FernUniversität in Hagen
Fakultät für Mathematik und Informatik

Masterarbeit

zur Erlangung des akademischen Grades

Master of Computer Science

Modellierung und Vergleich von Implementierungen einer REST-Anwendung in verschiedenen Sprachen

Vorgelegt von: Guido Kaiser
Matrikel-Nr.: 7829531
Erster Prüfer: Prof. Dr. Bernd Krämer
Betreuerin: Dipl.-Inf. Silvia Schreier
Lehrgebiet: Datenverarbeitungstechnik
Beginn der Arbeit: 01.09.2010
Abgabe der Arbeit: 08.01.2011

Ich erkläre hiermit, die folgende Masterarbeit selbständig verfasst zu haben. Andere als die angegebenen Quellen und Hilfsmittel habe ich nicht benutzt. Wörtliche und sinngemäße Zitate sind kenntlich gemacht.

Apensen, den 08.01.2011

Guido Kaiser

Widmung

Für meine Frau Katrin und meine frisch geborene Tochter Tarja.

„Ja, ich habe Karriere gemacht.
Aber neben meiner Familie erscheint sie mir unbedeutend.“

Lee Iacocca - Topmanager

Zusammenfassung

Der Architekturstil Representational State Transfer (REST) gewinnt bei der Verwendung von Webservices immer mehr an Bedeutung. Dabei fehlt es noch an einheitlichen Methoden, Techniken und Praktiken für den Entwurf und der Implementierung einer REST-Schnittstelle. Es existieren bereits verschiedene Frameworks für unterschiedliche Programmiersprachen, die eine aktive Unterstützung für die Entwicklung einer *RESTful*, also einer zu REST konformen, Schnittstelle bieten. Der Architekturstil definiert einige Prinzipien, auf denen auch das Hypertext Transfer Protocol (HTTP) basiert, um verteilte Anwendungen zu entwickeln.

Diese Masterarbeit befasst sich mit der Verwendbarkeit von verschiedenen Programmiersprachen, Technologien und der Kombination verschiedener Frameworks, um eine REST-Schnittstelle zu implementieren. Zum Vergleich werden die Sprachen Java, PHP und C# ausgewählt, in denen jeweils ein Client und ein Server implementiert werden. Das System sollte in Anlehnung an Flickr[1] und Picasa[2] ein Web-Fotoalbum realisieren. Im Verlauf der Arbeit werden verschiedene Frameworks vorgestellt, gegenübergestellt und für die Umsetzung der Anwendung verwendet. Abschließend werden die verwendeten Frameworks und die unterschiedlichen Implementierungen miteinander verglichen.

Diese Arbeit zeigt, dass es gute Frameworks gibt, mit denen es einfach ist, einen *RESTful*-Webservice zu entwickeln. Grundsätzlich ähneln sich die größeren Frameworks vom Funktionsumfang und von der Art und Weise der Realisierung. Manche Prinzipien, wie beispielsweise Hypermedia, werden bisher von den vorhandenen Frameworks nicht aktiv unterstützt. Dies kann allerdings durch den Entwickler mit Werkzeugen der Frameworks ausgeglichen werden. Die Vergleiche, der hier durchgeführten Implementierungen, haben gezeigt, dass die Frameworks sich bei der Generierung von HTTP-Status-Codes und in der Unterstützung der Repräsentationsformate unterscheiden. Zum Teil ist es sogar notwendig direkt in das Framework einzugreifen, um eine identische Repräsentation einer Ressource zu erhalten.

Der Vergleich, des in dieser Ausarbeitung entwickeltem Application Programming Interface (API) mit denen ähnlicher Projekte zeigt, dass es APIs gibt, die auf REST basieren sollen, aber nicht *RESTful* sind. Die Ursache hierfür ist, dass sie gegen die Prinzipien verstoßen oder beim Entwurf der Ressourcen nicht nach der Resource-oriented Architecture (ROA) vorgehen, sondern eher klassisch nach einer Service-oriented Architecture (SOA), wie beispielsweise bei einer Remote Procedure Call (RPC) Schnittstelle.

[1] `http://www.flickr.com`
[2] `http://picasaweb.google.com`

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Ich denke, dass es einen Weltmarkt
für vielleicht fünf Computer gibt.
Thomas Watson - IBM, 1943

1 Einleitung

Representational State Transfer (REST) und Resource-oriented Architecture (ROA) gewinnen heutzutage immer mehr an Bedeutung [vergleiche Rodriguez, 2008]. Bei REST handelt es sich um einen Architekturstil, der im Jahre 2000 von Fielding [2000] in seiner Dissertation vorgeschlagen wurde. Die wohl bekannteste Implementierung von REST ist Hypertext Transfer Protocol (HTTP) [siehe Fielding u. a., 1999]. Dieser Architekturstil ist allerdings nicht auf HTTP beschränkt, da er so abstrakt gehalten ist, dass er auch mit anderen Protokollfamilien angewendet werden kann.

REST basiert auf verschiedenen Prinzipien, die von einer REST-Anwendung eingehalten werden sollten, damit sie als *RESTful* bezeichnet werden kann. Diese Prinzipien sind laut Fielding [2000] die folgenden:

Über URIs ansprechbare Ressourcen Alle Ressourcen werden eindeutig über einen Mechanismus adressiert, wobei eine Ressource alles ist, was eindeutig über solch einen Mechanismus adressiert werden kann.

Verknüpfte Ressourcen (Hypermedia) Ressourcen sind miteinander verknüpft, so dass ein Anwender, oder eine andere Anwendung, sich von einer Ressource zur nächsten „hangeln“ kann.

Verwendung von Standard HTTP-Methoden Jede Ressource besitzt dieselben uniformen Methoden, die durch das Protokoll spezifiziert werden, wie beispielsweise im Falle von HTTP die beiden bekannten Methoden `GET` und `POST`.

Unterschiedliche Repräsentationen von Ressourcen Durch verschiedene Repräsentationen kann dieselbe Ressource unterschiedlich verwendet werden. Ein Anwender kann beispielsweise eine Repräsentation in Hypertext Markup Language (HTML) [siehe Connolly und Masinter, 2000] erhalten, die für ihn leicht lesbar ist. Eine Anwendung hingegen könnte eine Repräsentation als Comma-Separated Values (CSV) erhalten, um diese automatisiert zu verarbeiten.

Statuslose Kommunikation Der Anwendungszustand wird nicht im Webservice gespeichert. Hierdurch ist es leicht möglich die Anwendung über mehrere Server zu verteilen, da dieser nicht verteilt werden muss. Jede Anfrage soll daher alle notwendigen Informationen zur Abarbeitung der Anforderung enthalten.

1.1. Motivation

Es entstehen immer mehr REST-Frameworks für verschiedene Sprachen, wobei viele nur wenig Unterstützung für die Prinzipien von REST bieten (vergleiche Kapitel 2). Die Art und Weise der Unterstützung der REST-Prinzipien variiert zwischen den Frameworks bzw. zwischen den verschiedenen Sprachen (vergleiche Kapitel 6).

Erfahrungen und Vergleiche zwischen den verschiedenen Frameworks und Technologien hinsichtlich der Konformität der REST-Schnittstelle existieren bisher nur in geringem Umfang. Die Schnittstellen können sich in verschiedenen Punkten unterscheiden, wie beispielsweise bei der Unterstützung der Struktur eines Uniform Resource Identifier (URI) [siehe Berners-Lee u. a., 2005] für eine oder mehrere Ressourcen. Durch die aus dieser Ausarbeitung gewonnen Erfahrungen sollen zukünftige REST-Anwendungen mit einer höheren Qualität entwickelt werden können. Dies kann mithilfe von Vergleichen, Methodiken und Best Practices gewährleistet werden.

Da REST auf dem Prinzip des Webs basiert, welches bereits seit vielen Jahren erprobt und ausgereift ist, bietet es sich für die Kommunikation zwischen Web-Komponenten an. Zusätzlich bietet es weitere Vorteile, die für den Einsatz von REST sprechen. Dazu gehören unter anderem die Einfachheit und der geringe Mehraufwand (engl. Overhead) gegenüber anderer Technologien, wie beispielsweise Simple Object Access Protocol (SOAP) [siehe Gudgin u. a., 2007]. Die Vergleiche und Implementierungen einer *RESTful*-Anwendung in verschiedenen Sprachen und verschiedenen Frameworks soll dabei helfen, einheitliche Vorgehensweisen für den Entwurf und die Implementierung einer solchen Anwendung zu finden. In das zur Zeit an der *FernUniversität in Hagen* am Lehrgebiet *Datenverarbeitungstechnik* entstehende Metamodell für REST [siehe Schreier, 2010] sollen die in dieser Abschlussarbeit gewonnenen Erkenntnisse einfließen.

Diese Arbeit untersucht verschiedene Frameworks und deren Unterstützung für REST. Es wurden gängige Frameworks für unterschiedliche Programmiersprachen recherchiert und in Bezug auf deren Potenzial für die Implementierung einer *RESTful*-Anwendung miteinander verglichen. Für jede Programmiersprache wurde ein Framework für die Implementierungen gewählt.

Abschließendes Ziel ist es, eine *RESTful*-Anwendung in verschiedenen Programmiersprachen unter der Verwendung von verschiedenen Frameworks zu implementieren, die einander gegenübergestellt werden, um Erfahrungen der Benutzung und der Konformität der REST-Frameworks zu erhalten. Dafür werden die verschiedenen Implementierungen und die verwendeten Frameworks miteinander verglichen. Dabei ist ein Schwerpunkt die Unterstützung der Prinzipien von REST.

1.2. Aufbau der Arbeit

In Kapitel 2 wird der Architekturstil REST vorgestellt, der als Grundlage für diese Ausarbeitung dient. Prinzipien, auf denen dieser aufbaut, werden erläutert und durch ein kurzes Beispiel abgerundet. Das Ziel und die Vorgehensweise dieser Ausarbeitung wird in Kapitel 3 aufgezeigt. Dort werden unter anderem die Anforderungen an die dabei entstehende Anwendung definiert.

Eine Auswahl von umfangreicheren Frameworks für REST wird nach Sprachen gruppiert im Kapitel 4 vorgestellt. Für jede Sprache wird anschließend ein Framework für die Implementierung gewählt. Kapitel 5 befasst sich mit dem Entwurf und der Implementierung der Anwendung. Dort werden Architekturen, Schnittstellen, wichtige Entwurfsentscheidungen und Tests aufgezeigt. Das Kapitel 6 dient der Gegenüberstellung der verwendeten Frameworks in Bezug auf die Verwendbarkeit, Konformität und Kompatibilität zueinander. Abschließend wird in Kapitel 7 ein Fazit gezogen und ein Ausblick aufgezeigt.

Der Anhang beinhaltet ergänzende Informationen zu bestimmten Überlegungen, Entscheidungen und Implementierungen, die für das Ziel dieser Ausarbeitung hinausgehen, aber zum Verständnis beitragen können. Weiterhin wird im Anhang etwas detaillierter auf die Verwendung der eingesetzten Frameworks eingegangen. Im Anhang A werden weitere Frameworks präsentiert, die für diese Ausarbeitung weniger interessant waren, aber für eine vollständige Übersicht mit aufgenommen wurden. Weitere Entwurfsentscheidungen finden sich im Anhang B. Der Anhang C zeigt, wie die für diese Ausarbeitung relevanten Frameworks eingesetzt werden können. Wie die Schnittstelle im Detail getestet wurde, wird im Anhang D dargestellt. Zuletzt wird im Anhang E der Inhalt der beigefügten CD aufgeführt.

Internet ist nur ein Hype.
Bill Gates - Microsoft, 1995

2 Grundlagen

Nach einem kurzen Überblick wird dieser Abschnitt die notwendigen Grundlagen für diese Ausarbeitung liefern. Vermittelt werden zunächst wichtige Grundbegriffe und ein Überblick über REST. Dort wird zunächst erklärt was REST ist, wie es funktioniert und wie es im Vergleich zu anderen Technologien steht. Abgerundet wird das Kapitel mit einem kurzen Beispiel.

2.1. Grundbegriffe

Dieser Abschnitt beinhaltet eine Kurzeinführung der Fachbegriffe, die für das Verständnis dieser Ausarbeitung wichtig sind. Hier werden gemeinsame Definitionen festgelegt, um Missverständnissen vorzubeugen. Wenn in späteren Abschnitten der Ausarbeitung einer der folgenden Begriffe erscheint, so ist stets die hier zu findende Definition gemeint.

Webanwendung Bei einer Webanwendung [siehe Lorenz und Six, 2009; Vonhoegen, 2004] interagiert ein Benutzer mittels eines Browsers auf einer entfernten Applikation, ohne diese installieren zu müssen. Es handelt sich daher um eine Client-/Server-Anwendung, die in der Regel mittels HTML-Seiten die Benutzerschnittstelle realisiert. Sie unterscheidet sich in den folgenden Punkten von einer klassischen Anwendung:

- **Beliebige Programmabläufe** Durch manuelle Eingabe eines URI ist es möglich an beliebige Stellen des Programmablaufes zu springen. Dies ermöglicht zudem ein Zurückspringen zu vorherigen Seiten.

- **Erhöhter Kommunikationsaufwand** Durch Übertragungen zwischen Client und Server entsteht eine erhöhte Kommunikation.

- **Geringerer Aufwand für Installation und Wartung** Installationen und Wartungen sind einfacher, da diese nur zentral durchgeführt werden müssen. Daraus folgt, dass keine Versionsstände unterschieden werden müssen.

- **Erhöhte Portabilität** Erhöhte Portabilität wird dadurch erreicht, dass bereits beliebige Browser auf verschiedenen Plattformen existieren.

Durch diese Unterschiede ergeben sich unterschiedliche Anwendungsgebiete zwischen klassischen Anwendungen und Webanwendungen.

Uniform Resource Identifier (URI) Ein weltweit gültiger und eindeutiger Bezeichner für eine Ressource. Die im Internet am häufigsten anzutreffende Variante eines Uniform Resource Identifier (URI) ist der Uniform Resource Locator (URL), die eine Ressource über eine Ortsangabe spezifiziert [siehe Berners-Lee u. a., 2005; Berners-Lee, 1994].

Ressource Eine Ressource ist alles, was eindeutig durch einen URI identifiziert werden kann. Dies kann also beispielsweise eine Zeile in einer Datenbank oder das Ergebnis eines Algorithmus bzw. einer Berechnung sein [vergleiche Richardson und Ruby, 2007].

Hypertext Transfer Protocol (HTTP) Protokoll zum Übertragen von Daten in einem Netzwerk durch eine Anfrage (engl. Request) und eine zugehörige Antwort (engl. Response) [siehe Fielding u. a., 1999]. Es basiert auf TCP/IP, wobei das Transmission Control Protocol (TCP) dafür sorgt, dass Daten vollständig und ohne Seiteneffekte übertragen werden können. Das Internet Protocol (IP) sorgt dafür, dass die einzelnen Pakete, in welche die Daten zerlegt werden, beim richtigen Ziel ankommen [siehe Cerf u. a., 1974].

Remote Procedure Call (RPC) Mithilfe eines Remote Procedure Call (RPC) können Methoden von Programmen aufgerufen werden, die sich entweder auf anderen oder demselben Rechner befinden können. Hierfür wird in der Regel über das Netzwerk eine Information von einem Programm zu einem anderen geschickt. Der Erhalt einer Antwort kann entweder synchron oder asynchron erfolgen [vergleiche Birrel und Nelson, 1983; White, 1976]. Es existieren verschiedene Technologien, die dieses Vorgehen ermöglichen, wie beispielsweise die Common Object Request Broker Architecture (CORBA) [siehe Object Management Group, 2004] und die Java Remote Method Invocation (RMI) [siehe Oracle].

Extensionable Markup Language (XML) Bei Extensionable Markup Language (XML) [siehe Bray u. a., 2008] handelt es sich um eine Metasprache, die sich besonders zur Beschreibung hierarchischer Strukturen eignet. Vorteile sind die Trennung von Struktur und Inhalt, sowie Plattformunabhängigkeit. XML-Dateien sind baumartig aufgebaut. Ein Dokument, welches die Syntaxregeln von XML einhält wird als *wohlgeformt* bezeichnet. Ein wohlgeformtes Dokument, dessen Struktur den Regeln einer angegebenen Document Type Definition (DTD) [siehe Bray u. a., 2008] oder einer XML Schema Definition (XSD) [siehe Thompson u. a., 2004] entspricht, wird als *gültig* bezeichnet [siehe Bray u. a., 2008].

XML-Dateien können mittels der so genannten Extensible Stylesheet Language Transformation (XSLT) [siehe Clark, 1999] automatisiert in andere Datenformate umgewandelt werden, da die entsprechenden Abbildungsregeln in XSLT-Dateien definiert werden. Nach Tilkov [2009a] ist XML das vermutlich am weitesten verbreitete Repräsentationsformat.

Framework Bei einem Framework handelt es sich um ein Gerüst einer Applikation, welches von einem Entwickler an die jeweilige Situation angepasst werden kann. Im Gegensatz zu einer Klassenbibliothek legt ein Framework die Architektur der Applikation fest und die Klassen des Frameworks werden nicht von der Applikation genutzt, sondern umgekehrt. Daher wird von der *Inversion of Control* gesprochen. Ein Framework ist immer auf einen speziellen Einsatzbereich ausgerichtet. Ein geeignetes Framework verringert den Entwurfs- und Implementierungsaufwand erheblich und trägt außerdem den Applikationen auch eine identische Architektur auf [vergleiche Pree, 1997; Schneider, 2010].

2.2. Representational State Transfer (REST)

Bei REST handelt es sich um einen abstrakten Architekturstil für verteilte Anwendungen, der chronologisch gesehen auf den Kernprinzipien des HTTP basiert [vergleiche Fielding, 2000]. Durch REST wird eine leichtgewichtige Alternative gegenüber dem SOAP [siehe Gudgin u. a., 2007] und der Web Services Description Language (WSDL) [siehe Christensen u. a., 2001] bzw. dem *WS-*-Universum*[1] geboten. WSDL und SOAP selbst geben im Gegensatz zu REST für die Schnittstellen keine Standards vor, um eine Interoperabilität gewährleisten zu können. Das World Wide Web (WWW) kann als eine riesige REST-Anwendung angesehen werden, allerdings gibt es viele Webanwendungen und Dienste, die nicht als *RESTful* angesehen werden können. Als *RESTful* wird eine Anwendung bezeichnet, die konform zu REST ist, also die möglichst alle Prinzipien einhält.

[1] Sammlung von Spezifikationen für Web-Services, die den Präfix „WS-“ tragen.

Es handelt sich bei REST folglich um eine Client-/Server-Architektur, bei der die Clients Anfragen an einen Server stellen und dieser entsprechende Antworten liefert. Auf dem Server befinden sich Daten in Form von Ressourcen und die Geschäftslogik. Die Clients übernehmen die Repräsentation und verwalten eigenständig den Status der Anwendung und der Benutzerschnittstelle. Verbunden werden diese über die *uniforme Schnittstelle*[2] von REST. Auch wenn REST auf HTTP basiert, ist das HTTP für den Einsatz von REST nicht zwingend notwendig. Generell kann nahezu jede beliebige Protokollfamilie verwendet werden, mit dem sich die Prinzipien realisieren lassen. In der Praxis wird allerdings HTTP verwendet, wie das WWW zeigt.

REST wurde erstmals von Fielding [2000] in seiner Dissertation vorgestellt. Er war maßgeblich an der Spezifikation des HTTP in der Version `1.1` und anderen für das WWW elementaren Spezifikationen beteiligt. Ebenfalls war er Mitbegründer des *Apache HTTP-Servers*[3]. REST ist kein Produkt oder Standard, sondern es beschreibt, wie Webstandards (beispielsweise HTTP) in einer webgerechten Art und Weise eingesetzt werden können [vergleiche Bayer und Sohn, 2007]. Ein Beispiel für solch einen Webstandard ist Web-based Distributed Authoring and Versioning (WebDAV) [siehe Goland u. a., 1999], welches einen Standard zum Bereitstellen von Dateien über ein Netzwerk mittels HTTP beschreibt.

REST lässt sich sehr gut zusammen mit Asynchronous JavaScript and XML (AJAX) [siehe van Kesteren, 2010] einsetzen. Der Grund hierfür ist unter anderem die einfache Möglichkeit, mit Ressourcen zu interagieren. Diese Tatsache hat dazu beigetragen, dass sich REST in letzter Zeit immer weiter verbreitet hat [vergleiche Rodriguez, 2008]. Bekannte Webanwendungen mit *RESTful*-Schnittstellen sind laut Bayer und Sohn [2007] beispielsweise Amazon[4], Google[5], Facebook[6] und Flickr[7].

2.2.1. Funktionsweise

Eine Webanwendung auf Basis von REST stellt eine Ansammlung von Ressourcen dar. Die Nutzung von HTTP allein garantiert allerdings nicht, dass eine Anwendung *RESTful* ist. Vielmehr muss der Entwurf sorgfältig gegen die Prinzipien geprüft werden. Eine REST-Architektur basiert auf Ressourcen, URIs und HTTP (vergleiche Kapitel 2.1). Ressourcen sind in REST ein zentrales Konzept, daher wird eine solche Architektur auch oft

[2] Jede Ressource implementiert stets dieselben Methoden, die durch das verwendete Protokoll vorgegeben werden.
[3] `http://httpd.apache.org`
[4] `http://www.amazon.de`
[5] `http://www.google.de`
[6] `http://www.facebook.de`
[7] `http://www.flickr.com`

als Resource-oriented Architecture (ROA) bezeichnet. REST basiert auf einer Reihe von Prinzipien, die eingehalten werden sollten, damit eine Anwendung als *RESTful* bezeichnet werden kann. Dabei handelt es sich nach Fielding [2000] um die folgenden:

- **Über URIs ansprechbare Ressourcen** Die URIs bilden einen globalen Namensraum, wodurch sichergestellt wird, dass die Ressourcen weltweit eindeutig identifizierbar sind. Dieses *Namensschema* ist bereits vielfach erprobt und einfach zu verstehen. Aufgrund dieses Prinzips ist es möglich den URL einer Ressource zu speichern und ggf. zu verschicken. Dies kann beispielsweise ein URL zu einem Produkt aus einem Shop sein. Sinnvollerweise sollten für Menschen gut lesbare und leicht verständliche URIs verwendet werden, auch wenn diese größtenteils nur für die Kommunikation zwischen zwei Anwendungen benutzt werden.

- **Verknüpfte Ressourcen (Hypermedia)** Das Verknüpfen von Ressourcen ermöglicht einer Anwendung Zustandsübergänge zu realisieren. Im Web können solche Verknüpfungen anwendungs- und serverübergreifend verwendet werden, ohne dass der Anwender davon etwas mitbekommen muss. Ein wesentliches Ziel dieses Prinzips ist, dass ein Client ohne vorherige Kenntnis der URI-Struktur Ressourcen anfordern und verändern kann, da er alle notwendigen Informationen geliefert bekommt.

  Diese Verknüpfungen innerhalb einer REST-Architektur wird auch als *Verbindungshaftigkeit* bezeichnet. Es entsteht bei diesem Vorgehen das Prinzip eines Automaten, der durch Zustandsübergänge (in diesem Fall bereitgestellte Verknüpfungen) von einem Zustand in einen anderen wechseln kann. Das Automatenprinzip gilt dann für jede Ressource, für die sich der Zustand durch den Anwender ändern lässt. Durch die Änderung von Ressourcenzuständen ändert sich in der Regel auch der Anwendungszustand.

- **Verwendung von Standard HTTP-Methoden** Ein Browser weiß was er mit einem URI tun kann, weil jede Ressource die gleiche *uniforme Schnittstelle* unterstützt. Bei HTTP heißen diese Operationen *Verben*. Die für REST relevanten HTTP-Operationen sind:

  GET Fordert eine Ressource an, gilt als sicher[8] und ist idempotent[9].

  POST Hinzufügen neuer Daten bzw. (Sub-)Ressourcen zu einer bestehenden Ressource. Für diese Operation gibt es keine Garantien. In einem *RESTful*-Entwurf wird diese Methode laut Richardson und Ruby [2007] nur zum Anlegen von Ressourcen verwendet, die zu einer übergeordneten Ressource in einer Relation stehen soll.

[8] Ein Client geht durch die Verwendung keine Verpflichtungen ein.
[9] Die Operation kann beliebig oft wiederholt werden ohne Konsequenzen zu haben.

PUT Ändern oder Anlegen einer Ressource. Diese Methode sollte allerdings nur zum Verändern benutzt werden, da in den meisten Fällen der URI einer noch nicht vorhandenen Ressource noch nicht bekannt ist. Die `POST`-Methode besitzt dieses Problem nicht, da diese normalerweise auf eine bestehende Ressource (meist eine Listenressource, siehe Kapitel 5.2.2) ausgeführt wird. Diese Operation ist idempotent [vergleiche Richardson und Ruby, 2007].

DELETE Löscht eine Ressource und ist idempotent.

HEAD Fordert Metadaten, also nur die HTTP-Kopfzeilen ohne den Inhalt, einer Ressource an.

OPTIONS Fordert eine Liste, der für diese Ressource zur Verfügung stehenden Methoden und Eigenschaften, vom Server an. Dafür wird die HTTP-Kopfzeile `Allow` verwendet [siehe Fielding u. a., 1999]. Das Verhalten kann sich allerdings unterscheiden, je nachdem ob eine Authentifizierung für eine Methode der Ressource erforderlich ist oder nicht. Aus diesem Grund kann diese Methode auch für einfache Zugriffsberechtigungsprüfungen verwendet werden [vergleiche Richardson und Ruby, 2007].

PATCH Mittels der Methode `PATCH` [siehe Dusseault und Snell, 2010] ist es wie bei `PUT` möglich eine Ressource zu verändern. Der Unterschied liegt in der Art und Weise wie die Änderungen der Ressource zum Server übertragen und wie diese auf dem Server umgesetzt werden. Bei `PATCH` werden nur die Daten an den Server übertragen, die wirklich geändert werden sollen. Bei `PUT` hingegen wird immer die gesamte Repräsentation der Ressource übertragen, also auch die Werte, die sich nicht geändert haben.

Bei einer REST-Anwendung müssen die oben genannten Operationen ausreichen. Dafür ermöglicht dieses Konzept eine Kommunikation jeder Komponente miteinander, die das HTTP-Anwendungsprotokoll unterstützt. Da die Methoden `PUT` und `DELETE` häufig auf Webservern deaktiviert sind, müssen diese für eine *RESTful*-Anwendung zunächst aktiviert werden.

Streng genommen kommt REST, wie beispielsweise eine Datenbank auch, mit den so genannten **CRUD**-Operationen (Create, Retrieve, Update, Delete) aus. Die Tabelle 2.1 zeigt eine Gegenüberstellung der **CRUD**-Operationen im Vergleich zwischen Datenbanksystemen und REST. Es können allerdings trotz dieser Einschränkung komplexe Geschäftsprozesse abgebildet werden. Ein Beispiel hierfür stellt die Kombination der Anwendungsfälle *Bild als exzellent vorschlagen* und *Potenziell exzellentes Bild durch Experte annehmen/ablehnen* aus Abschnitt 5.1 dar.

CRUD	DBMS	REST
Create	INSERT *columns* INTO *table* ...	POST
Retrieve	SELECT *columns* FROM *table* ...	GET
Update	UPDATE *table* SET ...	PUT
Delete	DELETE FROM *table* ...	DELETE

Tabelle 2.1.: CRUD-Operationen in DBMS und REST

Im Gegensatz zu vielen RPC-Architekturen kodiert REST keine Methodeninformationen in dem URI, da sie Ort und Namen der Ressource angibt, nicht aber die Funktionalität, welche die Ressource anbietet.

- **Unterschiedliche Repräsentationen von Ressourcen** Eine Repräsentation ist eine Darstellung der Ressource, idealerweise in einem Standardformat. Ressourcen haben in REST nicht nur eine, sondern potenziell mehrere Repräsentationen, wie beispielsweise Grafik, Extensionable Markup Language (XML), JavaScript Object Notation (JSON) [siehe Crockford, 2006] oder Portable Document Format (PDF) [siehe Adobe Systems, 2008]. Unterschiedliche Clients können mittels der *Content Negotiation*[10] jeweils das Format anfordern, das am besten ihren Bedürfnissen entspricht. Für die *Content Negotiation* wird die Kopfzeile `Accept` verwendet.

- **Statuslose Kommunikation** REST schreibt vor, dass der Status entweder vom Client gehalten oder vom Server in einen Ressourcen-Status umgewandelt wird. Nicht gewollt ist ein auf dem Server abgelegter transienter (vorübergehender) clientspezifischer Status über die Dauer einer Anfrage hinweg. Gründe hierfür sind beispielsweise die Skalierbarkeit und die Kopplung zwischen Client und Server. Skalierbarkeit bedeutet in diesem Kontext, dass bei einer statuslosen Kommunikation eine Lastverteilung einfacher realisiert werden kann. Es müssen dabei, bis auf den Status der Ressource, keine Statusinformationen zwischen den Servern synchronisiert werden. Da die Anfragen nicht voneinander abhängig sind, können direkt hintereinander folgende Anfragen von verschiedenen Servern verarbeitet werden [vergleiche Richardson und Ruby, 2007].

  Ein Client kann sich bei der Kommunikation nicht darauf verlassen, dass ein in einer vorherigen Interaktion entstandener Kontext später immer noch vorhanden ist. Stattdessen müssen alle Informationen, die der Server zur Verarbeitung einer Anfrage benötigt, in dieser Anfrage auch enthalten sein. Bei einer alternativen Implementierung ohne REST würde man diese Zustandsinformationen beispielsweise in Datenbanken speichern, so dass diese ebenfalls möglichst serverübergreifend erreichbar sind.

[10] Angabe des gewünschten Repräsentationsformats über HTTP-Kopfzeilen.

Der Ressourcenzustand beinhaltet Informationen über eine vom Server verwaltete Ressource, dieser ist immer für alle Anwender zu einem bestimmten Zeitpunkt identisch. Der Anwendungszustand, der in der Regel vom Client verwaltet wird, bezieht sich auf einen Zustand, der exklusiv für jeden Client vorhanden ist. Eine gute und ausführliche Erläuterung zu den verschiedenen Zustandsarten liefern Richardson und Ruby [2007].

In der Praxis nutzen viele HTTP-basierte Anwendungen *Cookies*[11] und andere Techniken, um Zustandsinformationen zu behalten. Dies hat zur Folge, dass diese nicht als *RESTful* angesehen werden können. Durch die Verwendung von *Cookies* und anderen Techniken wird es problematischer, die Skalierbarkeit bzw. Flexibilität zu gewährleisten. Richardson und Ruby [2007] sagten hierzu passenderweise:

> *„Cookies break a web service client's back button."*

Es ist daher nicht *RESTful* eine klassische *Sitzung (engl. Session)* zu verwenden, bei der Daten in *Cookies* oder auf dem Server gespeichert werden und durch einen Schlüssel in dem URI zugeordnet werden. Allerdings ist es *RESTful* und auch durchaus üblich den Zustand in dem URI zu kodieren. Beispielsweise wäre das bei einer Liste die Angabe der aktuellen Position, also des Zustands, die mit dem Parameter `page=12` kodiert werden könnte. Also darf der URI den Zustand nicht unterstützen bzw. identifizieren, sondern muss diesen enthalten [vergleiche Richardson und Ruby, 2007].

Ein Client kann mit einer Ressource interagieren, wenn er den URI der Ressource und die durchzuführende Aktion kennt. Auch wenn REST die Menge von Operationen auf einige wenige beschränkt, können durchaus fortgeschrittene Anwendungsfälle implementiert werden. Dies wird möglich durch eine entsprechende Geschäftslogik, die sich hinter den Operationen verbergen kann.

Hypermedia as the Engine of Application State (HATEOAS) ist ein Prinzip nach dem REST arbeitet. Damit ist gemeint, dass nur Verknüpfungen für die Änderung des Ressourcenzustands verwendet werden dürfen, die zuvor beim Auslesen einer Ressource vom Server mitgeliefert wurden. Somit gibt der Server die verfügbaren Operationen dem Client vor, ohne dass der Client sie vorher kennen muss. Bei REST-Anwendungen übernimmt der Client normalerweise selber die Verwaltung des Anwendungszustands.

Somit ergeben sich nach Sletten [2008] die Ziele Einfachheit, Erweiterbarkeit, Skalierbarkeit, Performanz und Einheitlichkeit für den Architekturstil REST.

[11] Speichert Informationen beim Client; wird häufig in Verbindung mit Sitzungen verwendet.

2.2.2. Vorteile

Ein großer Vorteil bei REST ist, dass der bereits ausgefeilte *Caching-Mechanismus* vom HTTP für den Zugriff auf Ressourcen verwendet werden kann. Ebenso können Ressourcen so einfach gegen unbefugte Zugriffe eingeschränkt werden. Hierfür können verschiedene Techniken verwendet werden, wie beispielsweise die *HTTP-Basic-*, die *HTTP-Digest-Authentication* oder Sicherheits-Schlüssel für das API, so genannte *API-KEYs*. Die Verbindung kann gegen unbefugtes Mithören mittels Hypertext Transfer Protocol Secure (HTTPS) und Transport Layer Security (TLS) bzw. Secure Sockets Layer (SSL) gesichert werden. Diese Verfahren sind ebenfalls vielfach bewährte und erprobte Konzepte, die bereits zum HTTP-Konzept gehören.

URIs von Ressourcen können aufgrund der zuvor genannten Prinzipien einfach gespeichert und/oder verschickt werden. So können beispielsweise Artikel in einem Webshop oder Details zu einem bestimmten Benutzer dauerhaft abrufbar gehalten werden, vorausgesetzt die entsprechende Ressource (in diesem Fall der Artikel oder der Benutzer) existiert dann noch. Nicht zu unterschätzen ist die Tatsache, dass Ressourcen unterschiedliche Repräsentationen dank der *Content Negotiation* haben können. Auf diese Weise wird es Clients ermöglicht, sich eine Repräsentation *auszusuchen*, die sie am besten verarbeiten oder darstellen können.

Clients, die auf eine *RESTful*-Anwendung zugreifen wollen, benötigen nur die Möglichkeit HTTP-Anfragen zu versenden und HTTP-Antworten zu verarbeiten [vergleiche Hüffmeyer, 2010]. Diese Funktionen sind bei modernen Programmiersprachen bzw. Systemen bereits von Anfang an vorhanden. Als abschließender Vorteil sei noch zu nennen, dass HTTP-Anfragen in der Regel in einem Webserver von vornherein geloggt werden, so kann das vom Entwickler zu implementierende Logging für eine REST-Anwendung weniger komplex ausfallen, da bereits der Webserver einige wichtige Informationen loggt.

2.2.3. Grenzen

Laut Tilkov und Ghadir [2006] ist eines der größten Probleme von HTTP die fehlende Unterstützung asynchroner Kommunikation. Hier muss entweder auf Anwendungsmuster oder auf nicht standardisierte Lösungen zurückgegriffen werden. Es könnte beispielsweise in regelmäßigen Abständen oder nach einer vereinbarten Zeit erneut angefragt werden, ob das Ergebnis bereits zur Verfügung steht. Dieses Vorgehen erhöht allerdings die Menge der Anforderungen, da gegebenenfalls viele unnötige Anfragen mit entsprechenden Antworten hin- und hergesendet werden. Laut Fielding [2000] ist dieses so genannte *Pulling* im Sinne von REST, da ansonsten Mechanismen wie das *Caching* nicht mehr funktionieren würden.

