anfragen wird daher direkt von den Mechanismen der HTTP-Bibliothek von Curry übernommen, welche von Spicey genutzt wird. GET-Anfragen hingegen werden über die Routes aufgelöst.

Jede Anfrage wird zuerst von der `dispatcher`-Funktion im Modul `main` entgegen genommen. Dies entspricht dem *Front Controller* Pattern. Um zu entscheiden, welche Controller-Funktion für die Verarbeitung verantwortlich ist, wird der URL untersucht. Der erste GET-Parameter wird für die Verarbeitung durch die Routes benutzt, die restlichen werden an die ermittelte Controller-Funktion übergeben. Nun wird mit Hilfe der Routes eine Controller-Referenz ermittelt und diese in die entsprechende Controller-Funktion aufgelöst, welche dann aufgerufen wird (siehe Abbildung 6.7).

Das Ergebnis des Controller-Aufrufes (der wiederum einen View aufruft) ist ein Teil einer HTML-Seite, welche im Dispatcher zu einer vollständigen Seite ergänzt wird. Bei dieser Ergänzung können auch Teile der Seite, die immer gleich sein sollen, hinzugefügt werden.

6.5.3. Automatische Erzeugung eines Navigationsmenüs

Da in den Routes alle direkt über den URL zugreifbaren Seiten der Anwendung definiert werden, kann aus dieser Beschreibung automatisch ein Navigationsmenü erstellt

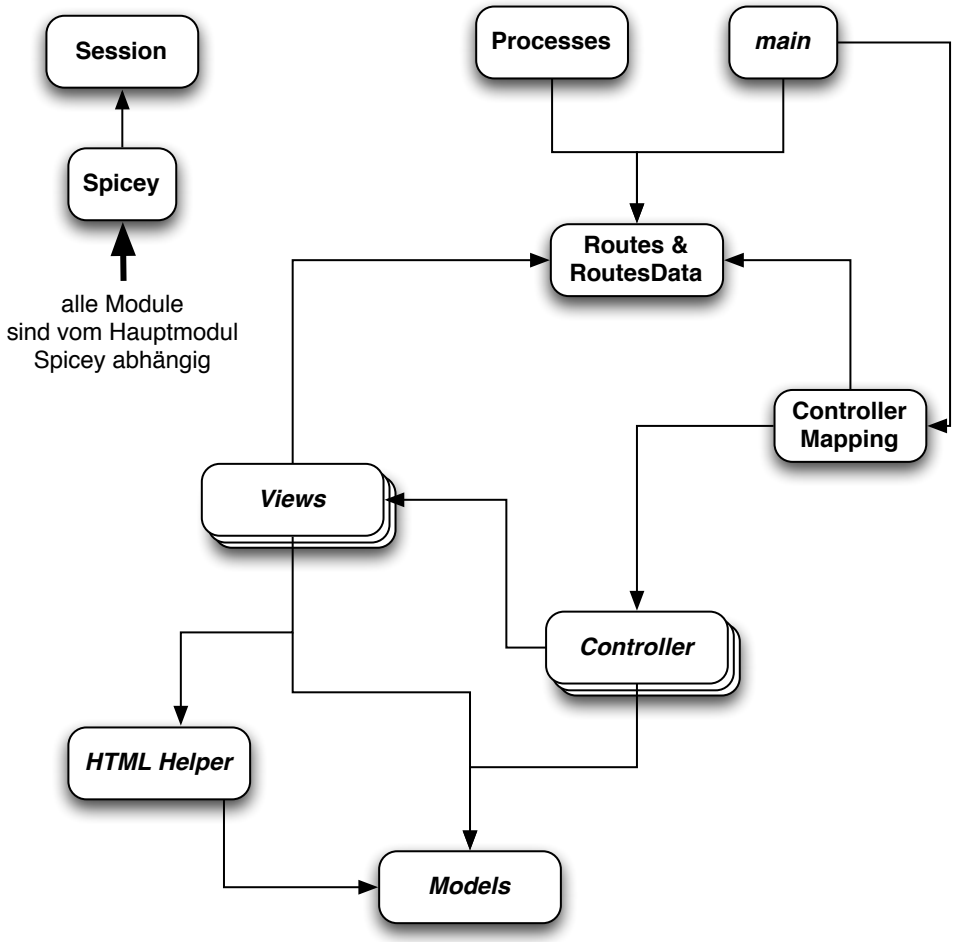

Abbildung 6.6.: Abhängigkeiten der Spicey Module

werden. Nach der Erzeugung des Projektes durch den Projektgenerator wird ein solches Navigationsmenü in jede Seite der Anwendung eingefügt, um sämtliche generierte Funktionalität zugreifbar zu machen. Das Navigationsmenü wird zur Laufzeit erzeugt und basiert ausschließlich auf den Definitionen der Routes. Somit werden alle Änderungen, die der Entwickler an den Routes vornimmt, direkt ins Navigationsmenü übernommen und neu erstellte Seiten sind sofort verfügbar. Dies verbessert die Übersichtlichkeit und die Geschwindigkeit, mit der neue Funktionen während der Entwicklung in die Anwendung eingebaut werden können.

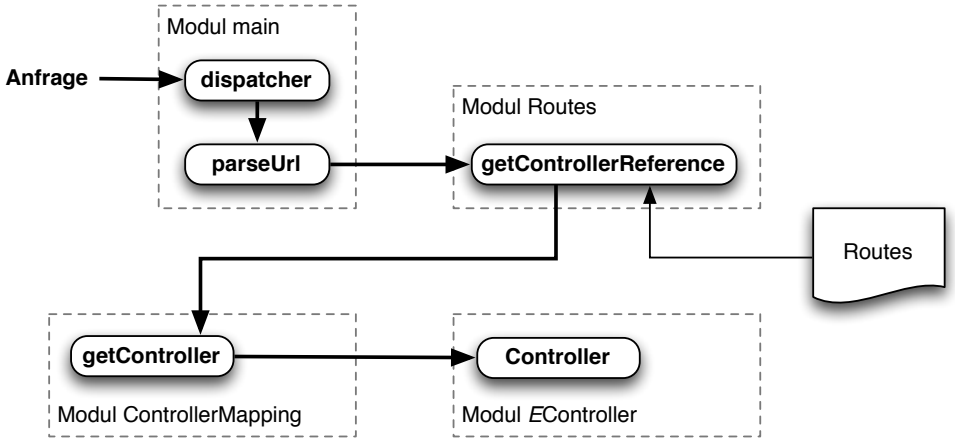

Abbildung 6.7.: Verarbeitung einer Anfrage

Abbildung 6.8.: Screenshot: automatisch erstelltes Navigationsmenü

6.6. Sessions

Eine häufige Anforderung in Web-Applikationen ist die Speicherung von bestimmten Daten über den Zeitraum einer HTTP-Anfrage hinaus. Die Anwendung soll sich trotz der Zustandslosigkeit des HTT-Protokolles verhalten, als bestünde eine ständige Verbindung zwischen der Anwendung und deren Benutzer. Der Zeitraum, über den der Benutzer mit der Anwendung interagiert, wird im Allgemeinen als Session bezeichnet. Eine Session definiert sich über die Verwendete Anwendung und den Benutzer der Anwendung. Jeder Benutzer einer Anwendung hat also eine eigene Session. Die Schwierigkeit besteht hier darin, den Benutzer über mehrere Anfragen hinweg eindeutig zu identifizieren.

6.6.1. Identifikation des Benutzers

Zur Identifikation von Benutzern im Web gibt es inzwischen eine Reihe von Ansätzen. Da der Benutzer von sich aus über keine von der Web-Anwendung abfragbaren Eigenschaften verfügt, anhand derer er sich eindeutig identifizieren ließe, muss die Anwendung jedem Benutzer ein eindeutiges Identifikationsmerkmal zuweisen. Hierfür sind zwei Umsetzungen möglich:

Die erste Möglichkeit ist die Einbettung des Identifikationsmerkmals in jeden Link und jedes Formular der Anwendung. Dadurch würde dieses bei jeder erneuten Anfrage durch den Benutzer als GET- beziehungsweise POST-Parameter mitgesendet und die Anwendung könnte es wieder auslesen. Problematisch ist dieses Verfahren, wenn der Benutzer über direktes Editieren des URL im Browser oder über die Verwendung von in einer früheren Session gespeicherten Lesezeichen navigiert. Hierbei kann das Identifikationsmerkmal verloren gehen. Weiterhin befindet sich das Identifikationsmerkmal direkt im URL der Seite, weshalb es durch Weitergabe des Links an andere Personen übernommen werden kann und der Benutzer so unbewusst leicht ein Sicherheitsrisiko eingeht.

Die zweite Möglichkeit ist die Nutzung von Cookies. Cookies sind kleine Datenpakete, welche von der Webseite an den Client-Browser geschickt werden und dort für eine von der Webseite festlegbare Zeitspanne gespeichert werden. Gespeicherte Cookies können von der Webseite wieder ausgelesen werden. Um zu verhindern, dass Webseiten auf beliebige Cookies von anderen Webseiten zugreifen können, kann zu jedem Cookie von der Webseite, die diesen an den Client-Browser schickt, eine Domain und ein Pfad angegeben werden. Dies verhindert das Auslesen dieses Cookies durch Webseiten unter anderen Domains beziehungsweise Pfaden. Durch das Setzen eines Cookies mit einem eindeutigen Identifikationsmerkmal ist es der Web-Applikation möglich, bei weiteren Anfragen auf den gesetzten Cookie zur Identifikation des Benutzers zuzugreifen. Wenn die Web-Applikation auch bei so identifizierten Benutzern den Cookie in jeder Anfrage (mit immer demselben Identifikationsmerkmal) neu setzt, so kann die festlegbare Lebensspanne des Cookies gleichzeitig als Zeitspanne genutzt werden, nach deren Ablauf ohne neue Anfragen des Benutzers die Session ungültig wird.

6.6.2. Speicherung der Session-Daten im Framework

Um die Details der Benutzeridentifizierung sollte sich der Entwickler bei der Verwendung des Frameworks nicht kümmern müssen. Der Entwickler möchte lediglich Daten in der Session des aktuellen Benutzers speichern können. Diese Funktionalität stellt im Spicey-Framework das Modul `Session` zur Verfügung.

Im Framework wird eine Benutzeridentifizierung durch Verwendung von Cookies durchgeführt. Bei jeder Benutzeranfrage wird überprüft, ob der Benutzer einen Cookie mit

einer Session-ID besitzt. Ist dies der Fall, so wird diese in der Anwendung zur Identifikation des Benutzers verwendet. Falls kein Cookie vorhanden ist, wird dem Benutzer eine neue Session-ID zugewiesen. Die Session-ID des Benutzers kann jederzeit durch die Funktion `getSessionId` ermittelt werden, sie wird in der Anwendung durch den Datentyp `SessionId` repräsentiert:

```
data SessionId = SessionId String
```

Der Konstruktor dieses Datentyps wird gemäß dem *Opaque Type* Pattern (siehe Antoy u. Hanus (2002)) jedoch nicht exportiert, wodurch ein Zugriff auf die interne Repräsentation nicht möglich ist. In der derzeitigen einfachen Implementierung ist die interne Repräsentation ein einfacher String, welcher eine eindeutige Zahl enthält. Für spätere aufwendigere Implementierungen zur Benutzeridentifikation kann der Datentyp jedoch so ohne Beeinflussung der restlichen Anwendung erweitert werden.

Session-Daten werden in einem globalen Datentyp gespeichert, welcher von der Curry-Bibliothek `Global` zur Verfügung gestellt wird und auf den Funktionen über eine IO-Aktion zugreifen können. Um einen globalen Datensatz anzulegen, muss er mit Hilfe der Funktion `global` definiert werden:

```
someString :: Global String
someString = global "initialer Wert" (Persistent "someString")
```

`global` erwartet die initialen Daten, welche gespeichert werden sollen, sowie die Angabe, ob es sich um Daten handeln soll, die auch über einen Neustart der Anwendung bestehen bleiben (`Persistent`) oder nur während der Laufzeit verfügbar sind (`Temporary`). Die Definition der zu speichernden Daten muss eindeutig sein, es dürfen hier also keine Typvariablen verwendet werden. Weiterhin muss bei persistenter Speicherung noch ein String zur Identifizierung angegeben werden (dieser wird in der jetzigen Implementierung als Dateiname zur Speicherung der Daten verwendet).

Da die zu speichernde Datenstruktur fest definiert werden muss, kann dem Entwickler keine konkrete Datenstruktur für alle Session-Daten vorgegeben werden. Dies entspräche auch nicht der starken Typisierung von Curry. Der Entwickler muss also selbst festlegen, welche Daten er in einer Session speichern möchte und eine entsprechende Datenstruktur hierfür definieren.