Die *Serialisierung* und die *Deserialisierung* muss durch den Programmierer selber übernommen werden oder es muss ein *XML-Binding-Framework*[12] eingesetzt werden, ganz im Gegenteil zu SOAP, bei dem dieses Verhalten bereits enthalten ist. Ebenfalls wird mit SOAP festgelegt, wie solche Nachrichten abzubilden und zu interpretieren sind. SOAP Frameworks übernehmen in der Regel bereits das Verpacken der Aufrufe in XML und das Serialisieren der Daten bevor diese verschickt werden. Auf der Empfängerseite werden umgekehrt zunächst die Daten deserialisiert und vom XML zu Methodenaufrufen umgesetzt. Wird kein XML verwendet, müssen andere Mechanismen verwendet werden, die das eingesetzte Format erzeugen und wieder zurückwandeln können. Zwischen spezialisierten lokalen Servern (beispielsweise *Anwendungsserver* und *Datenbankserver*) werden effizientere Protokolle wie beispielsweise die CORBA oder die Java RMI [vergleiche Bayer, 2002] benötigt.

Wenig verbreitete HTTP-Methoden, wie z.B. `PUT` und `DELETE` sind oft in der Server- bzw. Firewall-Konfiguration gesperrt bzw. in Webservern deaktiviert. Diese Tatsache könnte Probleme bei der Verwendung von *RESTful*-Anwendungen machen. Zur Zeit existieren keine Modellierungsmöglichkeiten für *RESTful*-Anwendungen, so dass es diverse Möglichkeiten gibt eine solche Schnittstelle zu beschreiben und umzusetzen. Eine weiteres Problem dabei ist, dass für bestimmte Situationen die zu liefernden HTTP-Status eine subjektive Wahl ist. Dieses Verhalten kann gut in Kapitel 6 im Vergleich der Frameworks beobachtet werden.

2.2.4. Beispiel

Um die Funktionsweise von REST zu verdeutlichen, wird in diesem Abschnitt ein kleines Fallbeispiel anhand eines Blogs durchgeführt. Hierbei wird sich auf die reine Blog-Funktionalität beschränkt. Diese beinhaltet einen Beitrag zu erstellen, zu verändern, zu löschen und zu kommentieren. Kommentare können ebenfalls erstellt, verändert und gelöscht werden. Dabei werden Benutzer und weiterführende Funktionalitäten ignoriert.

Klassische RPC-Schnittstelle

Bei einer klassischen RPC-Schnittstelle gibt es viele Möglichkeiten diese zu definieren. In diesem einfachen Beispiel wird eine *Fassade* verwendet, also eine große Schnittstelle, die alle Aufrufe kapselt. Dadurch kann die Modularisierung abstrahiert werden und es bestehen mehr Möglichkeiten eine Lastverteilung zu verwenden. In Abbildung 2.1 wird die Schnittstelle grafisch in Unified Modeling Language (UML) ähnlicher Notation dargestellt.

[12] Wandelt Anfragen und Antworten von und in XML um.

Abbildung 2.1.: Blog-Beispiel RPC-Schnittstelle

Jede Methode besitzt in der klassischen Form Parameter und Rückgabewerte. Diese Schnittstelle kann durch verschiedene RPC-Varianten implementiert werden (beispielsweise SOAP, CORBA oder RMI).

REST-Schnittstelle

Die REST-Schnittstelle besteht aus mehreren Komponenten, da jede Ressource ihre eigene Schnittstelle besitzt. Sie kann allerdings, ebenso wie auch die RPC-Schnittstelle, unterschiedlich entworfen werden und trotzdem grundsätzlich dieselben Ergebnisse liefern. In Abbildung 2.2 wird die Schnittstelle grafisch in UML ähnlicher Notation dargestellt.

In diesem Beispiel wurden vier Ressourcen identifiziert, um die Funktion eines rudimentären Blogs zu realisieren. Die Ressource `/beitraege` dient zum Auflisten aller Beiträge und zum Erstellen eines neuen Beitrags. Auf einzelne Beiträge kann durch die Ressource `/beitrag/{id}` zugegriffen werden, wobei die `id` ein eindeutiger Schlüssel für die vorhandenen Beiträge ist. Damit kann sie mittels `GET` angesehen, mit `PUT` bearbeitet oder mit `DELETE` gelöscht werden.

Ressource	**URI**	**Methode(n)**
Beitragsliste	/beitraege	GET, POST
Konkreter Beitrag	/beitrag/{id}	GET, PUT, DELETE
Kommentarliste pro Beitrag	/beitrag/{id}/kommentare	GET, POST
Konkreter Kommentar	/kommentar/{id}	GET, PUT, DELETE

Tabelle 2.2.: Blog - REST-Schnittstelle - URIs

Die Kommentare können über einen bestimmten Beitrag durch die Ressource `/beitrag/{id}/kommentare` aufgelistet und erstellt werden. Ansehen, Löschen und Bearbeiten von Kommentaren kann allerdings nur direkt über die Ressource `/kommentar/{id}` erfolgen. In Tabelle 2.2 werden alternativ zu Abbildung 2.2 alle Ressourcen und die verfügbaren Operationen in einer übersichtlichen Form dargestellt.

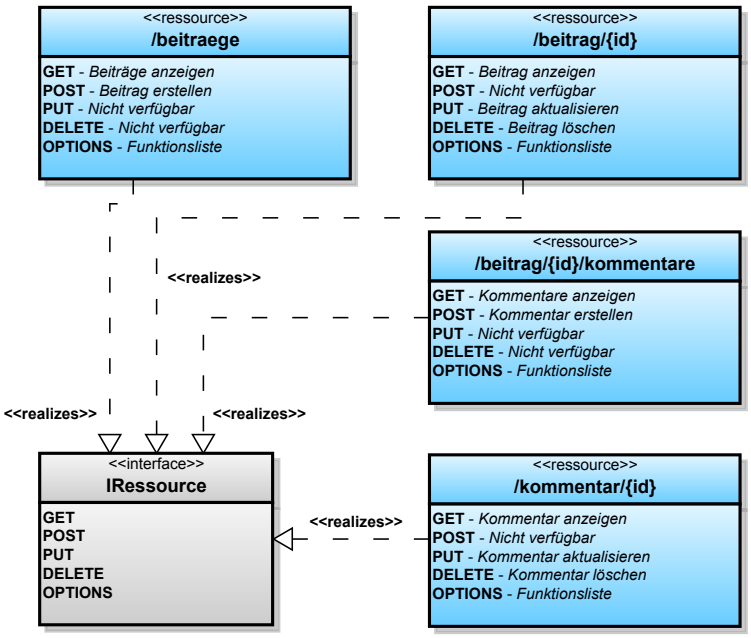

Abbildung 2.2.: Blog-Beispiel REST-Schnittstelle

2.2.5. Tipps und Best Practices

Auch für die Identifikation von Ressourcen für eine auf REST-basierende Anwendung haben sich bestimmte Vorgehensweisen und Prinzipien bewährt. Für den Entwurf der URI-Struktur nennt Rodriguez [2008] die folgenden Tipps und Best Practices:

- Dateinamenerweiterungen der serverseitig eingesetzten Technologien (beispielsweise `".jsp"`, `".php"` oder `".asp"`) sollten versteckt werden, damit diese jederzeit getauscht werden können, ohne dass die URIs sich ändern müssen.

- Alle URIs sollten komplett in Kleinbuchstaben gehalten werden.

- Leerzeichen sollten vermieden und durch Unterstriche „_“ oder Bindestriche „-“ ersetzt werden.

- Um die Übersichtlichkeit eines URI zu erhöhen, sollte so weit wie möglich auf Parameter verzichtet werden. Die notwendigen Informationen, die beim Speichern oder Versenden des URI erhalten bleiben müssen, sollten allerdings in dem URI kodiert werden.

Die folgenden Tipps und Best Practices für den Entwurf und die Implementierung stammen von Tilkov und Ghadir [2006], Rodriguez [2008] und Costello [2003]. Wenn diese eingehalten werden, sollte es nach Aussage der Autoren möglich sein eine *RESTful*-Anwendung zu entwerfen.

- Bei der Identifizierung von Ressourcen und deren Verknüpfungen untereinander ist von Bedeutung, was eindeutig durch einen URI identifiziert werden soll.

- Sammlungen (engl. Collections) sollten als Ressourcen modelliert werden, sofern die aufgelisteten Elemente wiederum mindestens eine Kurzbeschreibung besitzen sollen.

- Idealerweise sollte es einen Einstiegspunkt geben, welcher die nächsten erreichbaren Ressourcen über URIs enthält.

- Definieren der verfügbaren Methoden der zuvor identifizierten Ressourcen und der Repräsentationen der Ressourcen.

- Ressourcen sollten nach den auf ihnen ausführbaren Methoden klassifiziert werden (vergleiche Kapitel 5.2.2).

- Verfahren, die aus einem statuslosen System ein statusbehaftetes machen, sollten vermieden werden, auch wenn die Versuchung groß ist. Damit sind beispielsweise *Cookies* gemeint.

- Es sollten stets seiteneffektfreie `GET`-Methoden implementiert werden, also die Ressource darf in keiner Weise modifiziert werden.

- Zusammenhängende Ressourcen sollten durch Verknüpfungen in ihrer Repräsentation verbunden werden.

- Repräsentationen sollten mindestens in Extensionable Markup Language (XML), JavaScript Object Notation (JSON) oder in beiden Formaten verfügbar sein.

- Bei nicht vorhandenen Instanzen einer Ressource (beispielsweise eine bestimmte Person anhand ihrer ID), sollte kein `404 Not Found` HTTP-Status-Code, sondern stattdessen eine Standard-Seite oder -Ressource versandt werden [vergleiche Rodriguez, 2008]. Dieses Verhalten macht generell allerdings nur dann Sinn, wenn kein

Client verwendet wird, der die Antworten automatisch verarbeitet, sondern wenn beispielsweise ein Benutzer über einen Browser den Webservice mit einer HTML-Repräsentation benutzt.

Mithilfe der oben genannten Tipps und Best Practices sollte es nun möglich sein eine Schnittstelle, die als *RESTful* bezeichnet werden kann, zu entwerfen. Im Kapitel 5.2.2 werden weitere Aspekte betrachtet, wie beispielsweise die Typisierung von Ressourcen, sowie Erfahrungen die während dieser Ausarbeitung entstanden sind.

Irren ist menschlich, aber wenn
man richtigen Mist bauen will,
braucht man einen Computer.

Dan Rather - Journalist (USA)

3 Zielsetzung

Die Ziele und Kriterien für die nachfolgenden Kapitel werden in diesem Abschnitt definiert. Es werden einige vielversprechende Frameworks vorgestellt und anschließend wird jeweils ein Framework für Java, Personal Home Page Tools / Hypertext Preprocessor (PHP) und C# ausgewählt. Weitere Frameworks werden im Anhang A präsentiert. Nachdem Frameworks für die Implementierung ausgewählt wurden, werden die Anforderungen, Voraussetzungen und Funktionen der zu implementierenden Anwendung definiert und dargestellt. Anhand dieser Informationen wird im Kapitel 5 ein Entwurf erstellt und anschließend die Anwendung implementiert. Darauf folgt eine kurze Gegenüberstellung der gewählten Frameworks, vergleichbaren APIs und der durchgeführten Implementierung.

Die Auswahl der Frameworks geschieht unabhängig von den Anforderungen für die Implementierung, da diese Auswahl keine Frameworks ermitteln soll, die für diese Aufgabe am besten geeignet sind. Ziel dieser Evaluierung ist daher das Framework für die entsprechende Sprache zu finden, welches das größte Potenzial für die Entwicklung einer *RESTful*-Anwendung bietet. Später wird auf die Unterstützung der Prinzipien von REST durch die verwendeten Frameworks eingegangen und in Kapitel 6 zusammengefasst.

Für die Evaluierung, der zu verwendenden Frameworks, werden in diesem Abschnitt Kriterien definiert, die erfüllt werden sollten, damit die durchzuführenden Vergleiche eine solide Basis besitzen. Dabei werden sowohl notwendige, als auch optionale Kriterien definiert. Für die zu implementierende Webanwendung werden vier Arten von Kriterien definiert, die von notwendigen bis hin zu definitiv nicht umzusetzenden Kriterien reichen. Des Weiteren werden Voraussetzungen definiert und eine kurze Produktbeschreibung angeführt. Die abschließende Gegenüberstellung der Frameworks zueinander und der unterschiedlichen Implementierungen der zu entwickelnden Webanwendung wird ebenfalls anhand der hier definierten Kriterien vollzogen.

3.1. Kriterien für die Frameworkevaluation

Die Frameworks werden anhand einer groben Analyse der Funktionalität und Bedienbarkeit ausgewählt. In Anlehnung an Schneider [2010] werden Kriterien definiert, anhand deren die zu verwendenden Frameworks ausgewählt werden. Ein wichtiges Kriterium ist beispielsweise, dass alle Frameworks dieselbe URI-Struktur unterstützen. Es wird für jede zu verwendende Programmiersprache jeweils ein Framework ausgewählt, welches möglichst gut die nachfolgenden Anforderungen erfüllt. Die konkreten Kriterien sollen ermöglichen, dass die Frameworks gut miteinander verglichen werden können. Die *notwendigen Kriterien* sind zu erfüllen, damit das Framework in die engere Auswahl in Kapitel 4 aufgenommen werden kann. Die *optionalen Kriterien* wären zwar wünschenswert, aber sind nicht zwingend notwendig.

Notwendige Kriterien Bei den *notwendigen Kriterien*, die auf jeden Fall erfüllt werden müssen, handelt es sich um die folgenden:

- Bietet das Framework Unterstützung für alle Prinzipien von REST, so dass mit möglichst wenig Aufwand eine *RESTful*-Anwendung erstellt werden kann?
- Welche Darstellungsarten eines URI werden unterstützt bzw. wird die verzeichnisbasierte Variante unterstützt, die in dieser Ausarbeitung verwendet wird? Dieses Kriterium soll sicherstellen, dass sich die verschiedenen Implementierungen miteinander verbinden lassen.
- Unterstützung der HTTP-Methoden `GET`, `PUT`, `POST` und `DELETE`.
- Das Framework muss noch weiterentwickelt bzw. gepflegt werden.

Optionale Kriterien Frameworks, welche die *notwendigen Kriterien* gleich gut erfüllen, aber bei diesen Kriterien hervorstechen, werden bei der Auswahl bevorzugt. Bei den *optionalen Kriterien* handelt es sich um die folgenden:

- Einfache Einbindung in die eigene Anwendung bzw. die einfache Implementierung einer neuen Anwendung durch eine verständliches und gut dokumentiertes API. Idealerweise mit der Unterstützung einer automatischen Code-Generierung.
- Stabiles und voll funktionsfähiges Entwicklungs-Stadium des Frameworks, damit alle Anforderungen an die Anwendung korrekt getestet werden können.
- Unterstützung weiterer HTTP-Methoden, wie beispielsweise `OPTIONS`, `HEAD`

oder `PATCH`.

3.2. Produktbeschreibung

In diesem Abschnitt wird die Anwendung anhand von Kriterien, Anforderungen und einer Produktbeschreibung definiert. Diese bilden die Grundlage für den in Kapitel 5 beschriebenen Entwurf und die Implementierung. Die zu entwickelnde Anwendung soll ein Web-Fotoalbum ähnlich wie *Picasa*[1] oder *Flickr*[2] sein. Dabei soll es möglich sein Bilder in Alben zu hinterlegen. Es werden hierfür drei Clients und drei Server implementiert, die jeweils paarweise in unterschiedlichen Sprachen realisiert werden.

Die Server sollen soweit wie möglich konform zueinander sein, so dass theoretisch jeder Client mit jedem Server kommunizieren kann. Als Programmiersprachen werden PHP, Java und C# verwendet, wobei der PHP- und der Java-Client als eine Webanwendung und der C#-Client als eine native Windows-Anwendung realisiert wird. Diese wird nativ implementiert, um zu zeigen, dass die Verwendung eines REST-Webservices nicht auf Web-Clients beschränkt ist, sondern beliebig sein kann.

3.2.1. Anforderungen

Da im Mittelpunkt dieser Arbeit der Vergleich zwischen verschiedenen Frameworks und den Implementierungen der Schnittstelle mithilfe dieser Frameworks steht, werden einige Funktionen möglichst einfach gehalten und andere gar nicht berücksichtigt. Solche Funktionen werden später im Kapitel 7 als mögliche Erweiterungen bzw. Verbesserungen aufgeführt. Im folgenden Teil dieses Kapitels wird erläutert, welche Funktionen umgesetzt werden sollen und welche optional sind. Die Kriterien sind alle abhängig von der Realisierbarkeit durch die zu verwendenden Frameworks.

Musskriterien Die folgenden Kriterien müssen mindestens erfüllt werden, um das Ziel der Ausarbeitung zu erreichen, welches in diesem Kapitel näher erläutert wird.

- Die Anwendung soll *RESTful* sein, indem sie die Prinzipien von REST weitestgehend erfüllt.
- Alle Implementierungen sollen semantisch äquivalent sein, also dieselben Ergebnisse bei denselben Eingabedaten liefern.

[1] `http://picasaweb.google.com`
[2] `http://www.flickr.com`

- Es müssen mindestens die HTTP-Methoden `GET`, `PUT`, `POST`, `DELETE` und `OPTIONS` verwendet werden, allerdings muss nicht jede Ressource immer alle Methoden implementieren.

Sollkriterien Hier aufgeführte Kriterien sollten erfüllt werden, um ein möglichst gutes Ergebnis zu erlangen. Diese sind für ein verwertbares Ergebnis nicht zwingend erforderlich.

- Implementierung eines Arbeitsablaufs, bei dem bestimmte Bilder als *exzellent*[3] markiert werden können. Dabei soll ein Bild automatisiert und händisch als *potenziell exzellent* vorgeschlagen werden können. Ein so genannter *Experte*[4] muss dann den Vorschlag annehmen oder ablehnen.

- Jeder implementierte Client soll in der Lage sein mit jedem implementierten Server zu kommunizieren. Es muss also in jedem Client die Möglichkeit geben den Webservice zu wechseln. Diese Funktion sollte durch die nahezu überall einsetzbare Übertragung über HTTP und die Prinzipien von REST (siehe Kapitel 2) möglich sein. Das Wechseln muss allerdings nicht zur Laufzeit ermöglicht werden.

- Verwendung von Hinweisen und Best Practices zum Entwurf und zur Implementierung der Schnittstelle für die Entwicklung von *RESTful*-Webservices. Beispielsweise Ressourcentypen wie Primär-, Sub- und Filterressourcen (vergleiche Kapitel 5) und Entwurfsentscheidungen bzw. Vorgehen in Anlehnung an Tilkov [2009a].

Kannkriterien Die Kannkriterien sind optional und runden das Projekt ab. Sie können als künftige Erweiterungen angesehen oder direkt implementiert werden, sofern diese einfach umzusetzen sind.

- Verwendung verschiedener Formate für Anfragen und Antworten, beispielsweise zusätzlich zur Extensionable Markup Language (XML) die JavaScript Object Notation (JSON) oder das Atom Syndication Format (ASF).

- Anbieten von so genannten *eingebetteten Bildern* für fremde Webseiten. Hierbei soll ein bestimmtes Bild, zusammen mit seinem Titel, ggf. auch eine Beschreibung, ein Copyright-Hinweis und mindestens eine Verknüpfung enthalten sein.

[3] Ein besonders gutes Bild, welches entsprechend herausgehoben wird.
[4] Könnte beispielsweise ein Fotograf oder ein Künstler sein.

Abgrenzungskriterien Folgende Kriterien werden im Rahmen dieser Arbeit nicht umgesetzt, da sie zu zeitaufwendig wären und nicht notwendig sind für die Grundfunktionalität bzw. für die abschließenden Vergleiche:

- Ausführliche Authentifizierungsmechanismen und Rechtestrukturen für die Zugriffe auf Bilder und Alben inklusive Mechanismen, wie beispielsweise die Passwortgenerierung bei einem verlorenen Passwort.

- Komplexere Arbeitsabläufe für beispielsweise *exzellente Bilder* (siehe Kapitel 5), wie das Abstimmen von Experten für ein *potenziell exzellentes* Bild, bevor es wirklich als *exzellent* markiert wird.

Mehr zu diesen Punkten und weiteren Ideen zur Verbesserung der zu implementierenden Anwendung werden in der Zusammenfassung in Kapitel 7 vorgestellt.

Funktionale Anforderungen

Funktionale Anforderungen tragen den Präfix **F**. Die folgenden Anforderungen definieren die Funktionalität des Projekts:

- **F1**: Die Schnittstelle zwischen Client und Server soll *RESTful* sein und wird daher nach den in Kapitel 2 dargestellten Prinzipien entworfen.

- **F2**: Als Kommunikationsformat für die Metadaten wird als Standard valides XML verwendet, weitere Formate sind optional.

- **F3**: Einfaches Benutzermanagement zum Anlegen, Bearbeiten und Löschen von Benutzerkonten, zumindest serverseitig durch entsprechende Methoden der Ressourcen implementiert.

- **F4**: Verwalten von Alben und Bildern von Benutzern. Hierbei sollen Alben und Bilder erstellt, bearbeitet und gelöscht werden können. Zusätzlich können Bilder und Alben noch bewertet und anderen Benutzern vorgeschlagen werden. Bilder können zusätzlich noch kommentiert und automatisiert bzw. händisch als *exzellente Bilder* vorgeschlagen werden (siehe Kapitel 5).

- **F5**: Bilder können mit Benutzern verknüpft werden, damit eine Person als „auf dem Bild befindlich“ markiert werden kann.

Nicht-Funktionale Anforderungen

Nicht-Funktionale Anforderungen tragen den Präfix **N**. Die folgenden Anforderungen definieren, welche Eigenschaften die Anwendung haben soll:

- **N1**: Bei den Server-Implementierungen soll möglichst einfach eine Lastverteilung nachrüstbar sein, somit muss die Schnittstelle, die Speicherung der Bilder und die Datenbank entsprechend entworfen und implementiert werden.
- **N2**: Die Anwendungen sollen möglichst plattformunabhängig sein, sofern die notwendigen Technologien dies unterstützen.
- **N3**: Als Datenbank soll ein gemeinsames Relational Database Management System (RDBMS) für alle Lösungen zur Speicherung der Metadaten verwendet werden. Für die Speicherung der Bilder soll ein gemeinsames Repository im Dateisystem verwendet werden.
- **N4**: Die Anwendung soll vom Quelltext ausgehend leicht wartbar, gut dokumentiert und verständlich sein.
- **N5**: Drei Server und drei Clients sollen in jeweils drei verschiedenen Sprachen implementiert werden. Dabei wird jeweils ein Client und ein Server in PHP, Java und C# implementiert.

Benutzeroberfläche

Es werden zwei verschiedene Varianten der Benutzeroberfläche in dieser Ausarbeitung unterschieden. Es handelt sich dabei um eine Weboberfläche, die von dem PHP- und dem Java-Client implementiert werden und von einer nativen Windows-Oberfläche, die von dem C#-Client implementiert wird. Die folgenden Anforderungen werden dabei an die Benutzeroberflächen gestellt:

- Sowohl die Web-, als auch die native Anwendung soll eine einfache und übersichtliche Oberfläche bieten.
- Die Weboberfläche soll mindestens in den gängigsten Browsern (Firefox `3` und Internet Explorer `8`) darstellbar und benutzbar sein. Der native Client soll unter Windows lauffähig sein, sofern ein entsprechendes *.NET*-Framework installiert ist.

- Es sollen mindestens die Grundfunktionalitäten der Anwendung über die Weboberfläche benutzbar sein. Funktionen, die darüber hinausgehen, können im Webclient optional umgesetzt werden. Der native Client unter C# sollte mindestens die Funktionalität bieten Bilder zu einem bestimmten Album hochzuladen.

Schnittstelle

Die Schnittstelle zwischen den Clients und den Servern soll auf Basis von REST möglichst alle Prinzipien erfüllen, damit sie als *RESTful* bezeichnet werden kann (vergleiche Kapitel 2). Dadurch soll es möglich sein, die Clients und die Server beliebig zu mischen. Ebenfalls kann mittels Skripten oder fremder Clients auf die Server zugegriffen werden. Die Schnittstellendefinition wird in Kapitel 5 vorgestellt.

3.2.2. Voraussetzungen

Aufgrund der Tatsache, dass konkurrierende Technologien eingesetzt werden, entstehen einige Voraussetzungen für den Betrieb der Anwendung. Die webbasierten Clients können in der Regel nur mit einem Webbrowser im vollen Funktionsumfang genutzt werden. Der C#-Client benötigt das *.NET*-Framework ab Version `4.0`.

PHP Die PHP-Implementierung läuft am besten auf einem *Apache HTTP-Server* in der Version `2.2` oder neuer. Alternativ kann sie auch auf dem *Apache Tomcat*[5] oder auf dem *Microsoft Internet Information Services (IIS)*[6] verwendet werden.

Java Auf einem *Apache Tomcat* in der Version `6` läuft die Java-Variante problemlos. Alternativ kann ein Webserver wie *JBoss*[7] oder *Glassfish*[8] verwendet werden.

C# Der *Microsoft Internet Information Services (IIS)* in der Version `6` oder höher wird für die C#-Variante empfohlen. Weitere Alternativen existieren mit denselben Funktionsumfang eher nicht.

Die Hardwareanforderungen fallen gering aus. Empfohlen wird allerdings mindestens ein Dualcore-System mit 2 GB Hauptspeicher, um alle Varianten auf einem Rechner parallel ausführen zu können. Bis auf die C#-Implementierungen von Client und Server laufen die anderen Implementierungen auf einer Vielzahl von verschiedenen Betriebssystemen,

[5] `http://tomcat.apache.org`
[6] `http://www.iis.net`
[7] `http://jboss.org`
[8] `https://glassfish.dev.java.net`

da sowohl Java, als auch PHP unter dem *Apache HTTP-Server* sehr weit verbreitet sind. Für eine Weiterentwicklung der C#-Komponenten wird *Microsoft Visual Studio 2010*[9] oder neuer empfohlen, da sonst möglicherweise nicht der volle Funktionsumfang verwendet werden kann. Die PHP und die Java Komponenten können in beliebigen *Eclipse*[10] Versionen ab der Version `3.0` weiterentwickelt werden. Bei der PHP-Variante empfiehlt es sich das *PDT-Plugin*[11] zu installieren.

3.3. Kriterien für die abschließenden Gegenüberstellungen

Die abschließende Gegenüberstellung in Kapitel 6 soll die praktischen Erfahrungen in Bezug auf die Frameworks, REST und der Webanwendung darstellen und vergleichen. Zum Schluss wird das API von der hier entwickelten Anwendung mit denen von *Flickr* und *Picasa* verglichen. Dieser Abschnitt definiert die Kriterien für diesen Vergleich.

3.3.1. Frameworks

Die abschließende Gegenüberstellung der verwendeten Frameworks soll aufzeigen, in wie weit die hier definierten Erwartungen erfüllt werden konnten. Dabei handelt es sich um die folgenden Kriterien:

- Implementierung bzw. Realisierung von Ressourcen
- Routing bzw. Zuordnung von URIs zu Ressourcen
- Realisierung von Standard HTTP-Methoden
- Unterstützung der *Content Negotiation*
- Statuslose Kommunikation
- Unterstützung von Hypermedia
- Benutzbarkeit anhand eines *„Hello World“*-Beispiels

[9] `http://www.microsoft.com/germany/visualstudio`
[10] `http://www.eclipse.org`
[11] `http://www.eclipse.org/pdt`

3.3.2. Implementierung

Verglichen wird anhand verschiedener Kriterien, beispielsweise wie die Ressourcen aussehen müssen, damit sie die notwendigen Anforderungen erfüllen oder wie aufwendig bestimmte Teile der Implementierungen sind. Abschließend wird noch auf Schwierigkeiten, die bei der Implementierung auftraten, speziell auf das jeweilige Framework bezogen, eingegangen. Die Kriterien wurden in Anlehnung an Fielding [2000] in Bezug auf Konformität zu REST und an Schneider [2010] in Bezug auf Implementationsdetails definiert.

- Schwierigkeiten bei der Umsetzung der Anforderungen
- Aufwand der Implementierung
- Erfüllung der REST-Prinzipien
- Konformität der verschiedenen Implementierungen

3.3.3. APIs ähnlicher Projekte

Abschließend wird das API der entwickelten Anwendung mit denen von *Flickr* und *Picasa* verglichen. Hierfür werden entsprechende Vergleichskriterien definiert. Für den Vergleich werden einige ausgewählte Kriterien vorgegeben, um einen groben Überblick zwischen den APIs zu erhalten. Hierbei handelt es sich um die folgenden Kriterien:

- Authentifizierung
- Bibliotheken
- Funktionsumfang
- Formate und Technologien
- Erfüllung der REST-Prinzipien

Zu jedem der zuvor genannten Kriterien werden Gegenüberstellungen zum API des hier entwickelten Web-Fotoalbums gemacht.

Glücklich sind die Benutzer,
die nichts erwarten.
Sie werden nicht enttäuscht.
Edward A. Murphy - Ingenieur (USA)

4

Gegenüberstellung und Evaluierung ausgewählter REST Frameworks

Ausgewählte REST-Frameworks für die Programmiersprachen Java, PHP und C# werden in diesem Kapitel in einer Übersicht vorgestellt. Anschließend wird pro Programmiersprache ein Framework ausgewählt, welches für die jeweilige Implementierung verwendet wird. Die Auswahl der hier aufgeführten Kandidaten wurde anhand der in Kapitel 3.1 festgelegten Kriterien durchgeführt. Der Vollständigkeit halber wurden die Frameworks, die diese Kriterien nicht erfüllen, im Anhang A aufgeführt.

4.1. Wozu ein Framework?

Für die Verwendung eines Frameworks für diese Ausarbeitung gibt es zwei Gründe. Zum einen soll geprüft werden, wie groß der Funktionsumfang der bereits existierenden Frameworks ist (speziell bezogen auf die in Kapitel 2 vorgestellten Prinzipien von REST) und zum anderen, ob sich in der Praxis beliebige Frameworks mit beliebigen Programmiersprachen als Kommunikationspartner (Client bzw. Server) mischen lassen. Ein weiteres Argument für die Verwendung eines Frameworks ist die bereits investierte Entwicklungszeit und Ausgereiftheit, die eine saubere und gut funktionierende Implementation fördert. Als abschließendes Argument ist es im Sinne des *Software Engineerings*, vorhandene Frameworks zu verwenden, damit „das Rad nicht neu erfunden werden muss“.

Eine komplette Übersicht der recherchierten Frameworks findet sich im Anhang A. Die Frameworks im Anhang wurden weit weniger im Detail betrachtet als die in diesem Kapitel gelisteten.

4.2. Frameworks für PHP

Für PHP gibt es wenige REST-Frameworks, die den vollen Umfang von REST unterstützen. Bei PHP ist das größte Problem die URI-Adressierung im Stil von Verzeichnissen, wie beispielsweise `http://host/ressource/id`, zu unterstützen, da hier zunächst im Dateisystem geschaut wird, ob die angegebene Datei bzw. der angegebene Pfad vorhanden ist. Dieses Verhalten ist begründet in der Funktionsweise der Webserver, in denen ein PHP-Modul installiert ist. Eine mögliche Lösung dieses Problems bietet das Modul *mod_rewrite*[1] für den *Apache HTTP Server*. Mit diesem Modul ist es möglich einen URL automatisch, für den Anwender bzw. dem Client transparent, umzuschreiben.

4.2.1. Easyrest

Das Framework *Easyrest*[2] ist ein einfach zu verwendendes Framework, das sowohl den Client- als auch den Server-Teil unterstützt. Es existiert ein schlüssiges Beispiel vom Autor, welches allerdings entgegen der Prinzipien von REST arbeitet. Interessant an diesem Framework ist die einfache Möglichkeit mit einem API-Schlüssel und einem Sicherheits-Schlüssel eine implizite Authentifizierung der Clients mit dem Server zu verwenden. Das Framework steht unter der *GPLv3-Lizenz*[3].

Verwendete Technologien	PHP 4.3, XML-RPC
Letztes Update im Repository	Oktober 2009
Aktuelle Version	1.0 bzw. 1.1 beta
Unterstützung für Client/Server	Client und Server
Lizenz	GPLv3

Tabelle 4.1.: PHP-Framework Easyrest - Eckdaten

Easyrest unterstützt generell erstmal nur RPC über XML anstatt REST, wobei die Verwendung sehr einfach und überschaubar ist. Es kann mit wenigen Zeilen Quelltext die gesamte Schnittstelle implementiert werden. Es kann zwar auf REST umgeschaltet werden, allerdings ist die REST-Unterstützung eher rudimentär vorhanden. Der Autor sagt in Güvenç [2009], dass die Unterstützung für REST in der nächsten Version weiter ausgebaut werden soll.

[1] `http://httpd.apache.org/docs/2.2/mod/mod_rewrite.html`
[2] `http://code.google.com/p/easyrest`
[3] `http://www.gnu.org/licenses/gpl.html`

4.2.2. Recess

Bei dem Framework *Recess*[4] handelt sich um ein Framework, welches zunächst nur die Server-Seite unterstützt. Die Konfigurationsoberfläche ist leicht zu bedienen und es ist möglich sich die Kontrollklassen und die Modelle anhand von vorhandenen Datenbankstrukturen generieren zu lassen oder über die Weboberfläche händisch erstellen. Das Framework steht unter der *MIT-Lizenz*[5].

Verwendete Technologien	PHP 5.2.4, Apache HTTP Server, mod_rewrite
Letztes Update im Repository	August 2009
Aktuelle Version	0.2
Unterstützung für Client/Server	Nur Server
Lizenz	MIT

Tabelle 4.2.: PHP-Framework Recess - Eckdaten

Nachdem das Framework installiert wurde, können eigene Anwendungen über die mitgelieferte Konfigurationsoberfläche erstellt und verändert werden. Hier wird der Entwickler mittels kurzen *Wizards* durch die Prozesse geführt. Gleich mitgeliefert werden fertige `.htaccess`-Dateien, in denen für den *Apache HTTP Server* bereits fertige Vorgaben für das Umschreiben der URLs vorhanden sind.

Recess arbeitet ausschließlich objektorientiert und hält sich an das Prinzip Model-View-Controller (MVC), welches sowohl als Architektur als auch als ein Entwurfsmuster angesehen werden kann. Das Framework erwartet, dass, wie auch in Java empfohlen, nur eine Klasse pro Datei vorhanden ist. Dies wird dem Framework anhand des Dateinamens mitgeteilt (aus *beispiel.php* wird *beispiel.class.php*) [vergleiche Jordan, 2008a].

Mithilfe von Routen legt der Entwickler fest, bei welchen URI-Aufrufen welche Kontrollklassen und Methoden angesprochen werden. Das Routing wird mittels Annotationen in den Kontrollklassen für Ressourcen festgelegt und durch das Modul *mod_rewrite* für den *Apache HTTP-Server* unterstützt. Die Unterstützung wird durch das Umwandeln der URIs realisiert, so dass als Einstiegspunkt immer ein *Front-Controller*[6] [siehe Sun Microsystems, 2002] dient, der dann das Routing übernimmt. Das Framework kümmert sich nach Festlegung der Routen selbstständig darum, die entsprechenden Kontrollklassen und deren Methoden aufzurufen [vergleiche Jordan, 2008b]. Dieses Prinzip ähnelt dem Verhalten des *Struts-Frameworks*[7] bei Java Enterprise Edition (JEE).

[4] `http://recessframework.org`
[5] `http://www.opensource.org/licenses/mit-license.php`
[6] Architekturmuster, welches als Router für eine Webanwendung dient.
[7] `http://struts.apache.org`

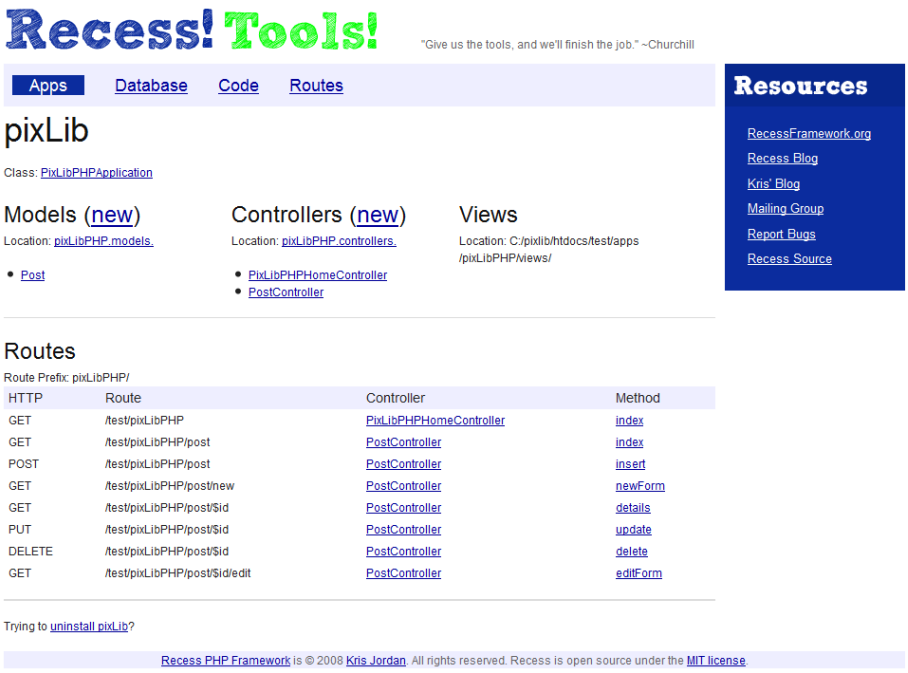

Abbildung 4.1.: Konfigurationsoberfläche von Recess

Das Framework benötigt eine Datenbank, um Informationen über die erstellten Anwendungen zu speichern. Die definierten Datenmodelle können einfach über das Framework in einer Datenbank verwaltet werden. Ebenfalls ist es möglich aus bereits vorhandenen Tabellen die zugehörigen Modelle automatisch zu generieren [vergleiche Jordan, 2009b]. *Recess* funktioniert ohne das Modul *mod_rewrite* nicht ordnungsgemäß. Dies äußert sich indem das Routing fehlschlägt und daher das zuvor erwähnte Problem bezüglich der URIs bei klassischen Webservern auftritt. Beispielsweise werden bei der Konfigurationsoberfläche die Bilder und Cascading Style Sheets (CSS)[8] nicht mehr geladen, da der URL ungültig ist. Eine gute unvollständige Übersicht bietet das Online-Buch vom Entwickler des Frameworks [vergleiche Jordan, 2009a].

Interessant an diesem Framework ist die automatische Generierung von Kontroll- und Modellklassen anhand von Datenbanktabellen. Der Nachteil dabei ist allerdings, dass bei komplexerem Verhalten die generierten Klassen vom Entwickler angepasst werden müssen. Für die Generierung durch die Konfigurationsseite von *Recess*, wird der Entwickler mittels *Wizards* durch den Prozess geführt. Hierbei können Namen und zu verwendende Tabellen angegeben oder sogar Tabellen generiert werden. Eine Einführung in die Verwendung von *Recess* kann in Anhang in Abschnitt C.1.2 nachgeschlagen werden.

[8] Formatvorlage, hier speziell für Webseiten.

4.2.3. Auswahl

Für die Implementierung der PHP-Variante wurde zunächst *easyREST* gewählt, da dieses sowohl die Client- als auch die Server-Seite unterstützt und da es einfach zu verwenden ist. Die REST-Unterstützung war leider zu ungenügend, da es nicht möglich ist, mit der oben genannten Version von *easyREST* einen *RESTful*-Webservice zu implementieren. Aus diesem Grund wurde frühzeitig zum Framework *Recess* gewechselt, welches für die Implementierung eines *RESTful*-Webservices geeignet ist.

4.3. Frameworks für Java

Da die meisten größeren REST-Frameworks für Java auf dem einheitlichen Standard *Java Specification Request (JSR) 311* [siehe Hadley und Sandoz, 2008] bzw. *Java API for RESTful Web Services (JAX-RS)* basieren, unterscheiden sie sich grundsätzlich in der Bedienbarkeit und in der Vollständigkeit der Implementierung dieses Standards. *JSR 311* ist ein API für *RESTful*-Webservices unter Java. Es arbeitet mithilfe von Annotationen und wurde im März 2008 finalisiert. Bei Java Webanwendungen existiert das Problem mit den verzeichnisbasierten URIs nicht, da diese sich selber um die Verarbeitung der URIs kümmern [vergleiche Mordani, 2009].

4.3.1. Restfulie

Restfulie[9] ist ein komplexes Framework, welches für Java, C# und Ruby on Rails existiert. Es unterstützt sowohl die Client- als auch die Server-Seite. Dieses Framework steht unter der *Apache License 2.0*[10].

Verwendete Technologien	Java, VRaptor3
Letztes Update im Repository	April 2010
Aktuelle Version	1.0.0 beta
Unterstützung für Client/Server	Client und Server
Lizenz	Apache License 2.0

Tabelle 4.3.: Java-Framework Restfulie - Eckdaten

[9] `http://restfulie.caelum.com.br`
[10] `http://www.apache.org/licenses/LICENSE-2.0.html`

Die Java-Variante von Restfulie benötigt zusätzliche Bibliotheken. Es basiert auf dem *VRaptor3*[11] Framework, welches für eine strenge MVC-Architektur in der Webanwendung sorgen soll. Zusätzlich bietet es Funktionen für beispielsweise *Content Negotiation*. Es ist damit nicht mehr notwendig auf die in der JEE definierten Klassen zuzugreifen. Das macht die Übertragung zu anderen Sprachen, für die dasselbe Framework existiert, einfacher. Mittlerweile ist *Restfulie* komplett in *VRaptor3* übergegangen, so dass es nicht mehr als eigenständiges Framework verfügbar ist. In der Java-Implementierung von *Restfulie* wird auf den Standard *JSR 311* gesetzt. Daher werden auch hier Annotationen[12] verwendet, um das Framework zu konfigurieren.

Das offizielle Beispielprojekt besitzt kein Skript zum Erstellen des Webcontainers. Ebenfalls gibt es zwei Klassen, die nicht ohne Anpassungen kompilierfähig sind. Um das Framework kompilierfähig und benutzbar zu machen, werden diverse Bibliotheken von Drittanbietern benötigt.

4.3.2. RESTeasy

RESTeasy[13] ist ein umfangreiches Framework für REST. Es wurde ursprünglich für *JBoss* entwickelt, soll aber laut Angabe der Entwickler in allen Webservern funktionieren, die JEE-Webcontainer unterstützen. Das Framework steht unter der *Apache License 2.0*.

Verwendete Technologien	Java 5, JBoss
Letztes Update im Repository	Juli 2010
Aktuelle Version	2.0.1
Unterstützung für Client/Server	Nur Server
Lizenz	Apache License 2.0

Tabelle 4.4.: Java-Framework RESTeasy - Eckdaten

Dieses Framework ist ein ausgereiftes Server-Framework mit vollständiger Implementierung der *JSR 311* und daher sehr flexibel. Es bietet eine gute Unterstützung für *RESTful*-Anwendungen und ist eine alleinstehende Anwendung, die als Web Application Archive (WAR)-Datei installiert wird. Speziell für JBoss gibt es ein Plugin, welches eine bessere Integration in den Webserver bietet. Als zusätzliche Repräsentationsformate neben XML unterstützt *RESTeasy* YAML Ain't Markup Language (YAML)[14] und XML-binary Optimized Packaging (XOP) [siehe Gudgin u. a., 2005] [vergleiche Schneider, 2010].

[11] `http://vraptor.caelum.com.br`
[12] Sprachelement aus Metadaten.
[13] `http://www.jboss.org/resteasy`
[14] `http://www.yaml.org`

Eine weitere Besonderheit gegenüber anderen Frameworks ist die von Haus aus integrierte *GnuZip*-Komprimierung für die Inhalte von Anforderungen und Antworten. Bei diesem Framework kann mit wenigen Zeilen Quelltext eine vollständige REST-Schnittstelle implementiert werden. Weitere Informationen und Beispiele können aus Schneider [2010] entnommen werden.

4.3.3. Jersey

Jersey[15] ist laut Angabe der Hersteller *die* Referenzimplementierung des *JSR 311* mit einer einfach zu handhabenden und erweiterbarem API. Das Framework steht unter der CDDL 1.1[16] und der *GPL 2 Lizenz*[17].

Verwendete Technologien	Java 5
Letztes Update im Repository	September 2010
Aktuelle Version	1.4
Unterstützung für Client/Server	Client und Server
Lizenz	CDDL 1.1 und GPL 2

Tabelle 4.5.: Java-Framework Jersey - Eckdaten

Hierbei handelt sich um ein mächtiges Framework, welches sehr einfach mittels Annotationen zu steuern ist. Die im WWW zu findenden Beispiele sind spärlich dokumentiert, so dass Vergleiche und die Entwicklung eigener Beispiel-Anwendungen notwendig ist. Nachteilig ist weiterhin, dass nur sehr wenig Dokumentation für den Einsatz unter *Apache Tomcat* vorhanden ist. Für die Verwendung von *Glassfish* und *NetBeans*[18] findet sich deutlich mehr Dokumentation. Die Implementierungen an sich sind davon allerdings nicht betroffen. Ein weiterer Nachteil ist, dass das Framework ausschließlich auf das HTTP aufsetzt, so dass kein Wechsel der Protokolle möglich ist. Weitere Details zur Verwendung des Frameworks finden sich im Anhang C.

4.3.4. Auswahl

RESTeasy läuft zwar in jedem Servlet-Container, aber entwickelt und optimiert wurde es für einen *JBoss*-Server. Da es sich bei allen drei Frameworks um Implementierungen des

[15] `http://jersey.java.net`
[16] `http://hub.opensolaris.org/bin/download/Main/licensing/cddllicense.txt`
[17] `http://www.gnu.de/documents/gpl-2.0.de.html`
[18] `http://www.netbeans.org`

JSR 311-Standards handelt, wurde die Auswahl zwischen diesen anhand von Eigenschaften und Funktionen, die über REST hinausgehen, getroffen. Darunter fällt beispielsweise die Menge an brauchbaren Beispielen und die Einfachheit der Verwendung.

Ein Argument für *Restfulie* ist, dass dieses auch für die Sprache C# existiert, welche ebenfalls verwendet werden soll. Weiterhin bietet *Restfulie* im Gegensatz zu *RESTeasy* eine Unterstützung für den Client, der den Aufwand der Implementierung auf der Clientseite mindern kann. Aus diesen Gründen wurde zunächst versucht eine Beispielanwendung mit *Restfulie* zu implementieren. Dabei ergaben sich beim Erstellen der Projekte Schwierigkeiten, die nicht in annehmbarer Zeit gelöst werden konnten. Bei den Problemen handelte es sich grundsätzlich um die folgenden:

- Aufgrund unzureichender Beispiele und Anleitungen im WWW war es schwer eine komplexere Anwendung zu entwickeln.

- Um eine Anwendung mittels *Restfulie* zu entwickeln, empfiehlt der Hersteller ein bereits vorbereitetes Projekt zu verwenden, welches sich genauso wie das eine angebotene Beispielprojekt nicht kompilieren ließ.

Zusätzlich zwingt das Framework *VRaptor3* eine unschöne Verzeichnisstruktur auf. Des Weiteren wird die Verwendung des MVC-Entwurfsmusters aufgezwungen, welches bei einer reinen Server-Anwendung nur ohne *Views* verwendet wird. Das Projekt *VRaptor3* ist zwangsweise mit *Restfulie* gekoppelt, da diese ineinander übergegangen sind.

Daher wurde als dritte Alternative *Jersey* ausgewählt. Die Vorteile bei *Jersey* sind, dass es weiter entwickelt und somit ausgereifter ist. Ein weiterer Vorteil ist, dass es sich ausschließlich auf REST bezieht und keine Vorgaben über die Struktur des Projekts oder der zu verwendenden Architekturmuster macht.

4.4. Frameworks für C#

Die Auswahl an Frameworks ist bei der Programmiersprache C# am kleinsten. Das mag daran liegen, dass *.NET*-Anwendungen häufiger im Bereich nativer Windows-Anwendungen statt bei Webanwendungen verwendet werden.

4.4.1. Restfulie

Die C#-Variante von *Restfulie* ist im Gegensatz zu den Varianten in Java und Ruby on rails zurzeit deutlich weniger weit entwickelt. Es findet sich so gut wie keine Dokumentation oder Beispiele für die Verwendung von *Restfulie* für C#.

Verwendete Technologien	.NET 4
Letztes Update im Repository	Juli 2010
Aktuelle Version	0.5
Unterstützung für Client/Server	Client und Server
Lizenz	Apache License 2.0

Tabelle 4.6.: C#-Framework Restfulie - Eckdaten

Das von *Restfulie* bereitgestellte Client-Framework bietet allerdings eine interessante Unterstützung für Hypermedia. Für alle URIs, die zu einer Ressource geliefert werden, existieren dann entsprechende Methoden, die genauso heißen wie der `rel`-Name eines solchen URI. Wenn also beispielsweise bei einer Ressource eine Verknüpfung `rel="andereRessource"` mitgeliefert wird, so kann der Entwickler im Client die Verknüpfung über `meineRessource.andereRessource()` verwenden. Dieses Verhalten wird durch den Datentyp `dynamic` [siehe msdn, b] ermöglicht. Dieser Typ ermöglicht es zur Kompilierungszeit Methoden aufzurufen, die zu diesem Zeitpunkt nicht existieren. Die Überprüfung findet daher erst zur Laufzeit statt.

Laut dem Hersteller gibt es auch einen Server-Teil für C#, der nur sehr schwer im WWW zu finden ist, da nur sehr wenig Dokumentation hierfür existiert. Beide Versionen werden als Dynamic Link Library (DLL) ausgeliefert, die dann im Projekt referenziert werden müssen. Eine Schnittstellenbeschreibung oder ähnliche ausführliche Dokumentationen sind für die C#-Variante im Internet nicht zu finden.

4.4.2. Windows Communication Foundation (WCF)

Bei der *WCF*[19] handelt es sich um eine dienstorientierte Kommunikationsplattform für verteilte Anwendungen. Das bedeutet, dass es kein reines Framework für REST ist. Es soll die Erstellung von Anwendungen mit einer SOA erleichtern und vereinheitlichen. Erstmalig wurde diese mit dem *.NET*-Framework in der Version `3.0` eingeführt.

[19] `http://msdn.microsoft.com/de-de/netframework/aa663324.aspx`

Verwendete Technologien	.NET
Aktuelle Version	4.0
Stand	April 2010
Unterstützung für Client/Server	Server
Lizenz	Proprietäre Software

Tabelle 4.7.: C#-Framework WCF - Eckdaten

Das Protokoll für die Übertragung und die Art der Serialisierung werden baukastenartig zusammengestellt, indem jeweils aus einer größeren Menge von verfügbaren Bausteinen gewählt werden kann. Hierbei können für die Übertragung beispielsweise Übertragungsprotokolle wie HTTP, TCP oder Simple Mail Transfer Protocol (SMTP) [siehe Postel, 1982] gewählt werden. Als Serialisierungsformat können beispielsweise SOAP, JSON oder ATOM gewählt werden. Ein interessanter Aspekt bei *WCF* ist, dass diese Auswahl auch zur Laufzeit veränderbar ist. Vergleiche hierzu einen Artikel von *Heise Developer* [siehe Schwichtenberg, 2010].

4.4.3. Auswahl

Aufgrund der begrenzten Auswahl an REST-Frameworks für C# fällt die Entscheidung für die Verwendung von *WCF* leicht. Alle anderen Frameworks bieten zu wenig Funktionalität oder Unterstützung für REST. *Restfulie* wurde nicht gewählt, da es so gut wie keine Beispiele und Dokumentationen gibt.

Programmieren Sie immer so, als wäre der Typ, der den Code pflegen muss, ein gewaltbereiter Psychopath, der weiß, wo Sie wohnen.

John F. Woods, Politiker (USA)

5

Entwurf und Implementierung

Nachdem ein Überblick über REST geliefert und Frameworks für die Implementierung gewählt wurden, kann jetzt die Webanwendung entworfen und implementiert werden. Die Webanwendung hat den Namen *pixLib* bekommen, welcher für *Bibliothek für Bilder* steht. Implementiert werden drei Server, die möglichst äquivalente Ergebnisse über die Schnittstelle liefern, sofern dieses Verhalten mit den verschiedenen Frameworks realisierbar ist. Zusätzlich werden drei Clients implementiert, wobei zwei eine Webanwendung und der dritte eine native Anwendung für Windows wird. Zunächst werden einigen Anwendungsfälle und die Architektur definiert. Darauf folgt die Implementierung und Informationen über die Tests zur Sicherstellung der geforderten Funktionalitäten. Abgeschlossen wird der Abschnitt mit einem kurzen Exkurs über die Verwendung der Anwendung und Schwierigkeiten bzw. Erfahrungen während des Entwurfs und der Implementierung.

5.1. Anwendungsfälle

Dieser Abschnitt stellt einige ausgewählte Anwendungsfälle für *pixLib* vor, bei denen der Fokus auf die komplexeren Vorgänge in *pixLib* gelegt wird. Exemplarisch werden noch zwei kleinere Anwendungsfälle vorgestellt, die das Grundverhalten von verschiedenen Funktionen zeigen sollen.

A1: Hochladen eines Bildes

Dieser Anwendungsfall beschreibt, wie ein neues Bild durch einen beliebigen Anwender in eines seiner eigenen Alben hochgeladen werden kann.

Beteiligte Akteure `Anwender`

Vorbedingungen Der `Anwender` muss ein gültiges Benutzerkonto besitzen.

Standardablauf Der `Anwender` authentifiziert sich mit seinen Anmeldedaten und wählt anschließend ein Album, zu dem das Bild hochgeladen werden soll, aus. Nachdem ein Album gewählt wurde, wird das Bild ausgewählt und weitere Informationen zu diesem Bild erfasst (beispielsweise ein Titel und eine Beschreibung). Abschließend wird das Bild mit den eingegebenen Informationen gespeichert.

Alternativer Ablauf Wenn das Album noch nicht existiert, muss es zunächst durch den `Anwender` angelegt werden. Weiterer Verlauf analog zum *Standardablauf*.

Nachbedingungen Das Bild wurde dem Album hinzugefügt.

Ausnahme Der `Anwender` authentifiziert sich nicht korrekt.

Nachbedingungen Das Bild wird abgelehnt und nicht gespeichert.

A2: Bild als exzellent vorschlagen

Dieser Anwendungsfall beschreibt den Vorgang, der ein Bild als *exzellent* vorschlägt.

Beteiligte Akteure `Anwender`, `System`, `Experten`

Vorbedingungen Das Bild darf noch nicht als *potenziell exzellentes Bild* vorgeschlagen worden oder als *exzellent* bzw. als *nicht exzellent* gekennzeichnet sein und der `Anwender` muss ein gültiges Benutzerkonto besitzen.

Standardablauf Ein `Anwender` authentifiziert sich mit seinen Anmeldedaten und wählt ein Bild aus, welches den bereits genannten Bedingungen entsprechen muss. Dieses Bild schlägt er als *exzellent* vor. Die `Experten` werden automatisch informiert, dass ein neues Bild als *exzellentes Bild* vorgeschlagen wurde.

Alternativer Ablauf Vorgang wird automatisiert durch das `System` bei Erreichung einer voreingestellten Anzahl von Bewertungen mit einer voreingestellten Mindestpunktzahl ausgelöst. Die `Experten` werden hier ebenfalls informiert, dass ein neues Bild als *exzellentes Bild* vorgeschlagen wurde.

Nachbedingungen Das Bild wurde als *exzellent* vorgeschlagen und die `Experten` wurden darüber informiert.

Ausnahme Der `Anwender` authentifiziert sich nicht korrekt.

Nachbedingungen Das Bild wird nicht vorgeschlagen.

A3: Potenziell exzellentes Bild durch Experte annehmen/ablehnen

Dieser Anwendungsfall beschreibt den Vorgang, der ein Bild als *exzellent* oder als *nicht exzellent* kennzeichnet.

Beteiligte Akteure `System`, `Experten`

Vorbedingungen Das Bild darf muss als *exzellentes Bild* vorgeschlagen worden sein und darf nicht als *exzellent* bzw. als *nicht exzellent* gekennzeichnet sein. Dieser Vorgang kann nur von `Experten` durchgeführt werden.

Standardablauf Ein `Experte` authentifiziert sich mit seinen Anmeldedaten und wählt ein *potenziell exzellentes* Bild aus. Nach einer Begutachtung des Bildes kann er es entweder als *exzellent* oder als *nicht exzellent* markieren.

Nachbedingungen Das Bild wurde als *exzellent* oder als *nicht exzellent* gekennzeichnet.

Ausnahme Der `Experte` authentifiziert sich nicht korrekt.

Nachbedingungen Das Bild wird nicht markiert.

A4: Löschen eines Bildes

Dieser Anwendungsfall beschreibt, wie ein Bild durch einen Anwender gelöscht werden kann.

Beteiligte Akteure `Anwender`

Vorbedingungen Der `Anwender` muss ein gültiges Benutzerkonto besitzen.

Standardablauf Zunächst authentifiziert der `Anwender` sich mit seinen Anmeldedaten und wählt anschließend das Album indem sich das zu löschende Bild befindet aus. Anschließend wählt er das entsprechende Bild aus und löscht es.

Nachbedingungen Das Bild wurde gelöscht.

Ausnahme Der `Anwender` authentifiziert sich nicht korrekt.

Nachbedingungen Bild wird nicht gelöscht.

Ausnahme Das zu löschende Bild existiert nicht.

Nachbedingungen Bild wird nicht gelöscht.

5.1.1. Exzellente Bilder

Der Ablauf für die Markierung eines Bildes als *exzellent* wird in den Anwendungsfällen A2 und A3 beschrieben. Wie im Anwendungsfall bereits beschrieben, kann der Vorschlag für ein *exzellentes Bild* entweder direkt von einem Anwender über eine entsprechende Verknüpfung oder indirekt durch das Bewerten eines Bildes ausgelöst werden. Sollte bei einer neuen Bewertung eine bestimmte Anzahl `x` an Bewertungen mit einem Durchschnittswert, der mindestens den Wert `y` beträgt, erreicht werden, wird das Bild automatisch vorgeschlagen.

Die Werte `x` und `y` können in der Systemkonfiguration festgelegt werden. Voraussetzung dafür ist, dass das Bild bisher noch nie als *exzellentes Bild* vorgeschlagen oder durch einen Experten als *exzellent* bzw. als *nicht-exzellent* markiert wurde. Die Abbildung 5.1 zeigt den Ablauf dieses Prozesses anhand eines Zustandsdiagramms, welches an das Fallbeispiel von Tilkov [2009a] angelehnt ist.

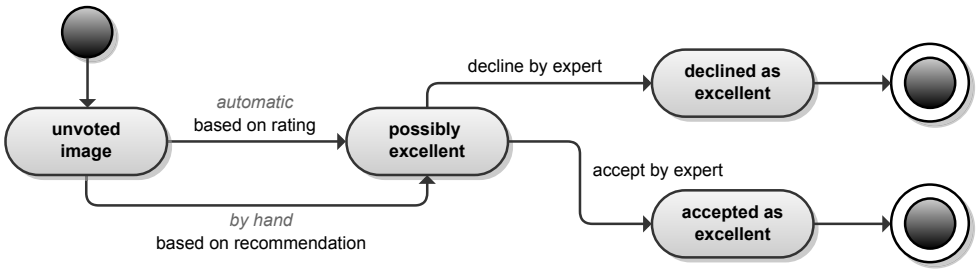

Abbildung 5.1.: Zustandsdiagramm für exzellente Bilder

Intern wird ein Bild als *potenziell exzellent* markiert, indem es einem *Phantom*-Benutzer vorgeschlagen wird. Dieser Benutzer kann nirgendwo angezeigt oder verwendet werden. Er dient lediglich zum Kennzeichnen der Bilder, indem eine weitere Funktion, nämlich das *Vorschlagen von Bildern*, verwendet wird. Ein Bild kann genau dann vorgeschlagen werden, wenn es weder vorgeschlagen noch markiert wurde. Das bedeutet, dass es für dieses Bild für den *Phantom*-Benutzer keinen Vorschlag geben darf und das Flag `bIsExcellent`

in der Tabelle der Bild-Metadaten muss `null` sein. Ein Bild gilt als *exzellent*, wenn das Flag `bIsExcellent` den Wert `1` hat, wobei der Wert `0` bedeutet, dass dieses Bild abgelehnt wurde. Sobald das Flag entweder auf `0` oder `1` gesetzt wird, wird ebenfalls der Zeitstempel der Aktion im Feld `tsMarkedExcellent` vermerkt.