Einige Eigenschaften sind jedoch bei Session-Daten immer gleich und können vom Framework vorgegeben werden. Da eine Session an einen Benutzer gebunden ist und die Anwendung höchstwahrscheinlich von mehreren Benutzern verwendet wird, wird für jeden Benutzer eine eigene Version des Datums in der Session benötigt. Es wird sich bei Session-Daten also immer um eine Liste handeln, wobei jedes Listenelement die Session-ID des Benutzers enthält, zu dem das Datum gehört. Da eine Session nicht explizit beendet wird, muss außerdem das Alter des Datums gespeichert werden, um ein automatisches Löschen

der Datensätze, welche zu einer bereits abgelaufenen Session gehören, zu ermöglichen. Ein Session-Daten-Eintrag besteht also immer aus einem Tripel: die Session-ID, die Zeit der Speicherung der Daten und die Daten selbst. Somit ergibt sich folgende Struktur, die dem Entwickler vorgegeben werden kann:

```
type SessionStore a = [(SessionId, Int, a)]
```

Der Entwickler muss nun also lediglich noch die zu speichernden Daten durch Setzen der Typvariable definieren. Um beispielsweise den zuletzt besuchten URL des Benutzers zu speichern (was ein einfacher String ist), wird folgende Deklaration benötigt:

```
lastUrl :: Global (SessionStore String)
lastUrl = global [] (Persistent "lastUrl")
```

Zum Laden und Speichern von Daten in die aktuelle Session existieren die Funktionen `getDataFromCurrentSession` und `putDataIntoCurrentSession`:

```
getDataFromCurrentSession :: Global (SessionStore a) -> IO (Maybe a)
putDataIntoCurrentSession :: a -> Global (SessionStore a) -> IO ()
```

Da diese Funktionen den jeweiligen `SessionStore` als Argument erwarten, sollten sie nicht direkt in der Anwendung verwendet werden. Die `SessionStore`s sollten im Modul `Session` gekapselt bleiben und nicht exportiert werden. Daher bietet es sich an, für jeden `SessionStore` eigene Zugriffsfunktionen zu schreiben, welche dann in der Anwendung genutzt werden können. Im Beispiel der zuletzt besuchten URL könnten diese wie folgt aussehen:

```
getLastUrl :: IO String
getLastUrl = do
  maybeUrl <- getDataFromCurrentSession lastUrl
  case maybeUrl of
    (Just url) -> return url
    Nothing -> return "Unknown"

saveLastUrl :: String -> IO ()
saveLastUrl url = putDataIntoCurrentSession url lastUrl
```

Eine solche Kapselung hat den Vorteil, dass die interne Datenrepräsentation geändern werden kann, ohne die Anwendung zu beeinflussen. Außerdem kann auch der Fall, dass ein Datensatz nicht gefunden wird, für jedes Datum einzeln behandelt werden. Dadurch können sinnvolle Entscheidungen über die in diesem Fall angebrachteste Verhaltensweise einmal an zentraler Stelle getroffen werden.

Eine Löschung von Daten aus abgelaufenen Sessions geschieht bei jedem Schreibzugriff auf den `SessionStore`, da hierbei wegen einer möglichen Ersetzung die Daten ohnehin durchgegangen werden müssen und eine Änderung der Daten stattfindet. Dies hat jedoch zur Folge, dass Session-Daten unter Umständen noch über den Ablauf der Session bestehen bleiben. Wenn der Client-Browser den Session-Cookie trotz der angegebenen Lebenszeit nicht löscht und der Benutzer dann auf die Anwendung zugreift, ohne dass ein Schreibzugriff auf den `SessionStore` stattfindet, hat die Anwendung noch Zugriff auf abgelaufene Daten. Allerdings sind diese Umstände eher unwahrscheinlich und daher wurde hier der Effizienz Vorrang gegeben.

6.7. Prozessmodellierung

Da Web-Anwendungen inzwischen für immer komplexer werdende Aufgaben eingesetzt werden, beschränkt sich die Erfüllung einer Aufgabe durch eine Web-Anwendung oft nicht nur auf die Anzeige einer Seite und die folgende Verarbeitung der auf dieser Seite getätigten Eingaben. Viele Anwendungen sollen komplexere Aufgaben erfüllen oder ganze Geschäftsprozesse in einer Firma abbilden.

Komplexere Aufgaben erfordern meist mehrere Schritte, die vom Anwender durchgeführt werden müssen. Diese Schritte sollten in der Anwendung auf mehrere Seiten aufgeteilt werden, um den Anwender bei der Konzentration auf den aktuellen Schritt zu unterstützen und nicht mit zu vielen Informationen zu überfordern. Die Abbildung von Geschäftsprozessen erfordert meist noch mehr Ausdrucksmöglichkeiten als die Implementierung einer Abfolge von Schritten in der Anwendung.

Im folgenden Abschnitt gehe ich auf die Möglichkeiten ein, die das Spicey-Framework dem Entwickler bietet, um komplexere Prozesse zu modellieren.

6.7.1. Abstrakte Definition eines Prozesses

Allgemein kann eine Prozess als Reihe von Zuständen und Transitionen zwischen diesen Zuständen definiert werden. Jeder Prozess hat genau einen Startzustand und beliebig viele Endzustände (ein Endzustand ist ein Zustand, aus dem keine Transitionen herausführen). An jede Transition ist eine Bedingung geknüpft. Soll ein Zustandsübergang stattfinden, werden die Bedingungen aller Transitionen, die aus dem aktuellen Zustand herausführen, unter Berücksichtigung eines Ergebnisses geprüft. Die Transition, bei der das Ergebnis deren Bedingung erfüllt, wird als Übergang in den neuen Prozesszustand

gewählt. Ist keine Bedingung erfüllt, findet kein Zustandswechsel statt. Um die Komplexität zu begrenzen, sollte die zu wählende Transition eindeutig sein, ein Ergebnis sollte also nicht mehrere Bedingungen gleichzeitig erfüllen können. Dieses Modell entspricht beispielsweise einem deterministischen Mealy-Automat. Es kann grafisch wie in Abbildung 6.9 dargestellt werden.

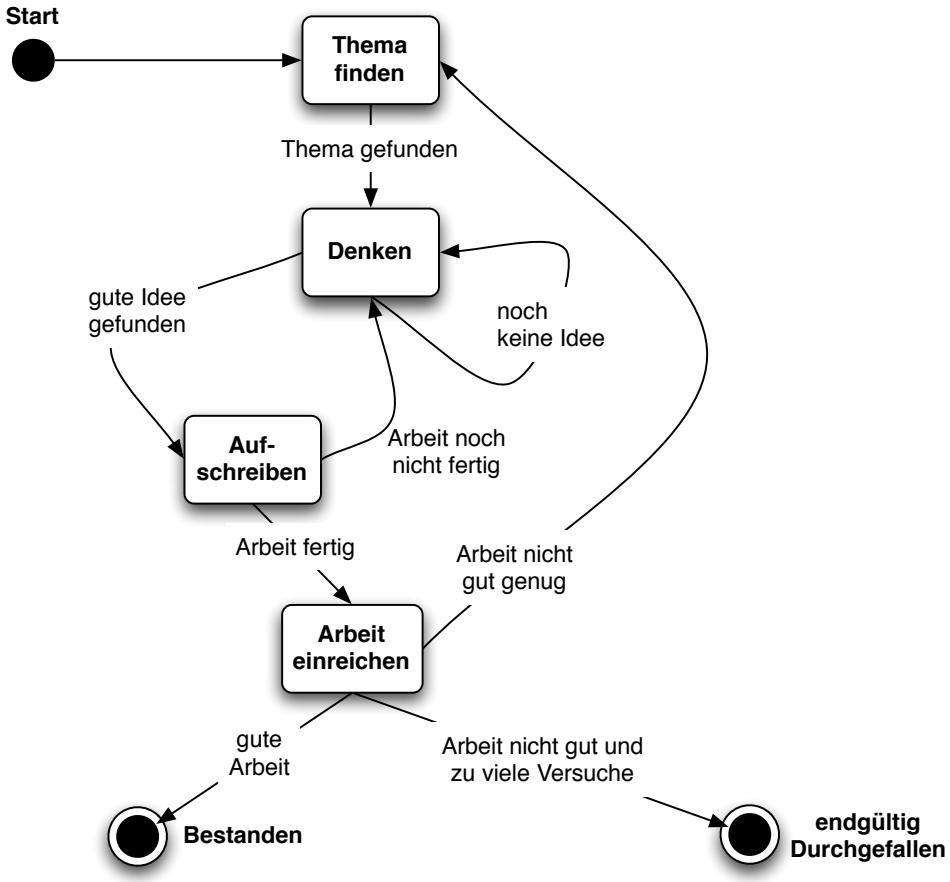

Abbildung 6.9.: Beispiel einer abstrakten Prozessdarstellung: Der Prozess des Schreibens einer Abschlussarbeit

Die Modellierung von Geschäftsprozessen kann natürlich noch wesentlich komplexer werden, als es mit diesem Modell abbildbar wäre. Parallele Bearbeitung von Aufgaben und Prozesse, welche mehrere Benutzer berücksichtigen, können nur mit zusätzlichem Implementierungsaufwand realisiert werden, jedoch ist dieses einfachere Modell in vielen Fällen bereits ausreichend.

6.7.2. Modellierung von Prozessen in der bestehenden HTML-Bibliothek

Mit der in Curry enthaltenen HTML-Bibliothek ist es bereits möglich, einen sich über mehrere Seiten erstreckenden Ablauf zu modellieren. In der HTML-Bibliothek wird der Benutzer über automatisch hinzugefügte versteckte Felder in den Formularen identifiziert. Da diese Felder beim Absenden des Formulars mitgeschickt werden, kann die Anwendung diese auslesen und so den Benutzer bestimmen. In der Anwendung wird ein solcher Ablauf über die Verschachtelung von Handler-Funktionen für die Formular-Buttons realisiert.

```
calcform = return $ HtmlForm "Erste Zahl"
  [htxt "Geben Sie die erste Zahl ein: ", textfield first "",
   button "Weiter" buttonHandler]
  where first free
      buttonHandler _ = return $ HtmlForm "Zweite Zahl"
          [htxt "Geben Sie die zweite Zahl ein: ", textfield second "",
           button "Weiter" secondButtonHandler]
        where second free
          secondButtonHandler env = return $ HtmlForm "Antwort"
            [htxt ("Erste Zahl: " ++ env first ++
                   ", Zweite Zahl: " ++ env second)]
```

Durch das Konzept des statischen Gültigkeitsbereiches (lexical scoping), welches in Curry verwendet wird, kann so auch auf die Formulardaten auf vorherigen Seiten zurückgegriffen werden.

Da die Verknüpfung der Seiten auf versteckten Formularfeldern basiert, ist ausschließlich eine Navigation durch Formular-Buttons möglich. Eine Navigation über Hyperlinks oder mittels der Vor- und Zurück-Funktion des Browsers ist nicht möglich. Diese Einschränkungen lassen sich jedoch leicht umgehen, wenn man eine auf Cookies basierende Lösung zur Identifikation des Benutzers (wie das Session-Modul von Spicey) einsetzt. Die Seiten werden direkt im Programmcode miteinander verknüpft. Für eine höhere Abstraktion muss der Entwickler selbst sorgen, indem er die Handler-Funktionen auslagert und in der Verschachtelung nur noch aufruft. Tut er dies nicht, vermischen sich Programmlogik und Prozessablauflogik und der Prozess kann nicht mehr so leicht angepasst werden, wie es bei einer abstrakten Prozessdefinition der Fall wäre. Da keine abstrakte Beschreibung genutzt wird, ist es ebenfalls nicht möglich, die Definition des Prozesses über einen grafischen Editor durchzuführen, der auf einer solchen abstrakten Beschreibung operieren könnte.

6.7.3. Prozesse durch Aneinanderreihung von Controller-Aufrufen

Im Spicey-Framework können einfache Abläufe modelliert werden, indem verschiedene Controller-Aufrufe aneinander gehängt werden. Im automatisch generierten Code befinden sich bereits Beispiele hierfür. So sind das Anlegen von Einträgen sowie das Editieren von Einträgen zweistufige Prozesse: Zuerst wird das Formular zum Anlegen beziehungsweise Editieren angezeigt und als zweites werden die neuen Daten beziehungsweise die Änderungen gespeichert.

```
newEntryController :: [String] -> IO ([HtmlExp])
newEntryController _ =
    do
     allTags <- getAllTags
     return (blankEntryView allTags createEntryController)

createEntryController :: (String,String,String,CalendarTime,[Tag]) ->
    IO ([HtmlExp])
createEntryController (title,text,author,date,tags) =
    do
     transResult <- runT (
        (newEntry title text author date) |>>=
        (addTagging tags)
     )
     either
        (\_ -> listEntryController [])
        (\error -> displayError (showTError error) [])
        transResult
```

Durch weitere Verkettung können so beliebig lange Seitenabläufe definiert werden. Hierbei kann jedoch nur auf Daten von vorherigen Schritten zurückgegriffen werden, sofern diese explizit weitergegeben werden. Durch Nutzung der WUI-Bibliothek in Spicey wird nicht mehr direkt auf Formularfelder zugegriffen, sondern es werden anwendungseigene Datenstrukturen über Formulare manipuliert. Die manipulierten Daten werden dann explizit an den nächsten Controller weitergegeben. Dadurch wird ein direkter Zugriff auf Formularfelder einer früheren Seite im Ablauf unnötig. Für Daten, die nicht direkt weitergegeben werden aber trotzdem zu einem späteren Zeitpunkt in der Anwendung noch zur Verfügung stehen sollen, kann die Session-Funktionalität eingesetzt werden.

Der genaue Ablauf von Prozessen, welche durch Aneinanderreihung von Controller-Aufrufen konstruiert werden, ist nur durch genaue Betrachtung des Programmcodes ersichtlich. Eine Änderung des Prozesses ist ebenfalls nur schwer möglich, da eine Verknüpfung der Controller an beliebigen und auch mehreren Stellen im Controller-Code erfolgen

kann. Diese Einschränkungen machen diese Methode der Prozessmodellierung nur für sehr einfache Abläufe sinnvoll.

6.7.4. Übergeordnete Prozessdefinitionen

Für eine Modellierung von komplexeren Prozessen, wie in Abbildung 6.9 dargestellt, wird eine Möglichkeit der abstrakteren Beschreibung von Prozessen in der Anwendung benötigt. Diese abstrakte Beschreibungsmöglichkeit bietet das Modul `Processes` des Spicey-Frameworks, es ist in der Projektstruktur im Verzeichnis `system` zu finden.