5.1.2. Eingebettete Bilder

Die Funktion der *eingebetteten Bilder* erlaubt es Bilder aus der Datenbank von *pixLib* auf anderen Seiten zu zeigen. Dafür wird eine bestimmte Ressource, nämlich `embeddedImage/{id}` angesprochen, welche ein HTML Dokument mit dem gewünschten Bild liefert. Beispielsweise kann dieses Bild über einen *iFrame*[1] in eine Webseite eingefügt werden, ein Beispiel hierfür bietet der Quelltext 5.1 und das zugehörige Ergebnis in Abbildung 5.2.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
2        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
3  <html>
4      <head>
5          <title>Embedded-Image-Test</title>
6      </head>
7      <body>
8          <h1>Beispiel für eingebettes Bild</h1>
9          Dies ist ein Text. <br />
10         <br />
11         <!-- Hier beginnt das eingebettete Bild -->
12         <iframe height="400" width="500"
               src="http://192.168.1.222:8080/pixLibJavaServer/embeddedImage/1" scrolling="no"></iframe>
13         <!-- Hier endet das eingebettete Bild -->
14         <br />
15         <br />
16         Dies ist noch ein Text!
17     </body>
18 </html>
```

Quelltext 5.1: Eingebettes Bild

5.2. Architektur

Für die Anwendung wurde eine Architektur ausgewählt, die problemlos mit allen drei Sprachen und den zugehörigen Frameworks umgesetzt werden kann. Zum Einsatz kommt eine Schichtenarchitektur, die an den Schnittstellen zu anderen Systemen mit Abstraktionsschichten arbeitet, so dass diese austauschbar sind, ohne dass andere Schichten als die Abstraktionen angepasst werden müssten. Die Architektur für den Server-Teil ist in allen Varianten identisch. Die Schichten beginnen mit der REST-Schnittstelle, führen in die Logik mit den zugehörigen Kontrollklassen und Anwendungsobjekten und schließlich über Abstraktionsschichten zur Persistenz.

[1] Fenster in einer Webseite, die beliebige Webseiteninhalte einbetten kann.

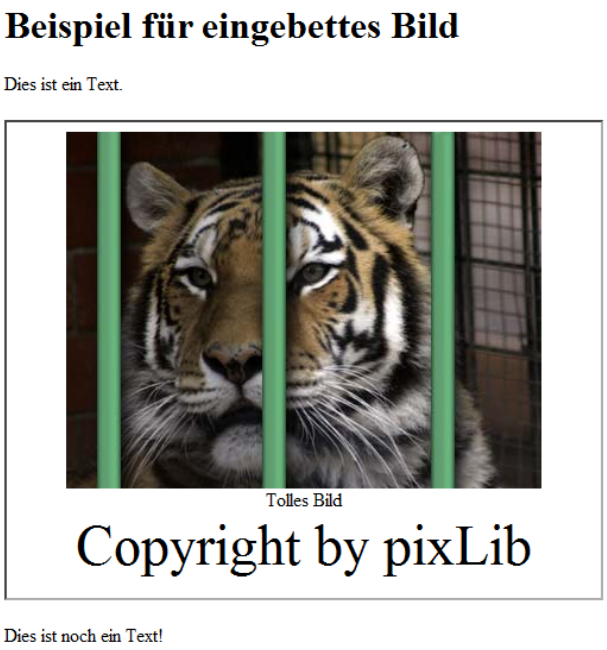

Abbildung 5.2.: Eingebettetes Bild

Server-Architektur Grundsätzlich übernehmen die Frameworks die Kommunikationsschicht, so dass eine von REST unabhängige Anwendung hinter dem Framework angesetzt wird. Es sollte daher generell möglich sein die Schnittstelle zwischen Client und Server zu entfernen um eine klassische Webanwendung oder eine auf SOAP basierte Lösung daraus zu machen. Zwischen der Kommunikationsschicht und der Geschäftslogik wurde auf eine weitere Abstraktionsebene verzichtet, da zum einen die Schnittstelle meist sowieso diejenige ist, die getauscht wird und nicht andersherum und zum anderen, weil es zum Teil schon schwer genug ist die Frameworks mit der eigenen Anwendung zu verknüpfen, da würde eine zusätzliche Schicht eher mehr Probleme machen, als sie nutzen würde.

Die Geschäftslogik besteht aus einer Sammlung von Kontrollklassen und Datenobjekten, die zusammen die Steuerung der Anwendung und der logischen Arbeitsabläufe (beispielsweise die Funktion *exzellente Bilder*) übernehmen. Hier werden in Anlehnung an die Prinzipien von REST keine Benutzersitzungen oder ähnliches verwaltet.

Die Implementierung der Persistenzschicht setzt sich aus der Datei- und der Datenbankschicht zusammen. Da die Bilder nicht in der Datenbank, sondern im Dateisystem gespeichert werden sollen, ist diese Trennung notwendig. Die Metadaten

der Bilder und alle sonstigen von der Anwendung benötigten Daten werden in der Datenbank gespeichert. Beide Teile sind wiederum in zwei weitere Schichten unterteilt. Die unterste Schicht kann ein Framework (beispielsweise Java Database Connectivity (JDBC) für Java, PHP Data Objects (PDO) für PHP oder ein Storage Resource Broker (SRB) [siehe Shen u. a., 2001] für ein beliebiges Dateisystem) sein. Die nächst höhere Schicht bildet eine Abstraktionsschicht zwischen der Anwendung und dem eingesetzten Persistenz-Framework. Auf diese Weise kann das Persistenz-Framework ausgetauscht werden, ohne dass die Geschäftslogik angepasst werden muss. Eine Übersicht der Server-Architektur wird in Abbildung 5.3 gezeigt.

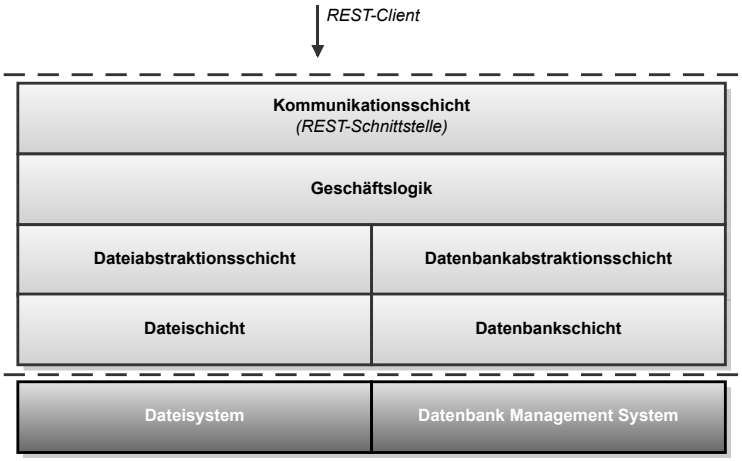

Abbildung 5.3.: pixLib Server-Architektur

Client-Architektur Die Web-Clients unterscheiden sich grundlegend in der obersten Schicht von den klassischen nativen Clients. Bei ersteren ist ein Webbrowser die oberste Schicht, die direkt mit dem Anwender interagiert. Bei der Schichten-Architektur gibt es ansonsten keine Unterschiede zwischen den Client-Varianten. Der Browser, der bei den webbasierten Varianten die Präsentation und die Interaktion mit dem Benutzer übernimmt, fällt bei der nativen Variante weg. Diese Aufgaben übernimmt dort eine Darstellungsschicht mithilfe der Benutzerschnittstelle des Betriebssystems.

Die Darstellungsschicht hängt stark mit Kontrollklassen der Darstellung zusammen, da diese Kontrollklassen die dargestellten Elemente steuern. Sie sollen beispielsweise Funktionalitäten wie das Ausgrauen eines Buttons in bestimmten Konstellationen übernehmen. Die client-spezifische Logik besteht aus einer Sammlung von Kontrollklassen und Datenobjekten, die zusammen die Steuerung des Clients übernehmen und bei Bedarf Anfragen über die REST-Schnittstelle ausführen. Hier befindet sich unter anderem auch die Sitzungsverwaltung, die an die Clients gebunden ist. In dieser Sitzungsverwaltung werden beispielsweise Informationen wie Benutzername, Passwort und der anzusprechende Webservice gespeichert.

Die Kommunikationsabstraktionsschicht bildet eine Abstraktion zwischen der Client-Anwendung und dem REST-Framework, damit das Framework (oder auch die gesamte Technologie dahinter) mit einfachen Mitteln ausgetauscht werden kann. Hierfür muss dann ebenfalls höchstens die Abstraktionsschicht angepasst werden und nicht der Client selber. Eine Übersicht der Client-Architektur wird in Abbildung 5.4 gezeigt.

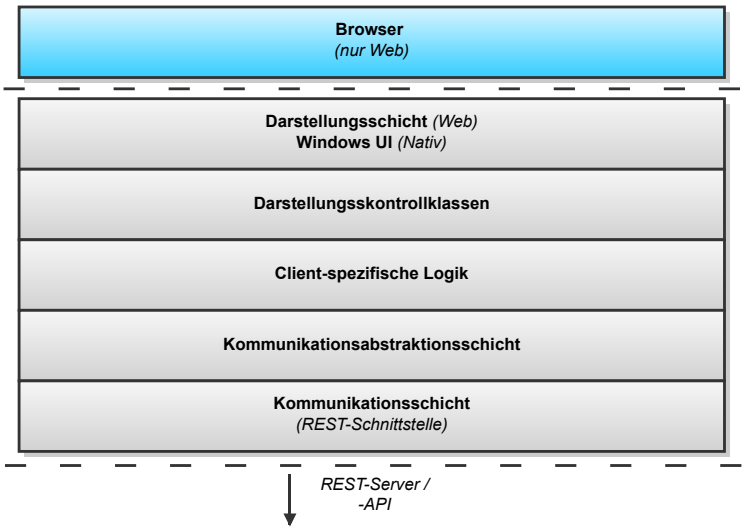

Abbildung 5.4.: pixLib Client-Architektur

5.2.1. Vereinheitlichungen

Die Klassen wurden weitestgehend so implementiert, dass sie nur durch Anpassungen der Syntax und wenigen Eigenheiten, möglichst einfach in andere Sprachen zu portieren sind. Dieses Vorgehen hat den Vorteil, dass nicht jede Variante der Anwendung komplett neu implementiert werden muss. Aufgrund der Tatsache, dass es hauptsächlich um die Frameworks und die Schnittstelle geht, ist dieses Vorgehen völlig ausreichend.

Die Datenhaltungen werden von allen Varianten gleichermaßen verwendet. Das betrifft sowohl die Datenbank, als auch das *Repository* für die hochgeladenen Bilder. So wird das Testen einfacher, da jede Variante der Anwendung dieselben Daten besitzt und verwenden kann. Dies gilt ebenso für die Datenbankabfragen. Die Konfigurationsdateien, die notwendigerweise vorhanden sein müssen (beispielsweise für die Zugangsdaten zur Datenbank), wurden in XML geschrieben, da heutzutage in nahezu jeder Programmiersprache einfach zu bedienende APIs vorhanden sind, um solche Dateien zu verarbeiten. Mehr Informationen über den Aufbau dieser Konfigurationsdateien stehen im Anhang B.

5.2.2. REST-Schnittstelle

Eine Schnittstelle kann auf verschiedene Art und Weise dokumentiert bzw. dargestellt werden. Bei einer grafischen Darstellung bietet sich eine Variante der UML [siehe Object Management Group, 2005] an. Alternative, durch Maschinen lesbare Arten eine REST-Schnittstelle zu beschreiben, sind beispielsweise die WSDL oder die Web Application Description Language (WADL) [siehe Hadley, 2009].

Für *pixLib* wurde auf den Einsatz von WSDL und WADL verzichtet und dafür eine Variante der UML verwendet. Diese ist zur Repräsentation der Zusammenhänge und der Übersicht besser geeignet, zumal hier keine maschinenlesbare Schnittstellendefinition notwendig sind. Die Ressourcentypen sind als Schnittstelle dargestellt und Ressourcen als Klassen. Jede Ressource beschreibt in ihrem Namen den zugeordneten URI und als Methoden alle HTTP-Methoden.

Beim Entwurf der Schnittstelle wurden weitestgehend die Tipps und Best Practices aus Kapitel 2 angewandt, um eine möglichst gute *RESTful*-Schnittstelle zu entwerfen. Als Leitfaden diente ebenfalls die Ausarbeitung von Marr [2006], die den Entwurf anhand eines einfachen Beispiels durchgängig beschreibt.

Typisierung von Ressourcen

Die Identifikation und die Typisierung der Ressourcen wurden in Anlehnung an Tilkov [2009a] durchgeführt. Die Typisierung dient dem besseren Verständnis der Schnittstelle und der Funktion der Ressource. Die Ressourcen können in die folgenden Typen eingeteilt werden:

Primärressourcen Primärressourcen sind meistens bei einem klassischen Entwurf Kandidaten für persistente Entitäten. Diese Ressourcen müssen nicht zwangsläufig `1:1` Datensätzen in einer Datenbank oder einem Objekt in der Anwendung entsprechen. Sie sind völlig losgelöst von der Implementierung und Speicherung und können daher auch aus komplexen Beziehungen bestehen (beispielsweise mehrere Tabellen und / oder Einträgen).

Subressourcen Diese sind Teil einer anderen Ressource. Dies kann beispielsweise eine Primärressource oder auch eine Subressource sein. Es gilt abzuwägen, ob diese Ressource als eigene Subressource implementiert oder in einer bestehenden Ressource eingebettet werden sollte. Diese Überlegung sollte sich der Frage widmen, ob die Ressource direkt adressierbar sein soll oder nicht.

Listen Für die meisten Primärressourcen gibt es Listen, was aber keine Vorschrift darstellt. Für Listen können die folgenden Untertypen definiert werden, um die Anzahl der Daten einzuschränken:

- **Filter:** Anhand von Filterkriterien werden bestimmte Einträge aus der Liste herausgefiltert bzw. überhaupt erst hereingenommen.
- **Paginierung:** Mittels der Paginierung können Ergebnisse seitenweise präsentiert werden.

Projektionen Eine Projektion enthält nur einen Teil der verfügbaren Informationen einer Primärressource oder eines Listenelements. Hierdurch kann die zu übertragende Datenmenge eingegrenzt werden, indem für den entsprechenden Kontext unwichtige Informationen nicht übertragen werden.

Aggregationen Eine Aggregation kombiniert Attribute verschiedener Ressourcen in eine einzige Ressource, um die Anzahl der Interaktionen zwischen Client und Server zu verringern.

Aktivitäten Aktivitäten können wie Arbeitsabläufe einen bestimmten Schritt innerhalb einer Verarbeitung oder aber auch ein ganzer Arbeitsauftrag sein.

Konzeptressourcen Diese Ressourcen sind keine Ressourcen im eigentlichen Sinne, denn sie repräsentieren nicht die Ressource selbst, sondern nur ein Konzept. Sie können beispielsweise anstatt unterschiedlicher Formate einer Ressource verwendet werden. Dies kann eine Person sein, die ein Bild und eine Visitenkarte hat. Hier hat das Bild und die Visitenkarte einen Bezug zu der Identität der Person, aber keins von beidem „ist“ diese Person.

Für *pixLib* wurde versucht die Typisierungen und die hierarchische Strukturierung der Ressourcen über die Schnittstelle möglichst gering zu halten, da diese zwar gute Erkenntnisse für den Schnittstellenentwurf bringen würden, aber für das eigentliche Ziel dieser Arbeit wenig Nutzen bringt. Fielding selbst beschreibt Ressourcen viel abstrakter, so dass er von keiner Typisierung spricht [vergleiche Fielding, 2000]. Als Mime-Types für die *Content Negotiation* unterstützt *pixLib* nur XML. Eine Ausnahme bildet die Ressource, die für das Hoch- und Herunterladen von Bildern zuständig ist. Diese Ressource unterstützt Joint Photographic Experts Group (JPEG), Portable Network Graphics (PNG) und Graphics Interchange Format (GIF). Es sollen bei dieser Architektur im Server keine Informationen für die Oberfläche vorhanden sein. Für die XML-Dokumente, die bei den Anfragen und Antworten verschickt werden, wurde keine explizite Schemadefinition erstellt, da dies sehr aufwendig ist und keine weiteren Erkenntnisse für diese Ausarbeitung bringen würde.

Ressourcen-Entwurf

Bei *pixLib* wurde die Schnittstelle in drei größere Ressourcengruppen unterteilt, die sich generell auf dieselben Objekte beziehen. Somit entstand eine Unterteilung in `Users` (Benutzer), `Libraries` (Alben) und `Images` (Bilder). Die Ressourcengruppe `Libraries` ist in Abbildung 5.5 dargestellt. Die gesamte Schnittstelle befindet sich als Vektorgrafik auf der beigefügten CD (siehe Anhang E).

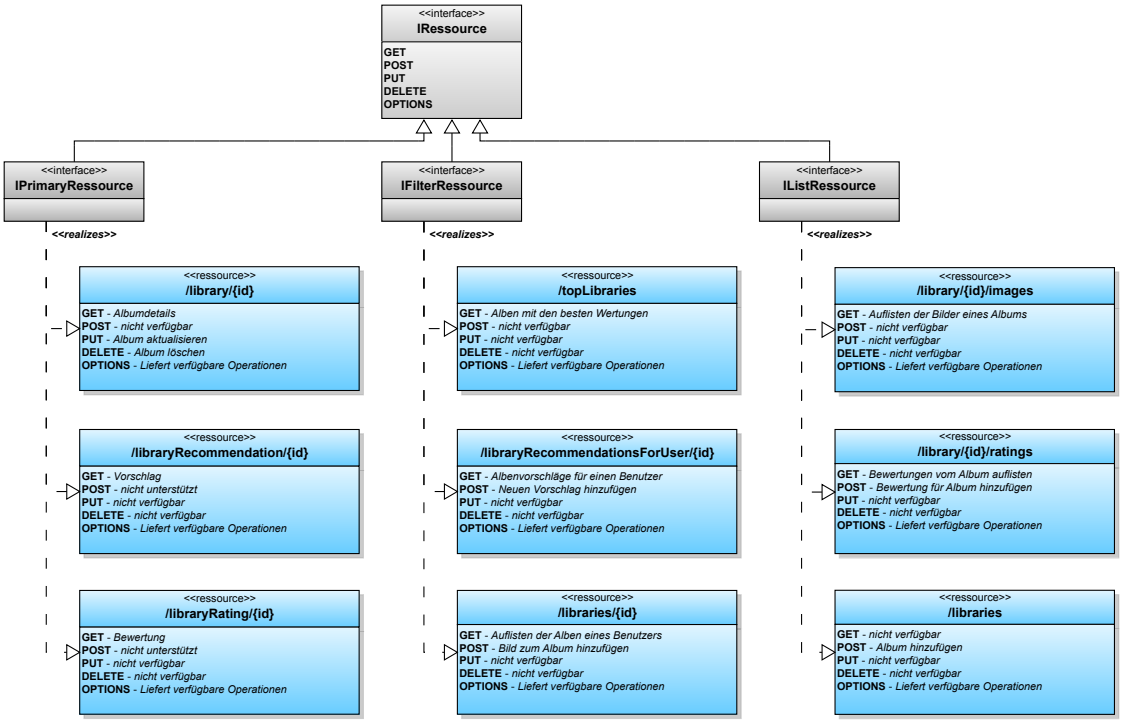

Abbildung 5.5.: pixLib Ressourcengruppe Libraries

Dabei umfasst eine solche Ressourcengruppe Primär-, Listen- und Filterressourcen. Die anderen beiden Ressourcengruppen ähneln dieser sehr stark. Der größte Unterschied liegt in der Anzahl der Ressourcen, da auf Benutzer beispielsweise weniger Filter oder andere Repräsentationen benötigt werden als für Alben oder Bilder. Auf komplexere Strukturen und Subressourcen wurde bewusst verzichtet, um die Beispielanwendung einfach und die Komplexität gering zu halten. Der URI-Entwurf der REST-Schnittstelle von *pixLib* wurde in Anlehnung an Tilkov [2009a] erstellt und ist in der Tabelle 5.1 dargestellt.

Ressource	URI	Methode(n)
Einstiegsressource	/	GET
Benutzerliste	/users	GET, POST
Benutzer	/user/{id}	GET, PUT, DELETE
Benutzer	/userByNickname/{nickname}	GET
Bild	/image/{id}	GET, PUT, DELETE
Kommentarliste/Bild	/image/{id}/comments	GET, POST
Bewertungsliste/Bild	/image/{id}/ratings	GET, POST
Verknüpfungsliste/Bild	/image/{id}/links	GET, POST
Bildvorschlagsliste/Benutzer	/imageReccomendationsForUser/{id}	GET, POST
Bildkommentar	/imageComment/{id}	GET
Bildbewertung	/imageRating/{id}	GET
Bestbewertete Bilder	/topImages	GET
Bildverknüpfungen/Benutzer	/imageLinksForUser/{id}	GET
Bildvorschlag	/imageRecommendation/{id}	GET
Bildverknüpfung	/imageLink/{id}	GET
Exzellente Bilder	/excellentImages	GET
Eingebettetes Bild	/embeddedImage/{id}	GET
Bilddatei	/binaryImage/{id}	GET, PUT
Potenziell exzellente Bilder	/maybeExcellentImages	GET, POST
Potenziell exzellentes Bild	/maybeExcellentImages/{id}	GET, PUT
Bestbewertete Alben	/topLibraries	GET
Albenvorschlagsliste/Benutzer	/libraryReccomendationsForUser/{id}	GET, POST
Neues Album	/libraries/	POST
Albenliste/Benutzer	/libraries/{id}	GET, POST
Album	/library/{id}	GET, PUT, DELETE
Bewertungsliste/Album	/library/{id}/ratings	GET, POST
Bilder/Album	/library/{id}/images	GET
Albumbewertung	/libraryRating/{id}	GET
Albumvorschlag	/libraryRecommendation/{id}	GET
29 Ressourcen		

Tabelle 5.1.: REST-Schnittstelle - URIs

Bei *pixLib* enthalten die Listenressourcen bereits alle Details zu jedem einzelnen Eintrag. In der Realität würden eher Projektionen mit weniger Details als Listenelemente verwendet werden. Beispielsweise einen Anzeigenamen, seinen URI und eventuell noch wenige aussagekräftige Attribute [vergleiche Tilkov, 2009a]. Da dieses Vorgehen keine Vorteile für die Implementierung oder die Vergleiche bietet, wurde auf entsprechende Projektionsressourcen verzichtet. Diese Projektionsressourcen sollten in solch einer Listenressource je-

weils eine Verknüpfung auf ihren Vorgänger und auf ihren Nachfolger haben [vergleiche Tilkov, 2009a], so dass eine Iteration der Liste möglich ist. Diese Funktionalität wird speziell bei einer *Paginierung* interessant, damit die nächste Seite mit dem richtigen Element beginnt. Die in *pixLib* vorhandenen Ressourcen bieten diese Funktionalität nicht, da dies zwar einem guten Entwurf entspricht, aber für die Implementierung einer *RESTful*-Anwendung nicht notwendig ist. Des Weiteren wird keine Paginierung eingesetzt.

Die Einstiegsressource wird in *pixLib* dreistufig implementiert, so dass je nach Authentifizierung weitere Verknüpfungen geliefert werden. Solch eine Ressource liefert alle Verknüpfungen, die als weitere Einstiegspunkte der Anwendung dienen sollen. Ein gutes Beispiel findet sich hierfür in Tilkov [2009a]. Die Einstiegsressource, in *pixLib* als `RootRessource` bezeichnet, ermöglicht es den gesamten Webservice bzw. die gesamte Anwendung über einen einzigen URI verfügbar zu machen.

Die erste Stufe beinhaltet wenige Verknüpfungen für alle Anwender, die sich bisher nicht authentifiziert haben. Weitere Verknüpfungen werden verfügbar, sobald sich ein Anwender erfolgreich authentifiziert. Sollte ein Anwender sich als Experte authentifizieren, so werden nochmals weitere Verknüpfungen geliefert. Mit einer komplexeren Rechteverwaltung kann auf diese Weise eine sehr dynamische Einstiegsressource entstehen. Die soeben beschriebene Einstiegsressource wird im Quelltext 5.2 dargestellt. Es handelt sich dabei um die einfachste Variante für nicht authentifizierte Benutzer. Der gezeigte Quelltext stellt die Einstiegsressource der *Jersey*-Implementierung unter Java dar, die *Recess*-Implementierung unter PHP unterscheidet sich nur minimal. Eine Einstiegsressource ist zwar nicht zwingend notwendig, um eine *RESTful*-Anwendung zu implementieren, allerdings wird dieses Vorgehen empfohlen, um das Prinzip *Hypermedia* zu unterstützen [vergleiche Tilkov, 2009a].

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <serviceDescription xml:base="http://192.168.1.222:8080/pixLibJavaServer/">
3   <link href="/topLibraries"     rel="topratedlibraries"  />
4   <link href="/topImages"        rel="topratedimages"     />
5   <link href="/excellentImages"  rel="excellentimages"    />
6 </serviceDescription>
```

Quelltext 5.2: pixLib Einstiegsressource

Jede Ressource erbt von der Klasse `RessourceSkeleton`, die zusätzliche Funktionalitäten für eine Ressource zur Verfügung stellt. Die Motivation für dieses Vorgehen ist, möglichst viel gemeinsamen Code zu faktorisieren, damit die durchzuführenden Implementierungen der einzelnen Ressourcen gering und wenig redundant ausfällt. Generell besitzt jede Ressource ihre eigene Kontrollklasse, welche die Verarbeitung der unterstützten HTTP-Methoden steuert. Als Ausnahme sind zum Teil die Subressourcen in den Kontrollklassen der zugehörigen Primärressourcen enthalten, so dass die Anzahl der Kontrollklassen übersichtlich bleibt. Wäre die Struktur komplexer oder automatisch generiert, sollten auch diese Subressourcen ihre eigenen Kontrollklassen erhalten. Eine weitere Besonderheit ist der Entwurf der Bildressource, die im Anhang B näher erläutert wird.

URI-Entwurf

Da URIs für Menschen lesbar und verständlich sein sollten [vergleiche Richardson und Ruby, 2007], wurde beim URI-Entwurf Wert auf eine einheitliche und einfache Struktur gelegt. Eine saubere Strukturierung erleichtert die zuvor genannten Ziele erheblich. Ob Parameter bzw. Informationen in Kopfzeilen oder direkt in URIs abgelegt werden, ist eine stark fallbezogene und zum Teil auch subjektive Entscheidung. Richardson und Ruby [2007] beschreibt, dass Informationen für bedingte Ressourcenrepräsentationen bei einer *RESTful*-Anwendung sowohl in den HTTP-Kopfzeilen, als auch in den URIs enthalten sein dürfen. URIs haben dabei den entscheidenden Vorteil, dass diese verschickt oder abgespeichert werden können. Die Kopfzeilen einer Anforderung gehen dabei allerdings verloren. Eine weitere Unterscheidung beim URI-Entwurf wird bei der Struktur gemacht. So kann ein URI wie eine Verzeichnisstruktur `http://host/ressource/{id}/top10` aufgebaut sein, oder alternativ mit Parametern `http://host/ressource?id={id}&filter=top10`. Ebenfalls ist eine Mischung aus beiden Varianten möglich und durchaus üblich `http://host/ressource/{id}/?filter=top10`. Alle drei Varianten sind *RESTful*.

Die Authentifizierung wurde in die Kopfzeilen verlagert, damit diese Daten auf jeden Fall beim Speichern oder Versenden eines URI verloren gehen und damit die Authentifizierungsinformationen nicht sofort ablesbar sind. Für die Ressourcen von *pixLib* wird ausschließlich die verzeichnisbasierte Variante als URI-Struktur verwendet. Im Anhang B wird das verwendete Authentifizierungskonzept beschrieben. Hüffmeyer [2010] beschreibt, dass ein falscher URI-Entwurf ungewollte Nebeneffekte haben kann. Beispielsweise würde ein `GET` auf `http://www.example.org/deletePerson?id=1234` vermutlich die Person mit der ID `1234` löschen, anstatt mit einem `DELETE` auf `http://www.example.org/person?id=1234`. So fasst ein menschlicher Leser den URI als eine Lösch-Anforderung aufgrund ihres Namens auf, die Maschine allerdings geht wegen dem Zugriff durch eine `GET`-Methode davon aus, dass diese Operation idempotent ist. Ein weiteres Problem dabei ist, dass solch ein Aufruf gecached werden kann, was bei einer Löschung sicherlich nicht erwünscht ist. Daher ist auch beim URI-Entwurf einer gewisse Sorgfalt einzuhalten.

Verknüpfungen

Verknüpfungen werden mit bis zu drei Attributen dargestellt, wobei der Medien-Typ des Ziels optional ist. Der Quelltext 5.3 stellt solch eine Verknüpfung dar. Für Verknüpfungen werden bei *pixLib* die folgenden Attribute verwendet, wobei die Reihenfolge beliebig sein kann:

rel Dieses Attribut legt einen Namen fest, damit diese Verknüpfung eindeutig zugeordnet werden kann. Dies erleichtert einem Client die Verarbeitung.

href Ein absoluter URI zu der verknüpften Ressource enthält dieses Attribut.

type (Optional) Sofern es erwünscht ist, kann dieses Attribut dafür verwendet werden, um dem Client mitzuteilen, welchen Medientyp die verknüpfte Ressource besitzt.

```
1 <!-- Verknuepfung mittels Attributen im XML-Tag -->
2 <link rel="linkName" href="http://link" type="application/xml" />
```

Quelltext 5.3: Verknüpfungen von Ressourcen

Jede Ressource besitzt einen URI auf sich selber, die so genannte `self`-Verknüpfung und beliebig viele Verknüpfungen zu anderen Ressourcen. Besonders sind in *pixLib* die Verknüpfungen zwischen Bildern und Benutzern, bei denen beliebige Benutzer als „auf dem Bild befindlich“ markiert werden können. Für solch eine Verknüpfung wird eine eigene Ressource verwendet und nicht nur eine einfache Verknüpfung von Ressourcen. Diese Entscheidung beruht auf der Tatsache, dass zu solch einer Verknüpfung zusätzliche Informationen gespeichert und abgerufen werden sollen (beispielsweise *wer* hat die Verknüpfung *wann* erstellt?). Des Weiteren sollen diese Verknüpfungen unabhängig von der Bearbeitung eines Bildes oder eines Benutzers sein und daher nicht über die Bild- bzw. die Benutzer-Ressource bearbeitet werden können.

5.2.3. Benutzer-Schnittstelle

Die Benutzer-Schnittstelle wurde übersichtlich und einfach in Bezug auf die Funktionen gehalten. Keine dieser Schnittstellen bietet den vollen Funktionsumfang der Server-Implementierungen. Diese Tatsache ist darin begründet, dass die Clients nur die grundsätzliche Kommunikation mit den Server-Implementierungen zeigen sollen. Zusätzlich sollen sie zeigen, dass sie aufgrund der Eigenschaften einer *RESTful*-Anwendung beliebig zwischen den Server-Implementierungen wechseln können.

Weboberfläche Damit die zu implementierenden Weboberflächen für einen Anwender leicht und verständlich zu bedienen sind, muss ein übersichtliches Menü existieren, welches die möglichen Grundfunktionen darstellt. Des Weiteren sollte eine Information vorhanden sein, wo der Anwender sich gerade befindet, wie beispielsweise der Pfad `Albumbesitzer>Albumname>Bildname` das Bild *Bildname* aus dem Album *Albumname* des Besitzers *Albumbesitzer* zeigt. Verfügbare Verknüpfungen, die von der aktuell angezeigten Ressource ausgehen, sollten in der Oberfläche dargestellt werden, so dass diese direkt verwendet werden können. Im Inhaltsbereich der Weboberfläche sind Listen, Alben und Bilder anzeigbar, wobei diese gut lesbar und übersichtlich gestaltet sein sollen.

Native Windows-Oberfläche Diese Anwendung soll einfach und intuitiv zu bedienen sein. Dabei sind nur die nötigsten Elemente einfach und übersichtlich darzustellen. Eine klare Gruppierung zwischen verschiedenen Funktionsbereichen soll kenntlich gemacht sein, so dass ein Anwender schnell und einfach zusammenhängende Funktionalitäten bzw. Felder erkennen kann. Die Verwendung eines Menüs oder einer Hilfeseite ist bei dieser Anwendung nicht notwendig.

5.3. Implementierung

Die Server-Varianten wurden alle drei mit dem vollen beschriebenen Funktionsumfang implementiert. Hierbei gab es kleine Abweichungen, die in Kapitel 5.6 mit Ursache und Lösung beschrieben werden. Die Client-Varianten enthalten alle drei verschiedene Funktionalitäten, so dass diese verschiedene Anwendungsgebiete abdecken. Für die Server-Varianten wurden die in Kapitel 4 gewählten Frameworks für die REST-Schnittstelle verwendet. Die Clients hingegen verwenden bis auf die Java-Variante keine Frameworks für die Kommunikation mit den Servern. Hier werden einfache HTTP-Klassen verwendet, um zu demonstrieren, wie einfach ein REST-Webservice anzusprechen ist. Die Java-Variante benutzt *Jersey* auch für den Client. Die Clients sind wie folgt unterteilt:

PHP - Benutzerportal Das Benutzerportal stellt die meisten Funktionen der Server-Varianten einem Anwender zur Verfügung. Dabei ist es möglich Alben und Bilder einzusehen, vorzuschlagen, zu bewerten und zu kommentieren. Zusätzlich können Alben und Bilder angelegt und gelöscht werden.

Java - Expertenportal Das Expertenportal dient den Experten dazu, potenziell *exzellente Bilder* einzusehen und diese als *exzellent* zu markieren oder abzulehnen. Des Weiteren können Bilder, die bereits als *exzellent* markiert wurden, betrachtet werden.

C# - Bilder hochladen Der in C# implementierte Client dient dem Hochladen von Bildern in ein Album eines bestimmten Benutzers. Hierbei muss der Benutzer sich authentifizieren, eines seiner Alben wählen und kann dann ein Bild von seinem Rechner mit einem Titel und einer Beschreibung hochladen.

Die Schichtenarchitektur der Clients (siehe Abbildung 5.4) wurde grundsätzlich in den Implementierungen verwendet. Die *Darstellungsschicht* besteht bei der nativen Anwendung aus Windows-UI-Komponenten und bei den Webanwendungen aus JavaServer Pages (JSP)[2] [siehe Delisle u. a., 2006] Seiten für die Java-Variante und aus der PHP Template

[2]Einfache Skriptsprache zur dynamischen Erzeugung von HTML.

Engine Smarty[3] bei der PHP-Variante. Die *Darstellungskontrollklassen* sind aus Gründen der Übersichtlichkeit mit der *Darstellungsschicht* verschmolzen, so dass es keine zusätzlichen Klassen gibt, die sich um die Steuerung der Benutzeroberfläche kümmern. Eine *Client-spezifische Logik* wurde bei den Webanwendungen auf verschiedene Kontrollklassen verteilt, die sich logisch nach Aufgabengebiet aufteilen. Bei der nativen Anwendung existiert eine einzige Klasse, die die gesamte Geschäftslogik des Clients enthält. Die *Kommunikationsabstraktionsschicht* besteht aus einer Factory[4] [vergleiche Gamma u. a., 2009], die das Instantiieren einer beliebigen Implementierung der Schnittstelle enthält. Durch diesen Ansatz kann jederzeit die *Kommunikationsschicht* durch eine andere ersetzt werden.

Bei der Schichtenarchitektur der Server (siehe Abbildung 5.3) wurde ebenfalls der Entwurf mit wenigen Anpassungen auf die Implementierung übertragen. Dabei wurde die *Kommunikationsschicht* und die *Geschäftslogik* größtenteils innerhalb der Ressourcen vereint, da viele Vorgänge keine eigene Logik für diese Implementierung benötigten. Als Ausnahme gelten die Arbeitsabläufe der Funktion *exzellente Bilder*, bei der eine eigene Kontrollklasse implementiert wurde. Um die Anwendung übersichtlicher zu halten, wurde die *Dateiabstraktionsschicht* weggelassen. Die *Datenbankabstraktionsschicht* wurde analog zu der *Kommunikationsabstraktionsschicht* der Clients mittels einer Schnittstelle und einer Factory realisiert.

5.4. Tests

Die Herausforderung beim Testen war der Einsatz verschiedener Programmiersprachen und Technologien und die Vermeidung der daraus entstehenden Redundanz von Testfällen. Dieser Abschnitt gibt einen groben Überblick über die verwendeten Techniken und Frameworks zum Testen der Anwendung. Weitere Details können dem Anhang D entnommen werden.

Zum Teil wurde der *Test-First*-Ansatz [siehe Wells, 2000] verfolgt, bei dem die Testfälle vor der eigentlichen Implementierung geschrieben werden. Dieser Ansatz soll die Befangenheit des Entwicklers durch die Implementierung weitestgehend umgehen. Zunächst wurde in jedem Framework exemplarisch jede zu verwendende HTTP-Methode für ein bis zwei Ressourcen implementiert, um das Verhalten zu beobachten. Bei diesen Implementierungen wurde der *Test-First*-Ansatz nicht verwendet. Für die weiteren Implementierungen hingegen wurde er eingesetzt.

[3] `http://www.smarty.net`
[4] Entwurfsmuster zur Verringerung der Kopplung von Klassen.

Unit-Tests Für die *Unit-Tests* wurden entsprechende Frameworks der verwendeten Programmiersprachen eingesetzt, die grundsätzlich alle auf *JUnit*[5] basieren. Diese Test-Frameworks bieten die Möglichkeit Testfälle mit erwarteten Ergebnissen zu definieren, so dass diese automatisiert ausgeführt werden können. Sie geben ebenfalls Auskunft über den Testverlauf. Für die Tests der Implementierung in Java wurde *JUnit* in der Version `4.5` verwendet. Die PHP-Variante wird mittels des Testframeworks *PHPUnit*[6] in der Version `3.5` getestet. Die Tests der C#-Variante wurden mit *NUnit*[7] in der Version `2.5` durchgeführt. Die vorhanden Quelltexte wurden nur rudimentär mit Unit-Tests kontinuierlich getestet, da das Hauptaugenmerk auf die Schnittstelle gelegt wurde und die Methoden sehr einfach gehalten wurden.

Test der Schnittstelle Um die Schnittstellen der drei Server zu testen, wurde ein kleines Framework auf Basis von *JUnit 4* und *Jersey* implementiert, um die große Menge an Testfällen handhaben zu können. Verwendet wird hierbei das Prinzip eines *Data Providers* [siehe Bergmann, 2009], bei dem eine Menge von Eingabewerten definiert und automatisisert in die Testklasse injiziert wird. Durch diesen Mechanismus kann relativ einfach eine große Menge an Testfällen durch verschiedene Parameter definiert werden, ohne diverse sich ähnelnde Methoden zu implementieren.

Gruppiert wurden die Tests nach den HTTP-Methoden, die implementiert werden. Folglich existieren fünf Test-Klassen, jeweils eine für `GET`, `PUT`, `POST`, `DELETE` und `OPTIONS`. Diese Tests sollen allerdings keine komplexen Abläufe testen, sondern die reine Funktionalität der Ressourcen. Im Anhang D wird dieses Framework detaillierter vorgestellt.

Test der Anwendungsfälle Das *Debugging* und *Tracing* wurde zusammen mit empirischen Tests kombiniert. Häufige Anwendungsfälle wurden durchgespielt und anschließend in der Datenbank und über Log-Einträge verifiziert, ob die Anwendung richtig funktioniert. War dies nicht der Fall oder waren Auffälligkeiten (beispielsweise Methodenaufrufe, Parameter, etc.) oder sogar ein Fehlverhalten des Programms ersichtlich, wurde mithilfe von zusätzlichen Ausgaben mögliche Ursachen gesucht. Nach Auffindung der Ursache wurde diese beseitigt und die Anwendung anschließend erneut getestet.

[5] `http://www.junit.org`
[6] `http://www.phpunit.de`
[7] `http://www.nunit.org`

5.5. Benutzung der Anwendung

In diesem Abschnitt wird übersichtsweise erklärt, wie die Anwendung *pixLib* verwendet werden kann. Dabei wird zunächst auf eine Verwendung ohne einen eigenen Client eingegangen und anschließend auf die Verwendung mit den Webclients und dem nativen Client für Windows.

Ohne Client Die Server können auch ohne einen extra Client verwendet werden. Aufgrund der Tatsache, dass nicht alle durch den Server angebotenen Funktionen durch die entwickelten Clients verwendet werden können, bietet sich diese Methode an, um diese Funktionen dennoch nutzen oder ausprobieren zu können. Eine gängige, einfache und sehr flexible Variante diese Zugriffe zu realisieren, bietet das Kommandozeilenprogramm *cURL*[8]. Eine weitere Möglichkeit, sehr einfach lesend auf Ressourcen zuzugreifen, ist ein klassischer Browser.

Webclients Die Webclients sind vom Aufbau der Benutzeroberfläche her identisch. Sie unterscheiden sich lediglich im Funktionsumfang. Aufgeteilt ist die Oberfläche in die vier logischen Bereiche *Kopfzeile*, *Lokationsanzeige*, *Inhalt* und *Fußzeile*. Diese vier Bereiche beinhalten dynamische Anzeigeelemente und werden in Abbildung 5.6 dargestellt. In der *Kopfzeile* befindet sich das Logo und ein Menü. Dieses Menü wird dynamisch bei jedem Aufruf neu generiert und beinhaltet Verknüpfungen, die durch die Einstiegsressource vorgegeben werden. Beim *Expertenportal* werden einige dieser Verknüpfungen herausgefiltert, so dass nur die vorhandenen Funktionen erscheinen.

Die *Lokationsanzeige* zeigt dem Anwender die aktuelle Position in der Anwendung. Sie ist in der Regel in zwei Stufen aufgeteilt, bei der die erste Stufe das aktuelle Modul namentlich dargestellt und in der zweiten Stufe die Auswahl in diesem Modul. Dies kann beispielsweise der Titel eines Bildes sein, welches gerade ausgewählt ist. Der *Inhalt* stellt die aktuellen Inhalte dar. Hier finden sich ebenfalls ausgehende Querverweise zu anderen Teilen der Anwendung. In der *Fußzeile* wird der zur Zeit ausgewählte Webservice dargestellt.

Beim *Benutzerportal* können fast alle Funktionen lesend ohne eine Anmeldung verwendet werden. Zum Hochladen oder Kommentieren von Bildern ist beispielsweise eine Anmeldung zwingend erforderlich. Das *Expertenportal* hingegen benötigt für alle Funktionen eine Anmeldung durch einen *Experten*. Solange kein Benutzer angemeldet ist, befindet sich im Menü am Ende der Eintrag „Anmelden“. Sobald ein Benutzer sich erfolgreich angemeldet hat, steht an dieser Stelle der Benutzername und der Eintrag „Abmelden“.

[8] `http://curl.haxx.se`

Abbildung 5.6.: Webclient - Oberflächenbereiche

Als besondere Aktion im *Expertenportal* ist das Zustimmen bzw. das Ablehnen von *exzellenten Bildern*. Ein Bild, welches als *exzellent* vorgeschlagen wurde, bekommt in der Detailansicht zwei zusätzliche Symbole. Über diese Symbole kann der Experte bestimmen, ob das Bild *exzellent* ist oder nicht. Ein Bild, welches bereits *exzellent* ist, wird in beiden Oberflächen mit einem entsprechenden Symbol in der Detailansicht gekennzeichnet. Diese drei Symbole sind in Abbildung 5.7 dargestellt.

Abbildung 5.7.: Symbole für exzellente Bilder

Das linke Symbol aus Abbildung 5.7, welches einen grünen Daumen nach oben zeigt, dient im *Benutzerportal* dazu, ein Bild als *exzellent* vorzuschlagen. Im *Expertenportal* hingegen dient er dem Markieren eines Bildes als *exzellent*. Der rote Daumen nach unten existiert nur im *Expertenportal*. Dieser ermöglicht es einem Experten ein Bild als *nicht exzellent* zu markieren. Das mittlere Symbol kennzeichnet in beiden Portalen ein Bild als *exzellent*. Da die Verwendung der Weboberfläche intuitiv möglich und sehr einfach gehalten ist, werden hier keine weiteren Details zur Benutzung dargestellt.

Nativer Windowsclient Bei diesem Client handelt es sich um eine reguläre Windows-Anwendung. Die Oberfläche ist in drei vertikal angeordnete Teile getrennt. Der oberste Teil *Benutzerinformationen*, die Felder *Benutzername* und *Passwort* zur Authentifizierung und Identifizierung eines Benutzers beim Server und dem Feld *Webservice* um einen bestimmen Server zu verwenden. Als Standard-Server ist die Java-Variante ausgewählt.

Der mittlere Teil *Benutzeralben* enthält die Liste aller Alben eines Benutzers, den *Titel* und die *Beschreibung* des aktuell ausgewählten Albums und den Knopf *„Alben laden“* zum Laden der Alben. Der letzte Teil *Bild hochladen* beinhaltet die Felder *Titel*, *Beschreibung* und *Bilddatei*, um ein hochzuladendes Bild zu spezifizieren. Über den Knopf *„. . . “* kann über einen Dateiauswahl-Dialog ein beliebiges Bild aus dem Dateisystem gewählt werden. Über den Knopf *„Bild hochladen“* wird das Bild dann an den Server übermittelt.

Die Anwendung ist von oben nach unten zu verwenden, also muss zunächst ein Benutzer angegeben werden, anschließend die Alben geladen und eins ausgesucht werden. Im letzten Schritt müssen Informationen zu einem Bild eingegeben werden. In Abbildung 5.8 wird die Oberfläche gezeigt.

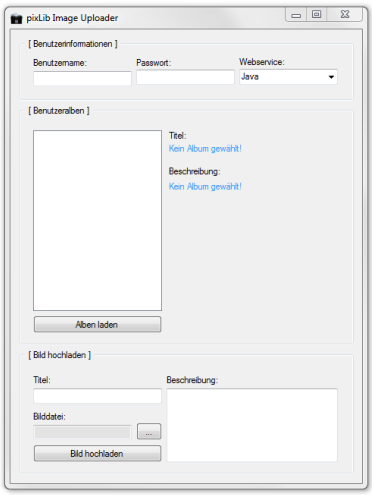

Abbildung 5.8.: Benutzeroberfläche - C#-Client

Fehlende Eingaben werden abgefangen und dem Benutzer mit einem Informationsdialog kenntlich gemacht. Diese Meldung wird in der Abbildung 5.9 dargestellt. Bei dieser Anwendung müssen die Felder *Benutzername* und *Passwort* gefüllt sein, damit die Albenliste geladen werden kann. Um ein Bild zu speichern, muss zusätzlich noch ein Album und eine Bilddatei ausgewählt werden. Die beiden Felder *Titel* und *Beschreibung* müssen gefüllt sein.

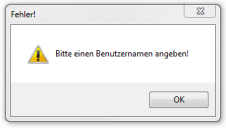

Abbildung 5.9.: Benutzeroberfläche - C#-Client - Fehler-Dialog

Fehlermeldungen vom Server oder Kommunikationsfehler werden dem Anwender ebenfalls mit Informations-Dialogen gemeldet. Das Programm schreibt seine Logdatei unter `C:/pixlib/logs/pixLib/client-csharp.log`. Dort können konkrete Ergebnisse und der Programmfluß eingesehen werden. Die Konfigurationsdatei mit den Informationen über die verschiedenen Webservices muss sich unter `C:/pixlib/config/client/webservice.config.xml` befinden, da der Client sonst keine Verbindung zu dem Server aufbauen kann.

5.6. Schwierigkeiten und Erfahrungen

Bei der Implementierung der Anwendung gab es einige Probleme, die sich grundsätzlich auf die Unterschiede der Frameworks und auf die Umsetzung der REST-Prinzipien beziehen. Letzteres ist darin begründet, dass es wenige gute Beispiele gibt, die als *RESTful* angesehen werden können.

5.6.1. Entwurf

In diesem Abschnitt wird gezielt auf die Schwierigkeiten und Erfahrungen beim Entwurf eingegangen. Dabei wird zunächst die REST-Schnittstelle, anschließend die verwendeten Frameworks und zum Schluss, die von den Frameworks unabhängigen, Implementierungen betrachtet.

REST-Schnittstelle Das Identifizieren der Ressourcen gestaltete sich in zweierlei Hinsicht als Herausforderung. Zum einen war diese Art der Herangehensweise und die Art der Schnittstelle im Vergleich zu klassischen RPC-Schnittstellen aufgrund der REST-Eigenschaft *Uniforme Schnittstelle* kaum vergleichbar und erforderte eine andere Denkweise. Dabei ist der Aspekt einer ROA gegenüber klassischen Architekturen gemeint. Zum anderen war es schwierig zu definieren, welche Ressourcen wie verwendet werden sollen und somit den Typ der Ressource zu definieren. Um eine Ressource zu typisieren, können die folgenden Fragen gegebenenfalls behilflich sein:

- Soll eine Ressource als Liste darstellbar sein (beispielsweise eine Liste von Benutzern)? Dann sollte eine entsprechende Listenressource zur vorhandenen Primärressource erstellt werden.

- Soll eine Ressource unabhängig von anderen Ressourcen neu angelegt werden können? Dann sollte eine entsprechende Listenressource erstellt werden.

- Soll eine Liste auf bestimmte Einträge anhand von bestimmten Kriterien reduziert bzw. gefiltert werden können (beispielsweise alle Benutzer die am 10.11. Geburtstag haben)? Dann sollte eine Filterressource verwendet werden.

- Soll eine Ressource veränderbar oder gar löschbar sein? Dann sollte eine Primärressource angelegt werden, ansonsten könnte es unter Umständen ausreichen diese Informationen in einer anderen Ressource eingebettet zu verwenden.

- Soll eine Ressource verknüpft werden können (beispielsweise um die Verknüpfung zu speichern oder um sie zu versenden)? Dann sollte für diese Ressource eine eigene Primärressource angelegt werden.

Dennoch sind einige Typ-Entscheidungen nur rein subjektiv durchführbar und andere wurden für diese Ausarbeitung außen vor gelassen (beispielsweise Subressourcen), um die Komplexität der Schnittstelle nicht unnötig zu erhöhen.

Datenbankabfragen Die Datenbankabfragen werden über *vorbereitete Anweisungen* (engl. Prepared-Statements) [siehe Oracle, 2010] an das RDBMS gestellt. Optimal wäre eine einheitliche Definition dieser Abfragen (beispielsweise über eine XML-Datei) oder das Verwenden von *gespeicherten Prozeduren* (engl. Stored-Procedures) [siehe Oracle, 2010]. Für die derzeitige Implementierung wurden die Abfragen einmalig definiert und in alle drei Server-Varianten fest eingebaut. Die zuvor erwähnten besseren Lösungen sind im Sinne des *Software Engineerings* zwar gut geeignet, aber haben keinen Einfluss auf die Vergleiche der Frameworks.

5.6.2. Verwendung der Frameworks

Bei der Verwendung der Frameworks gab es unterschiedliche Probleme, die größtenteils spezifisch für die Frameworks zu sehen sind. Manche Details des Entwurfs lassen sich mit einem Framework leichter und mit einem anderen schwerer umsetzen.

Allgemeines

Der C# und der PHP-Client verwenden keine Frameworks oder Bibliotheken für die Kommunikation mit den Servern. Es werden reine HTTP-Klassen verwendet, die bereits zur Verfügung stehen. Dies soll zeigen, dass es mit einfachen Mitteln möglich ist auf einen REST-Webservice zuzugreifen. Der Java-Client verwendet für die Kommunikation mit dem REST-Webservice die Client-Bibliotheken von *Jersey*.

Die automatische Quelltext-Erzeugung von *Recess* zeigt, dass es generell möglich ist Ressourcen und Datenmodelle automatisiert zu erzeugen. Dennoch steckt noch viel mehr Potenzial in diesem Vorgehen, welches von *Recess* noch nicht voll ausgenutzt wird. Dies würde sich mit einem Metamodell ergänzen [vergleiche Schreier, 2010]. Für komplexere Ressourcen, Datenmodelle und Arbeitsabläufe werden deutlich mehr Informationen benötigt als die, die *Recess* verwendet. Diese müssten auf einer Entwurfsebene, die noch nicht die betreffenen Zielplattformen betrachtet, bzw. in einer Metasprache definiert werden. Anschließend wäre es möglich, daraus beliebige Implementierungen zu erzeugen.

Diese Quelltext-Erzeugung wird durch Hilfsklassen unterstützt, die bereits viele Aufgaben, die beispielsweise in den Ressourcen stattfinden, abstrahiert zur Verfügung stellen. So muss eine Ressource die Aufgaben nur an Hilfsklassen delegieren und enthält dadurch deutlich weniger Zeilen Quelltext. Solche eine Hilfsklasse könnte wie der `RessourceSkeleton` von *pixLib* aussehen, der Aufgaben, wie die Authentifizierung oder das Generieren von Verknüpfungen zwischen Ressourcen, übernimmt. Diese Abstraktion wird dadurch ermöglicht, dass die Ressourcen zu einem Großteil identischen Quelltext enthalten. Die Quelltext-Erzeugung würde daher dem Entwickler viel Fleißarbeit abnehmen, speziell bei größeren Anwendungen. Zusätzlich würde die Qualität der Quelltexte verbessert und die Fehlerwahrscheinlichkeit verringert.

PHP - Recess

Einige der von *Recess* mitgelieferten Funktionalitäten werden für *pixLib* nicht benötigt, diese können aber für andere Anwendungen sehr wertvoll sein. Dazu zählen beispielsweise die automatisch generierten Datenmodelle und Ressourcen anhand von Tabellenstrukturen. Bei der Verwendung des Frameworks sind die folgenden Punkte aufgefallen:

Content Negotiation Die Verwendung von XML als Format für Anforderungs- und Antwortdokumente wird generell nicht vom Framework unterstützt. Ein Entwickler bietet eine entsprechende Klasse an, welche die automatisierte Umwandlung von und in XML unter *Recess* ermöglicht. Diese Klasse ist aus einem Versionskontrollsystem[9] beziehbar. Eine kurze Anleitung zur Verwendung dieser Klasse befindet sich im Anhang C. Ohne weitere Anpassungen der Klasse bietet sie **keine** Unterstützung für verschachtelte Objekte, Attribute innerhalb eines XML-Tags oder eine Repräsentation von booleschen Werten als `true` bzw. `false`. Die erste Schwäche kann eliminiert werden, indem eine Rekursion eingeführt wird, sobald das aktuell zu schreibende Attribut ein Objekt ist. Eine Anpassung in Bezug auf boolesche Werte kann entweder in dieser Klasse geschehen oder in den Datenmodellen die solche Attribute enthalten.

[9] `git://github.com/rday/recess.git`

HTTP-Status-Codes Für die gängigsten HTTP-Status-Codes, wie beispielsweise `200 Ok` oder `201 Created` existieren bereits entsprechende Methoden in einer abstrakten Oberklasse, dem `AbstractController`. Diese können aufgerufen werden, damit die entsprechende Antwort generiert wird. Andere Status-Codes besitzen keine fertigen Methoden, so dass entweder solche Methoden implementiert werden müssen, oder diese Antworten müssen in der jeweiligen Methode instantiiert werden. Um einer einheitlichen Vorgehensweise zu folgen, wurden im `RessourceSkeleton` solche Methoden für `401 Unauthorized`, `403 Forbidden` und weitere implementiert. Diese wurde in Anlehnung an die Implementierung in *Jersey* durch das Werfen einer `Exception` realisiert.

Ressourcen Ressourcen können in *Recess* leider nur begrenzt strukturiert werden, da bei diesem Framework nur ein einzelnes Verzeichnis für Ressourcen spezifiziert werden kann, welches nicht rekursiv durchsucht wird. Somit müssen alle Ressourcen innerhalb einer Anwendung in demselben Verzeichnis liegen. Dies kann bei vielen Ressourcen schnell unübersichtlich werden. Weiterhin müssen Ressourcen von einer gemeinsamen Oberklasse, dem `AbstractController`, ableiten. Ein weiterer Nachteil ist, dass die zu verwendende Ressource bei jeder Anforderung erneut gesucht werden muss, indem die Annotationen der vorhandenen Ressourcen ausgelesen werden. Dies kann bei sehr vielen Ressourcen ein Performanzproblem darstellen.

Java - Jersey

Das Framework *Jersey* bietet viele nützliche Funktionen, von denen die meisten verwendet wurden. Bei der Verwendung des Frameworks sind die folgenden Punkte aufgefallen:

Content Negotiation Die Verwendung von JSON als Format für Anforderungs- und Antwortdokumente funktioniert in der verwendeten Version nicht wie beschrieben. Diverse Internetquellen äußern sich zu diesem Problem ganz unterschiedlich. Auf manchen wird beschrieben, dass es ausreicht nur den Typ in den Annotationen anzugeben und alles Weitere wird vom Framework übernommen. Andere Quellen wiederum schreiben, dass entsprechende *Provider* benötigt werden, damit die Umwandlung funktioniert. Möglicherweise waren unterschiedliche Versionen des Frameworks gemeint, dies wurde aus den Quellen jedoch nicht ersichtlich. Daher wurde auf JSON verzichtet, da es sich lediglich um ein *Kannkriterium* handelt (siehe Kapitel 3).

C# - WCF

Das Framework *WCF* bietet ähnliche Ansätze wie *Jersey*. Allerdings gibt es auch einige Unterschiede und somit andere Schwierigkeiten. Bei der Verwendung des Frameworks sind die folgenden Punkte aufgefallen:

Platzhalter in URIs Im Gegensatz zu *Jersey* ist es nicht möglich für Platzhalter innerhalb einer URI, beispielsweise eine `ID`, einen numerischen Datentyp zu wählen. Es kann nur der Datentyp `string` verwendet werden. Somit muss eine Umwandlung durch den Entwickler erfolgen. Bei *Recess* spielt dies aufgrund der *dynamischen Typisierung* keine Rolle.

Content Negotiation Mit den Parametern `RequestFormat` und `ResponseFormat` der Attribute `WebGet` oder `WebInvoke` können die Repräsentationsformate vorgegeben werden. Leider hatten diese Parameter bei den verwendeten Projekteinstellungen keine Wirkung. Es wurde immer ein XML-Dokument zurückgeliefert, allerdings ohne den notwendigen Starttag `<?xml ...?>`, daher konnte die Antwort nicht validiert werden. Zusätzlich wurden die expliziten Namensangaben für die XML-Elemente ignoriert. Durch das Attribut `[XmlSerializerFormat()]` kann das Serialisierungs-Framework gewechselt werden, so dass beide Probleme behoben werden.

URIs Das Framework *WCF* verhält sich beim Aufruf von URIs etwas anders als *Jersey* oder *Recess*. Ressourcen, für die ein Routing definiert wurde (vergleiche Anhang im Abschnitt C.3.2), müssen beim Aufruf in dem URI einen abschließenden Schrägstrich besitzen (beispielsweise `http://host/meineRessource/`), wobei alle untergeordneten Ressourcen keinen abschließenden Schrägstrich in dem URI besitzen dürfen (beispielsweise `http://host/meineRessource/1`). *WCF* stellt bei Nichtbeachtung dieser Regel zwar eine Weiterleitungs-Seite bereit, allerdings kommen die implementierten Clients damit nicht zurecht.

Kopplung der Frameworks

Aufgrund der Implementierung des gemeinsamen Architekturstils REST sollte es einfach sein die Schnittstellen konform zueinander zu implementieren. Bis auf kleinere Hürden hat sich diese Theorie auch in der Praxis bewahrheitet. Diese Tatsache erleichterte die Implementierung und Verwendung der Clients auf allen Server-Varianten. Die folgenden Punkte stellen die zuvor erwähnten Hürden dar:

HTTP-Status-Codes Bei Ressourcen, die zwar an sich existieren, aber so angesprochen werden, dass diese kein richtiges Ergebnis liefern sollten, antworten die Frameworks mit verschiedenen HTTP-Status-Codes. Damit ist gemeint, dass eine Ressource, deren einziger URI mit `<RessourceName>/{id}` definiert ist, aber nur durch `<RessourceName>` angesprochen wird. Hierbei wirft das Java-Framework *Jersey* einen `405 Not allowed` und das PHP-Framework *Recess* einen `404 Not Found` Status-Code. Beide Varianten sind korrekt, da die Definition [vergleiche Fielding u. a., 1999] der beiden Status-Codes entsprechende Freiheiten lässt.

`POST` und `PUT` Ein auf *Jersey* basierender Client meldet bei einer *Recess* Server-Anwendung nach einem `POST` bzw. `PUT` den Fehler `Der Medientyp "" konnte nicht geparst werden`. Der Vorgang selber wird erfolgreich abgeschlossen, nur die Antwort kann nicht richtig ausgelesen werden.

Weitere Probleme, wie beispielsweise die Einheitlichkeit der XML-Repräsentationen, werden hier nicht aufgeführt, da diese nicht direkt mit den Frameworks zusammenhängen.

5.6.3. Weitere Implementierung

Ein interessanter und ungewohnter Aspekt bei der Implementierung einer *RESTful*-Anwendung ist, dass die definierten Ressourcen in der Regel nicht direkt auf die Datenstrukturen der Anwendung abgebildet werden können. Diese Tatsache ist vor allem darin begründet, dass dieselben Informationen über verschiedene Ressourcen angesprochen werden können. Dabei kann die Darstellung dieser Informationen sich bei verschiedenen Ressourcen unterscheiden. Daher sollte auch der Schnittstellenentwurf sinnvollerweise unabhängig vom Datenbankentwurf gemacht werden.

Ein Beispiel hierzu wird bei *pixLib* bei den *Bildern eines Albums* gezeigt. Auf der Datenbankseite sind *alle* Bild-Metadaten in einer Tabelle abgelegt und anhand einer `id` einem Album zugeordnet. Aus Sicht der Hierarchie und üblicherweise auch in den Datenmodellen im Kern der Anwendung existiert ein Album, welches eine Liste der zugehörigen Bilder beinhaltet. Über eine REST-Schnittstelle kann üblicherweise eine Kombination der vorgestellten Varianten angeboten werden. Beispielsweise könnte eine Filterressource auf die Liste aller Bilder mit dem Einschränkungs-Kriterium der `id` des Albums verwendet werden. Alternativ kann die hierarchische Variante gewählt werden, bei der es ein konkretes Album als Ressource gibt, welches die Liste der Bilder oder auch konkrete Bilder aus dem Album als Subressource enthält. Weiterhin wäre es denkbar, dass die Liste der Bilder keine eigene Ressource darstellt, sondern nur als Verknüpfungen innerhalb der Repräsentation eines Albums auftaucht.

Das Problem der logischen Gruppierung von Ressourcen in Bezug auf die Datenmodelle, die Ressourcen und die Persistenzebene ist daher nicht trivial. Wichtig beim Entwurf ist, dass die Performanz und Skalierbarkeit, die durch REST erreicht werden kann, nicht behindert wird.

Eine weitere Schwierigkeit, die bei der Entwicklung viel Zeit kostete, war die Tatsache, dass die XML-Serialisierungen zwischen PHP und Java bzw. C# stark abwichen. Grundsätzlich geht es dabei um die Namen der Elemente und die Gruppierungen in der Hierarchie. Dies ist hauptsächlich dadurch begründet, dass die Serialisierungen unter Java und C# durch Annotationen gesteuert werden können und bei PHP nicht, zumindest bei dem hier verwendeten Vorgehen. Dadurch hatten die Clients Probleme die XML-Dateien zu verarbeiten, da sie unterschiedliche Strukturen und Namen beinhalteten. Aus diesem Grund weicht die Implementierung der Datenmodelle in der PHP-Variante von den anderen ab.

Als Herausforderung stellte sich auch die Verwendung des Prinzips *Hypermedia* auf Seite der Clients heraus. Das Problem bei diesem Prinzip ist, dass es bei der Verwendung eines zusätzlichen Clients, wie bei *pixLib*, die Verknüpfungen der Ressourcen nur zwischen dem Client und dem Server nutzbar sind. Dem Benutzer müssen die Verknüpfungen anderweitig zur Verfügung gestellt werden. Bei einem Client, der als Webanwendung implementiert ist, müsste der Client Verknüpfungen generieren, die von der Funktionalität her denen der Verknüpfung entsprechen, aber auf ein anderes Ziel zeigen. Der Anwender soll innerhalb des Clients bleiben und nicht durch eine Verknüpfung, die auf den Server zeigt, aus dem Client herausspringen. Hierfür ist es notwendig ein Mapping zu erstellen und diese Verknüpfungen geeignet zwischenzuspeichern.

Die Software wird schneller langsamer
als die Hardware schneller.
Niklaus Wirth - Informatiker (Schweiz), 1995

6

Abschließende Gegenüberstellungen

Die verwendeten Frameworks werden in diesem Kapitel, anhand der in dieser Ausarbeitung gesammelten Erfahrungen, miteinander verglichen. Dabei wird grundsätzlich die Benutzbarkeit für Entwickler und die Konformität zu REST betrachtet. Zunächst werden die drei verschiedenen Implementierungen von *pixLib* miteinander verglichen, wobei speziell auf die unterschiedlichen Handhabungen der Frameworks eingegangen wird. Abgeschlossen wird dieses Kapitel mit einer Gegenüberstellung der APIs von *pixLib* und zwei weiteren Web-Fotoalben, die ebenfalls eine REST-Schnittstelle anbieten.

6.1. Frameworks

Die Benutzbarkeit und die Konformität der verwendeten Frameworks zu REST spielen für diese Ausarbeitung eine große Rolle. Daher ist ein Vergleich der Frameworks untereinander und eine Untersuchung der Unterstützung der REST-Prinzipien notwendig.

6.1.1. Implementierung bzw. Realisierung von Ressourcen

Verglichen wird die Art und Weise, wie die Ressourcen der verwendeten Frameworks definiert werden. Damit die Ressourcen verwendet werden können, muss bei allen Frameworks mindestens eine HTTP-Methode, wie in Abschnitt 6.1.2 dargestellt, definiert werden.

PHP - Recess Eine Ressource wird unter *Recess* von der Klasse `Controller` abgeleitet, welche bereits einige Funktionalitäten bereitstellt. Eine Plain Old PHP Object (POPO) Klasse reicht nicht aus, da die Klassen durch das Framework überprüft werden, ob sie von der `Controller`-Klasse ableiten. Weiterhin müssen mindestens die Annotationen `!Prefix Routes` und `!RespondsWith` an die Klasse gebunden sein. Die Klassen unterliegen der Einschränkung, dass sie in dem definierten Verzeichnis für Ressourcen liegen müssen und nicht in einem Unterverzeichnis davon. Ein Beispiel für die Definition einer Ressource bei *Recess* ist im Quelltext 6.1 dargestellt. *Recess* sucht zur Laufzeit, also bei einer ankommenden Anfrage, nach der zu verwendenden Ressource.

```
1 <?php
2   Library::import('recess.framework.controllers.Controller');
3
4   /**
5    * !RespondsWith Layouts
6    * !Prefix Routes: /url
7    */
8   class MyResource extends Controller { /* ... */ }
9 ?>
```

Quelltext 6.1: Ressourcenvergleich - Recess

Java - Jersey Bei *Jersey* wird eine Ressource als Plain Old Java Object (POJO) angelegt. Erkannt wird diese vom Framework, sobald die Annotation `@Path` an die Klasse oder an mindestens eine Methode der Klasse gebunden wird. Die Position der Klassen innerhalb des Projektes ist beliebig. Ein Beispiel für die Definition einer Ressource mit *Jersey* ist im Quelltext 6.2 dargestellt. *Jersey* kann entweder automatisch beim Laden der Applikation nach Ressourcen suchen oder die Ressourcen werden von Hand im *Web Descriptor* definiert.

```
1 import javax.ws.rs.*;
2
3 @Path("/url")
4 public class MyResource { /* ... */ }
```

Quelltext 6.2: Ressourcenvergleich - Jersey

C# - Windows Communication Foundation (WCF) Die Realisierung von Ressourcen bei *WCF* ist ähnlich zu *Jersey*. Diese werden bei C# in Verbindung mit *WCF* über Attribute [siehe msdn, a], das Pendant zu Annotationen in Java, gesteuert. Eine Ressourcenklasse kann daher eine Plain Old Common Language Runtime Object (POCO) Klasse sein. Für die Namensgebung und Positionierung innerhalb des Projektes gibt es keine weiteren Einschränkungen, bis auf die durch die Programmiersprache vorgegebenen. Solch eine Ressource wird in Quelltext 6.3 dargestellt.

```
1 [ServiceContract]
2 [AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
3 [ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
4 public class MyResource { /* ... */ }
5
6 // ----------------------------
7 // Routing aus Einstiegsklasse:
8 RouteTable.Routes.Add(new ServiceRoute("url", new WebServiceHostFactory(), typeof(MyResource)));
```

Quelltext 6.3: Ressourcenvergleich - WCF

Im Gegensatz zu *Recess* können bei *Jersey* und *WCF* die Ressourcen als POJOs bzw. als POCOs implementiert werden. Alle drei Frameworks nutzen Annotationen um das Verhalten der Ressourcen festzulegen. Das Framework *Jersey* bietet hierfür die meisten Möglichkeiten.

6.1.2. Routing bzw. Zuordnung von URIs zu Ressourcen

Das Routing eines URI zu einer Ressource bzw. einer Methode von einer Ressource, muss bei jedem Framework explizit definiert werden. Dieser Abschnitt zeigt, wie diese Zuordnung bei den eingesetzten Frameworks realisiert wird.

PHP - Recess Das Framework *Recess* benötigt die Annotation `!Prefix Routes`, welche an die Klasse gebunden wird. Ohne diese Angabe kann das Framework die Ressource nicht verwenden. Methoden, die auf denselben URI reagieren sollen, wie sie für die Ressource definiert wurde, benötigen keine weiteren Angaben für das Routing. Methoden können auf von der Ressource abweichende URIs reagieren, indem ein relativer oder absoluter Pfad angegeben wird. Hier muss keine extra Annotation verwendet werden. Hinter der HTTP-Methode muss getrennt durch ein Komma der Pfad angegeben werden. Ein Beispiel wird in Quelltext 6.4 dargestellt.

```
1  <?php
2    Library::import('recess.framework.controllers.Controller');
3
4    /**
5     * !RespondsWith Layouts
6     * !Prefix Routes: /
7     */
8    class MyResource extends Controller {
9
10       /** !Route GET, myressource/$nummer */
11       function getMyRessource( $nummer ) {
12           return $this->ok();
13       }
14
15   }
16 ?>
```

Quelltext 6.4: Routingvergleich - Recess

So kann beispielsweise die Klasse den URI `/myressource` zugewiesen bekommen und eine Methode dieser Klasse `listAll`, so dass die Methode über `/myressource/listAll` angesprochen wird.

Java - Jersey Die Annotation `@Path` bindet einen URI an eine Ressource oder eine Methode einer Ressource. Es kann eine Hierarchie definiert werden, indem eine Klasse und deren Methoden einen URI zugewiesen bekommen. Für eine Methode können relative oder absolute URIs, analog zum Beispiel von *Recess*, definiert werden. Sollte eine Methode denselben URI besitzen, die bereits für die Klasse definiert wurde, so muss dieser bei der Methode nicht erneut angegeben werden. Alternativ kann der

URI bei der Klassendefinition weggelassen und nur bei den Methoden der Klassen angegeben werden. Platzhalter in den URIs werden mit geschweiften Klammern umschlossen `name`. Es muss dafür zusätzlich ein Methodenparameter definiert werden, der über die Annotation `@PathParam("name")` mit dem Platzhalter verknüpft wird. Ein einfaches Beispiel wird in Quelltext 6.5 dargestellt.

```
1  import javax.ws.rs.*;
2
3  @Path("/")
4  public class MyResource {
5
6    @GET
7    @Path("myressource/{nummer}")
8    public Response getMyRessource( @PathParam("nummer") Integer iNummer ) {
9      return Response.ok().build();
10   }
11
12 }
```

Quelltext 6.5: Routingvergleich - Jersey

C# - Windows Communication Foundation (WCF) Die Definition des Basis-URI einer Ressource findet im Gegensatz zu *Recess* oder *Jersey* nicht innerhalb der Ressource, sondern in der Einstiegsklasse statt. Dort wird in eine *Routingtabelle* eingetragen, welcher URI zu welcher Ressource gehört. Die Route zu einer Methode wird allerdings in der Ressource definiert. Dies geschieht mittels Attributen der Methoden. Hier wird anhand der HTTP-Methoden unterschieden, denn für `GET` wird das Attribut `WebGet(UriTemplate = "url")` und bei den Methoden `POST`, `PUT` und `DELETE` wird das Attribut `WebInvoke(UriTemplate = "url", Method = "methode")` verwendet. Platzhalter in den URIs werden mit geschweiften Klammern umschlossen `name`. Es muss dafür zusätzlich ein Methodenparameter definiert werden, der denselben Namen wie der Platzhalter besitzt. Ein einfaches Beispiel wird in Quelltext 6.6 dargestellt.

```
1  [ServiceContract]
2  [AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
3  [ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
4  public class MyResource
5  {
6      [WebGet(UriTemplate = "myressource/{nummer}")]
7      public String getMyRessource( string nummer )
8      {
9          return nummer;
10     }
11 }
12
13 // ----------------------------
14 // Routing aus Einstiegsklasse:
15 RouteTable.Routes.Add(new ServiceRoute("", new WebServiceHostFactory(), typeof(MyResource)));
```

Quelltext 6.6: Routingvergleich - WCF

Das Zuordnen von URIs zu Ressourcen findet bei *Jersey* und *Recess* über Annotationen, die an die Ressourcenklasse gebunden werden, statt. Bei *WCF* hingegen muss diese Zuordnung beim Starten der Anwendung manuell vorgenommen werden. Für Methoden einer Ressource erfolgt diese Zuordnung bei allen drei Frameworks über Annotationen, die an die entsprechende Methoden gebunden sind.

6.1.3. Realisierung von Standard HTTP-Methoden

In diesem Abschnitt wird die Verwendung der HTTP-Methoden `GET`, `PUT`, `POST`, `DELETE`, `HEAD` und `OPTIONS` für die verwendeten Frameworks beschrieben.

PHP - Recess Von den vier **CRUD**-Methoden (vergleiche Kapitel 2) werden nur die zugelassen, welche explizit für eine Ressource über Annotationen defininiert wurden. Alle anderen Aufrufe werden mit `405 Method not allowed` abgewiesen. Die Methode `HEAD` wird zur Zeit von *Recess* nicht unterstützt[1]. Dasselbe gilt für die Methode `OPTIONS`[2]. Wird versucht, eine dieser Methoden auf eine Ressource anzuwenden, so liefert der Server ebenfalls den Status `405 Method not allowed`.

Java - Jersey Durch das transparente Routing des Frameworks ist es möglich die HTTP-Methoden, die eine Ressource anbieten soll, zu implementieren. Alle HTTP-Methoden, die nicht explizit innerhalb der Ressource definiert wurden, werden mit dem Fehler *405 - Method not allowed* abgewiesen. Ausnahmen sind beispielsweise die HTTP-Methoden `OPTIONS` und `HEAD`, da diese vom Framework *Jersey* bereits verarbeitet werden. Somit muss sich der Entwickler nicht um diese beiden Methoden kümmern. Es ist jedoch möglich, mit den Annotationen `@HEAD` bzw. `@OPTIONS`, diese Methoden selber zu implementieren.

C# - Windows Communication Foundation (WCF) Mit *WCF* ist es ebenfalls möglich, die hier betrachteten HTTP-Methoden zu verwenden. Im Gegensatz zu *Jersey* werden die Methoden `HEAD` und `OPTIONS` nicht bereits durch das Framework verarbeitet und beantwortet. Diese müssen explizit durch den Entwickler mit dem Attribut `[WebInvoke(Method = "HEAD")]` bzw. `[WebInvoke(Method = "OPTIONS")]` implementiert werden. Alle HTTP-Methoden, die nicht explizit innerhalb der Ressource definiert wurden, werden mit *405 - Method not allowed* abgewiesen.

Die Realisierung der betrachteten Standard HTTP-Methoden wird sowohl von *Jersey* als auch von *WCF* unterstützt. Das Framework *Jersey* hat den Vorteil, dass die Methoden `HEAD` und `OPTIONS` bereits automatisch durch das Framework verarbeitet werden, so dass der Entwickler sich darum nicht kümmern muss. *Recess* hingegen unterstützt nur die vier **CRUD**-Methoden, die allerdings auch ausreichen, um einen *RESTful*-Webservice zu implementieren.

[1] `https://github.com/recess/recess/issues`
[2] `http://recess.lighthouseapp.com/projects/19507/tickets/32-options-method`

6.1.4. Unterstützung der Content Negotiation

Dieser Abschnitt vergleicht, welche Repräsentationsformate die Frameworks bereits unterstützen und ob es möglich ist, weitere zu definieren. Weiterhin wird dabei kurz angeschnitten, wie einfach bzw. schwierig die Implementierung weiterer Formate ist.

PHP - Recess Die *Content Negotiation* funktioniert automatisiert mithilfe von Annotationen. Auch hier können, mit eigenen Klassen für die Generierung der *Views*, eigene Formate oder bisher nicht unterstützte Formate in das Framework integriert werden. Definiert werden die unterstützten Formate für die Antwortdokumente mit der Annotation `!RespondsWith`. Hier können beliebig viele Formate, getrennt durch jeweils ein Komma, angegeben werden.

Nachteilig bei diesem Framework ist, dass diese Annotation nicht für jede Methode, sondern nur für die gesamte Ressource definiert werden kann. Dies schränkt den Entwurf einer Ressource ein, sofern der Entwickler nicht für jede Methode der Ressource dieselben Formate unterstützen will. Die Implementierung einer eigenen *View* ist relativ einfach. Hierfür muss eine Klasse implementiert werden, die von `recess.framework.AbstractView` ableitet. Diese muss zusätzlich noch in der Klasse `recess.framework.DefaultPolicy` registriert werden.

Wird ein nicht definiertes Format bei einer Anforderung übertragen, so löst das Framework einen *500 - Internal Server Error* Fehler aus. Bei einem geforderten Antwort-Format, welches nicht für diese Methode definiert wurde, liefert das Framework *406 - Not Acceptable* an den Client. Wird kein bestimmtes und unterstütztes Antwort-Format angefordert, liefert das Framework das zuerst definierte Format an den Client zurück.

Zu jedem angegebenen Format muss eine entsprechende *View* im Framework existieren, die selbstständig die gewünschte Umwandlung durchführt. Für eingehende Formate gibt es keine Annotation, so dass unterstützte Formate akzeptiert werden, aber ungewollte Formate nicht explizit „verboten“ werden können, indem sie nicht aufgeführt werden. Das Framework *Recess* beinhaltet bereits die *Views* für JSON und HTML.