Prozessdefinition

Für die Beschreibung von Prozessen wird eine Datenstruktur definiert, die einen Graphen abbildet. Ein Graph besteht aus Knoten und Kanten, wobei eine Kante zwei Knoten verbindet (siehe Abbildung 6.10):

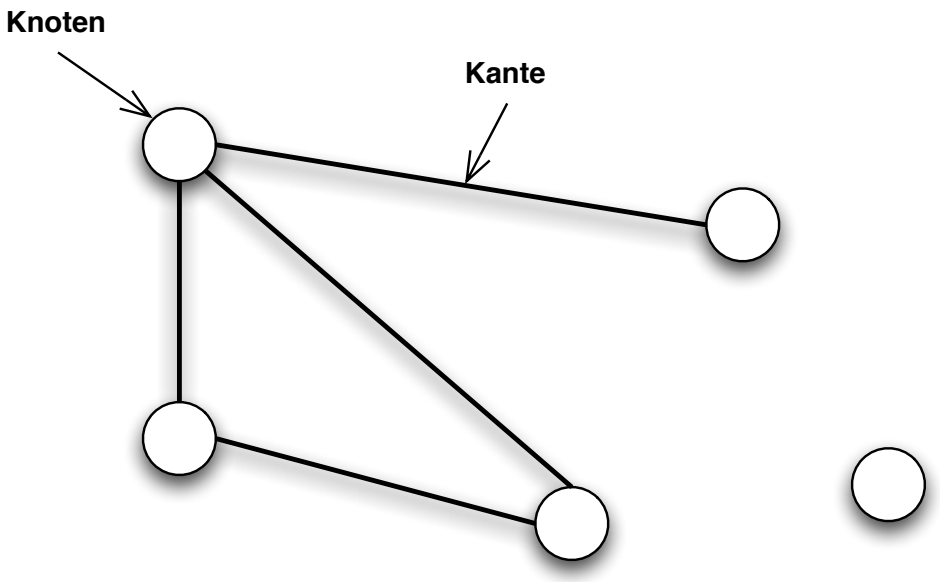

Abbildung 6.10.: Beispiel eines Graphen

Im Falle des Prozesses sind die Knoten des Graphen die Zustände des Prozesses und die Kanten die Transitionen. Die Kanten sind hier gerichtet, das heisst eine Kante verbindet zwei Knoten nur in eine bestimmte Richtung. Zur Verbindung von zwei Knoten in beide Richtungen werden zwei Kanten benötigt.

```
data Process = Process String State [State] [Transition]
```

Ein Prozess besteht aus einem String, welcher den Namen des Prozesses festlegt, dem Startzustand des Prozesses und einer Liste von Zuständen und Transitionen. Um sicherzustellen, dass es genau einen Startzustand gibt, wird dieser gesondert aufgeführt. Es gibt drei Arten von Zuständen:

```
data State =
  StartState StateId ControllerFunctionReference |
  State StateId ControllerFunctionReference |
  EndState StateId ControllerFunctionReference
```

Die Datenstruktur definiert genau drei Zustände: nämlich Startzustände (`StartState`), Zwischenzustände (`State`) und Endzustände (`EndState`). Die enthaltenen Daten unterscheiden sich in allen drei Fällen nicht. Die Aufteilung in drei verschiedene Konstruktoren wurde lediglich als Hilfe bei der Definition von Prozessen gewählt, um die Art eines Zustandes direkt erkennen zu können, ohne überprüfen zu müssen, ob es nur eingehende Transitionen im Falle eines Endzustandes gibt, und im Falle des Startzustandes um die Lesbarkeit von Prozessdefinitionen weiter zu erhöhen. Alle Zustandsdefinitionen enthalten eine `StateId` (welche in der aktuellen Implementierung ein einfacher String ist) zur eindeutigen Identifizierung des Zustandes und eine Referenz zur Controller-Funktion, welche aufgerufen werden soll, wenn sich der Prozess im jeweiligen Zustand befindet. Die `StateId` ist im Falle des Startzustandes nötig, auch wenn es nur genau einen Startzustand gibt, um auch Transitionen zurück zum Startzustand zu ermöglichen (zum Beispiel falls gleich am Anfang des Prozesses eine Schleife modelliert werden soll).

Zur Modellierung der Zustandsübergänge existiert der Datentyp `Transition`:

```
data Transition =
  Transition StateId StateId TransitionGuard
```

Eine Transition (`Transition`) beinhaltet die `StateId` des Anfangs- und Zielzustandes. Außerdem enthält sie eine Bedingung `TransitionGuard`, welche entscheidet, ob eine Transition für einen Zustandsübergang verwendet werden soll oder nicht (siehe Abschnitt 6.7.1). Dies ist eine Funktion, welche einen String als Parameter nimmt (das Ergebnis, das für den Zustandsübergang verwendet werden soll) und einen Wahrheitswert zurückgibt, der bestimmt, ob die Transition für den Zustandsübergang verwendet werden soll.

```
type TransitionGuard = (String -> Bool)
```

In diesem Modell ist es theoretisch möglich, dass für ein Ergebnis bei einem Zustandsübergang die Bedingungen mehrerer Transitionen erfüllt sind. Es muss durch den Entwickler sichergestellt werden, dass dies nicht passiert, indem grundsätzlich sich gegenseitig ausschließende Bedingungen verwendet werden. Das Verhalten für mehrere gleichzeitig erfüllte Bedingungen von Transitionen aus einem Zustand ist nicht definiert.

Die vorgesehene Verwendung ist, dass lediglich Vergleiche des Ergebnis-Strings von den Bedingungs-Funktionen durchgeführt werden. Einige solcher einfachen Bedingungs-Funktionen sind bereits vordefiniert:

```
always :: TransitionGuard
always _ = True

on_success :: TransitionGuard
on_success value = (value == "ok")

on_error :: TransitionGuard
on_error value = (value == "error")
```

Um mit diesem Modell eine Entscheidung zu definieren, welche einen Übergang von einem Anfangszustand in verschiedene Zielzustände anhand von bestimmten Bedingungen beschreibt, muss für jeden Zielzustand eine eigene Transition aus dem Anfangszustand definiert werden:

```
[
  Transition "beginning State" "target State 1" on_success,
  Transition "beginning State" "target State 2" on_error,
  Transition "beginning State" "target State 1" on_unknown,
]
```

Da dies häufig vorkommt, gibt es hierfür eine Hilfsfunktion `decision`, welche die Definition einer solchen Gruppe von Transitionen mit demselben Anfangszustand vereinfacht:

```
decision :: StateId -> [(StateId, TransitionGuard)] -> [Transition]
```

Die Funktion nimmt die `StateId` eines Anfangszustandes und eine Liste von `StateId` der Zielzustände mit zugehörigen Bedingungen und generiert daraus die Liste benötigter Transitionen. Das obige Beispiel kann damit übersichtlicher definiert werden:

```
decision "beginning State" [
  ("target State 1", on_success),
  ("target State 2", on_error),
  ("target State 3", on_unknown)
]
```

Verwendung von Prozessen

Um nun einen Prozess in der Anwendung abzubilden, muss dieser zunächst mit Hilfe der im vorherigen Abschnitt vorgestellten Prozessdefinition beschrieben werden. Im Beispiel des Weblogs wäre ein Prozess die Erstellung eines Weblog-Eintrags und danach die Erstellung mehrerer Kommentare zu diesem Eintrag. Abbildung 6.11 stellt diesen Prozess grafisch dar.

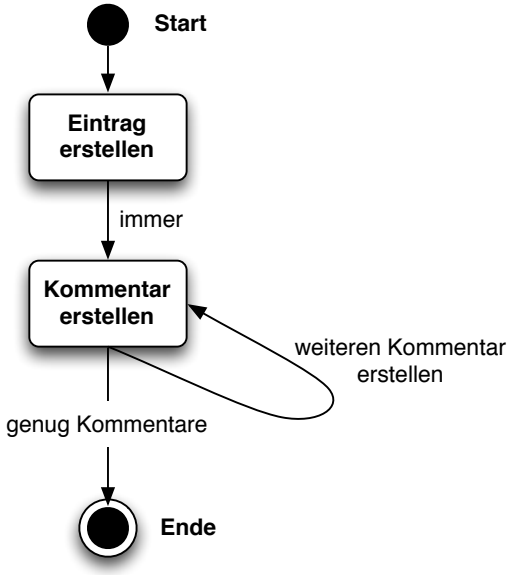

Abbildung 6.11.: Prozess zum Anlegen eines Weblog-Eintrags und Kommentaren

Der Prozess kann durch folgende Definition beschrieben werden:

```
testProcess = Process "Erstellung eines Eintrags und Kommentaren"
  (StartState new_entry NewEntryController)
  [State new_comment NewCommentController,
   EndState end ListCommentController]
  (
```

```
    [ Transition new_entry new_comment (always) ] ++
    decision new_comment [(end, on_finished), (new_comment, on_another)]
  )
  where
    new_entry = "new_entry"
    new_comment = "new_comment"
    end = "end"
```

Um sicherzustellen, dass die Zustandsnamen für gleiche Zustände in den Transitionen auch denen in den Zuständen entsprechen, wurden diese als lokale Deklarationen definiert. Würden in der Prozessdefinition ausschließlich Strings verwendet, würde ein Tippfehler zu einem fehlerhaften Prozess führen. Durch die Verwendung von lokalen Deklarationen findet der Compiler etwaige Tippfehler und auch versehentliche Benutzung desselben Zustandsnamens für mehrere Zustände ist so unwahrscheinlicher, solange man den lokalen Variablennamen und den zugehörigen String gleich wählt.

Als zweites müssen die Controller-Funktionen, die zum Prozess gehören, entsprechend angepasst werden. Es wird festgelegt, dass die Controller-Funktion zum Anlegen eines neuen Blog-Eintrags, welche die erste im Prozess ist, auch den Prozess starten soll. Das heisst, sobald die Seite zum Anlegen eines neuen Eintrags aufgerufen wird, wird auch der Prozess gestartet. Das Starten eines neuen Prozesses geschieht über die Funktion `startProcess`, welche als Parameter den Namen des zu startenden Prozesses erwartet.

```
newEntryController :: [String] -> IO ([HtmlExp])
newEntryController _ =
    do
     startProcess "Erstellung eines Eintrags und Kommentaren"
     allTags <- getAllTags
     return (blankEntryView allTags createEntryController)
```

Der Start des Prozesses kann an einer beliebigen Stelle in einem beliebigen Controller geschehen. Es ist nicht notwendig, dass der Prozess in der Controller-Funktion gestartet wird, welche auch die erste im Prozess ist.

Da das Anlegen eines Eintrags aus der Verkettung von `newEntryController` und `createEntryController` besteht (siehe Abschnitt 6.7.3), läuft nun dieser „Subprozess“ eigenständig ab. Nach der Erstellung des Eintrags im `createEntryController` wird durch Aufruf von `advanceInProcess` und Übergabe eines Ergebnisses ein Zustandsübergang eingeleitet.

```
createEntryController :: (String,String,String,CalendarTime,[Tag])
                         -> IO ([HtmlExp])
```

```
createEntryController (title,text,author,date,tags) =
    do
     transResult <- runT ((newEntry title text author date)
                          |>>= (addTagging tags))
     advanceInProcess ""
     either
      (\_ -> nextInProcessOrDefault)
      (\error -> displayError (showTError error) [])
      transResult
```

Als Ergebnis wird in diesem Fall ein leerer String übergeben, da nach Prozessdefinition für jedes Ergebnis ein Zustandsübergang stattfindet. Anstatt nun nach dem Anlegen des Eintrags auf die Liste der Einträge zurückzugehen, wie es in der generierten Controller-Funktion der Fall war, wird `nextInProcessOrDefault` aufgerufen, was bewirkt, dass nach der Prozessdefinition vorgegangen wird und die Controller-Funktion, welche im aktuellen Zustand angegeben ist, aufgerufen wird. In diesem Fall ist dies immer `newCommentController`. Da `newCommentController` ebenfalls wieder einen „Subprozess“ einleitet, bleibt dieser unverändert:

```
newCommentController :: [String] -> IO ([HtmlExp])
newCommentController _ =
    do
     allEntrys <- getAllEntrys
     return (blankCommentView allEntrys createCommentController)
```

Im `createCommentController` findet nun wieder ein Zustandsübergang statt.

```
createCommentController :: (String,String,CalendarTime,Entry)
                           -> IO ([HtmlExp])
createCommentController (text,author,date,entry) =
    do
     if (text == "letzter")
      then advanceInProcess "finished"
      else advanceInProcess "another"
     transResult <- runT
        (newCommentWithEntryCommentingKey text author date (entryKey entry))
     either
      (\_ -> nextInProcessOrDefault)
      (\error -> displayError (showTError error) [])
      transResult
```

Hier wird beispielhaft anhand des eingegebenen Textes für den Kommentar über das Ergebnis der Controller-Funktion entschieden. Wurde als Text „letzter“ eingegeben, ist das Ergebnis „finished“, ansonsten „another“. Je nach Ergebnis findet nach der Prozessdefinition ein anderer Zustandsübergang statt. Beim Ergebnis „finished“ wird als nächste Seite die Liste der Kommentare angezeigt, beim Ergebnis „another“ kann ein weiterer Kommentar eingegeben werden.

6.7.5. Nutzung der Prozessdefinitionen als Navigation

Ähnlich wie die Routes (siehe Abschnitt 6.5.3) können auch die definierten Prozesse für ein Navigationsmenü genutzt werden. Hierbei erfolgt eine Abkehr von der Sicht einer Web-Anwendung als Menge von Seiten ab und sie wird als Menge von Arbeitsabläufen betrachtet, die der Benutzer mit dieser Anwendung durchführen kann. Falls man dieses Konzept in der Anwendung nutzen möchte, bietet das Spicey Framework bereits entsprechende integrierte Funktionalität hierfür an. In jedem generierten Projekt gibt es eine Seite mit der Auflistung aller vorhandenen Prozesse (siehe Abbildung 6.12). Diese können über den entsprechenden Menüeintrag des automatisch erzeugten Navigationsmenüs erreicht werden. Ein Klick auf einen Prozessnamen startet den entsprechenden Prozess.

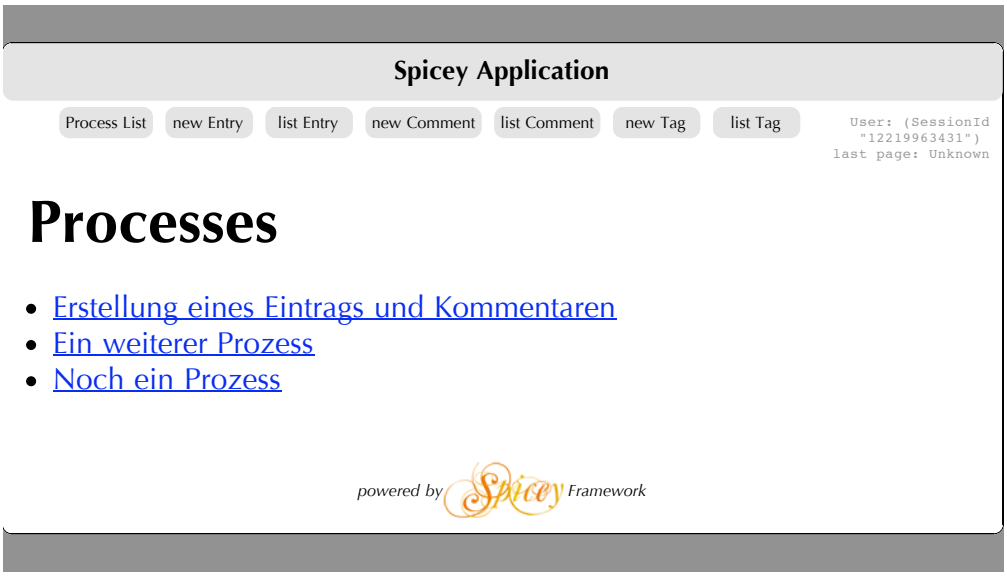

Abbildung 6.12.: Screenshot: Liste aller verfügbaren Prozesse

Selbst wenn man diese Strukturierung der Anwendung nicht Nutzen möchte, ist die Prozessliste eine einfache Möglichkeit, erstellte Prozesse direkt zu testen, da sie automatisch in der angezeigten Prozessliste auftauchen. Die Seite zum Auflisten aller Prozesse

ist, genau wie alle anderen Seiten, als eigener View implementiert, welcher von einer Controller-Funktion gesteuert wird. Dieser View ist im Modul `SpiceySystemView` und der Controller in `SpiceySystemController` in den entsprechenden Verzeichnissen enthalten. Diese Module können in der Zukunft um weitere systemspezifische Funkionen erweitert werden.

7. Vergleich mit bestehenden Frameworks

Nachdem nun alle Aspekte des ausgearbeiteten und implementierten Web-Frameworks vorgestellt wurden, bietet sich eine Betrachtung der Eigenschaften des Frameworks unter Berücksichtigung von anderen, bereits existierenden Frameworks an.

Hierbei sind zuerst die großen Unterschiede der zugrunde liegenden Programmiersprachen und damit auch Programmierparadigmen zu nennen. Während die meisten am Anfang dieser Arbeit untersuchten Frameworks eine imperative, objektorientierte Sprache verwenden, ist die Basis von Spicey eine logisch-funktionale Sprache. Die Vorteile dieses Paradigmenwechsels wurden bereits in Abschnitt 4.2 erläutert.

Die Umsetzung des *Model View Controller* Patterns sowie die automatische Generierung einer Projektstruktur sind auch bei Ruby on Rails, Django und mit Hilfsmitteln bei Seam möglich. Jedoch geschieht dies in keinem der Frameworks aus einer so abstrakten Beschreibung wie einem ER-Diagramm.

Das Konzept von Routes, wie es auch in Spicey umgesetzt wurde, existiert in allen Frameworks. Der Abstraktionsgrad der Beschreibung dieser URL-Controller-Verknüpfungen variiert jedoch stark.

Alle Frameworks bis auf Seaside setzen auf das *Template View* Pattern anstatt des robusteren *Transform View* Patterns. Der Grund hierfür ist, dass Designer, die die Gestaltung der Seiten übernehmen, eher mit Werkzeugen arbeiten, die HTML-Code erzeugen, und nicht direkt am Programmcode arbeiten können. Die eigentliche Arbeit des Designers sollte dieser jedoch durch Erstellung der Stylesheets erledigen können, welche dann auf von der Web-Applikation generierten HTML-Code angewendet wird. Daher eignet sich das *Transform View* Pattern hier besser.

Der geringe Fokus auf Konsistenzerhaltung der Datenbank ist ein sehr großes Problem in allen Frameworks, selbst in Seam, welches auf die hochentwickelte Enterprise Java Bean 3.0 Technologie von Sun setzt. In der von Spicey verwendeten ERD-Bibliothek wird dagegen viel Wert auf Einhaltung von Konsistenzbedingungen gelegt. Dies kommt auf lange Sicht der Erstellung einer soliden Anwendung zu gute.

Das Konzept von abstrakten Prozessmodellierungen wird einzig in Seam aufgegriffen und dank der unabhängigen Entwicklung von jBPM sehr gut unterstützt. In Spicey ist

bereits eine Grundlage vorhanden, die sich zwar keinesfalls mit jBPM messen kann, jedoch bereits eine nützliche Hilfe bei der Beschreibung von in der Anwendung ablaufenden Prozessen darstellt. In allen anderen untersuchten Frameworks gibt es noch keine Ansätze für eine solche Unterstützung des Entwicklers. Aufgrund seiner Konzeption ist es im Seaside-Framework allerdings möglich, beispielsweise Formulare auf mehrere Seiten aufzuteilen, da auch in Seaside eine Berechnung über mehrere Requests „fortgesetzt“ werden kann, wie es auch in der HTML-Bibliothek vorgesehen ist (siehe Abschnitt 6.7.2).

8. Ausblick

Das in dieser Arbeit beschriebene und implementierte Web-Framework Spicey ermöglicht eine schnelle und dennoch strukturierte Entwicklung von Web-Anwendungen. Ein Prototyp kann automatisch aus der abstrakten Beschreibung des Datenmodells generiert werden und erspart dem Entwickler zeitraubende und monotone Implementierung. Dabei wird eine für Web-Anwendungen vielfach bewährte Architektur und Struktur der Anwendung vorgegeben. Die Nutzung von bereits bestehenden, sorgfältig entwickelten Bibliotheken und die konsequente Weiterentwicklung zu einem vollständigen Framework zur Erstellung von Web-Anwendungen geben dem Programmierer ein Werkzeug in die Hand, welches die häufigsten Anforderungen der Web-Entwicklung bereits abdeckt.

Trotz allem existiert natürlich noch viel Raum zur Verbesserung und Weiterentwicklung des Frameworks.

Die durchgängige aktive Unterstützung des Entwicklers, auch nach dem Generieren des Projektes, kann noch weiter verbessert werden. Die nachträgliche, erneute Generierung von Teilkomponenten unter Berücksichtigung von bereits durchgeführten Anpassungen und Ergänzungen an der Funktionalität der Models würde eine noch flexiblere Entwicklung ermöglichen.

Eine weitere Verbesserung wäre eine Möglichkeit, die Datenbank an die Veränderungen einer sich entwickelnden Anwendung leichter anpassen zu können. Im Ruby on Rails Framework wird dies mit sogenannten Migrationen versucht. Diese erlauben eine Art Versionierung des Datenbank-Schemas. Darüber hinaus kann der Entwickler Skripte schreiben, welche das Datenbank-Schema an die aktualisierte Anwendung anpassen und gleichzeitig die bereits enthaltenen Daten einer Produktivversion der Anwendung migrieren. So wird eine Aktualisierung der Anwendung im Produktivbetrieb vereinfacht und kann dadurch öfter durchgeführt werden.

Die im Spicey-Framework bereits enthaltene einfache Prozessmodellierung kann zur Unterstützung von komplexeren Prozessen noch erweitert werden. Hier sind eventuell auch Aspekte der logischen Programmierung bei der Definition von Zustandsübergängen anwendbar. Auch wenn die enthaltenen Modellierungsmöglichkeiten bereits viele Anforderungen abdecken, können sie noch erweitert werden, um noch komplexere Prozesse einfach abbilden zu können.

Literaturverzeichnis

Antoy u. Hanus 2002
ANTOY, Sergio ; HANUS, Michael: Functional Logic Design Patterns. In: *In Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002*, Springer LNCS, 2002, S. 67–87

Antoy u. Hanus 2007
ANTOY, Sergio ; HANUS, Michael: *Curry - A Tutorial Introduction*. Draft, dec 2007

Berners-Lee u. a. 1994
BERNERS-LEE, T. ; MASINTER, L. ; MCCAHILL, M.: *Uniform Resource Locators (URL)*. RFC 1738 (Proposed Standard). `http://www.ietf.org/rfc/rfc1738.txt`. Version: Dezember 1994 (Request for Comments). – Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986

Bos u. a. 2007
BOS, Bert ; ÇELIK, Tantek ; LIE, Håkon W. ; HICKSON, Ian: Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification / W3C. 2007. – Candidate Recommendation. – http://www.w3.org/TR/2007/CR-CSS21-20070719

Brassel u. a. 2008
BRASSEL, B. ; HANUS, M. ; MÜLLER, M.: High-Level Database Programming in Curry. In: *Proc. of the 10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, Springer LNCS 4902, 2008, S. 316–332

Burke u. Monson-Haefel 2006
BURKE, Bill ; MONSON-HAEFEL, Richard: *Enterprise JavaBeans 3.0 (5th Edition)*. O'Reilly Media, Inc., 2006. – ISBN 059600978X

Buschmann u. a. 2007
BUSCHMANN, Frank ; HENNEY, Kevlin ; SCHMIDT, Douglas: *Pattern Oriented Software Architecture: On Patterns and Pattern Languages (Wiley Software Patterns Series)*. John Wiley & Sons, 2007. – ISBN 0471486485

Cowlishaw 1999
COWLISHAW, Mike: ECMAScript Language Specification / Ecma International. http://www.ecma-international.org/publications/standards/Ecma-262.htm, dec 1999. – Forschungsbericht

Farley 2007
FARLEY, Jim: *Practical JBoss®Seam Projects (Practical).* Berkely, CA, USA : Apress, 2007. – ISBN 1590598636

Fielding u. a. 1999
FIELDING, R. ; GETTYS, J. ; MOGUL, J. ; FRYSTYK, H. ; MASINTER, L. ; LEACH, P. ; BERNERS-LEE, T.: *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616 (Draft Standard). `http://www.ietf.org/rfc/rfc2616.txt`. Version: Juni 1999 (Request for Comments). – Updated by RFC 2817

Fischer 2005
FISCHER, Sebastian: A functional logic database library. In: *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming.* New York, NY, USA : ACM, 2005. – ISBN 1–59593–069–8, S. 54–59

Fowler 2002
FOWLER, Martin: *Patterns of Enterprise Application Architecture.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0321127420

Hanus 2001
HANUS, M.: High-Level Server Side Web Scripting in Curry. In: *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, Springer LNCS 1990, 2001, S. 76–92

Hanus 2006
HANUS, Michael: Type-oriented construction of web user interfaces. In: *PPDP '06: Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming.* New York, NY, USA : ACM, 2006. – ISBN 1–59593–388–3, S. 27–38

Hanus u. a. 2006
HANUS, Michael ; ANTOY, Sergio ; BRASSEL, Bernd ; KUCHEN, Herbert ; LÓPEZ-FRAGUAS, Francisco J. ; LUX, Wolfgang ; NAVARRO, Juan José M. ; STEINER, Frank: *Curry - An Integrated Functional Logic Language*, 2006. (0.8.2)

Pemberton 2002
PEMBERTON, Steven: XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition) / W3C. 2002. – W3C Recommendation. – http://www.w3.org/TR/2002/REC-xhtml1-20020801

Ray 2003
RAY, Erik T.: *Learning XML.* Second Edition. O'Reilly Media, Inc., 2003

A. Inhalt der CD

Die dieser Arbeit beiliegende CD enthält den Quellcode des implementierten Frameworks sowie diese Arbeit in digitaler Form als PDF. Es folgt eine genaue Auflistung der Verzeichnisstruktur auf der CD.

- `thesis.pdf` – diese Arbeit als PDF-Datei
- `blog.term` – die Term-Datei zur Beschreibung des ER-Diagramms für das Weblog-Beispiel, welches in dieser Arbeit verwendet wurde
- `spicey_framework` – das implementierte Framework
  - `project_generator` – enthält das Programm zur Generierung eines neuen Projektes
    - `new.curry` Programm, welches die Generierung eines neuen Projektes anhand der enthaltenen Strukturbeschreibung steuert
    - `spiceup.sh` Shell-Script zum Starten der Projektgenerierung
    - `resource_files` enthält alle Dateien, welche nicht anhand des ER-Diagrammes generiert werden müssen
  - `scaffolding` – enthält den Programmcode der Generatoren für Models, Controller und Views
- `example_application` – die aus der `blog.term` generierte Anwendung

Listings im Anhang

B. Vollständiger Quellcode der Beispielanwendung

Die in dieser Arbeit verwendete Beispielanwendung kann komplett aus folgender ERD Term Definition generiert werden:

Listing B.1: Term-Notation des ER-Diagrammes

```
(ERD "Blog"
[
 (Entity "Entry"
        [
         (Attribute "Title" (StringDom Nothing) Unique False),
         (Attribute "Text" (StringDom Nothing) NoKey False),
         (Attribute "Author" (StringDom Nothing) NoKey False),
         (Attribute "Date" (DateDom Nothing) NoKey False)
        ]
 ),
 (Entity "Comment"
        [
         (Attribute "Text" (StringDom Nothing) NoKey False),
         (Attribute "Author" (StringDom Nothing) NoKey False),
         (Attribute "Date" (DateDom Nothing) NoKey False)
        ]
 ),
 (Entity "Tag"
  [
         (Attribute "Name" (StringDom Nothing) Unique False)
        ]
 )
]
[
 (Relationship "Commenting"
  [
   REnd "Entry" "commentsOn" (Exactly 1),
   REnd "Comment" "isCommentedBy" (Range 0 Nothing)
  ]
 ),
 (Relationship "Tagging"
  [
   REnd "Entry" "tags" (Range 0 Nothing),
   REnd "Tag" "tagged" (Range 0 Nothing)
  ]
 )
]
)
```

Aus dieser Definition werden folgende Module generiert.

B.1. Model-Schicht

Der Code der Model-Schicht wird vollständig von der ERD-Bibliothek generiert und wird hier lediglich zu Referenzzwecken gezeigt.