Java - Jersey Die *Content Negotiation* wird durch die Annotationen `@Consumes` und `@Produces` realisiert. Bei beiden können ein oder mehrere Formate anhand deren MimeTypes definiert werden. Die Annotation `@Consumes` definiert welche Formate für ein Anforderungsdokument verwendet werden dürfen und `@Produces` definiert die unterstützten Formate für das Antwortdokument. Diese Annotationen können sowohl auf die gesamte Ressource, als auch auf einzelne Methoden der Ressource angewandt werden. Wird ein nicht definiertes Format bei einer Anforderung über-

tragen, so löst das Framework einen *415 - Unsupported Media Type* Fehler aus. Bei einem geforderten Antwort-Format, welches nicht für diese Methode definiert wurde, liefert das Framework *406 - Not Acceptable* an den Client. Wird kein bestimmtes und unterstütztes Antwort-Format angefordert, liefert das Framework das zuerst definierte Format an den Client zurück.

XML als Antwortformat kann das Framework automatisch aus beliebigen Java-Objekten generieren, sofern diese serialisierbar sind. Es muss dafür das entsprechende Objekt als Rückgabewert der Methode spezifiziert werden. Über weitere Annotationen, des XML-Frameworks der JEE, kann die Serialisierung gesteuert werden. Beispielsweise kann mit den Annotationen `@XmlElement(name="elementName")` bzw. `@XmlAttribute(name="attributeName")` eine Klasseneigenschaft als XML-Element oder Attribut mit einem abweichenden Namen definiert werden.

Eigene Formate können über sogenannte *EntityProvider* realisiert werden. Es muss dann jeweils eine Klasse zum Erzeugen und zum Auslesen des neuen Formates erstellt werden. Dafür werden die Schnittstellen `MessageBodyReader` und `MessageBodyWriter` verwendet. Die weitere Implementierung der Klassen ist relativ einfach. Die Annotation `@Provider` ermöglicht zudem, dass diese Klasse automatisch registriert wird. Siehe hierzu Beispiele und Erläuterungen von Tilkov [2009b].

C# - Windows Communication Foundation (WCF) Um bei *WCF* die Content Negotiation zu verwenden, müssen einige Umwege gegangen werden. Mit den Parametern `RequestFormat` und `ResponseFormat` der Attribute `WebGet` oder `WebInvoke` kann zwischen einer XML- und einer JSON-Repräsentation gewählt werden. Alternativ kann der verwendete XML-Serialisierer durch `[XmlSerializerFormat()]` gewechselt werden. Durch den Wechsel des Serialisierers können die Attribute für bestimmte Benamungen von Elementen und Attributen analog zu *Jersey* verwendet werden. Über den Umweg eines `MemoryStreams` können beliebige eigene Formate verwendet werden.

Die beste und flexibelste Unterstützung der Content Negotiation wird von *Jersey* geliefert. Hier kann das Format einfach und effektiv über Annotationen bestimmt werden und eigene Formate können einfach hinzugefügt werden. Bei *Recess* ist der entscheidende Nachteil, dass die Content Negotiation nur für eine Ressource, nicht aber für eine Methode einer Ressource spezifiziert werden kann. Zusätzlich gibt es keine Möglichkeit das Repräsentationsformat für den Inhalt einer Anfrage, beispielsweise bei einem `POST`, einzuschränken. *WCF* bietet die schlechteste Unterstützung, da die Implementierung eigener Formate sehr umständlich ist.

6.1.5. Statuslose Kommunikation

Die statuslose Kommunikation kann von keinem Framework erzwungen, sondern nur unterstützt werden. Dies ist grundsätzlich darin begründet, dass sich verschiedene Elemente der Programmiersprachen nicht sperren lassen. So kann beispielsweise unter PHP nicht verhindert werden, dass Daten in einer Sitzungsvariable gespeichert werden.

PHP - Recess Unter PHP wird keine aktive Unterstützung dieses Prinzips benötigt, da bei dieser Skriptsprache von vornherein bei jeder Anfrage die notwendigen Klassen instantiiert, erneut verarbeitet und nach Abarbeitung wieder zerstört werden. Es können daher keine Statusinformationen in Attributen der Klassen gehalten werden. Nicht verhindert werden kann die Verwendung von Dateien, Datenbanken oder der Variable `$_SESSION` durch den Entwickler. Das Prinzip der statuslosen Kommunikation wird daher bereits von der Programmiersprache unterstützt.

Java - Jersey Im Gegensatz zu PHP geht Java anders mit solchen Objekten und Anfragen um. Beispielsweise werden bei *Servlets* die Klassen einmalig beim Programmstart instantiiert und die gesamte Laufzeit über verwendet. Dabei greift jede Anfrage immer auf ein und dieselbe Instanz zu. Das Framework *Jersey* hingegen umgeht dieses Problem, indem es eine Ressource bei jeder Anfrage neu instantiiert und diese nach Beendigung der Anfrage wieder zerstört. So wird vermieden, dass innerhalb dieser Klasse ein Zustand in Attributen gehalten werden kann. Die verschiedenen Gültigkeitsbereiche (Scopes) [siehe Gabhart, 2003] der JEE können nicht verwendet werden, da die Ressourcen nicht aus Servlets bestehen und somit keinen Zugriff auf diese haben [vergleiche Shannon und Chinnici, 2008].

C# - Windows Communication Foundation (WCF) Bei *WCF* wird dieses Prinzip ebenfalls aktiv unterstützt, indem wie bei *Jersey* die notwendigen Ressourcen für jede Anforderung neu instantiiert werden. Nachdem die Anfrage beendet ist, werden diese Instanzen wieder zerstört. Um die Einhaltung dieses Prinzips zu erzwingen, können zwei Einstellungen für eine Ressource gesetzt werden. Dabei handelt es sich um `[ServiceContract(SessionMode = SessionMode.NotAllowed)]`, welches explizit dafür sorgt, dass keine Sitzungen verwendet werden können und das Attribut `[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]`, welches dafür sorgt, dass für jeden Aufruf eine neue Instanz der Ressource erzwungen wird.

Für die statuslose Kommunikation bieten sowohl *Jersey* als auch *WCF* die Instantiierung einer Ressourcenklasse für jeden Aufruf. Bei *Recess* ist dies nicht notwendig, da dieses Prinzip bereits in der Natur von PHP-Anwendungen liegt.

6.1.6. Unterstützung von Hypermedia

Dieser Abschnitt betrachtet, welche Hilfsmittel die verwendeten Frameworks bieten, um Ressourcen miteinander verknüpfen zu können. Als Beispiel dient die zuvor vorgestellte Ressource `MyResource`. Für eine konkrete Repräsentation wird die Instanz mit `nummer = 1` verwendet.

PHP - Recess Das Framework *Recess* bietet keine aktive Unterstützung, um Ressourcen zu verknüpfen. Es ist allerdings möglich anhand des Klassennamens, deren Methoden und Parametern, einen URI zu generieren. So kann für die Beispielimplementierung aus Quelltext 6.1 ein Aufruf, wie im Quelltext 6.7 durchgeführt, genutzt werden.

```php
<?php

    // URI generieren, wobei $this eine Instanz von recess.framework.AbstractController sein muss
    $this->urlTo( 'MyResource::getMyRessource', 1 );

?>
```

Quelltext 6.7: Beispiel - Hypermedia mit Recess

Weitere Implementierungen, wie beispielsweise eine Verknüpfungen an die Ressource anhängen oder die Präsentation der Verknüpfungen, bleiben dem Entwickler überlassen.

Java - Jersey Bei *Jersey* gibt es wie auch bei *Recess* keine aktive Unterstützung für Hypermedia. Hier kann die Annotation `@Path` der Klasse bzw. der Methoden ausgelesen werden, um den URI zu ermitteln. Durch einen `UriBuilder` ist es dann möglich, sehr einfach den URI zu erstellen. So kann für die Beispielimplementierung aus Quelltext 6.2 ein Aufruf, wie im Quelltext 6.8 dargestellt, durchgeführt werden.

```java
/**
 * Beispiel für Hypermedia unter Jersey
 */
public class URIExample {

  /** Kontext für die Erzeugung der URI */
  public @Context UriInfo uriInfo;


  /**
   * @return Beispiel-URI
   */
  public String getExampleURI() {
    return   uriInfo.getBaseUriBuilder()
            .path( RessourceHelper.getPathNoSlash( MyResource.class. HttpMethod.GET ) )
            .path( String.valueOf( 1 ) )
            .build()
            .toString();
  }

}
```

Quelltext 6.8: Beispiel - Hypermedia mit Jersey

Weitere Implementierungen, wie beispielsweise eine Verknüpfungen an die Ressource anhängen oder die Präsentation der Verknüpfungen, bleiben dem Entwickler überlassen.

C# - Windows Communication Foundation (WCF) Bei *WCF* gibt es ebenfalls keine aktive Unterstützung für Hypermedia. Hier kann der URI aus der Routingtabelle anhand der Ressourcenklasse geladen werden und mit dem Basis-URI und gegebenenfalls mit weiteren Pfaden generiert werden. Die Generierung kann entweder mit einer einfachen Konkatenation oder mithilfe der `UriTemplates` von C# realisiert werden.

Letztere Variante ist die bessere Lösung, da sie flexibler gegenüber Änderungen ist. Allerdings wurde für *pixLib* zunächst die erste Variante gewählt, da diese einfacher zu realisieren ist und die andere Variante keinen Mehrwert bringt. Das Ergebnis ist bei beiden Varianten identisch. So kann für die Beispielimplementierung aus Quelltext 6.3 ein Aufruf, wie im Quelltext 6.9 dargestellt, durchgeführt werden.

```
1  /// <summary>
2  /// Beispiel für Hypermedia unter WCF
3  /// </summary>
4  public class URIExample {
5
6      /// <returns>
7      /// Beispiel-URI
8      /// </returns>
9    public String getExampleURI() {
10     return     RessourceController.getRouteForType( typeof( MyResource.class ) )
11              + "/"
12              + 1;
13   }
14
15 }
```

Quelltext 6.9: Beispiel - Hypermedia mit WCF

Weitere Implementierungen, wie beispielsweise eine Verknüpfungen an die Ressource anhängen oder die Präsentation der Verknüpfungen, bleiben dem Entwickler überlassen.

Für das Prinzip Hypermedia bietet kein Framework eine aktive Unterstützung. Allerdings können mitgelieferte `UriBuilder` oder Hilfsklassen zum Auslesen eines URI von einer Ressource, den Entwickler bei der Implementierung dieses Prinzips passiv unterstützen.

6.1.7. Benutzbarkeit anhand eines *„Hello World“*-Beispiels

Hier werden die Frameworks anhand der Implementierung aus Kapitel 5 und anhand eines hier präsentierten Minimalbeispiels vorgestellt. Die Minimalbeispiele basieren auf einem klassischen *„Hello World“*-Beispiel. Weitergehende Gegenüberstellungen werden anhand der Implementierungen von *pixLib* vorgenommen.

PHP - Recess Bei *Recess* muss zunächst das Framework installiert und konfiguriert werden, bevor eine Webanwendung entwickelt werden kann. Nachdem dies geschehen ist, können mithilfe der mitgelieferten Weboberfläche eine neue Anwendung, zugehörige Ressourcen, Objekte und Routen erstellt werden. Dabei werden die Klassen größtenteils automatisch generiert, so dass diese nur noch mit eigener Logik angereichert werden müssen. Die Schritte und Möglichkeiten der Konfigurationsoberfläche können dem Anhang C oder der Herstellerwebseite[3] entnommen werden.

Eine Webanwendung in *Recess* ist ein Modul des Frameworks und kann nicht ohne Modifikation der Funktionsweise des Frameworks an eine andere Stelle gelegt werden. Das bedeutet, dass es ein Verzeichnis innerhalb der Framework-Struktur gibt, in das alle Anwendungen gelegt werden. Dabei handelt es sich normalerweise um das Verzeichnis `<recess-install>/app/`. Solch ein Modul basiert auf dem MVC-Prinzip und besteht daher aus mehreren Komponenten.

Als Einstiegspunkt wird eine *application*-Klasse generiert, wie beispielhaft im Quelltext 6.10 gezeigt wird. Diese Klasse gehört zu den automatisch generierten Klassen und muss im Normalfall nicht verändert werden. Die Darstellung dient nur dem Verständnis der Arbeitsweise des Frameworks. Die Applikationsklasse aus Quelltext 6.10 übernimmt hier nur die Aufgabe der Konfiguration der Anwendung. Es werden nur der Name und die zu verwendenden Pfade definiert.

```
1  <?php
2  Library::import('recess.framework.Application');
3
4  class HelloRecessApplication extends Application {
5      public function __construct() {
6
7          $this->name = 'HelloRecess';
8
9          $this->viewsDir = $_ENV['dir.apps'] . 'helloRecess/views/';
10
11         $this->assetUrl = $_ENV['url.base'] . 'apps/helloRecess/public/';
12
13         $this->modelsPrefix = 'helloRecess.models.';
14
15         $this->controllersPrefix = 'helloRecess.controllers.';
16
17         $this->routingPrefix = 'helloRecess/';
18
19     }
20 }
21 ?>
```

Quelltext 6.10: Hello Recess Anwendung

Ressourcen bestehen aus einem *Controller* und beliebig vielen *Views*. Dabei übernimmt der *Controller* die Verarbeitung einer Anfrage und die *Views* die Darstellung der Ressource. Ein *Controller* ist beispielhaft im Quelltext 6.11 dargestellt. Hier können anhand der Annotation `!Route` die unterstützten HTTP-Methoden auf bestimmte Methoden der Kontrollklasse abgebildet werden.

[3] `http://www.recessframework.org`

```php
<?php
Library::import('recess.framework.controllers.Controller');

/**
 * !RespondsWith Layouts
 * !Prefix Views: home/, Routes: /
 */
class HelloRecessHomeController extends Controller {

    /** !Route GET */
    function index() {

        $this->flash = 'Welcome to your new Recess application!';

    }

}
?>
```

Quelltext 6.11: Hello Recess Controller

Im Gegensatz zu *Jersey* ist die Aufteilung bzw. Strukturierung der Klassen nicht so flexibel (vergleiche Abschnitt 6.1.1). Die Namenswahl der Klassen ist allerdings frei, sie sollten aber von der entsprechenden Kontrollklasse abgeleitet werden. Interessanterweise werden auch Klassen, die weder von der Kontrollklasse ableiten, noch entsprechende Annotationen besitzen, in der Routing-Darstellung der Konfigurationsoberfläche angezeigt, sofern sie in dem spezifizierten Verzeichnis für Kontrollklassen liegen. Als Ressourcen können Klassen in diesem Verzeichnis allerdings nur verwendet werden, wenn sie vom entsprechenden `Controller` ableiten, da dies mittels Typprüfung sichergestellt wird.

Die *Views* sind als HTML Datei gekennzeichnet und werden als Standard-Repräsentation angeboten. Das Framework bietet Hilfsklassen an, um HTML-Formulare zum Erstellen bzw. Bearbeiten von generierten Datenmodellen zu erstellen. Die Beispielanwendung akzeptiert nur `GET`, welches in 6.12 exemplarisch aufgeführt ist.

```
# Request:
GET /server/recess/helloRecess HTTP/1.1
Host: 192.168.1.222

# Response:
HTTP/1.1 200 OK
Date: Thu, 07 Oct 2010 04:47:47 GMT
Server: Apache/2.2.15 (Win32) PHP/5.2.13
X-Powered-By: PHP/5.2.13
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html

Hello Recess
```

Quelltext 6.12: Hello Recess HTTP-Anfrage und -Antwort

Die Routing-Darstellung für die Beispielanwendung wird in Abbildung 6.1 gezeigt. Dort wird die Art des Aufrufes, hier die Methode `GET`, die Route, also den relativen Pfad der Ressource, und die zugehörige Kontrollklasse bzw. Methode dargestellt. Weitere Details zur Erstellung eines Projektes können dem Anhang C.1.2 entnommen werden.

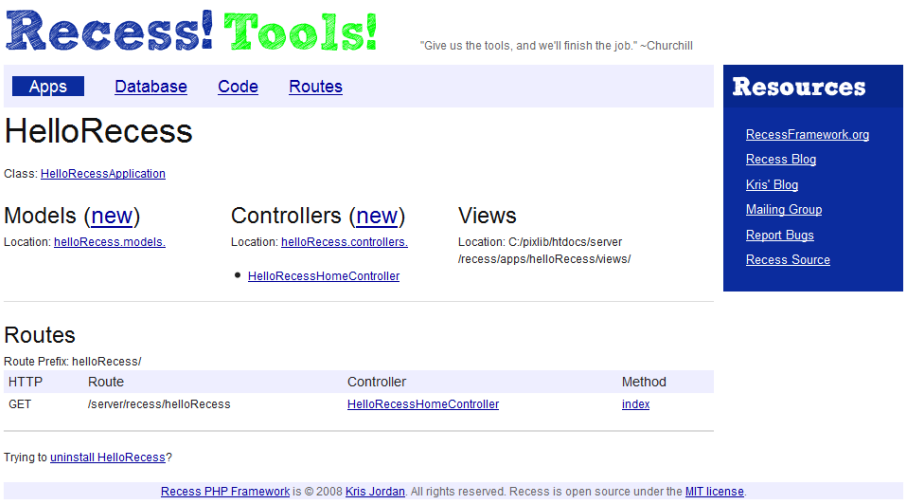

Abbildung 6.1.: Routingtabelle von Recess

Java - Jersey Für jeden Webservice, der mit *Jersey* entwickelt wird, werden die Bibliotheken benötigt, die im Anhang C näher erläutert werden. Dort wird ebenfalls beschrieben, wie ein Projekt unter *Eclipse* für die Verwendung von *Jersey* konfiguriert werden muss. Die beispielhaft implementierte Webanwendung *„Hello Jersey“* besteht aus zwei Dateien. Dem *Web Descriptor*, der im Quelltext C.4 dargestellt ist und der Ressource, die in Form einer POJO implementiert ist. Im *Web Descriptor* wird festgelegt, dass *Jersey* für alle Ressourcen aktiviert werden soll. Weitere Details zum Inhalt des *Web Descriptors* können dem Anhang C.2.2 entnommen werden.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
       http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
3    <display-name>HelloJersey</display-name>
4      <servlet>
5            <servlet-name>JerseyTest</servlet-name>
6             <servlet-class>
7                 com.sun.jersey.spi.container.servlet.ServletContainer
8             </servlet-class>
9          <load-on-startup>1</load-on-startup>
10     </servlet>
11
12     <!-- Jersey registrieren -->
13     <servlet-mapping>
14           <servlet-name>JerseyTest</servlet-name>
15         <url-pattern>/*</url-pattern>
16     </servlet-mapping>
17 </web-app>
```

Quelltext 6.13: Hello Jersey Descriptor

Diese Ressource wird mithilfe der Annotation `@Path` als Ressource kenntlich gemacht werden. Mit den Annotationen `@GET`, `@PUT`, `@POST` und `@DELETE` werden Methoden der Klasse als Methoden der Ressource aus der uniformen Schnittstelle definiert. Die beschriebene Ressource wird im Quelltext 6.14 dargestellt.

```
1  package de.pixlib.server.jerseytest;
2
3  import javax.ws.rs.*;
4
5  /**
6   * Beispielressource
7   */
8  @Path ("/hellojersey")
9  public class HelloJerseyResource {
10
11   /**
12    * Ausgeben von "Hello Jersey"
13    * @return "Hello Jersey"
14    */
15   @GET
16   @Produces ("text/plain")
17   public String sayHello() {
18     return "Hello Jersey";
19   }
20 }
```

Quelltext 6.14: Hello Jersey Ressource

Die Beispielanwendung akzeptiert folglich nur ein `GET`, welches im Quelltext 6.15 exemplarisch aufgeführt ist.

```
1  # Request:
2  GET /HelloJersey/hellojersey HTTP/1.1
3  Host: 192.168.1.222:8080
4
5  # Response:
6  HTTP/1.1 200 OK
7  Server: Apache-Coyote/1.1
8  Content-Type: text/plain
9  Transfer-Encoding: chunked
10 Date: Wed, 06 Oct 2010 16:36:23 GMT
11
12 Hello Jersey
```

Quelltext 6.15: Hello Jersey HTTP-Anfrage und -Antwort

C# - Windows Communication Foundation (WCF) Um einen Webservice mit *WCF* zu erstellen, wird empfohlen die im Anhang C.3.2 vorgeschlagene Vorlage zu verwenden. Dieses minimalistische Beispiel besteht aus einer Konfigurationsdatei für den IIS, einer Einstiegsklasse, welche die Routen der URIs zu den Ressourcen definiert und einer Ressourcenklasse. Die Konfigurationsdatei, die im Quelltext 6.16 dargestellt ist, beinhaltet einige Einstellungen und Informationen für den Webserver. Hier wird beispielsweise auch die verwendete Version des *.NET*-Frameworks angegeben.

```
1  <?xml version="1.0"?>
2  <configuration>
3
4    <system.web>
5      <compilation debug="true" targetFramework="4.0" />
6    </system.web>
7
8    <system.webServer>
9      <modules runAllManagedModulesForAllRequests="true">
10       <add name="UrlRoutingModule" type="System.Web.Routing.UrlRoutingModule, System.Web,
               Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
11     </modules>
12   </system.webServer>
13
14   <system.serviceModel>
15     <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
16     <standardEndpoints>
17       <webHttpEndpoint>
18         <!--
```

```
19              Configure the WCF REST service base address via the global.asax.cs file and the
                    default endpoint
20              via the attributes on the <standardEndpoint> element below
21          -->
22          <standardEndpoint name="" helpEnabled="true" automaticFormatSelectionEnabled="true"/>
23        </webHttpEndpoint>
24      </standardEndpoints>
25    </system.serviceModel>
26
27 </configuration>
```

Quelltext 6.16: Hello WCF Konfigurationsdatei

Die Einstiegsklasse wird zunächst nur verwendet, um festzulegen welche Klassen bzw. Ressourcen für welche URIs aufgerufen werden sollen. Es werden also Routen definiert. Solch eine Einstiegsressource wird in Quelltext 6.17 dargestellt.

```
1  using System;
2  using System.ServiceModel.Activation;
3  using System.Web;
4  using System.Web.Routing;
5
6  namespace HelloWCF
7  {
8      /// <summary>
9      /// Einstiegsklasse
10     /// </summary>
11     public class Global : HttpApplication
12     {
13
14         /// <summary>
15         /// Eintrittspunkt der Anwendung
16         /// </summary>
17         /// <param name="sender">
18         /// Objekt, welches die Applikation startet
19         /// </param>
20         /// <param name="e">
21         /// Start-Event
22         /// </param>
23         void Application_Start(object sender, EventArgs e)
24         {
25             RegisterRoutes();
26         }
27
28
29         //------------------------------------------------------------------------
30         /// <summary>
31         /// Registrieren der Routen
32         /// </summary>
33         private void RegisterRoutes()
34         {
35             RouteTable.Routes.Add(new ServiceRoute("hellowcf", new WebServiceHostFactory(),
                       typeof(HelloJerseyResource)));
36         }
37     }
38 }
```

Quelltext 6.17: Hello WCF Einstiegspunkt

Die Beispielressource ist im Quelltext 6.18 dargestellt.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.ServiceModel;
5  using System.ServiceModel.Activation;
6  using System.ServiceModel.Web;
7  using System.Text;
8
9  namespace HelloWCF
10 {
11     /// <summary>
12     /// Beispielressource
13     /// </summary>
```

```
14      [ServiceContract]
15      [AspNetCompatibilityRequirements(RequirementsMode =
            AspNetCompatibilityRequirementsMode.Allowed)]
16      [ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
17      public class HelloWCFResource
18      {
19
20          /// <summary>
21          /// Ausgeben von "Hello WCF"
22          /// </summary>
23          /// <returns>
24          /// "Hello WCF"
25          /// </returns>
26          [WebGet(UriTemplate = "")]
27          public String sayHello()
28          {
29              return "Hello WCF!";
30          }
31
32      }
33  }
```

Quelltext 6.18: Hello WCF Ressource

Die Beispielanwendung akzeptiert folglich nur ein `GET`, welches im Quelltext 6.19 exemplarisch aufgeführt ist.

```
1  # Request:
2  GET /hellowcf/ HTTP/1.1
3  Host: localhost:56995
4
5  # Response:
6  HTTP/1.1 200 OK
7  Server: ASP.NET Development Server/10.0.0.0
8  Date: Sat, 11 Dec 2010 09:01:13 GMT
9  X-AspNet-Version: 4.0.30319
10 Content-Length: 87
11 Cache-Control: private
12 Content-Type: text/html; charset=utf-8
13 Connection: Close
14
15 Hello WCF
```

Quelltext 6.19: Hello WCF HTTP-Anfrage und -Antwort

Die Erstellung und Konfiguration eines solchen Projektes kann dem Anhang C.3.2 entnommen werden.

Die obigen Beispiele bieten einen kleinen Einblick, wie unterschiedlich die Frameworks arbeiten. Von den grundsätzlichen Konzepten unterscheiden sie sich allerdings nur geringfügig. Detailliertere Information können im Anhang C nachgeschlagen werden.

6.1.8. Fazit

Für die Implementierung einer REST-Schnittstelle in der Programmiersprache PHP ist das Framework *Recess* gut geeignet und zu empfehlen. Die automatisierte Codegenerierung hilft speziell bei einfachen Schnittstellen und Ressourcen, die *1:1* auf eine Datenbanktabelle abgebildet werden. Bei komplexeren Ressourcen kann diese für eine kürzere Entwicklungszeit sorgen, sofern die Grundstrukturen erhalten bleiben.

Jersey bietet eine sehr gute Konformität zu REST, da der vorgegebene Standard *JAX-RS* (vergleiche Kapitel 4.3) laut Herstellerangabe vollständig implementiert wurde. Bei der Implementierung von *pixLib* wurden keine Anzeichen eines nicht konformen Verhaltens oder fehlenden Implementierungen gefunden. Wie anhand der obigen Vergleiche gut zu erkennen ist, bietet *Jersey* eine gute und umfangreiche Unterstützung für Entwickler, um Webanwendungen zu erstellen, die als *RESTful* bezeichnet werden können. Es werden darüber hinaus noch weitere Funktionen, wie beispielsweise die WADL-Generierung, angeboten. Diese sind für diesen Vergleich allerdings nicht wichtig und werden daher hier nicht erläutert.

Mit *WCF* lassen sich ebenfalls *RESTful*-Anwendungen implementieren, allerdings kann das Framework, speziell für die Content Negotiation und dem Zuordnen von Ressourcen zu einem URI, noch deutlich optimiert werden. Diese sind wesentliche Kritikpunkte bei dem Framework.

Somit können alle drei Frameworks verwendet werden, um eine *RESTful*-Anwendung zu implementieren. Wenn die Wahl zwischen den verschiedenen Sprachen und Frameworks möglich ist, sollte möglichst *Jersey* für Java verwendet werden, da dieses Framework die größte Flexibilität und Unterstützung für REST bietet. Zusätzlich ist es sehr einfach zu verwenden und es gibt viel Dokumentation im Internet. *Recess* bietet sich an, wenn keine Java-Unterstützung auf dem Webserver vorhanden ist, was bei vielen Anbietern für Webseiten-Hosting der Fall ist. Von dem Framework *WCF* für C# ist eher abzuraten, da mangels Beispielen und der geringen Verbreitung des IIS als Webserver, die Verwendung uninteressant ist.

6.2. Implementierung

Dieser Abschnitt befasst sich mit den Server-Varianten von *pixLib* und vergleicht die Implementierungen anhand der in Kapitel 3 definierten Kriterien. Die Clients werden nicht betrachtet, da sie, bis auf die Java-Variante, keine REST-Frameworks verwenden.

6.2.1. Schwierigkeiten bei der Umsetzung der Anforderungen

Die größte Schwierigkeit beim Entwurf lag bei der REST-Schnittstelle, da die Identifizierung und Gruppierung von Ressourcen einen großen Teil des gesamten Entwurfs ausmacht. Bei der Umsetzung der Anforderungen aus Kapitel 3 war die Realisierung der semantischen Äquivalenz der Anwendungen eine der größten Herausforderungen. Dies ist vor allem darin begründet, dass beispielsweise eine XML-Serialisierung unter PHP zunächst kein äquivalentes Dokument zu einer XML-Serialisierung unter Java liefert. Hier müssen

die Datenmodelle und Serialisierungs-Frameworks angepasst bzw. konfiguriert werden, so dass die Dokumente äquivalent werden. Dies geschieht bei Java und C# mittels Annotationen. Bei PHP hingegen entsteht hierfür deutlich mehr Programmieraufwand.

6.2.2. Aufwand der Implementierung

Der Implementierungsaufwand war für alle drei Server ähnlich groß. Generell war zumindest die Implementierung der Ressourcenklassen eine Fleißarbeit, da beim Entwurf bereits viel aus diesen Klassen herausgenommen wurde (vergleiche Kapitel 5). So enthielten diese Klassen nur noch Aufrufe von anderen Kontrollklassen oder von Persistenzklassen und die entsprechende Fehlerbehandlung.

pixLib PHP Bei der PHP-Variante ist der Aufwand für wenig komplexe *RESTful*-Anwendungen sehr gering, da hier die automatische Quelltext-Erzeugung verwendet werden kann. Händische Implementierungen sind hier im Gegensatz zu der Entwicklung mit *Jersey* oder *WCF* am aufwendigsten, da das Verhalten des Frameworks andere Ansätze erfordert, speziell bezogen auf die Fehlerbehandlung. Bei den anderen Varianten kann einfach mit einer entsprechenden *Ausnahme* abgebrochen werden, bei *Recess* hingegen muss ein Antwort-Objekt zurückgeliefert werden.

pixLib Java Aufgrund der einfachen Verwendung von *Jersey*, kann in sehr kurzer Zeit eine *RESTful*-Anwendungen implementiert werden. Ressourcen und Datenmodelle sind dank Annotationen sehr schnell implementiert und gleich einsatzbereit. Ein großer Vorteil bietet die einfach zu verwendende Content Negotiation. Wenn die automatische Quelltext-Erzeugung von *Recess* nicht verwendet wird, ist diese Kombination die einfachste und schnellste.

pixLib C# Der Aufwand für die Implementierung von *pixLib* mit *WCF* war höher als bei den anderen Implementierungen, was vermutlich an der Tatsache liegt, dass dieses Framework kein reines REST-Framework ist und daher viel flexibler sein muss. Ein wichtiger Punkt, der den Aufwand bei einer größeren Anzahl von Ressourcen nach oben treibt, ist die Tatsache, dass jede Ressource händisch in der Routingtabelle eingetragen werden muss. Ein Automatismus, wie der von *Jersey*, wäre hier wünschenswert.

Der geringste Aufwand für die Implementierung kann bei einfachen Webservices mit *Recess* durch die automatische Quelltext-Erzeugung erreicht werden. Bei komplexeren Anwendungen bietet sich *Jersey* an.

6.2.3. Erfüllung der REST-Prinzipien

Welche Prinzipien mehr oder weniger gut bei *pixLib* umgesetzt wurden, wird in diesem Abschnitt dargestellt. Dabei wird auf die Unterstützung durch die Frameworks Bezug genommen (vergleiche Kapitel 6.1). Da die verschiedenen Implementierungen größtenteils zueinander konform sind, wird in diesem Abschnitt nicht nach den verschiedenen Implementierungen unterschieden.

Über URIs ansprechbare Ressourcen Die Ressourcen werden über verzeichnisbasierte URIs identifiziert (vergleiche Kapitel 2). Ebenfalls wurde eine sinnvolle Zuordnung von gut lesbaren und eindeutigen URIs verwendet. Sie sind problemlos speicherbar und beinhalten alle notwendigen Informationen für einen späteren Abruf.

Verknüpfte Ressourcen (Hypermedia) Es wird von jeder Implementierung eine Einstiegsressource geliefert, durch die sich ein Client zu weiteren Ressourcen „hangeln“ kann (vergleiche Kapitel 2). Des Weiteren besitzen alle Ressourcen eine Verknüpfung auf sich selber und zum Teil zu anderen Ressourcen. So besitzt beispielsweise ein Bild eine Verknüpfung auf den Besitzer des Bildes und auf das zugehörige Album.

Verwendung von Standard HTTP-Methoden Je nach Ressourcentyp werden alle vier **CRUD**-Methoden wie vorgesehen verwendet (vergleiche Kapitel 2). Die Java-Implementierung unterstützt zudem noch die HTTP-Methoden `HEAD` und `OPTIONS`. *Recess* unterstützt diese beiden Methoden in der aktuellen Version gar nicht und bei *WCF* müssten diese für jede Ressource explizit implementiert werden.

Unterschiedliche Repräsentationen von Ressourcen Grundsätzlich bieten alle *pixLib* Server nur XML als Repräsentationsformat an. Die Ressource, die ein *eingebettetes Bild* liefert, verwendet HTML statt XML als Repräsentationsformat. Die einzige Ressource, die mehrere Formate für Anfragen und Antworten unterstützt ist die, welche die binäre Repräsentation eines Bildes repräsentiert. Dort werden die Formate GIF, JPEG und PNG in beide Richtungen angeboten.

Statuslose Kommunikation Die Frameworks *Jersey* und *WCF* bieten bereits eine aktive Unterstützung für die statuslose Kommunikation. *Recess* benötigt dies nicht, da PHP dies bereits unterstützt. Somit können keine Statusinformationen über eine Anfrage hinaus innerhalb einer Ressourcenklasse gespeichert werden. Weiterhin wurde in allen drei Implementierungen auf *Cookies*, *Benutzersitzungen* und weitere Möglichkeiten der Statushaltung verzichtet.

Alle drei Server-Implementierungen von *pixLib* können als *RESTful* bezeichnet werden, da sie die Prinzipien von REST erfüllen bzw. implementieren.

6.2.4. Konformität der verschiedenen Implementierungen

Dieser Vergleich zeigt überblicksartig, in welchen Punkten die verschiedenen Implementierungen nicht konform zueinander sind. Eine detailliertere Beschreibung zu diesen Problemen findet sich in Kapitel 5.6. Als Referenzimplementierung wird die Java-Anwendung genommen, um Vergleiche zu ziehen.

Datenbankabfragen Die Datenbankabfragen sind in der Java- und der PHP-Variante identisch, da dieselben Platzhalter (ein Fragezeichen) für *vorbereitete Anweisungen* verwendet werden können. Als Beispiel „`SELECT * FROM ?`". Bei der C#-Variante hingegen muss zusätzlich noch ein Name angegeben werden, beispielsweise „`SELECT * FROM ?tabelle`". Daher müssen zumindest für die C#-Variante modifizierte Datenbankabfragen existieren.

Content Negotiation Bei der *Recess*-Anwendung können zwar die drei Bildformate JPEG, GIF und PNG verwendet werden, allerdings kann aufgrund von Kompatibilitätsproblemem der verwendeten Grafikbibliothek *GD2*[4] keine Vorschaubilder für das GIF-Format erstellt werden. Im Gegensatz zu den anderen beiden Varianten liefert das Serialisierungs-Framework von C# einen Namensraum für das XML-Dokument. Diese Tatsache hat erstmal keine Auswirkungen, da die Namensräume nicht geprüft werden.