Listing B.2: Modul Blog.curry

```
module Blog(Entry, Comment, Tag, EntryKey, CommentKey, TagKey, entryTitle,
    setEntryTitle, entryText, setEntryText, entryAuthor, setEntryAuthor, entryDate
    , setEntryDate, commentText, setCommentText, commentAuthor, setCommentAuthor,
    commentDate, setCommentDate, commentEntryCommentingKey,
    setCommentEntryCommentingKey, tagName, setTagName, entry, entryKey, newEntry,
    updateEntry, deleteEntry, getEntry, comment, commentKey,
    newCommentWithEntryCommentingKey, updateComment, deleteComment, getComment,
    tag, tagKey, newTag, updateTag, deleteTag, getTag, tagging, newTagging,
    deleteTagging, tagged, tags, isCommentedBy, commenting, commentsOn,
    checkAllData, checkTagging, checkEntry, checkComment, checkTag, saveAllData,
    restoreAllData) where

import ERDGeneric
import Database
import Time

data Entry
     = Entry Key String String String CalendarTime

data Comment
     = Comment Key String String CalendarTime Key

data Tag
     = Tag Key String

data EntryKey
     = EntryKey Key

data CommentKey
     = CommentKey Key

data TagKey
     = TagKey Key

data Tagging
     = Tagging Key Key

--- Sets the value of attribute "EntryTaggingKey" in a Tagging entity.
setTaggingEntryTaggingKey :: Tagging -> EntryKey -> Tagging
setTaggingEntryTaggingKey (Tagging _ x2) x = Tagging (entryKeyToKey x) x2

--- Sets the value of attribute "TagTaggingKey" in a Tagging entity.
setTaggingTagTaggingKey :: Tagging -> TagKey -> Tagging
setTaggingTagTaggingKey (Tagging x1 _) x = Tagging x1 (tagKeyToKey x)

--- Sets the value of attribute "Key" in a Entry entity.
setEntryKey :: Entry -> Key -> Entry
setEntryKey (Entry _ x2 x3 x4 x5) x = Entry x x2 x3 x4 x5

--- Gets the value of attribute "Title" of a Entry entity.
entryTitle :: Entry -> String
entryTitle (Entry _ x _ _ _) = x

--- Sets the value of attribute "Title" in a Entry entity.
```

```
setEntryTitle :: Entry -> String -> Entry
setEntryTitle (Entry x1 _ x3 x4 x5) x = Entry x1 x x3 x4 x5

--- Gets the value of attribute "Text" of a Entry entity.
entryText :: Entry -> String
entryText (Entry _ _ x _ _) = x

--- Sets the value of attribute "Text" in a Entry entity.
setEntryText :: Entry -> String -> Entry
setEntryText (Entry x1 x2 _ x4 x5) x = Entry x1 x2 x x4 x5

--- Gets the value of attribute "Author" of a Entry entity.
entryAuthor :: Entry -> String
entryAuthor (Entry _ _ _ x _) = x

--- Sets the value of attribute "Author" in a Entry entity.
setEntryAuthor :: Entry -> String -> Entry
setEntryAuthor (Entry x1 x2 x3 _ x5) x = Entry x1 x2 x3 x x5

--- Gets the value of attribute "Date" of a Entry entity.
entryDate :: Entry -> CalendarTime
entryDate (Entry _ _ _ _ x) = x

--- Sets the value of attribute "Date" in a Entry entity.
setEntryDate :: Entry -> CalendarTime -> Entry
setEntryDate (Entry x1 x2 x3 x4 _) x = Entry x1 x2 x3 x4 x

--- Sets the value of attribute "Key" in a Comment entity.
setCommentKey :: Comment -> Key -> Comment
setCommentKey (Comment _ x2 x3 x4 x5) x = Comment x x2 x3 x4 x5

--- Gets the value of attribute "Text" of a Comment entity.
commentText :: Comment -> String
commentText (Comment _ x _ _ _) = x

--- Sets the value of attribute "Text" in a Comment entity.
setCommentText :: Comment -> String -> Comment
setCommentText (Comment x1 _ x3 x4 x5) x = Comment x1 x x3 x4 x5

--- Gets the value of attribute "Author" of a Comment entity.
commentAuthor :: Comment -> String
commentAuthor (Comment _ _ x _ _) = x

--- Sets the value of attribute "Author" in a Comment entity.
setCommentAuthor :: Comment -> String -> Comment
setCommentAuthor (Comment x1 x2 _ x4 x5) x = Comment x1 x2 x x4 x5

--- Gets the value of attribute "Date" of a Comment entity.
commentDate :: Comment -> CalendarTime
commentDate (Comment _ _ _ x _) = x

--- Sets the value of attribute "Date" in a Comment entity.
setCommentDate :: Comment -> CalendarTime -> Comment
setCommentDate (Comment x1 x2 x3 _ x5) x = Comment x1 x2 x3 x x5

--- Gets the value of attribute "EntryCommentingKey" of a Comment entity.
commentEntryCommentingKey :: Comment -> EntryKey
commentEntryCommentingKey (Comment _ _ _ _ x) = EntryKey x

--- Sets the value of attribute "EntryCommentingKey" in a Comment entity.
setCommentEntryCommentingKey :: Comment -> EntryKey -> Comment
```

```
setCommentEntryCommentingKey (Comment x1 x2 x3 x4 _) x = Comment x1 x2 x3 x4 (
    entryKeyToKey x)

--- Sets the value of attribute "Key" in a Tag entity.
setTagKey :: Tag -> Key -> Tag
setTagKey (Tag _ x2) x = Tag x x2

--- Gets the value of attribute "Name" of a Tag entity.
tagName :: Tag -> String
tagName (Tag _ x) = x

--- Sets the value of attribute "Name" in a Tag entity.
setTagName :: Tag -> String -> Tag
setTagName (Tag x1 _) x = Tag x1 x

--- Dynamic predicate representing the relation between keys and Entry entities.
entry :: EntryKey -> Entry -> Dynamic
entry key obj
     | key =:= (entryKey obj)
     = entryEntry obj

entryEntry :: Entry -> Dynamic
entryEntry = persistent "file:/Users/darkuni/test3/EntryDB"

--- Gets the key of a Entry entity.
entryKey :: Entry -> EntryKey
entryKey (Entry x _ _ _ _) = EntryKey x

entryKeyToKey :: EntryKey -> Key
entryKeyToKey (EntryKey k) = k

maybeEntryKeyToKey :: Maybe EntryKey -> Maybe Key
maybeEntryKeyToKey Nothing = Nothing
maybeEntryKeyToKey (Just (EntryKey k)) = Just k

--- Inserts a new Entry entity.
newEntry :: String -> String -> String -> CalendarTime -> Transaction Entry
newEntry title_p text_p author_p date_p = (unique entryTitle entryEntry title_p)
    |>> (ERDGeneric.newEntry (entryKeyToKey . entryKey) setEntryKey entryEntry (
    Entry 0 title_p text_p author_p date_p))

--- Updates an existing Entry entity.
updateEntry :: Entry -> Transaction ()
updateEntry entry_p = (uniqueUpdate entryKey entryTitle entry entry_p) |>> (
    ERDGeneric.updateEntry entryKey entry entry_p)

--- Deletes an existing Entry entity.
deleteEntry :: Entry -> Transaction ()
deleteEntry entry_p = (requiredForeignDBKey taggingEntryTaggingKey taggingEntry (
    entryKey entry_p)) |>> ((requiredForeignDBKey commentEntryCommentingKey
    commentEntry (entryKey entry_p)) |>> (ERDGeneric.deleteEntry entryKey entry
    entry_p))

--- Gets a Entry entity stored in the database with the given key.
getEntry :: EntryKey -> Transaction Entry
getEntry key = ERDGeneric.getEntry key entry

--- Dynamic predicate representing the relation between keys and Comment entities.
comment :: CommentKey -> Comment -> Dynamic
comment key obj
     | key =:= (commentKey obj)
     = commentEntry obj
```

```
commentEntry :: Comment -> Dynamic
commentEntry = persistent "file:/Users/darkuni/test3/CommentDB"

--- Gets the key of a Comment entity.
commentKey :: Comment -> CommentKey
commentKey (Comment x _ _ _ _) = CommentKey x

commentKeyToKey :: CommentKey -> Key
commentKeyToKey (CommentKey k) = k

maybeCommentKeyToKey :: Maybe CommentKey -> Maybe Key
maybeCommentKeyToKey Nothing = Nothing
maybeCommentKeyToKey (Just (CommentKey k)) = Just k

--- Inserts a new Comment entity.
newCommentWithEntryCommentingKey :: String -> String -> CalendarTime -> EntryKey
    -> Transaction Comment
newCommentWithEntryCommentingKey text_p author_p date_p entryCommentingKey_p = (
    existsDBKey entryKey entryEntry entryCommentingKey_p) |>> (ERDGeneric.newEntry
     (commentKeyToKey . commentKey) setCommentKey commentEntry (Comment 0 text_p
    author_p date_p (entryKeyToKey entryCommentingKey_p)))

--- Updates an existing Comment entity.
updateComment :: Comment -> Transaction ()
updateComment comment_p = (existsDBKey entryKey entryEntry (
    commentEntryCommentingKey comment_p)) |>> (ERDGeneric.updateEntry commentKey
    comment comment_p)

--- Deletes an existing Comment entity.
deleteComment :: Comment -> Transaction ()
deleteComment comment_p = ERDGeneric.deleteEntry commentKey comment comment_p

--- Gets a Comment entity stored in the database with the given key.
getComment :: CommentKey -> Transaction Comment
getComment key = ERDGeneric.getEntry key comment

--- Dynamic predicate representing the relation between keys and Tag entities.
tag :: TagKey -> Tag -> Dynamic
tag key obj
     | key =:= (tagKey obj)
     = tagEntry obj

tagEntry :: Tag -> Dynamic
tagEntry = persistent "file:/Users/darkuni/test3/TagDB"

--- Gets the key of a Tag entity.
tagKey :: Tag -> TagKey
tagKey (Tag x _) = TagKey x

tagKeyToKey :: TagKey -> Key
tagKeyToKey (TagKey k) = k

maybeTagKeyToKey :: Maybe TagKey -> Maybe Key
maybeTagKeyToKey Nothing = Nothing
maybeTagKeyToKey (Just (TagKey k)) = Just k

--- Inserts a new Tag entity.
newTag :: String -> Transaction Tag
newTag name_p = (unique tagName tagEntry name_p) |>> (ERDGeneric.newEntry (
    tagKeyToKey . tagKey) setTagKey tagEntry (Tag 0 name_p))
```

```
--- Updates an existing Tag entity.
updateTag :: Tag -> Transaction ()
updateTag tag_p = (uniqueUpdate tagKey tagName tag tag_p) |>> (ERDGeneric.
    updateEntry tagKey tag tag_p)

--- Deletes an existing Tag entity.
deleteTag :: Tag -> Transaction ()
deleteTag tag_p = (requiredForeignDBKey taggingTagTaggingKey taggingEntry (tagKey
    tag_p)) |>> (ERDGeneric.deleteEntry tagKey tag tag_p)

--- Gets a Tag entity stored in the database with the given key.
getTag :: TagKey -> Transaction Tag
getTag key = ERDGeneric.getEntry key tag

taggingEntryTaggingKey :: Tagging -> EntryKey
taggingEntryTaggingKey (Tagging x _) = EntryKey x

taggingTagTaggingKey :: Tagging -> TagKey
taggingTagTaggingKey (Tagging _ x) = TagKey x

--- Dynamic predicate representing the Tagging relation between Entry entities and
     Tag entities
tagging :: EntryKey -> TagKey -> Dynamic
tagging (EntryKey key1) (TagKey key2) = taggingEntry (Tagging key1 key2)

--- Inserts a new Tagging relation between a Entry entity and a Tag entity
newTagging :: EntryKey -> TagKey -> Transaction ()
newTagging key1 key2 = (existsDBKey entryKey entryEntry key1) |>> ((existsDBKey
    tagKey tagEntry key2) |>> ((unique2 taggingEntryTaggingKey
    taggingTagTaggingKey taggingEntry key1 key2) |>> (newEntryR key1 key2 tagging)
    ))

--- Deletes an existing Tagging relation between a Entry entity and a Tag entity
deleteTagging :: EntryKey -> TagKey -> Transaction ()
deleteTagging key1 key2 = deleteEntryR key1 key2 tagging

taggingEntry :: Tagging -> Dynamic
taggingEntry = persistent "file:/Users/darkuni/test3/TaggingDB"

--- Dynamic predicate representing role "tagged".
tagged :: EntryKey -> TagKey -> Dynamic
tagged = tagging

--- Dynamic predicate representing role "tags".
tags :: TagKey -> EntryKey -> Dynamic
tags = flip tagging

--- Dynamic predicate representing role "isCommentedBy".
isCommentedBy :: EntryKey -> CommentKey -> Dynamic
isCommentedBy key1 key2 = (comment key2 f2) |&> (key1 =:= (
    commentEntryCommentingKey f2))
   where
     f2 free

--- Dynamic predicate representing the Commenting relation between Entry entities
    and Comment entities
commenting :: EntryKey -> CommentKey -> Dynamic
commenting = isCommentedBy

--- Dynamic predicate representing role "isCommentedBy".
commentsOn :: CommentKey -> EntryKey -> Dynamic
commentsOn = flip isCommentedBy
```

```
--- Checks the consistency of the complete database.
checkAllData :: Transaction ()
checkAllData = checkTagging |>> (checkEntry |>> (checkComment |>> checkTag))

--- Checks the consistency of the database for Tagging entities.
checkTagging :: Transaction ()
checkTagging = (getDB (queryAll taggingEntry)) |>>= (mapT_ checkTaggingEntry)

--- Checks the consistency of the database for Entry entities.
checkEntry :: Transaction ()
checkEntry = (getDB (queryAll entryEntry)) |>>= (mapT_ checkEntryEntry)

--- Checks the consistency of the database for Comment entities.
checkComment :: Transaction ()
checkComment = (getDB (queryAll commentEntry)) |>>= (mapT_ checkCommentEntry)

--- Checks the consistency of the database for Tag entities.
checkTag :: Transaction ()
checkTag = (getDB (queryAll tagEntry)) |>>= (mapT_ checkTagEntry)

checkTaggingEntry :: Tagging -> Transaction ()
checkTaggingEntry tagging_p = (existsDBKey entryKey entryEntry (
    taggingEntryTaggingKey tagging_p)) |>> ((existsDBKey tagKey tagEntry (
    taggingTagTaggingKey tagging_p)) |>> (unique2C taggingEntryTaggingKey
    taggingTagTaggingKey taggingEntry (taggingEntryTaggingKey tagging_p) (
    taggingTagTaggingKey tagging_p)))

checkEntryEntry :: Entry -> Transaction ()
checkEntryEntry entry_p = (duplicateKeyTest entry) |>> (uniqueC entryTitle entry
    entry_p)

checkCommentEntry :: Comment -> Transaction ()
checkCommentEntry comment_p = (duplicateKeyTest comment) |>> (existsDBKey entryKey
     entryEntry (commentEntryCommentingKey comment_p))

checkTagEntry :: Tag -> Transaction ()
checkTagEntry tag_p = (duplicateKeyTest tag) |>> (uniqueC tagName tag tag_p)

--- Saves the complete database as Curry terms.
--- The first argument is the directory where the term files should be stored.
saveAllData :: String -> IO ()
saveAllData path = (saveDBTerms path "Tagging" taggingEntry) >> ((saveDBTerms path
     "Entry" entryEntry) >> ((saveDBTerms path "Comment" commentEntry) >> (
    saveDBTerms path "Tag" tagEntry)))

--- Restore the complete database from files containing Curry terms.
--- The first argument is the directory where the term files are stored.
restoreAllData :: String -> IO ()
restoreAllData path = (restoreDBTerms path "Tagging" taggingEntry) >> ((
    restoreDBTerms path "Entry" entryEntry) >> ((restoreDBTerms path "Comment"
    commentEntry) >> (restoreDBTerms path "Tag" tagEntry)))
```

B.2. Controller-Schicht

Listing B.3: Modul CommentController.curry

```
module CommentController(newCommentController, createCommentController,
    editCommentController, updateCommentController, deleteCommentController,
    listCommentController, getAllEntrys, getCommentingEntry) where

import Spicey
import Dynamic
import Time
import Blog
import CommentView
import Maybe

--- Shows a form to create a new Comment-entity.
newCommentController :: [String] -> IO ([HtmlExp])
newCommentController _ =
    do
     allEntrys <- getAllEntrys
     return (blankCommentView allEntrys createCommentController)

--- Persists a new Comment-entity to the database.
createCommentController :: (String,String,CalendarTime,Entry) -> IO ([HtmlExp])
createCommentController (text,author,date,entry) =
    do
     transResult <- runT (newCommentWithEntryCommentingKey text author date (
         entryKey entry))
     either (\_ -> listCommentController []) (\error -> displayError (showTError
         error) []) transResult

--- Shows a form to edit the given Comment-entity.
editCommentController :: Comment -> IO ([HtmlExp])
editCommentController commentToEdit =
    do
     allEntrys <- getAllEntrys
     commentingEntry <- runQ (getCommentingEntry commentToEdit)
     return (editCommentView (commentToEdit) (fromJust commentingEntry) allEntrys
         updateCommentController)

--- Persists modfications of a given Comment-entity to the database.
updateCommentController :: Comment -> IO ([HtmlExp])
updateCommentController comment =
    do
     transResult <- runT (updateComment comment)
     either (\_ -> listCommentController []) (\error -> displayError (showTError
         error) []) transResult

--- Deletes a given Comment-entity and calls the list-controller-function after
    that.
deleteCommentController :: Comment -> IO ([HtmlExp])
deleteCommentController comment =
    do
     transResult <- runT (deleteComment comment)
     either (\_ -> listCommentController []) (\error -> displayError (showTError
         error) []) transResult

--- Lists all Comment-entities with buttons to delete or edit an entity.
listCommentController :: [String] -> IO ([HtmlExp])
listCommentController _ =
    do
```

```
     comments <- runQ (queryAll (\c -> let
          key free
           in (comment key c)))
     return (listCommentView comments editCommentController
         deleteCommentController)

--- Gets all Entry entities.
getAllEntrys :: IO ([Entry])
getAllEntrys = runQ (queryAll (\e -> let
     key free
      in (entry key e)))

--- Gets the associated Entry-entity for a given Comment entity.
getCommentingEntry :: Comment -> Query (Maybe Entry)
getCommentingEntry cEntry = queryOne (\e -> let
     ckey free
     ekey free
      in ((comment ckey cEntry) <> ((entry ekey e) <> (isCommentedBy ekey ckey))))
```

Listing B.4: Modul EntryController.curry

```
module EntryController(newEntryController, createEntryController,
    editEntryController, updateEntryController, deleteEntryController,
    listEntryController, addTagging, removeTagging, getAllTags, getTaggingTags)
    where

import Spicey
import Dynamic
import Time
import Blog
import EntryView
import Maybe

--- Shows a form to create a new Entry-entity.
newEntryController :: [String] -> IO ([HtmlExp])
newEntryController _ =
    do
     allTags <- getAllTags
     return (blankEntryView allTags createEntryController)

--- Persists a new Entry-entity to the database.
createEntryController :: (String,String,String,CalendarTime,[Tag]) -> IO ([HtmlExp
    ])
createEntryController (title,text,author,date,tags) =
    do
     transResult <- runT ((newEntry title text author date) |>>= (addTagging tags)
         )
     either (\_ -> listEntryController []) (\error -> displayError (showTError
         error) []) transResult

--- Shows a form to edit the given Entry-entity.
editEntryController :: Entry -> IO ([HtmlExp])
editEntryController entryToEdit =
    do
     allTags <- getAllTags
     taggingTags <- runQ (getTaggingTags entryToEdit)
     return (editEntryView (entryToEdit,taggingTags) allTags updateEntryController
         )

--- Persists modfications of a given Entry-entity to the database.
updateEntryController :: (Entry,[Tag]) -> IO ([HtmlExp])
updateEntryController (entry,tagsTagging) =
```

```
    do
     transResult <- runT ((updateEntry entry) |>> (((getDB (getTaggingTags entry))
          |>>= (\oldTaggingTags -> removeTagging oldTaggingTags entry)) |>> (
         addTagging tagsTagging entry)))
     either (\_ -> listEntryController []) (\error -> displayError (showTError
         error) []) transResult

--- Deletes a given Entry-entity and calls the list-controller-function after that
    .
deleteEntryController :: Entry -> IO ([HtmlExp])
deleteEntryController entry =
    do
     transResult <- runT (((getDB (getTaggingTags entry)) |>>= (\oldTaggingTags ->
          removeTagging oldTaggingTags entry)) |>> (deleteEntry entry))
     either (\_ -> listEntryController []) (\error -> displayError (showTError
         error) []) transResult

--- Lists all Entry-entities with buttons to delete or edit an entity.
listEntryController :: [String] -> IO ([HtmlExp])
listEntryController _ =
    do
     entrys <- runQ (queryAll (\e -> let
          key free
           in (entry key e)))
     return (listEntryView entrys editEntryController deleteEntryController)

--- Associates given entities with the Entry-entity.
addTagging :: [Tag] -> Entry -> Transaction ()
addTagging tags entry = sequenceT_ (map (\t -> newTagging (entryKey entry) (tagKey
     t)) tags)

--- Removes association to the given entities with the Entry-entity.
removeTagging :: [Tag] -> Entry -> Transaction ()
removeTagging tags entry = sequenceT_ (map (\t -> deleteTagging (entryKey entry) (
    tagKey t)) tags)

--- Gets all Tag entities.
getAllTags :: IO ([Tag])
getAllTags = runQ (queryAll (\t -> let
     key free
      in (tag key t)))

--- Gets the associated Tag-entities for a given Entry entity.
getTaggingTags :: Entry -> Query ([Tag])
getTaggingTags eTag = queryAll (\t -> let
     ekey free
     tkey free
      in ((entry ekey eTag) <> ((tag tkey t) <> (tagging ekey tkey))))
```

Listing B.5: Modul TagController.curry

```
module TagController(newTagController, createTagController, editTagController,
    updateTagController, deleteTagController, listTagController) where

import Spicey
import Dynamic
import Time
import Blog
import TagView
import Maybe

--- Shows a form to create a new Tag-entity.
```

```
newTagController :: [String] -> IO ([HtmlExp])
newTagController _ =
    do
     return (blankTagView createTagController)

--- Persists a new Tag-entity to the database.
createTagController :: (String) -> IO ([HtmlExp])
createTagController (name) =
    do
     transResult <- runT (newTag name)
     either (\_ -> listTagController []) (\error -> displayError (showTError error
         ) []) transResult

--- Shows a form to edit the given Tag-entity.
editTagController :: Tag -> IO ([HtmlExp])
editTagController tagToEdit =
    do
     return (editTagView (tagToEdit) updateTagController)

--- Persists modfications of a given Tag-entity to the database.
updateTagController :: Tag -> IO ([HtmlExp])
updateTagController tag =
    do
     transResult <- runT (updateTag tag)
     either (\_ -> listTagController []) (\error -> displayError (showTError error
         ) []) transResult

--- Deletes a given Tag-entity and calls the list-controller-function after that.
deleteTagController :: Tag -> IO ([HtmlExp])
deleteTagController tag =
    do
     transResult <- runT (deleteTag tag)
     either (\_ -> listTagController []) (\error -> displayError (showTError error
         ) []) transResult

--- Lists all Tag-entities with buttons to delete or edit an entity.
listTagController :: [String] -> IO ([HtmlExp])
listTagController _ =
    do
     tags <- runQ (queryAll (\t -> let
          key free
           in (tag key t)))
     return (listTagView tags editTagController deleteTagController)
```

B.3. View-Schicht

Listing B.6: Modul CommentView.curry

```
module CommentView(wComment, tuple2Comment, comment2Tuple, wCommentType,
    createCommentView, editCommentView, blankCommentView, listCommentView) where

import WUI
import Time
import Spicey
import Blog
import BlogEntitiesToHtml
```

```
--- The WUI-specification for the Comment-entity-type. Includes fields for
    associated entities and labels for all fields.
wComment :: [Entry] -> WuiSpec ((String,String,CalendarTime,Entry))
wComment entryList = withRendering (w4Tuple wString wString wDateType (wSelect
    entryTitle entryList)) (renderLabels commentLabelList)

--- Transformation from data of a WUI-Form to Comment-entity-type
tuple2Comment :: Comment -> (String,String,CalendarTime,Entry) -> Comment
tuple2Comment commentToUpdate (text,author,date,entry) = (setCommentText (
    setCommentAuthor (setCommentDate (setCommentEntryCommentingKey commentToUpdate
     (entryKey entry)) date) author) text)

--- Transformation from Comment-entity-type to a tuple which can be used in WUI-
    Specifications
comment2Tuple :: Entry -> Comment -> (String,String,CalendarTime,Entry)
comment2Tuple entry (comment) = (commentText comment,commentAuthor comment,
    commentDate comment,entry)

--- WUI Type for editing or creating Comment entities. Includes fields for
    associated entities.
wCommentType :: Comment -> Entry -> [Entry] -> WuiSpec Comment
wCommentType comment entry entryList = transformWSpec (tuple2Comment comment,
    comment2Tuple entry) (wComment entryList)

--- Supplies a WUI-Form to create a new Comment entity. Takes default values to be
     prefilled in the form fields.
createCommentView :: String -> String -> CalendarTime -> Entry -> [Entry] -> ((
    String,String,CalendarTime,Entry) -> IO ([HtmlExp])) -> [HtmlExp]
createCommentView defaultText defaultAuthor defaultDate defaultEntry
    possibleEntrys controller = let
     (hexp,handler) = wui2html (wComment possibleEntrys) (defaultText,
         defaultAuthor,defaultDate,defaultEntry) (nextControllerForData controller
         )
      in [(h1 [(htxt "new Comment")]),hexp,(wuiHandler2button "create" handler)]

--- Supplies a WUI-Form to edit the given Comment entity. Takes also associated
    entities and a list of possible associations for every associated entity type.
editCommentView :: Comment -> Entry -> [Entry] -> (Comment -> IO ([HtmlExp])) -> [
    HtmlExp]
editCommentView comment relatedEntry possibleEntrys controller = let
     (hexp,handler) = wui2html (wCommentType comment relatedEntry possibleEntrys)
         comment (nextControllerForData controller)
      in [(h1 [(htxt "edit Comment")]),hexp,(wuiHandler2button "change" handler)]

--- Supplies a WUI-Form to create a new Comment entity. The presented fields are
    empty
blankCommentView :: [Entry] -> ((String,String,CalendarTime,Entry) -> IO ([HtmlExp
    ])) -> [HtmlExp]
blankCommentView possibleEntrys controller = createCommentView " " " " (
    CalendarTime 30 9 1981 0 0 0 0) (head possibleEntrys) possibleEntrys
    controller

--- Supplies a list view for a given list of Comment entities. Shows also buttons
    to delete or edit entries. Takes the controller-functions to delete and edit.
listCommentView :: [Comment] -> (Comment -> IO ([HtmlExp])) -> (Comment -> IO ([
    HtmlExp])) -> [HtmlExp]
listCommentView comments editCommentController deleteCommentController = [(h1 [(
    htxt "CommentList")]),(table ([(take 3 commentLabelList)] ++ (listComments
    comments)))]
   where
     listComments :: [Comment] -> [[[HtmlExp]]]
     listComments [] = []
```

```
    listComments (comment:commentList) = (((commentToListView comment) ++ [[(
        button "edit" (nextControllerWithoutRefs (editCommentController comment))
        ),(button "delete" (nextControllerWithoutRefs (deleteCommentController
        comment)))]]):(listComments commentList))
```