HTTP-Status-Codes *Recess* verhält sich in Ausnahmefällen unterschiedlich bei den zurückgelieferten HTTP-Status-Codes. Das unterschiedliche Verhalten tritt nur auf, wenn eine Ressource, die zwar existiert, aber so angesprochen wird, dass diese kein richtiges Ergebnis liefern soll (vergleiche Kapitel 5). Das Framework *WCF* verhält sich hier genauso wie *Jersey*. Dieses Verhalten hat allerdings keine Auswirkungen auf die *pixLib*-Clients.

Die Implementierungen sind, bis auf die oben genannten Punkte, zueinander konform.

6.2.5. Fazit

Die drei verschiedenen Implementierungen konnten konform zueinander implementiert werden. Dabei ist sowohl die REST-Schnittstelle, als auch die Persistenz (Datenbank und Dateisystem) gemeint. Diese Aussage wird durch die einwandfreie Verwendung der drei verschiedenen Clients auf jeder beliebigen Server-Variante unterstützt. In den Clients, werden aus Sicht der Implementierung, keine Unterschiede zwischen den drei Servern gemacht. Dadurch kann jederzeit der Server gewechselt werden.

[4] `http://www.libgd.org/Main_Page`

6.3. APIs

In diesem Abschnitt werden zwei APIs von konkurrierenden Web-Fotoalben verglichen und mit dem in dieser Ausarbeitung entwickeltem API verglichen. Dafür werden die in Kapitel 3 definierten Kriterien verwendet. Es werden die folgenden Web-Fotoalben betrachtet:

Flickr Das Web-Fotoalbum *Flickr* stammt von der Firma *Yahoo!*[5]. Das API [siehe Yahoo!] bietet die Möglichkeit mit selbstgeschriebenen Anwendungen lesend und schreibend auf Bilder, Alben und weitere Ressourcen zuzugreifen. Eine kommerzielle Nutzung des API muss vorher mit *Yahoo!* abgesprochen werden.

Picasa *Picasa* ist ebenfalls ein Web-Fotoalbum, welches von der Firma *Google*[6] stammt. Das API [siehe Google] bietet ebenfalls die Möglichkeit lesend und schreibend auf Bilder, Alben und weitere Ressourcen zuzugreifen. In diesem Vergleich wird die *Version 2.0* betrachtet.

6.3.1. Authentifizierung

Die Authentifizierung ist zwar für eine *RESTful*-Webanwendung nicht ausschlaggebend, aber der Vergleich ist interessant, da mit der Authentifizierung das Prinzip der statuslosen Kommunikation eng zusammenhängt. Wird beispielsweise eine klassische Authentifizierung über Sitzungen verwendet, so besteht die Gefahr, dass auch andere Informationen in den Sitzungen gespeichert werden.

Flickr Die Authentifizierung findet über einen *Token* und über eine *Signatur* statt. In der Spezifikation finden sich verschiedene Vorgehensweisen für die Authentifizierung von Web-, Desktop- und Mobilen-Anwendungen. Diese unterscheiden sich nur minimal in der Realisierung. Da für diese Ausarbeitung nur das Vorgehen für Webanwendungen interessant ist, wird nur auf diese eingegangen.

Zunächst muss ein *API-Schlüssel* über ein Benutzerportal angefordert werden, der pro Anwendung eindeutig sein muss. Nachdem dieser Schlüssel verfügbar ist, muss er über dasselbe Portal konfiguriert werden. Hierzu zählen Titel, Beschreibung, Logo und Authentifizierungstyp. Nachdem die Konfiguration abgeschlossen ist, kann der Authentifizierungsmechanismus verwendet werden. Da dieser Schlüssel an ein Benutzerkonto gebunden ist, muss keine zusätzliche Authentifizierung mit einem Benutzernamen und einem Passwort erfolgen.

[5] `http://de.yahoo.com`
[6] `http://www.google.de`

Der Token muss zunächst generiert werden. Hierfür muss ein Authentifizierungsaufruf mit den Parametern API-Schlüssel, dem gewünschten Recht (read, write oder delete) und der aufzurufenden Methode, durchgeführt werden. Aus den zuvor genannten Informationen wird eine *MD5-Prüfsumme*[7] [siehe Rivest, 1992] generiert, die abschließend als Signatur angehängt wird. Die Authentifizierungsinformationen werden immer als Paramter an die URL angehängt. Hierbei handelt es sich um den *Token*, den *API-Schlüssel* und die *Signatur*.

```
1 Schema zum Anfordern eines Tokens:
2 http://flickr.com/services/auth/?api_key=[key]&perms=[perms]&api_sig=[sig]
3
4 Schema mit Authentifizierung:
5 http://flickr.com/services/rest/?method=[method]&api_key=[key]&auth_token=[token]&api_sig=[sig]
```

Quelltext 6.20: Anforderung eines Tokens und Authorisierung bei Flickr

Picasa Bei diesem API gibt es ebenfalls verschiedene Authentifizierungsvarianten. Es wird unterschieden nach Einzel- und Mehrbenutzer-Anwendungen. Betrachtet wird hier die Mehrbenutzer-Variante für Webanwendungen. Dafür wird das *AuthSub System* verwendet, wobei auf die verschiedenen Techniken, wie z.B. *OAuth*, *openID* und *hybrid* nicht weiter eingegangen wird. Diese können in der API-Dokumentation [siehe Google] in detaillierter Form eingesehen werden.

Bei diesem Authentifizierungsverfahren muss die Webanwendung zunächst eine Anmelde-Seite von *Google* zusammen mit der URL der ersten Seite, die nach erfolgter Authentifizierung angezeigt werden soll, aufrufen. Der Anwender muss sich dann über die Anmelde-Seite authentifizieren, um anschließend die angegebene Aktion durchführen zu können. Dabei wird ein Gültigkeitsbereich (Scope) angegeben, für den das angeforderte Token gilt.

```
1 Schema zum Anfordern der Anmelde-Seite:
2 https://www.google.com/accounts/AuthSubRequest?scope=[scope]&session=1&secure=0&next=[nexturl]
```

Quelltext 6.21: Anforderung eines Tokens bei Picasa

Die obige Variante funktioniert allerdings nur für eine einzige Aktion. Für mehrere Aktionen, die hintereinander durchgeführt werden sollen, kann der zuvor generierte Token hochgestuft werden, indem dieser explizit über eine weitere `GET`-Anfrage abgerufen wird. Die erste Authentifizierung muss allerdings bereits erfolgt sein und der Parameter `session=1` muss gesetzt sein, da sonst der *Token* gleich wieder ungültig wird.

```
1 GET https://www.google.com/accounts/accounts/AuthSubSessionToken
2 Authorization: AuthSub token="[token]"
```

Quelltext 6.22: Hochstufung eines Tokens

[7] Nicht umkehrbare Prüfsumme aus 32 hexadezimalen Zeichen.

Bei Besitz eines hochgestuften *Tokens*, kann dieser als zusätzliche Kopfzeile bei Anfragen verwendet werden. Diese Kopfzeile sieht genauso aus wie beim Hochstufen des *Tokens*.

```
1 Authorization: AuthSub token="[token]"
```

Quelltext 6.23: Authorisierung von Anforderungen

pixLib Bei *pixLib* wird eine einfache Art der Authentifizierung verwendet. Diese ähnelt der *HTTP-Basic-Authentication* mit dem Unterschied, dass das Passwort als *MD5-Prüfsumme* verschlüsselt wird. Der Benutzername und das Passwort werden als zusätzliche Kopfzeilen bei jeder Anfrage erneut übertragen. Dabei wird der Benutzername in der Kopfzeile `username` und das Passwort in der Kopfzeile `passhash` übertragen. Eine detailliertere Beschreibung des verwendeten Mechanismus kann im Anhang B.1 gefunden werden. Diese Implementierung ähneln denen von *Flickr* und *Picasa*.

Dieser Vergleich zeigt, dass es eine Reihe verschiedener Möglichkeiten gibt, eine Authentifizierung zu implementieren. Ob *Flickr* und *Picasa* Sitzungen intern speichern, ist aus dem API nicht ersichtlich, daher kann nicht mit Sicherheit angegeben werden, ob die Implementierung *statuslos* arbeitet. Speziell der hochgestufte *Token* bei *Picasa* scheint Sitzungs-ähnliche Informationen auf dem Server zu speichern. *pixLib* hingegen ist vollständig *statuslos* implementiert.

6.3.2. Bibliotheken

Ob für die vorgestellten APIs fertige Bibliotheken existieren, um die Implementierung von Clients zu vereinfachen, wird in diesem Abschnitt betrachtet.

Flickr Bei *Flickr* gibt es eine Reihe von sogenannten *API-Kits*. Diese wurden allerdings von Dritten entwickelt und werden daher nicht von *Yahoo!* verwaltet oder in irgendeiner Weise unterstützt. Es existieren zum Stand dieser Ausarbeitung *API-Kits* für die folgenden Sprachen bzw. Technologien: ActionScript, C, Cold Fusion, Common Lisp, cUrl, Delphi, Java, .NET, Objective-C, Perl, PHP, Python, REALbasic und Ruby.

Picasa Für *Picasa* gibt es deutlich weniger Bibliotheken (hier *Client Libraries* genannt), dafür sind diese nicht von Drittanbietern und werden daher voll unterstützt. Für die *API-Version 2.0* gibt es zur Zeit nur die Bibliothek für Java. Für die *API-Version 1.0* gibt es Bibliotheken für Java, .NET, PHP und Python. Objective-C wird ebenfalls angeboten, allerdings wird diese Bibliothek nicht vollständig von *Google* unterstützt.

pixLib *pixLib* bietet zunächst keine Bibliotheken an. Da allerdings drei verschiedene Implementierungen auf dieselbe Schnittstelle zugreifen, kann bei der *Kommunikationsabstrakionsschicht* im weiteren Sinne von einer Bibliothek gesprochen werden. Diese könnte losgelöst von den Clients als Bibliothek zur Verfügung gestellt werden.

Die größte Auswahl an Bibliotheken bietet *Flickr* an, diese sind allerdings von Dritten implementiert und werden nicht von *Yahoo!* unterstützt.

6.3.3. Funktionsumfang

In diesem Abschnitt wird der Funktionsumfang der untersuchten APIs miteinander verglichen.

Flickr Der Funktionsumfang umfasst eine Reihe von Kategorien. Darunter fallen zusätzlich zu den grundlegenden Funktionen von einem Web-Fotoalbum noch Kategorien wie Blogs, Favoriten, Gruppen, Statistiken und Orte. Diese Fülle von Kategorien ermöglicht deutlich mehr Möglichkeiten Fotos zu verwalten, zu präsentieren und zu suchen.

Picasa *Picasa* bietet einen kleineren Funktionsumfang als beispielsweise *Flickr*. Der Umfang beschränkt sich auf ein ähnliches Maß wie der Funktionsumfang von *pixLib*. Das API von *Picasa* bietet zusätzlich die Möglichkeit *Tags* für Fotos zu erstellen, um diese einfacher wiederzufinden. *pixLib* hingegen bietet dafür die Möglichkeit andere Benutzer auf einem Bild zu markieren.

pixLib Das Web-Fotoalbum *pixLib* bietet die Grundfunktionen Alben und Bilder zu erstellen, zu ändern und zu löschen. Als weitergehende Funktionen sind das Bewerten und Kommentieren von Alben und Bildern, Bilder anderen Benutzern vorschlagen, Bilder als exzellent zu markieren und Benutzer auf einem Bild zu markieren (vergleiche Kapitel 3 und Kapitel 5).

Der Funktionsumfang von *Picasa* und *pixLib* ist sehr ähnlich. *Flickr* hingegen bietet eine Vielzahl weiterer Funktionalitäten an.

6.3.4. Formate und Technologien

Die verwendeten bzw. unterstützten Formate und Technologien der APIs werden in diesem Abschnitt dargestellt.

Flickr Das API von *Flickr* bietet eine Reihe verschiedener Zugriffsmöglichkeiten. Als Anforderungsformate werden REST, XML-RPC und SOAP unterstützt. Antwortformate sind REST, XML-RPC, SOAP, JSON und PHP. Um den Bezug zu dieser Ausarbeitung beizubehalten, wird nur die REST-Variante für Anforderung und Antwort betrachtet. Weitere Informationen zu den anderen Formaten können der Dokumentationsseite[8] entnommen werden. Zum Hoch- und Herunterladen von Bildern werden zusätzlich die Formate GIF, JPEG und PNG angeboten. Zum Hochladen werden weitere Formate wie beispielsweise das Tagged Image File Format (TIFF) unterstützt. Diese werden allerdings automatisch auf dem Server in JPEG konvertiert. Weitere Formate sind derzeit nicht implementiert oder vorgesehen.

Die Unterstützung von REST beschränkt sich allerdings auf `GET`- und `POST`-Methoden. REST-Anforderungen werden mittels eines bestimmten Endpunktes abgesetzt und REST-Antworten werden entweder implizit durch eine REST-Anforderung oder durch Angabe des Parameters `format=rest` angefordert. Ein Beispielaufruf findet sich in Quelltext 6.24.

```
1 Endpunkt für REST-Anforderungen:
2 http://api.flickr.com/services/rest/
3
4 Beispiel einer REST-Anforderung:
5 http://api.flickr.com/services/rest/?method=flickr.test.echo&name=value
```

Quelltext 6.24: REST-Beispiel bei Flickr

Das API wirkt, als wenn versucht wurde auf klassischen RPC-Methoden eine REST-Schnittstelle aufzusetzen. Dabei gibt es keine *Content Negotiation* im klassischen Sinne. Sie kann durch das Setzen des Parameters `format` simuliert werden, wobei hier nicht auf die Funktionalität des HTTP mittels Kopfzeilen wie `Accept` gesetzt wird. Es können bei dem API von *Flickr* nur wenige Elemente gelöscht werden. Dabei handelt es sich beispielsweise um Kommentare. Diese werden auch bei der Schnittstelle für REST mit einem `POST`-Aufruf gelöscht, was aufgrund der uniformen Schnittstelle nicht *RESTful* ist, da hierfür die Methode `DELETE` vorgesehen ist.

Picasa Neben dem regulären API bietet *Picasa* Feeds wie Really Simple Syndication (RSS) [siehe RSS Advisory Board, 2009] und ASF [siehe Nottingham und Sayre, 2005] für das Abrufen und Ändern von Daten an. Weitere Details finden sich hierzu in der API-Dokumentation [siehe Google]. Des Weiteren unterstützt *Picasa* die direkte Verwendung des HTTP, um ohne Bibliotheken auf die Schnittstelle zugreifen zu können. Die Zugriffe über das Protokoll basieren ebenfalls auf ASF. Das bedeutet, dass bei `GET`-Anforderungen ein XML-Dokument im Format ASF geliefert wird und bei `POST`- und `PUT`-Anforderungen ein solches Dokument gesendet werden muss. Bei *Picasa* befinden sich die Alben eines Benutzers „unterhalb“ der Benutzerressource. Es wird also eine Subressource verwendet.

[8] `http://www.flickr.com/services/api`

In einem experimentellen Stadium befindet sich die `PATCH`-Methode zum partiellen Verändern von Inhalten. Hierfür können dieselben URIs wie für die `PUT`-Anforderungen verwendet werden. Es müssen beim partiellen Update die Attribute, die verändert werden sollen, angegeben werden[9]. Zum Hoch- und Herunterladen von Bildern werden zusätzlich die Formate Windows Bitmap (BMP), GIF, JPEG und PNG angeboten. Weitere Formate sind derzeit nicht implementiert oder vorgesehen.

pixLib *pixLib* benutzt aus Komplexitätsgründen flache Hierarchien für die URIs, so dass die Datenstrukturen intern hierarchisch dagestellt, aber nach außen über die Schnittstelle flach gehalten werden. So kann auf ein Album über seine `ID` zugegriffen werden, ohne dass zusätzlich der Besitzer bekannt sein muss. Ein Vergleich zu den stark hierarchischen URIs von *Picasa* kann dem Quelltext 6.25 entnommen werden.

```
1 Picasa:
2 http://picasaweb.google.com/data/feed/api/user/[userID]/[albumID]
3
4 PixLib:
5 http://[webservice_uri]/library/[libraryID]
```

Quelltext 6.25: Vergleich URIs pixLib und Picasa

Als Repräsentationsformat für Anfragen und Antworten wird in der Regel XML verwendet. Zum Hoch- und Herunterladen von Bildern werden zusätzlich die Formate GIF, JPEG und PNG angeboten. Weitere Formate sind derzeit nicht implementiert oder vorgesehen.

Flickr bietet eine Unterstützung für eine Vielzahl von Repräsentationsformaten und verschiedene Technologien zur Kommunikation an. *Picasa* und *pixLib* hingegen bieten deutlich weniger Repräsentationsformate an.

6.3.5. Erfüllung der REST-Prinzipien

Der Vergleich mit dem API von *pixLib* wurde in dieser Gegenüberstellung nicht erneut aufgeführt, da dieser bereits zuvor in Kapitel 6.2.3 dargestellt wurde. Bei *Flickr* und *Picasa* wird auf die bisher in diesem Kapitel gewonnenen Erkenntnisse zurückgegriffen. Die Schnittstelle von *Flickr* ist im Gegensatz zu *pixLib* und *Picasa* nicht *RESTful*. Dafür spricht, dass die uniforme Schnittstelle nicht implementiert bzw. fälschlicherweise eine klassische SOA statt einer ROA (vergleiche Kapitel 6.3.4) verwendet wird. Je nachdem, ob die Anwendung bei Verwendung eines „hochgestuften Tokens“ zustandslos arbeitet oder nicht, kann sie als *RESTful* oder nicht bezeichnet werden. Dies kann ohne intensive Tests oder Betrachtung der Quelltexte nur spekuliert werden. Solange der „einfache Token“ verwendet wird, besteht dieses Problem nicht.

[9] `http://code.google.com/intl/de-DE/apis/picasaweb/docs/2.0/developers_guide_protocol.html`

6.3.6. Fazit

Der Vergleich verschiedener APIs zeigt, dass es durchaus Webservices gibt, die als *RESTful* bzw. als REST-Schnittstelle angepriesen werden, es aber nicht sind. Weiterhin ist zu erkennen, dass Schnittstellen, die auf klassische RPCs ausgelegt und später versucht wurden zu einer REST-Anwendung zu portieren, sehr wahrscheinlich keiner ROA, sondern einer SOA entsprechen. Daher sollte für die Implementierung einer *RESTful*-Schnittstelle versucht werden, diese neu aufzubauen, anstatt eine vorhandene auf SOA basierende, zu migrieren.

Das Web-Fotoalbum *Flickr* bietet zwar eine REST-Schnittstelle an, allerdings werden die Prinzipien von REST kaum beachtet und die Schnittstelle ist nach einer SOA für eine RPC-basierte Verwendung implementiert. Vermutlich wurde lediglich die SOAP-Schnittstelle erweitert statt einen neuen Entwurf anhand einer ROA durchzuführen. *Picasa* kann als *RESTful* angesehen werden, sofern die richtige Authentifizierungsmethode gewählt wird (vergleiche Kapitel 6.3.1). Die Schnittstelle von *pixLib* ist ebenfalls als *RESTful* anzusehen.

Weiterhin zeigt der Vergleich der Implementierungen von *pixLib*, dass flexible Frameworks zusammen mit einem sauberen Schnittstellenentwurf in verschiedenen Programmiersprachen und Technologien eine *RESTful*-Implementierung ermöglichen. Damit ist gemeint, dass sowohl die Schnittstelle *RESTful* ist, als auch die Konformität der XML-Repräsentationen untereinander gewährleistet ist, so dass diese völlig transparent und kompatibel mit einem für den Webservice entwickelten Client ist.

Das Internet?
Gibt's diesen Blödsinn immer noch?
Homer Simpson - KKW Springfield, 2006

7 Zusammenfassung

Der zentrale Bestandteil dieser Arbeit war es eine *RESTful*-Beispielanwendung in verschiedenen Programmiersprachen mit REST-Frameworks zu entwerfen und zu implementieren. Im Anschluss daran sollten die verschiedenen Frameworks und die Implementierungen der Beispielanwendung miteinander verglichen werden. Das Ziel dabei war es zu prüfen, wie gut die Unterstützung für *RESTful*-Anwendungen durch verschiedene Frameworks für unterschiedliche Programmiersprachen ist. Weitere Ziele waren die Sammlung von Erfahrungen für den Entwurf einer *RESTful*-Anwendung und für die unterschiedlichen Handhabungen der Frameworks.

Zunächst wird auf den Endzustand der implementierten Anwendung eingegangen, indem aufgezeigt wird, welche Kriterien und Anforderungen komplett, welche nur teilweise und welche gar nicht umgesetzt werden konnten. Im Anschluss daran wird ein Fazit über die Ergebnisse dieser Ausarbeitung gezogen. Abgeschlossen wird dieses Kapitel mit einem Ausblick und Ideen für die Weiterentwicklung der Anwendung.

7.1. Endzustand der Anwendung

Hier werden die in Kapitel 3 definierten Anforderungen und Kriterien mit der Implementierungen, die in Kapitel 5 beschrieben wurde, miteinander verglichen. Dieser Vergleich wird in der Tabelle 7.1 dargestellt.

Wie in Kapitel 5.5 zu sehen ist, wurden beide Oberflächen übersichtlich und funktional gestaltet. Die Weboberfläche ist mit den aktuellen Browsern lauffähig und wurde mit dem *Internet Explorer* in der Version `8` und mit dem *Firefox* in der Version `3.5` getestet.

Anforderung	Stichwort	PHP	Java	C#
F1	RESTful-Anwendung	✓	✓	✓
F2	Valides XML-Format	✓	✓	✓
F3	Benutzerverwaltung (Server)	✓	✓	✓
F4	Verwaltung von Alben und Bildern	✓	✓	✓
F5	Bildverknüpfungen	✓	✓	✓
N1	Lastverteilung möglich?	✓	✓	✓
N2	Plattformunabhängigkeit	✓	✓	✗
N3	Gemeinsames RDBMS	✓	✓	✓
N4	Wartbarkeit und Dokumentation	✓	✓	✓
N5	Implementierung	✓	✓	✓

Tabelle 7.1.: Umgesetzte Anforderungen

Beide Oberflächen bieten die jeweiligen geforderten Funktionalitäten übersichtlich an. Eine kurze Übersicht über noch offene Probleme, die nicht gelöst werden konnten bzw. die nicht notwendigerweise für diese Ausarbeitung gelöst werden mussten, werden im Folgenden angesprochen. Eine detailliertere Aufstellung über die offenen und auch die bereits gelösten Probleme kann in Kapitel 5 eingesehen werden.

Offene Probleme Die Unterstützung der HTTP-Methoden `HEAD` und `OPTIONS` wird nur durch *Jersey* automatisiert geleistet. Bei *WCF* können diese noch selber implementiert werden, was bei *Recess* hingegen nicht möglich ist. Damit auch *Recess* diese Methoden unterstützen kann, muss das Framework angepasst werden.

Bei der Server-Implementierung mittels *WCF* bleiben zwei Probleme offen, die nicht mit dem Framework oder der Implementierung zusammenhängen. Dabei handelt es sich um falsche Zeitstempel, die beim Auslesen aus der Datenbank entstehen (Problem mit der Zeitzone) und um die deaktivierten HTTP-Methoden `DELETE` und `PUT` im IIS (Diese Methoden sind grundsätzlich im IIS deaktiviert). Beide Probleme können durch entsprechende Konfigurationen gelöst werden.

Persönliche Anmerkungen Die Implementierungen haben gezeigt, dass selbst für eine verhältnismäßig kleine Anwendung wie *pixLib* viele Ressourcen entstehen, die sich von der Implementierung her sehr stark ähneln. Da es zur Zeit für die meisten Frameworks keine automatische Code-Generierung gibt, ist die Implementierung mit viel Fleißarbeit verbunden. Die Code-Generierung von *Recess* bietet hier bereits etwas Unterstützung, allerdings nur für einfache Ressourcen und Anwendungen.

7.2. Fazit

Diese Ausarbeitung hat gezeigt, dass es mit verschiedenen Frameworks in verschiedenen Programmiersprachen möglich ist relativ einfach eine *RESTful*-Webanwendung zu implementieren. Dabei bieten ausgereifte Frameworks für die meisten Prinzipien von REST eine gute Unterstützung an. Lediglich die Unterstützung für Hypermedia ist bei fast allen Frameworks noch so gut wie gar nicht vorhanden. Besonders gut ist die Zuordnung von URIs zu Ressourcen. Dies geschieht bei den meisten Frameworks mittels Annotationen oder vergleichbaren Metadaten.

Trotz des geringen Funktionsumfangs, des hier implementierten Fallbeispiels *pixLib*, entstand eine relativ große Anzahl von Ressourcen, die sich grundsätzlich nur in Kleinigkeiten unterscheiden. Aufgrund der Menge der Ressourcen ist die Implementierung einer REST-Schnittstelle zunächst sehr mühsam. Daher würde eine automatisierte Quelltextgenerierung einer REST-Schnittstelle einen großen Nutzen für Entwickler bieten. Zusätzlich bringt eine solche automatisierte Quelltextgenerierung noch weitere Vorteile mit sich. Darunter fallen beispielsweise die geringere Fehleranfälligkeit und eine höhere Qualität der Software.

Durch die Implementierungen und Vergleiche, die in dieser Ausarbeitung durchgeführt wurden, wird ersichtlich, dass die eigentliche Schnittstelle sehr schlank in Bezug auf den Quelltext gehalten werden kann. Dies wird grundsätzlich dadurch möglich, dass die Frameworks dem Entwickler bereits vieles an Arbeit abnehmen, wie beispielsweise die Content Negotiation oder das Routing eines URI zu einer Ressource. Somit muss sich der Entwickler mit diesen Vorgängen in der Regel nicht befassen und kann sich auf die Entwicklung der zu implementierenden Anwendung konzentrieren. Diese Tatsache erleichtert die automatisierte Generierung einer solchen Schnittstelle bereits erheblich.

Weitere Schnittstellen, die als auf REST-basierend vorgestellt werden, sind teilweise nicht *RESTful*, weil die Prinzipien von REST nur ungenügend oder gar nicht eingehalten werden. Dies zeigt der API-Vergleich aus Kapitel 6. Hieran lässt sich erkennen, dass von einigen Entwicklern entweder das Prinzip von REST nicht richtig verstanden oder versucht wurde auf eine bereits vorhandene „klassische“ Schnittstelle REST „aufzusetzen“. Dieses Problem kann größtenteils durch eine automatisierte Quelltextgenerierung vermieden werden.

An der *FernUniversität in Hagen* wird am Lehrgebiet *Datenverarbeitungstechnik* zurzeit ein Metamodell für REST [siehe Schreier, 2010] entwickelt. Dieses Metamodell soll es ermöglichen eine *RESTful*-Anwendung unabhängig von den Details der Implementierung und unabhängig von einer Programmiersprache zu entwickeln. Anhand dieses Modells wäre es dann möglich mittels einer automatisierten Quelltextgenerierung die Schnittstelle bereits größtenteils fertig zur Verfügung zu stellen.

7.3. Ausblick

Eine Übersicht über Verbesserungen durch neue Funktionalitäten oder Optimierungen, die für *pixLib* noch möglich sind, werden in diesem Abschnitt vorgestellt. Dabei wird zwischen funktionalen und nicht-funktionalen Verbesserungsvorschlägen unterschieden.

Funktionale Verbesserungsvorschläge Der Arbeitsablauf für die Bestimmung von *exzellenten Bildern* könnte durch etwas mehr Funktionalität deutlich verbessert werden. Mittels eines *Review-Prozesses* könnten potenzielle *exzellente Bilder* einer Abstimmung von Experten unterzogen werden. Dann würde ein solches Bild nur dann als *exzellent* markiert werden, wenn mindestens `x` Experten dafür und maximal `y` Experten dagegen stimmen.

Weiterhin könnten Experten durch Administratoren und/oder automatisch durch eine Mindestanzahl von eigenen *exzellenten Bildern* bestimmt werden. Sprich ein Benutzer muss eine bestimmte Anzahl von *exzellenten Bildern* eingestellt oder mindestens `x` Bilder mit guten Bewertungen in seinen Alben haben, damit er als Experte vorgeschlagen bzw. gewählt werden kann.

Für die Bilder könnten diverse weitere Funktionen implementiert werden, wie beispielsweise eine automatische Konvertierung der Bildformate oder das automatisierte Extrahieren verschiedener weiterer Metadaten aus den Bildern (beispielsweise Farbtiefe, Datum und Ort der Aufnahme, etc.).

Verknüpfungen von Bildern und Benutzern könnten dahingehend erweitert werden, dass auch die Position der Person auf dem Bild gespeichert und gezeigt wird. Weiterhin wäre es denkbar, dass ein Benutzer solch einer Verknüpfung erst zustimmen muss, bevor diese veröffentlicht wird. Als zusätzliche Reaktion könnte die Anfrage auch ignoriert werden, also weder zustimmen noch ablehnen.

Um eine bessere Lesbarkeit von aufgelisteten Ressourcen zu erlangen, sollten Listenressourcen *paginiert* (siehe Kapitel 5) werden, so dass immer nur eine begrenzte Zahl von Listenelementen zur Zeit präsentiert werden. Eine andere und unter Umständen auch bessere Unterteilung der Bilder kann über zusätzliche Kategorien gewährleistet werden. Die E-Mailbenachrichtigungen für die Experten könnten gesammelt erfolgen, so dass nicht bei jedem neuen potenziell *exzellenten Bild* eine einzelne E-Mail verschickt wird. Dabei würde ein zeitgesteuerter Prozess alle neuen Kandidaten seit der letzten Benachrichtigung zusammenfassen und als E-Mail versenden.

Als wirklich sinnvolle Erweiterung sollten Ressourcen beim Erstellen und beim Bearbeiten verschiedenen Validierungen unterzogen werden können, so dass beispielswei-

se nicht gefüllte, aber notwendige, Felder erkannt werden oder Werte in bestimmten Feldern im richtigen Format und/oder Wertebereich liegen. Eine weitere gute Erweiterung wäre eine komplexere Rechtestruktur, bei der ein Benutzer bestimmen kann, wer seine Alben und Bilder sehen kann. Mögliche Varianten wären dabei *alle*, *alle angemeldete* oder *ausgewählte Personen*.

Nicht-funktionale Verbesserungsvorschläge Um eine bessere Performanz zu erlangen, können aufgrund der *statuslosen Kommunikation* und der Verwendung von *Hypermedia* bei einer *RESTful*-Anwendung die Ressourcen auf mehrere Server verteilt werden. Weitere Verteilungen in Bezug auf Last und Redundanz der Daten können durch einen SRB für die im Dateisystem abgelegten Bilder und über entsprechende *Cluster*-Mechanismen von Datenbanken für die Metadaten erreicht werden.

Eine höhere Sicherheit in Bezug auf Authentifizierung und *Mithören* von Daten kann mittels einer SSL bzw. TLS Verbindung ermöglicht werden. Solch eine Verbindung ist grundlegend unabhängig von den Implementierungen, da diese bereits durch den verwendeten Webserver realisiert wird. Sie kann daher jederzeit nachträglich hinzugefügt werden.

Die Structured Query Language (SQL) Anweisungen sollten in *gespeicherten Prozeduren* in der Datenbank hinterlegt werden, so dass diese unabhängig von der Implementierung der Anwendung sind. Dies hat den Vorteil, dass bei verschiedenen Implementierungen die SQL Anweisungen nicht redundant vorhanden sind. Bei Änderungen der Daten-Struktur müssen dann nur die *gespeicherten Prozeduren* angepasst werden und nicht die SQL Anweisungen aller Implementierungen.

A

Weitere REST-Frameworks

Ein grober Überblick über weitere Frameworks für REST wird in diesem Abschnitt vorgestellt, die während der Recherche ermittelt wurden. Einige davon werden nicht mehr weiterentwickelt oder besitzen einen zu geringen Funktionsumfang, um als konkreter Kandidat für diese Ausarbeitung in Frage zu kommen.

A.1. Frameworks für PHP

Die Frameworks *Easyrest* und *Recess* wurden bereits in Kapitel 4 vorgestellt und werden daher in diesem Abschnitt nicht erneut aufgeführt.

Tonic - A RESTful Web App Development PHP Library Die Bibliothek *Tonic*[1] ist, wie der Name schon vermuten lässt, kein Framework, sondern eine Bibliothek. Diese soll dem Entwickler bei der Implementierung einer REST-Anwendung unterstützen. Es gibt daher keinen *Front-Controller* oder ähnliche Mechanismen, die das Routing und weitere vorgelagerte Arbeiten durchführen, wie beispielsweise die *Content Negotiation*.

Die Bibliothek basiert auf der PHP-Version `4.3`, bei der die Unterstützung der Objektorientierung nur rudimentär vorhanden ist. Eine Funktionalität, die die Bibliothek bietet, ist eine Authentifizierung mittels der *HTTP-Basic-Authentication*. Die letzte Aktualisierung in der Versionsverwaltung war im *Mai 2010* für die Version `2`. Die Bibliothek steht unter der *MIT-Lizenz*[2].

[1] `http://tonic.sourceforge.net`
[2] `http://www.opensource.org/licenses/mit-license.php`

ZEND-Framework mit REST-Plugin Das *ZEND-Framework*[3] ist ein großes Framework für PHP-Anwendungen, welches nicht speziell für REST entwickelt wurde. Es bietet eine ausführliche Unterstützung für Anwendungen, welche die MVC-Architektur implementieren. Für REST-Anwendungen gibt es das zusätzliche Plugin *ZEND_Rest*[4], das dem Framework hinzugefügt werden kann. Es steht unter einer Variante der *BSD-Lizenz*[5] und benötigt eine PHP-Installation in der Version `5`.

Für diese Ausarbeitung ist das *ZEND-Framework* nicht interessant, da der Overhead sehr hoch ist und REST nur als Plugin unterstützt wird aber nicht direkt vom Framework. Des Weiteren handelt es sich, in Bezug auf REST, eher um eine Bibliothek, als um ein Framework. Die letzte Aktualisierung in der Versionsverwaltung war im *Oktober 2010* für die Version `1.10`.

Konstrukt *Konstrukt*[6] ist kein Framework für REST, sondern für Kontrollklassen unter PHP. Es wird vom Hersteller ebenfalls ein *URI-to-controller-mapping* angeboten, welches nach demselben Prinzip arbeitet wie das Routing bei *Jersey* oder *Recess*. Es werden Funktionen wie die *Content Negotiation* unterstützt, so dass es mit diesem Framework möglich ist *RESTful*-Anwendungen zu implementieren. Die letzte Aktualisierung in der Versionsverwaltung war im *März 2010* für die Version `2.3.1`.

Chinchilla Das Framework *Chinchilla*[7] bietet einen *Front-Controller*, der HTTP-Anfragen an Ressourcen weiterleitet. Wobei es hier nicht möglich ist die Methoden selber zu bestimmen. So werden beispielsweise alle `GET`-Anfragen auf eine Ressource automatisch an die Methode `get_index()` und alle `POST`-Anfragen an die Methode `post_index()` weitergeleitet. Die letzte Aktualisierung in der Versionsverwaltung war im *Oktober 2010*.