Listing B.7: Modul EntryView.curry

```
module EntryView(wEntry, tuple2Entry, entry2Tuple, wEntryType, createEntryView,
    editEntryView, blankEntryView, listEntryView) where

import WUI
import Time
import Spicey
import Blog
import BlogEntitiesToHtml

--- The WUI-specification for the Entry-entity-type. Includes fields for
    associated entities and labels for all fields.
wEntry :: [Tag] -> WuiSpec ((String,String,String,CalendarTime,[Tag]))
wEntry tagList = withRendering (w5Tuple wString wString wString wDateType (
    wMultiCheckSelect (\tag -> [(htxt (tagName tag))]) tagList)) (renderLabels
    entryLabelList)

--- Transformation from data of a WUI-Form to Entry-entity-type
tuple2Entry :: Entry -> (String,String,String,CalendarTime,[Tag]) -> (Entry,[Tag])
tuple2Entry entryToUpdate (title,text,author,date,tags) = (setEntryTitle (
    setEntryText (setEntryAuthor (setEntryDate entryToUpdate date) author) text)
    title,tags)

--- Transformation from Entry-entity-type to a tuple which can be used in WUI-
    Specifications
entry2Tuple :: (Entry,[Tag]) -> (String,String,String,CalendarTime,[Tag])
entry2Tuple (entry,tags) = (entryTitle entry,entryText entry,entryAuthor entry,
    entryDate entry,tags)

--- WUI Type for editing or creating Entry entities. Includes fields for
    associated entities.
wEntryType :: Entry -> [Tag] -> [Tag] -> WuiSpec ((Entry,[Tag]))
wEntryType entry tags tagList = transformWSpec (tuple2Entry entry,entry2Tuple) (
    wEntry tagList)

--- Supplies a WUI-Form to create a new Entry entity. Takes default values to be
    prefilled in the form fields.
createEntryView :: String -> String -> String -> CalendarTime -> [Tag] -> [Tag] ->
     ((String,String,String,CalendarTime,[Tag]) -> IO ([HtmlExp])) -> [HtmlExp]
createEntryView defaultTitle defaultText defaultAuthor defaultDate defaultTags
    possibleTags controller = let
     (hexp,handler) = wui2html (wEntry possibleTags) (defaultTitle,defaultText,
         defaultAuthor,defaultDate,defaultTags) (nextControllerForData controller)
      in [(h1 [(htxt "new Entry")]),hexp,(wuiHandler2button "create" handler)]

--- Supplies a WUI-Form to edit the given Entry entity. Takes also associated
    entities and a list of possible associations for every associated entity type.
editEntryView :: (Entry,[Tag]) -> [Tag] -> ((Entry,[Tag]) -> IO ([HtmlExp])) -> [
    HtmlExp]
editEntryView (entry,tags) possibleTags controller = let
     (hexp,handler) = wui2html (wEntryType entry tags possibleTags) (entry,tags) (
         nextControllerForData controller)
      in [(h1 [(htxt "edit Entry")]),hexp,(wuiHandler2button "change" handler)]

--- Supplies a WUI-Form to create a new Entry entity. The presented fields are
    empty
```

```
blankEntryView :: [Tag] -> ((String,String,String,CalendarTime,[Tag]) -> IO ([
    HtmlExp])) -> [HtmlExp]
blankEntryView possibleTags controller = createEntryView " " " " " " " (CalendarTime
     30 9 1981 0 0 0 0) [] possibleTags controller

--- Supplies a list view for a given list of Entry entities. Shows also buttons to
     delete or edit entries. Takes the controller-functions to delete and edit.
listEntryView :: [Entry] -> (Entry -> IO ([HtmlExp])) -> (Entry -> IO ([HtmlExp]))
     -> [HtmlExp]
listEntryView entrys editEntryController deleteEntryController = [(h1 [(htxt "
    EntryList")]),(table ([(take 4 entryLabelList)] ++ (listEntrys entrys)))]
   where
     listEntrys :: [Entry] -> [[[HtmlExp]]]
     listEntrys [] = []
     listEntrys (entry:entryList) = (((entryToListView entry) ++ [[(button "edit"
         (nextControllerWithoutRefs (editEntryController entry))),(button "delete"
          (nextControllerWithoutRefs (deleteEntryController entry)))]]):(
         listEntrys entryList))
```

Listing B.8: Modul TagView.curry

```
module TagView(wTag, tuple2Tag, tag2Tuple, wTagType, createTagView, editTagView,
    blankTagView, listTagView) where

import WUI
import Time
import Spicey
import Blog
import BlogEntitiesToHtml

--- The WUI-specification for the Tag-entity-type. Includes fields for associated
    entities and labels for all fields.
wTag :: WuiSpec ((String))
wTag = withRendering wString (renderLabels tagLabelList)

--- Transformation from data of a WUI-Form to Tag-entity-type
tuple2Tag :: Tag -> (String) -> Tag
tuple2Tag tagToUpdate (name) = (setTagName tagToUpdate name)

--- Transformation from Tag-entity-type to a tuple which can be used in WUI-
    Specifications
tag2Tuple :: Tag -> (String)
tag2Tuple (tag) = (tagName tag)

--- WUI Type for editing or creating Tag entities. Includes fields for associated
    entities.
wTagType :: Tag -> WuiSpec Tag
wTagType tag = transformWSpec (tuple2Tag tag,tag2Tuple) wTag

--- Supplies a WUI-Form to create a new Tag entity. Takes default values to be
    prefilled in the form fields.
createTagView :: String -> ((String) -> IO ([HtmlExp])) -> [HtmlExp]
createTagView defaultName controller = let
     (hexp,handler) = wui2html wTag (defaultName) (nextControllerForData
         controller)
      in [(h1 [(htxt "new Tag")]),hexp,(wuiHandler2button "create" handler)]

--- Supplies a WUI-Form to edit the given Tag entity. Takes also associated
    entities and a list of possible associations for every associated entity type.
editTagView :: Tag -> (Tag -> IO ([HtmlExp])) -> [HtmlExp]
editTagView tag controller = let
```

```
     (hexp,handler) = wui2html (wTagType tag) tag (nextControllerForData
         controller)
      in [(h1 [(htxt "edit Tag")]),hexp,(wuiHandler2button "change" handler)]

--- Supplies a WUI-Form to create a new Tag entity. The presented fields are empty
blankTagView :: ((String) -> IO ([HtmlExp])) -> [HtmlExp]
blankTagView controller = createTagView " " controller

--- Supplies a list view for a given list of Tag entities. Shows also buttons to
    delete or edit entries. Takes the controller-functions to delete and edit.
listTagView :: [Tag] -> (Tag -> IO ([HtmlExp])) -> (Tag -> IO ([HtmlExp])) -> [
    HtmlExp]
listTagView tags editTagController deleteTagController = [(h1 [(htxt "TagList")])
    ,(table ([(take 1 tagLabelList)] ++ (listTags tags)))]
   where
     listTags :: [Tag] -> [[[HtmlExp]]]
     listTags [] = []
     listTags (tag:tagList) = (((tagToListView tag) ++ [[(button "edit" (
         nextControllerWithoutRefs (editTagController tag))),(button "delete" (
         nextControllerWithoutRefs (deleteTagController tag)))]]):(listTags
         tagList))
```

Listing B.9: Modul BlogEntitiesToHtml.curry

```
module BlogEntitiesToHtml(entryToListView, entryToShortView, entryToDetailsView,
    entryLabelList, commentToListView, commentToShortView, commentToDetailsView,
    commentLabelList, tagToListView, tagToShortView, tagToDetailsView,
    tagLabelList) where

import WUI
import Time
import Spicey
import Blog

entryToListView :: Entry -> [[HtmlExp]]
entryToListView entry = [[(stringToHtml (entryTitle entry))],[(stringToHtml (
    entryText entry))],[(stringToHtml (entryAuthor entry))],[(calendarTimeToHtml (
    entryDate entry))]]

entryToShortView :: Entry -> String
entryToShortView entry = entryTitle entry

entryToDetailsView :: Entry -> [HtmlExp]
entryToDetailsView entry = [(table (map (\(label,value) -> [label,value]) (zip
    entryLabelList (entryToListView entry))))]

entryLabelList :: [[HtmlExp]]
entryLabelList = [[(textstyle "label label_for_type_string" "Title")],[(textstyle
    "label label_for_type_string" "Text")],[(textstyle "label
    label_for_type_string" "Author")],[(textstyle "label
    label_for_type_calendarTime" "Date")],[(textstyle "label
    label_for_type_relation" "Tag")]]

commentToListView :: Comment -> [[HtmlExp]]
commentToListView comment = [[(stringToHtml (commentText comment))],[(stringToHtml
     (commentAuthor comment))],[(calendarTimeToHtml (commentDate comment))]]

commentToShortView :: Comment -> String
commentToShortView comment = commentText comment

commentToDetailsView :: Comment -> [HtmlExp]
```

```
commentToDetailsView comment = [(table (map (\(label,value) -> [label,value]) (zip
     commentLabelList (commentToListView comment))))]

commentLabelList :: [[HtmlExp]]
commentLabelList = [[(textstyle "label label_for_type_string" "Text")],[(textstyle
     "label label_for_type_string" "Author")],[(textstyle "label
    label_for_type_calendarTime" "Date")],[(textstyle "label
    label_for_type_relation" "Entry")]]

tagToListView :: Tag -> [[HtmlExp]]
tagToListView tag = [[(stringToHtml (tagName tag))]]

tagToShortView :: Tag -> String
tagToShortView tag = tagName tag

tagToDetailsView :: Tag -> [HtmlExp]
tagToDetailsView tag = [(table (map (\(label,value) -> [label,value]) (zip
    tagLabelList (tagToListView tag))))]

tagLabelList :: [[HtmlExp]]
tagLabelList = [[(textstyle "label label_for_type_string" "Name")]]
```

B.4. Routes

Listing B.10: Modul ControllerMapping.curry

```
module ControllerMapping(getController) where

import Spicey
import Routes
import RoutesData
import EntryController
import CommentController
import TagController
import SpiceySystemController -- system

getController :: ControllerFunctionReference -> ControllerFunction
getController fktref = case fktref of
     ProcessListController -> processListController -- system
     NewEntryController -> newEntryController
     ListEntryController -> listEntryController
     NewCommentController -> newCommentController
     ListCommentController -> listCommentController
     NewTagController -> newTagController
     ListTagController -> listTagController
     _ -> displayError "getController: no mapping found"
```

Listing B.11: Modul RoutesData.curry

```
module RoutesData(ControllerFunctionReference(..), UrlMatch(..), routes) where



data ControllerFunctionReference
     = NewEntryController
     | ListEntryController
     | NewCommentController
```

```
     | ListCommentController
     | NewTagController
     | ListTagController
     | ProcessListController -- system

data UrlMatch
     = Exact String
     | Matcher (String -> Bool)
     | Always

routes :: [(String,UrlMatch,ControllerFunctionReference)]
routes = [
  ("Process List", Exact "spiceyProcesses", ProcessListController), -- system
  ("new Entry",Exact "newEntry",NewEntryController),
  ("list Entry",Exact "listEntry",ListEntryController),
  ("new Comment",Exact "newComment",NewCommentController),
  ("list Comment",Exact "listComment",ListCommentController),
  ("new Tag",Exact "newTag",NewTagController),
  ("list Tag",Exact "listTag",ListTagController),
  ("default",Always,ListEntryController)]
```

B.5. Systemmodule

Listing B.12: Modul Spicey.curry

```
module Spicey (
  module System,
  module HTML,
  module ReadNumeric,
  module Database,
  nextController, nextControllerWithoutRefs, nextControllerForData, getForm,
      wDateType, displayError,
  nextInProcessOrDefault,
  renderLabels,
  stringToHtml, intToHtml, calendarTimeToHtml
  ) where

-- general libs
import System
import HTML
import ReadNumeric
import Database
import WUI
import Time
import Session
import Routes


--------------------------- vvvv -- Framework functions -- vvvv
    --------------------------------

-- a viewable can be turned into a representation which can be displayed as
    interface
-- here: a representation of a html page
type Viewable = HtmlPage

type ViewBlock = [HtmlExp]
```

```
-- Controllerfunctions should contain all logic and their result should be a
    Viewable
type ControllerFunction = [String] -> IO ViewBlock

nextController :: ControllerFunction -> [CgiRef] -> HtmlHandler
nextController controller refs env = do
  view <- controller (map env refs)
  getForm view

nextControllerWithoutRefs :: IO ViewBlock -> HtmlHandler
nextControllerWithoutRefs controller _ = do
  view <- controller
  getForm view

-- for wuis
nextControllerForData :: (a -> IO ([HtmlExp])) -> a -> IO HtmlForm
nextControllerForData controller param =
  do
    view <- controller param
    getForm view

nextInProcessOrDefault :: IO [HtmlExp]
nextInProcessOrDefault = return [htxt ""] -- triggers redirect


-- returns html list of all controllers
{-
controllerIndex :: HtmlExp
controllerIndex =
  ulist (controllerLinkList routes)
  where
    controllerLinkList :: [Route] -> [[HtmlExp]]
    controllerLinkList ((url, _):restList) =
      ([(href ("?" ++ url) [(htxt url)])]:controllerLinkList restList)
    controllerLinkList [] =
      []
-}
wDate :: WuiSpec (Int, Int, Int)
wDate = wTriple (wSelectInt [1..31]) (wSelectInt [1..12]) (wSelectInt
    [1900..2100])

tuple2date :: (Int, Int, Int) -> CalendarTime
tuple2date (day, month, year) = CalendarTime year month day 0 0 0 0

wDateType :: WuiSpec CalendarTime
wDateType = (adaptWSpec tuple2date) wDate

dummyHandler :: (CgiRef -> String) -> IO HtmlForm
dummyHandler _ = getForm []

addLayout :: ViewBlock -> ViewBlock
addLayout viewblock =
  [blockstyle "header" [h1 [htxt "Spicey Application"]], routeMenu] ++ viewblock
      ++ [blockstyle "footer" [par [htxt "powered by", image "images/spicey-logo.
      png" "Spicey", htxt "Framework"]]]

getForm :: ViewBlock -> IO HtmlForm
getForm viewBlock =
  case viewBlock of
    [HtmlText ""] -> return (HtmlAnswer "text/html" "<html><head><meta http-equiv
        =\"refresh\" content=\"3; url=/cgi-bin/spicey.cgi\"></head><body><p>Sie
        werden weitergeleitet...</p></body></html>")
```

```
    _ -> do
      cookie <- sessionCookie
      sid <- getSessionId
      lastUrl <- getLastUrl
      return (HtmlForm "form" [cookie, FormCSS "css/style.css"] (addLayout ([
          blockstyle "debug" [par [htxt ("User: "++(show sid)), breakline, htxt ("
          last page: "++lastUrl)]]] ++ viewBlock)))

-- dummy-controller to display an error
displayError :: String -> ControllerFunction
displayError msg _ =
  if (null msg) then
    return [htxt "general error, shown by function displayError in spicey.curry"]
  else
    return [htxt msg]

-- like renderTaggedTuple from WUI Library but takes list of HtmlExp instead of
    list of strings
renderLabels :: [[HtmlExp]] -> Rendering
renderLabels labels hexps =
 table (map (\(l, h) -> [l, [h]]) (zip labels hexps))

-- convert standard-datatype-values to html representation
stringToHtml :: String -> HtmlExp
stringToHtml s = textstyle "type_string" s

intToHtml :: Int -> HtmlExp
intToHtml i = textstyle "type_int" (show i)

calendarTimeToHtml :: CalendarTime -> HtmlExp
calendarTimeToHtml ct = textstyle "type_calendartime" (toDayString ct)
------------------
```

Listing B.13: Modul Routes.curry

```
module Routes(
  getControllerReference,
  linkToController,
  routeMenu
) where

import HTML
import RoutesData

type Route = (String, UrlMatch, ControllerFunctionReference)

getControllerReference :: String -> ControllerFunctionReference
getControllerReference url =
  findControllerReference routes url
  where
    findControllerReference :: [Route] -> String -> ControllerFunctionReference
    findControllerReference ((name, matcher, fktref):restroutes) url =
      case matcher of
        Exact string ->
          if (url == string) then
            fktref
          else
            findControllerReference restroutes url
        Matcher fkt ->
          if (fkt url) then
            fktref
          else
```

```
            findControllerReference restroutes url
        Always -> fktref
    findControllerReference [] url = error ("findControllerReference: No reference
         found for " ++ url)

-- returns link to controller, can only find urls of type Exact
linkToController :: ControllerFunctionReference -> [HtmlExp] -> HtmlExp
linkToController controllerRef text = href ("?" ++ getRoute routes controllerRef)
    text
  where
    getRoute :: [Route] -> ControllerFunctionReference -> String
    getRoute ((name, matcher, fktref):restroutes) refToSearchFor =
      case matcher of
        Exact string ->
          if (fktref == refToSearchFor) then
            string
          else
            getRoute restroutes refToSearchFor
        _ -> getRoute restroutes refToSearchFor
    getRoute [] refToSearchFor = error ("getRoute: No url found for " ++ (show
        refToSearchFor))

-- generates menu for all route entries of type Exact
routeMenu :: HtmlExp
routeMenu = ulist (getLinks routes)
  where
    getLinks :: [Route] -> [[HtmlExp]]
    getLinks ((name, matcher, fktref):restroutes) =
      case matcher of
        Exact string ->
          [(href ("?" ++ string) [htxt name])]:(getLinks restroutes)
        _ -> getLinks restroutes
    getLinks [] = []
```

Listing B.14: Modul Session.curry

```
module Session (
  SessionId,
  getSessionId,
  sessionCookie,
  saveLastUrl,
  getLastUrl,
  -- exported for module Processes, don't use elsewhere:
  getDataFromCurrentSession,
  putDataIntoCurrentSession,
  dummySessionId
  ) where

import HTML
import Time
import Random
import Global
import System
import List

data SessionId = SessionId String

type SessionStore a = [(SessionId, Int, a)]

sessionLifespan = 60 -- minutes

-- saves time and last id
```

```
lastId :: Global (Int, Int)
lastId = global (0, 0) (Persistent sessionCookieName)

getUnusedId :: IO SessionId
getUnusedId = do
  last <- readGlobal lastId
  clockTime <- getClockTime
  if (clockTimeToInt clockTime /= (fst last))
    then writeGlobal lastId (clockTimeToInt clockTime, 0)
    else writeGlobal lastId (clockTimeToInt clockTime, (snd last)+1)
  return (SessionId (show (clockTimeToInt clockTime) ++ (show $ (snd last) + 1)))

sessionCookieName = "spiceySessionId"

getSessionId :: IO SessionId
getSessionId = do
    cookies <- getCookies
    case (lookup sessionCookieName cookies) of
      (Just sessionCookieValue) -> return (SessionId sessionCookieValue)
      Nothing -> getUnusedId

getId :: SessionId -> String
getId (SessionId i) = i

dummySessionId = SessionId "dummy" -- used by Processes.curry

sessionCookie :: IO FormParam
sessionCookie = do
    sessionId <- getSessionId
    clockTime <- getClockTime
    return (FormCookie sessionCookieName (getId (sessionId)) [CookiePath "/",
        CookieExpire (addMinutes sessionLifespan clockTime)])


getDataFromCurrentSession :: Global (SessionStore a) -> IO (Maybe a)
getDataFromCurrentSession sessionData = do
    sid <- getSessionId
    sdata <- readGlobal sessionData
    return (findInSession sid sdata)
  where
    findInSession :: SessionId -> (SessionStore a) -> (Maybe a)
    findInSession si ((id, _, storedData):rest) =
      if ((getId id) == (getId si)) then
        (Just storedData)
      else
        findInSession si rest
    findInSession _ [] = Nothing



putDataIntoCurrentSession :: a -> Global (SessionStore a) -> IO ()
putDataIntoCurrentSession newData sessionData = do
  sid <- getSessionId
  sdata <- readGlobal sessionData
  currentTime <- getClockTime
  case findIndex (\(id, _, _) -> id == sid) sdata of
    (Just i) ->
      writeGlobal sessionData (replace (sid, clockTimeToInt currentTime, newData)
          i (cleanup currentTime sdata))
    Nothing ->
      writeGlobal sessionData ((sid, clockTimeToInt currentTime, newData):(cleanup
           currentTime sdata))
```

```
-- expects that clockTimeToInt converts time into ascending integers!
-- we should write our own conversion-function
cleanup :: ClockTime -> SessionStore a -> SessionStore a
cleanup currentTime sessionData =
  filter (\(_, time, _) -> (time > (clockTimeToInt (addMinutes (0 -
      sessionLifespan) currentTime)))) sessionData

-- example
lastUrl :: Global (SessionStore String)
lastUrl = global [] (Temporary)

getLastUrl :: IO String
getLastUrl = do
  maybeUrl <- getDataFromCurrentSession lastUrl
  case maybeUrl of
    (Just url) -> return url
    Nothing -> return "Unknown"

saveLastUrl :: String -> IO ()
saveLastUrl url = putDataIntoCurrentSession url lastUrl
```

Listing B.15: Modul Processes.curry

```
import Global
import Maybe

import Routes
import RoutesData
import Session

data State =
  StartState StateId ControllerFunctionReference |
  State StateId ControllerFunctionReference |
  EndState StateId ControllerFunctionReference

type TransitionGuard = (String -> Bool)
type StateId = String

data Transition =
  Transition StateId StateId TransitionGuard

data Process = Process String State [State] [Transition]

-- example process:

testProcess = Process "Erstellung eines Eintrags und Kommentaren"
  (StartState new_entry NewEntryController)
  [State new_comment NewCommentController,
   EndState end ListCommentController]
  (
    [ Transition new_entry new_comment (always) ] ++
    decision new_comment [(end, on_finished), (new_comment, on_another)]
  )
  where
    new_entry = "new_entry"
    new_comment = "new_comment"
    end = "end"


decision :: StateId -> [(StateId, TransitionGuard)] -> [Transition]
decision startState targetStates =
```

```
  map (\(targetState, guard) -> Transition startState targetState guard)
      targetStates

always :: TransitionGuard
always _ = True

on_success :: TransitionGuard
on_success value = (value == "ok")

on_error :: TransitionGuard
on_error value = (value == "error")

on_finished :: TransitionGuard
on_finished value = (value == "finished")

on_another :: TransitionGuard
on_another value = (value == "another")

data CurrentProcess = CProcess String
data CurrentState = CStartState | CState StateId

type ProcessState = (CurrentProcess, CurrentState)

availableProcesses :: [Process]
availableProcesses = [testProcess]

initialProcessState :: ProcessState
initialProcessState = (CProcess "", CStartState)

currentProcess :: Global (SessionStore ProcessState)
currentProcess = global [(dummySessionId, 0, initialProcessState)] (Persistent "
    spicey_current_process")

getCurrentProcess :: IO ProcessState
getCurrentProcess = do
  curProc <- getDataFromCurrentSession currentProcess
  case curProc of
    (Just procInfo) -> return procInfo
    Nothing -> return initialProcessState

saveCurrentProcess :: ProcessState -> IO ()
saveCurrentProcess proc =
  putDataIntoCurrentSession proc currentProcess

findState :: StateId -> [State] -> State
findState id (state:rest) =
  case state of
    (State sid ref) -> if (id == sid) then (State sid ref) else findState id rest
    (EndState sid ref) -> if (id == sid) then (State sid ref) else findState id
        rest
    _ -> findState id rest
{- the following would be the nice version but the compiler doesn't like it
    st@(State sid _)    | id == sid -> st
    es@(EndState sid _) | id == sid -> es
    _ -> findState id rest
-}
findState _ [] = error $ "findState: State not found for " ++ show id

findProcess :: String -> [Process] -> (Maybe Process)
findProcess processName (p@(Process name _ _ _):rest) =
  if (name == processName)
    then (Just p)
```

```
    else findProcess processName rest
findProcess _ [] = Nothing

findTransition :: String -> StateId -> [Transition] -> (Maybe Transition)
findTransition outcome stateId (trans:rest) =
  case trans of
    t@(Transition fromStateId _ guard) ->
      if ((fromStateId == stateId) && (guard outcome)) then (Just t) else
          findTransition outcome stateId rest
    _ -> findTransition outcome stateId rest
findTransition _ _ [] = Nothing

findTransitionFromStart :: State -> String -> [Transition] -> (Maybe Transition)
findTransitionFromStart startState outcome (trans:rest) =
  case trans of
    t@(Transition transStartState _ guard) ->
      if (((getStateId startState) == transStartState) && (guard outcome)) then (
          Just t) else findTransitionFromStart startState outcome rest
    _ -> findTransitionFromStart startState outcome rest
findTransitionFromStart _ _ [] = Nothing

getStateId :: State -> StateId
getStateId (StartState stateId _) = stateId
getStateId (State stateId _) = stateId
getStateId (EndState stateId _) = stateId

getStates :: Process -> [State]
getStates (Process _ _ states _) = states

getStartState :: Process -> State
getStartState (Process _ start _ _) = start

getTransitions :: Process -> [Transition]
getTransitions (Process _ _ _ trans) = trans

getControllerReferenceFromState :: State -> ControllerFunctionReference
getControllerReferenceFromState (StartState _ ref) = ref
getControllerReferenceFromState (State _ ref) = ref
getControllerReferenceFromState (EndState _ ref) = ref

getCurrentState :: IO (Maybe State)
getCurrentState = do
  (CProcess curProcName, curState) <- getCurrentProcess
  case (findProcess curProcName availableProcesses) of
    (Just process) ->
      case curState of
        (CStartState) -> return (Just (getStartState process))
        (CState curStateId) -> return (Just (findState curStateId ((getStartState
            process):(getStates process))))
    Nothing -> return Nothing

startProcess :: String -> IO ()
startProcess processName =
  let
    maybeProcess = findProcess processName availableProcesses
  in
    if (maybeProcess == Nothing)
      then error $ "startProcess: process not found: " ++ processName
      else saveCurrentProcess (CProcess processName, CStartState)

advanceInProcess :: String -> IO ()
advanceInProcess outcome = do
```

```
    (CProcess curProcName, curState) <- getCurrentProcess
    case (findProcess curProcName availableProcesses) of
      (Just process) ->
        case curState of
          (CStartState) -> if ((findTransitionFromStart (getStartState process)
              outcome (getTransitions process)) /= Nothing)
            then saveCurrentProcess (CProcess (pName process), CState (tid (
                fromJust (findTransitionFromStart (getStartState process) outcome
                (getTransitions process)))))
            else done
          (CState sid) -> if ((findTransition outcome sid (getTransitions process)
              ) /= Nothing)
            then saveCurrentProcess (CProcess (pName process), CState (tid (
                fromJust (findTransition outcome sid (getTransitions process)))))
            else done
      Nothing -> done -- no active process, do nothing
  where
    pName (Process name _ _ _) = name
    tid (Transition _ toStateId _) = toStateId

processCompleted = advanceInProcess

nextControllerRefInProcessOrForUrl :: String -> IO ControllerFunctionReference
nextControllerRefInProcessOrForUrl url = do
  (CProcess curProcName, curState) <- getCurrentProcess
  case (findProcess curProcName availableProcesses) of
    (Just process) ->
      case curState of
        (CStartState) -> return (getControllerReferenceFromState (getStartState
            process))
        (CState sid) -> return (getControllerReferenceFromState (findState sid (
            getStates process)))
    Nothing -> return $ getControllerReference url
```

Listing B.16: Eintrittspunkt der Anwendung main.curry

```
import ControllerMapping
import Spicey
import WUI
import HTML
import Routes
import RoutesData

import Processes
import Session

splitUrl :: String -> [String]
splitUrl url =
  let
    (ys,zs) = break (== '/') url
  in
    if null zs then
      [ys]
    else
      ys : splitUrl (tail zs)

parseUrl :: String -> (String, [String])
parseUrl url =
  let
    list = splitUrl url
  in
    (head list, tail list)
```

```
dispatcher :: IO (HtmlForm)
dispatcher =
  do
    -- get url
    urlParam <- getUrlParameter -- alternative: getEnviron "QUERY_STRING"

    --controller <- (getController (getControllerReference (fst (parseUrl urlParam
        )))) (snd (parseUrl urlParam))

    -- with process definitions:
    controllerRef <- nextControllerRefInProcessOrForUrl (fst (parseUrl urlParam))
    controller <- getController controllerRef (snd (parseUrl urlParam))

    form <- getForm controller
    saveLastUrl urlParam
    return form

main = dispatcher
```