PHPRestSQL Bei *PHPRestSQL*[8] handelt es sich um eine reine Datenbankschnittstelle, die es ermöglicht über REST direkt mit einer Datenbank zu interagieren. So kann beispielsweise mit einem `DELETE` direkt eine Zeile in einer Tabelle gelöscht werden, ohne dass ein komplexer Webservice dahinter steckt. Dieses Vorgehen ist allerdings nur dann sinnvoll, wenn sich die Ressourcen komplett in einer Datenbank abbilden lassen. Als Formate für das Antwortdokument werden XML und HTML unterstützt; es werden die HTTP-Methoden `GET`, `POST`, `PUT` und `DELETE` unterstützt. Dieses Projekt steht unter der *GPL-Lizenz*[9]. Die letzte Aktualisierung in der Versionsverwaltung war im *Januar 2009*.

[3] `http://framework.zend.com`
[4] `http://framework.zend.com/manual/en/zend.rest.html`
[5] `http://framework.zend.com/license`
[6] `http://konstrukt.dk`
[7] `http://restfulchinchilla.com`
[8] `http://phprestsql.sourceforge.net`
[9] `http://www.gnu.de/documents/gpl.de.html`

Tiny REST Framework Beim *Tiny REST Framework*[10] handelt es sich um ein kleines Framework, welches laut Angabe des Herstellers eine einfache Implementierung von *RESTful*-Anwendungen ermöglichen soll. Es steht unter der *Regular CodeCanyon Lizenz*[11] und kostet `5 US $`. Es wird eine PHP-Installation in der Version `5` benötigt.

Unterstützt werden als Ausgabeformate XML, JSON, CSV, HTML und die HTTP-Methoden `GET`, `POST`, `PUT` und `DELETE`. Ebenfalls wird die *HTTP-Basic-* und *Digest-Authentication* unterstützt. Es kann sowohl eine verzeichnisbasierte, als auch eine parameterbasierte URI-Struktur verwendet werden. Auf der oben genannten Webseite befindet sich ein kurzes Beispiel für die Verwendung des Frameworks. Die letzte Aktualisierung in der Versionsverwaltung war im *April 2010*.

escobar PHP REST framework Das Framework *escobar*[12] ist sehr einfach gehalten und für kleinere Anwendungen vorgesehen. Nachteilig ist, dass keine URIs in verzeichnisbasierter Form möglich sind, da alle Anforderungen über eine zentrale Datei laufen und keine Funktionen vorhanden sind, die diesen Nachteil vermeiden. Dies könnte beispielsweise, wie auch bei *Recess*, über ein URL-Rewriting gelöst werden. Das Framework basiert auf *Sinatra*[13] für *Ruby on Rails*[14]. Es wird eine PHP-Installation in der Version `5` benötigt.

Die offizielle Webseite des Projekts[15] existiert mittlerweile nicht mehr. Das Projekt ist nur noch über die Versionsverwaltung zu erreichen. Das Framework scheint ansonsten relativ wenig Funktionalitäten zu bieten. Es wird ein Beispiel mitgeliefert, das die Verwendung des Frameworks demonstriert. Die letzte Aktualisierung in der Versionsverwaltung war im *Oktober 2009*. Die Bibliothek steht unter der *MIT-Lizenz*.

A.2. Frameworks für Java

Hier aufgeführte Frameworks sind für die Programmiersprache Java vorgesehen. Die Frameworks *Restfulie*, *RESTeasy* und *Jersey* wurden bereits in Kapitel 4 vorgestellt und werden in diesem Abschnitt daher nicht erneut aufgeführt.

[10] `http://codecanyon.net/item/tiny-rest-framework/99263`
[11] `http://codecanyon.net/wiki/support/legal-terms/licensing-terms`
[12] `http://github.com/yizzreel/escobar`
[13] `http://www.sinatrarb.com`
[14] `http://rubyonrails.org`
[15] `http://www.ohloh.net/p/escobar-php`

Restlet Bei *Restlet*[16] handelt sich um ein Framework für Client- und Server-Anwendungen. Es kann in *Servlets* verwendet werden, da es auf dem API von *Servlets* aufsetzt. Aktuell ist die Version `2.0`, die einige Neuerungen und Modernisierungen im Gegensatz zur Vorgängerversion beinhaltet. *Restlet* unterstützt ebenfalls Annotationen und ist laut Webseite des Herstellers *„Ready for the Semantic Web (Web 3.0)“*, bei dem mittels erweiterten Metadaten Zusammenhänge zwischen Ressourcen gebildet werden können.

Die letzte Aktualisierung in der Versionsverwaltung war im *Juli 2010*. Das Framework steht wahlweise unter einer der folgenden Lizenzen: *LGPL 2.1*[17], *LGPL 3.0*[18], *CDDL 1.0*[19] oder *EPL 1.0*[20]. Weitere Informationen und Beispiele sind der Webseite des Herstellers oder Schneider [2010] zu entnehmen.

Axis 2 Das Apache-Projekt *Axis 2*[21] ist ein umfangreiches Framework zur Erstellung von Webservices. Es werden SOAP, WSDL, REST und weitere unterstützt. Daher handelt es sich folglich nicht um ein reines REST-Framework. Das Framework implementiert nicht den Standard *JAX-RS* [siehe Hadley und Sandoz, 2008], sondern verfolgt eigene Konzepte. Die letzte Aktualisierung war im *September 2010*. Das Framework steht unter der *Apache 2.0-Lizenz*[22].

Dyuproject Das Framework *Dyuproject*[23] ist eine auf *Servlets* basierende Implementierung, die ebenfalls mit Annotationen arbeitet. Die letzte Aktualisierung in der Versionsverwaltung war im *Mai 2010* für die Version `1.1.7`. Das Framework steht unter der *Apache 2.0-Lizenz*.

SerfJ - Simple Ever REST Framework for Java *SerfJ*[24] ist ein einfach zu verwendendes Framework, welches die MVC-Architektur implementiert. Es wird laut Angabe des Herstellers der *JAX-RS*-Standard weitestgehend implementiert, allerdings soll es hier kleinere Abweichungen geben. Grundlegend fehlt *SerfJ* eine Vielzahl an Funktionen aus dem zuvor genannten Standard. Es befindet sich allerdings noch in einem frühen Entwicklungsstadium. Es existiert eine Unterstützung für Client und für Server. Die letzte Aktualisierung in der Versionsverwaltung war im *Juli 2010* für die Version `0.3.0`. Das Framework steht unter der *Apache 2.0-Lizenz*.

[16] `http://www.restlet.org`
[17] `http://www.opensource.org/licenses/lgpl-2.1.php`
[18] `http://www.opensource.org/licenses/lgpl-3.0.html`
[19] `http://www.opensource.org/licenses/cddl1.php`
[20] `http://www.opensource.org/licenses/eclipse-1.0.php`
[21] `http://ws.apache.org/axis2`
[22] `http://www.apache.org/licenses/LICENSE-2.0.html`
[23] `http://code.google.com/p/dyuproject`
[24] `http://serfj.sourceforge.net`

Spring MVC Das Framework *Spring MVC*[25] implementiert nicht den *JAX-RS*-Standard, da es laut Aussage des Herstellers mit *Jersey*, *RESTeasy* und *Restlet* bereits genug Implementierungen gibt. Die Annotationen und die Verwendung des Frameworks zeigen jedoch gewisse Parallelen. Ebenso wie bei *Recess* werden die Klassen, die die Ressourcen repräsentieren, als *Controller* bezeichnet. Es werden ebenfalls Annotationen verwendet. Zu beachten ist bei diesem Framework, dass die Zustandslosigkeit nicht durch das Framework unterstützt wird, da die Ressourcen nicht jedes Mal aufs Neue instantiiert werden, sondern nur einmalig und dann weiter verwendet werden [vergleiche Schneider, 2010]. Weitere Informationen und Beispiele sind der Webseite des Herstellers oder der Diplomarbeit von Schneider [2010] zu entnehmen.

A.3. Frameworks für C#

Hier aufgeführte Frameworks sind für die Programmiersprache C# vorgesehen. Die Frameworks *Restfulie* und *Windows Communication Foundation* wurden bereits in Kapitel 4 vorgestellt und werden in diesem Abschnitt daher nicht erneut aufgeführt.

RestLess Das Framework *RestLess*[26] arbeitet mit *Attributen* von C#, ähnlich zu Annotationen bei *Jersey* unter Java. Leider unterstützt das Framework zur Zeit ausschließlich die HTTP-Methode `GET`. Es wird als DLL ausgeliefert und kann als Bibliothek eingebunden werden. Die letzte Aktualisierung war im *April 2008*.

Snooze Bei *Snooze*[27] handelt es sich um ein Framework für *RESTful*-Anwendungen mittels *ASP .NET MVC*. Das Routing wird durch einfaches Zusammensetzen von Zeichenketten und *URL-Klassen* realisiert. Die letzte Aktualisierung in der Versionsverwaltung war im *September 2009*. Die Bibliothek steht unter der *BSD-Lizenz*[28].

[25] `http://blog.springsource.com/2009/03/08/rest-in-spring-3-mvc`
[26] `http://ndepth.net/blog/restless-a-simple-rest-framework`
[27] `http://www.assembla.com/wiki/show/snooze`
[28] `http://www.gnu.org/philosophy/bsd.html`

B

Detaillierter Entwurf

In diesem Anhang werden weiterführende Konzepte dargestellt, die in Kapitel 5 bereits kurz erwähnt wurden, aber keinen Einfluss auf die Ausarbeitung oder deren Ergebnis haben. Sie sollen dem interessierten Leser tiefere Einblicke in die Anwendung geben und gegebenenfalls das Verständnis für bestimmte Entwurfsentscheidungen bzw. Gedankengänge verbessern.

Es wird im Folgenden auf allgemeine Entscheidungen eingegangen, gefolgt von Entscheidungen über die REST-Schnittstelle. Abschließend werden noch client- und server-spezifische Details vorgestellt.

B.1. Allgemeines

In diesem Abschnitt geht es um allgemeine bzw. grundsätzliche Entscheidungen und detaillierte Informationen. Darunter fällt alles, was nicht eindeutig dem Server, dem Client oder der Schnittstelle zugeordnet werden kann.

Konfigurationsdateien Eine Konfigurationsdatei enthält grundlegende Einstellungen für das System. Alle drei Implementierungen benutzen dieselben Konfigurationsdateien. Sie sind in XML ohne eine DTD implementiert. Eine DTD ist optional später zu erstellen. Der grundlegende Aufbau einer solchen Konfigurationsdatei kann dem Quelltext B.1 entnommen werden. Es existiert keine Schnittstelle zum Verwalten der Konfigurationsdateien, diese müssen daher mit einem Editor direkt bearbeitet werden. Sowohl der Client, als auch der Server besitzen Konfigurationsdateien.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<pixlib>
    <!-- Es kann einzelne Einstellungen geben -->
    <setting1>Wert</setting1>

    <!-- Ebenfalls sind gruppierte Einstellungen moeglich -->
    <settingGroup1>
        <setting2>Wert</setting2>
    </settingGroup1>
</pixlib>
```

Quelltext B.1: Aufbau einer Konfigurationsdatei

Die Clients besitzen eine Konfigurationsdatei, die die notwendigen Informationen für die verschiedenen Webservices enthält (siehe Quelltext B.2). Darunter fallen beispielsweise Informationen, über welchen Basis-URI diese angesprochen werden können.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pixlib>
<pixlib>
    <webservice>
        <!-- Allgemeine Einstellungen -->
        <host>192.168.1.222</host>
        <current>java</current>

        <!-- Einstellungen fuer den JAVA-Webservice -->
        <java>
            <port>8080</port>
            <relpath>/pixLibJavaServer</relpath>
            <ssl>false</ssl>
        </java>

        <!-- Einstellungen fuer den PHP-Webservice -->
        <php>
            <port>80</port>
            <relpath>/recess/pixLib</relpath>
            <ssl>false</ssl>
        </php>

        <!-- Einstellungen fuer den CSharp-Webservice -->
        <csharp>
            <port>8090</port>
            <relpath>/abc</relpath>
            <ssl>false</ssl>
        </csharp>
    </webservice>
</pixlib>
```

Quelltext B.2: Webservicekonfiguration

Die Server-Implementierungen von *pixLib* verwenden alle dieselben zwei Konfigurationsdateien. Beispielsweise werden hier die Zugangsdaten für die Datenbank gespeichert (siehe Quelltext B.3), damit sich die Anwendung mit dem RDBMS verbinden kann.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pixlib>
<pixlib>
    <!-- Zur Zeit wird nur mySQL unterstuetzt -->
    <database>
        <hostname>exampleHost</hostname>
        <type>mysql</type>
        <port>3306</port>
        <username>exampleUser</username>
        <password>examplePassword</password>
        <databasename>exampleDatabase</databasename>
    </database>
</pixlib>
```

Quelltext B.3: Datenbankkonfiguration

In der zweiten Konfigurationsdatei werden die Zugangsdaten für einen Mailserver gespeichert (siehe Quelltext B.4), über den Benachrichtigungen verschickt werden.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <pixlib>
3      <mail>
4          <!-- Mailserver Einstellungen -->
5          <type>smtp</type>
6          <host>smtp.example.de</host>
7          <port>25</port>
8          <username>exampleUser</username>
9          <password>examplePassword</password>
10     </mail>
11 </pixlib>
```

Quelltext B.4: E-Mail-Konfiguration

Weitere Konfigurationen werden für die Server nicht benötigt.

Authentifizierung und Sicherheit Die Authentifizierung für *pixLib* ist durch eine eigene Implementierung realisiert worden, die aus mehreren Stufen besteht. Generell hat jeder Anwender ohne eine Anmeldung die Möglichkeit auf alle Ressourcen *lesend* zuzugreifen. Die einzige Ausnahme bildet hier die Liste der potenziell *exzellenten Bilder* (vergleiche Kapitel 5), da diese nur von Experten eingesehen werden darf. Jeder angemeldete Benutzer darf alle Ressourcen bearbeiten und löschen, wobei auch hier die Einschränkung für die potenziell *exzellenten Bilder* besteht. Weitere Abstufungen sind möglich, aber wurden nicht implementiert.

Realisiert wird die Authentifizierung durch das Versenden von zusätzlichen Kopfzeilen. Dabei ist in der Kopfzeile `username` der Benutzername und in `passhash` die *MD5-Prüfsumme* des Passwortes enthalten. Sollten diese nicht gesetzt, das Benutzerkonto deaktiviert oder die Kombination Benutzername/Passwort nicht gültig sein, so gilt ein Benutzer als nicht angemeldet. Sind beide Kopfzeilen nicht gesetzt, so wird ein `401 Unauthorized` geliefert, wobei eine nicht gültige Anmeldung ein `403 Forbidden` produziert.

Eine alternative Vorgehensweise wäre die Verwendung von den bereits vorhandenen Mechanismen *HTTP-Basic-* und *HTTP-Digest-Authentication*, die durch die zusätzliche Verwendung von SSL bzw. TLS die Anwendung auf einen hohen Sicherheitsstandard bringen können. Diese Art der Authentifizierung bringt allerdings erstmal keine Abstufungen für Benutzerrollen oder für Datenbankabfragen mit sich. Des Weiteren werden diese nicht von jedem Framework unterstützt, so dass diese selbst implementiert oder durch den Webserver übernommen werden müssen. In der Regel sollte der Webserver diese Authentifizierung übernehmen, noch bevor die Anwendung oder das Framework angesprochen werden. In manchen Frameworks kann dieses Verhalten ebenfalls erzeugt werden, allerdings funktioniert es fast analog zu dem

verwendeten Mechanismus. Das Verhalten und die Verwendung der in *pixLib* verwendeten Implementierung ist stark an die *HTTP-Basic-Authentication* angelehnt. Ausführliche Diskussionen über die Sicherheit von *RESTful*-Webanwendungen liefert Tilkov [2009a].

B.2. REST-Schnittstelle

Dieser Abschnitt befasst sich mit der REST-Schnittstelle für *pixLib*.

Bildressourcen Bei diesem Punkt geht es speziell um das Laden mittels `GET` und das Speichern mittels `POST` bzw. `PUT` von Bildern. Die Bildressource ist in zwei Ressourcen aufgeteilt. Eine Ressource, die das eigentliche Bild als Binärdatei beinhaltet, sowie eine weitere Ressource für die Metadaten des Bildes. Dieses Vorgehen hat den Hintergrund, dass das Format der Metadaten konform zu allen anderen Ressourcen-Daten gehalten werden soll. So können die Metadaten einfach über XML gehandhabt werden. Zusätzlich erlaubt die gewählte Variante das getrennte Hochladen und die Änderung bzw. Erstellung der Metadaten eines Bildes. So kann beispielsweise der Titel geändert werden, ohne dass das Bild erneut hochgeladen werden muss.

Andere APIs benutzen hierfür nur eine Ressource, wobei dann entweder der Inhalt der Anforderung die Binärdaten enthält und die Metadaten als Parameter übertragen werden, oder es wird ein *Multipart*-Inhalt verwendet, der sowohl die Metadaten in einer XML-Form, als auch die Binärdaten des Bildes enthält. Diese Variante ist allerdings aufwendiger zu implementieren und bringt keine nennenswerten Erkenntnisse für den Vergleich der Frameworks, daher wurden zwei Ressourcen zur Lösung des Problems gewählt.

Die bei *pixLib* verwendete Aufteilung hat zur Folge, dass das Hinzufügen eines Bildes zu einem Album in zwei Schritten durchgeführt werden muss. Im ersten Schritt werden nur die Metainformationen an die entsprechende Ressource übertragen. Diese Anforderung liefert bei Erfolg auch die dem Bild zugewiesene `ID` zurück, welche anschließend im zweiten Schritt, also beim Hochladen der Binärdatei auf eine andere Ressource, verwendet wird. Dadurch kann *pixLib* das hochgeladene Bild zu den Metainformationen zuordnen.

Die Metainformationen *Titel* und *Beschreibung* werden durch den Benutzer im ersten Schritt angegeben. *Abmessungen*, getrennt in *Breite* und *Höhe* und die *Dateigröße* in Bytes werden automatisiert nach dem Hochladen im zweiten Schritt gesetzt.

Verknüpfungen von Ressourcen Das Verknüpfen von Ressourcen ist ein noch wenig durch Frameworks unterstütztes Prinzip von REST. Es werden zwar zum Teil Mechanismen, auch *URI-Builder* genannt, angeboten, die den Entwickler bei der Implementierung unterstützen können, allerdings nehmen sie dem Entwickler noch zu wenig Arbeit ab. Für *pixLib* wurde ein Mechanismus entwickelt, der an einer zentralen Stelle, nämlich dem `RessourceSkeleton`, für jedes Datenmodell die notwendigen Verknüpfungen bei Bedarf generieren kann. Dies geschieht durch Auslesen des Ressourcen-URI der jeweiligen Ressourcen-Kontrollklasse und gegebenenfalls dem Hinzufügen der entsprechenden `ID` der Zielressource.

Durch den oben beschriebenen Mechanismus können die Verknüpfungen bei Bedarf dynamisch generiert werden, so dass diese nicht persistent gespeichert werden müssen. Durch das Injizieren der Verknüpfungen in die Datenmodelle müssen weder die Ressourcen, noch das Datenmodell Kenntnis von anderen Ressourcen besitzen. Die konkrete Implementierung von diesem Mechanismus unterscheidet sich stark durch die unterschiedliche Unterstützungen der Frameworks bzw. der Sprachen. Der Quelltext B.5 zeigt, dass bei *Recess* anhand des *Klassennamens* der Ressource, der Angabe des *Methodennamens* und der entsprechenden *Parameter*, das Framework automatisch der richtige URI ermittelt. Die Methodennamen können dank den Annotationen frei gewählt werden. Bei *pixLib* wurden die Methoden in Anlehnung an *Servlets* mit einem `do` vor dem Namen der verwendeten HTTP-Methode benannt. Bei einem `GET` würde die Methode dann `doGet()` heißen.

```
1  <?php
2
3    /**
4     * URIs im Model vervollständigen
5     *
6     * @param $oModel Model eines Albums
7     */
8    protected function completeLibraryModel( LibraryModel $oModel ) {
9
10       // Setzen der self-Verknüpfung
11       $oModel->setLink(
12               'self',
13               $this->urlTo(
14                       'LibraryRessourceController::doGet',
15                       $oModel->id
16               )
17       );
18
19       // Setzen der Verknüpfung des Besitzers
20       $oModel->setLink(
21               'creator',
22               $this->urlTo(
23                       'UserRessourceController::doGet',
24                       $oModel->creatorID
25               )
26       );
27
28   }
29
30 ?>
```

Quelltext B.5: Recess - Ressourcen verknüpfen

Bei *Jersey* musste eine Hilfsklasse implementiert werden, die aus einer Ressource die Annotation `@Path` ausliest. Diese Klasse wird in Quelltext B.6 dargestellt.

```java
package de.pixlib.server.utils;

import javax.ws.rs.Path;
import javax.ws.rs.core.UriBuilder;


/**
 * Hilfsklasse zum Auslesen von Routing-Annotationen
 *
 * @author Guido Kaiser
 * @version 1.0
 */
public class RessourceHelper {

  /**
   * Standardkonstruktor
   */
  private RessourceHelper() {}



  /**
   * Liefert den Pfad einer Ressource
   *
   * @param clazz Ressource
   * @return Pfad
   */
  @SuppressWarnings("unchecked")
  public static String getPath( Class clazz ) {
    return UriBuilder.fromResource( clazz ).build().toString();
  }


  /**
   * Liefert den Pfad einer Ressource ohne den führenden Slash "/"
   *
   * @param clazz Ressource
   * @return Pfad ohne führenden Slash
   */
  @SuppressWarnings("unchecked")
  public static String getPathNoLeadingSlash( Class clazz ) {
    return UriBuilder.fromResource( clazz ).build().toString().replaceFirst( "/", "" );
  }

}
```

Quelltext B.6: Jersey - Ressourcenpfad auslesen

Der Quelltext B.7 zeigt die Verwendung der Hilfsklasse, um die Verknüpfungen zu erstellen. Dabei wird der mitgelieferte *URI-Builder* von *Jersey* verwendet, der mithilfe des *Basis-Pfads*, des *Pfades zur Ressource* und gegebenenfalls mit einer konkreten `ID` der Zielressource, die Verknüpfung generiert.

```java
  /**
   * URIs im Model vervollständigen
   *
   * @param oModel Model eines Albums
   */
  protected void completeURIS( LibraryModel oModel ) {

    // Setzen der self-Verknüpfung
      oModel.setSelf(
          new URIModel(
              "self",
              uriInfo.getBaseUriBuilder().path(
                  RessourceHelper.getPathNoSlash( LibraryRessourceController.class ) )
                  .path( String.valueOf( oModel.getId() ) )
                  .build()
                  .toString()
          )
      );

      // Setzen der Verknüpfung des Besitzers
      oModel.setCreator(
          new URIModel(
              "creator",
```

```
24                uriInfo.getBaseUriBuilder().path(
25                    RessourceHelper.getPathNoSlash( UserRessourceController.class ) )
26                    .path( String.valueOf( oModel.getCreatorId() ) )
27                    .build()
28                    .toString()
29            )
30        );
31
32    }
```

Quelltext B.7: Jersey - Ressourcen verknüpfen

Der Einfachheit halber sind erstmal alle `IDs` bzw. *Fremdschlüssel* aus der entsprechenden Datenbanktabelle Kandidaten für eine Verknüpfung. Es können darüber hinaus beliebige weitere Verknüpfungen, die nicht direkt aus der Datenbankstruktur ableitbar sind, hinzugefügt werden.

Hypermedia Die Verwendung von *Hypermedia* in dem Sinne, dass die Ressourcen alle weiterführenden Verknüpfungen gleich liefern, stellt bei einem REST-Client eine große Herausforderung dar. Dies ist vor allem darin begründet, dass diese Verknüpfungen nicht direkt verwendet werden können, da diese ja auf den Server zeigen und nicht auf den Client. Es muss also intern im Client ein Mapping einer Aktion auf eine Verknüpfung durchgeführt werden. Aufgrund der Tatsache, dass dieses Vorgehen eine komplexe Verwaltung benötigt und keinen nennenswerten Mehrwert für diese Ausarbeitung bietet, wurde die Unterstützung für *Hypermedia* in den Clients so gering wie möglich gehalten.

Der C#-Client unterstützt kein *Hypermedia*, dies ist darin begründet, dass dieser keinen eigenen Zustand besitzt und alle Aufrufe neu generiert, mit den Informationen, die in der Oberfläche eingestellt werden. Der Java- und der PHP-Client hingegen unterstützen intern das Prinzip *Hypermedia* in dem Sinne, dass alle möglichen Start-Verknüpfungen, die von der *Einstiegsressource* geliefert werden, später im Menü erscheinen. Auf diese Weise kann der Funktionsumfang, den das Portal bietet, durch den Server anhand der Authentifizierungsinformationen dynamisch erzeugt werden. Der Java-Client filtert allerdings alle Verknüpfungen heraus, die nicht zum Funktionsumfang des Expertenportals gehören.

B.3. Dateipfade für Bilder

Redundante Datenhaltung sowie Verteilung der Bilder ist nicht implementiert. Solche Funktionalitäten können bei Bedarf mittels eines Storage Resource Broker (SRB) [siehe Shen u. a., 2001] nachträglich hinzugefügt werden. Der Dateipfad wird anhand einer einfachen Struktur automatisch zusammengesetzt und nicht persistent gespeichert. Dieser setzt sich aus der hierarchischen Struktur der Metadaten eines Bildes zusammen und besteht demnach aus: `<Repository-Pfad>/<UserID>/<AlbumID>/<BildID>`

`.<Erweiterung>`. Der Repository-Pfad ist zur Zeit fest im Quelltext vorhanden. Da anhand dieser Informationen die Dateierweiterung nicht bekannt sein kann, werden weitere Dateien mit demselben Pfad und der Erweiterung `.ext` angelegt, in der die Dateierweiterung des Bildes enthalten ist. Bei einem JPEG-Bild mit `ID = 1` würden dann die Dateien `1.ext` und `1.jpg` im Verzeichnis des entsprechenden Albums vorhanden sein. Auf diese Weise müssen keine Informationen in der Datenbank gespeichert werden.

Durch diese vereinheitlichte Speicherung der Daten können alle Implementierungsvarianten auf dieselben Dateien zugreifen, da alle benötigten Informationen in der globalen Datenbank vorhanden sind. Alternativ kann, wie bereits beschrieben, ein SRB verwendet werden. Dabei handelt es sich um eine Speicherverwaltung, die es ermöglicht, die Daten auf verschiedenen Servern abzulegen und dabei auch mehrere Kopien, oder so genannte *Replikationen* der Dateien zu erstellen.

B.4. Datenbankentwurf

Hier wird ein Überblick über die Entscheidungen für bestimmte Datenbank-Konstrukte oder Maßnahmen aufgeführt und begründet. Die Abbildung B.1 zeigt das gesamte Datenbankschema der *pixLib*-Datenbank. Verschiedene parallele Änderungs-Anforderungen stellt hier kein Problem dar, weil atomare `UPDATE`-Anweisungen verwendet werden und keine eindeutigen `ID`s veränderbar sind.

MyISAM Da für die wenig komplexe Datenhaltung zurzeit keine Transaktionen verwendet werden müssen, wird das Format *MyISAM* verwende. Der Geschwindigkeitsvorteil von *InnoDB* bei gleichzeitigen Lese- und Schreibzugriffen ist für dieses Projekt vernachlässigbar, da es nicht für viele parallele Benutzer gedacht ist und da nicht viele Schreibzugriffe auf die Datenbank stattfinden. Sollte die Anwendung für eine höhere Benutzeranzahl skaliert werden, so sollte das Datenbankformat auf *InnoDB* umgestellt werden.

Mehrsprachigkeit Um die Komplexität gering zu halten, wurde bewusst auf Unterstützung von Mehrsprachigkeit verzichtet. Das bedeutet, dass weder beim Zeichensatz, noch bei den Namen oder Beschreibungen von Datenbankattributen darauf Rücksicht genommen wurde. Um eine Mehrsprachigkeit nachträglich zu realisieren, werden diverse weitere Tabellen und andere Zeichensätze benötigt.

Primärschlüssel Alle Primärschlüssel wurden als `unsigned BigInt` mit der Eigenschaft `AutoIncrement` definiert.

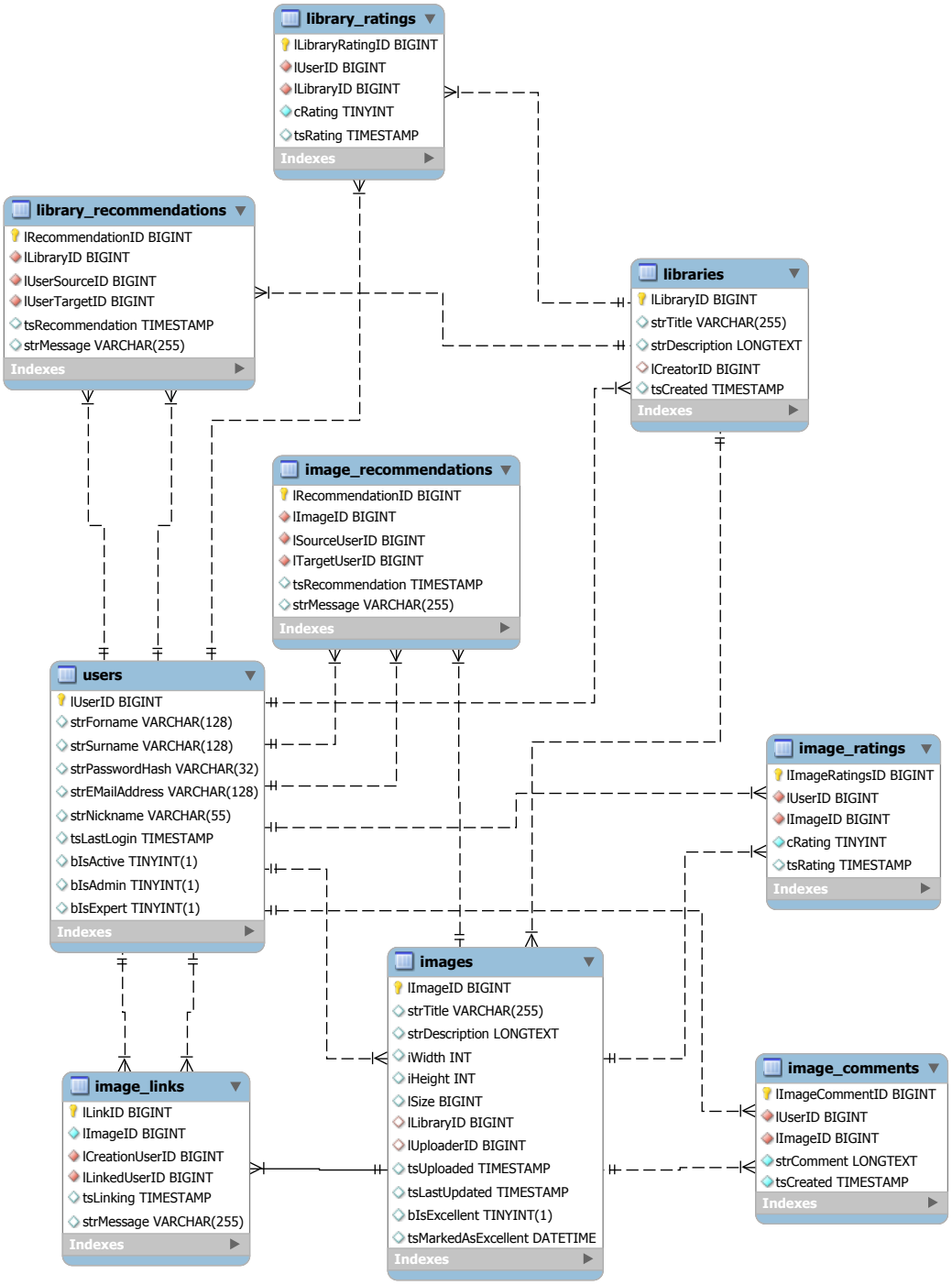

Abbildung B.1.: pixLib Datenbankschema

Sekundärschlüssel Zur Wahrung der Datenintegrität wurden auf alle `IDs` Sekundärschlüssel gelegt, welche die Eigenschaft `on delete cascade` besitzen. Diese wird verwendet, um sicherzustellen, dass beim Löschen von Objekten keine „Überreste“ von referenzierenden Tabellen erhalten bleiben. Ein weiterer Grund ist die Vereinfachung der Komplexität der Anwendung, damit innerhalb der Anwendung keine zusätzlichen Abfragen, Informationen oder händisch ausgelöste Löschungen durchgeführt werden müssen (beispielsweise müssen erst alle Kommentare und Bewertungen durch den Benutzer gelöscht werden, bevor ein Bild gelöscht werden kann).

Gespeicherte Prozeduren Um die Entwicklungszeit zu minimieren, wurde bewusst darauf verzichtet die SQL-Befehle in gespeicherte Prozeduren zu verlagern. Dies hat allerdings zur Folge, dass gegebenenfalls für die verschiedenen Implementierungen der Server die SQL-Befehle modifiziert werden müssen, damit diese in dem jeweiligen System die richtige Syntax für die verwendeten *vorbereitete Anweisungen* besitzen.

C

Verwendung der ausgewählten Frameworks

Dieser Anhang soll einen Überblick über die Verwendung der ausgewählten Frameworks geben. Es wird beschrieben, wie diese installiert und verwendet werden können. Vertiefende Anleitungen und Tipps sind den jeweiligen Webseiten der Hersteller oder alternativen Quellen zu entnehmen. Es wird von einem vorkonfiguriertem System ausgegangen, sprich bei einem Framework, das in einem *Apache Tomcat* laufen soll, wird ein vorinstalliertes System mit einer funktionierenden Installation des *Apache Tomcat* erwartet.

Für jedes Framework wird die notwendige vorinstallierte Software angegeben. Spezifische Einstellungen eines Frameworks für das System werden ausgehend von einer Standard-Installation der Komponenten beschrieben.

C.1. PHP - Recess

Dieser Abschnitt befasst sich mit der Installation und der Verwendung des Frameworks *Recess* für PHP. Die Anleitungen und Informationen richten sich nach den Angaben des Herstellers[1], den Hinweisen aus Jordan [2009a] und den Erfahrungen des Autors, die bei dieser Ausarbeitung entstanden sind.

[1] `http://www.recessframework.org/page/documentation`

C.1.1. Installation

Das Framework *Recess* benötigt einen *Apache HTTP-Server*[2] mit vorinstalliertem PHP in der Version *5.2.4* oder höher. Es wird empfohlen, dass das Apache-Modul *mod_rewrite* installiert und konfiguriert ist, da sonst das Framework nicht korrekt funktioniert. Dies äußert sich beispielsweise durch nicht geladene Bilder oder CSS. Für die Installation sind die folgenden Schritte notwendig:

1. Entpacken des heruntergeladenen Archivs in das Dokumentenverzeichnis des Webservers (z.B. `C:/Programme/ApacheSoftwareFoundation/Apache2.2/htdocs` unter Windows oder `/srv/www/htdocs` unter Linux).

2. Wenn es sich um eine Entwicklungsumgebung handelt, müssen einige Verzeichnisse für den PHP-Prozess beschreibbar sein, damit die Konfigurationsoberfläche in der Lage ist, die Kontroll- und Modell-Klassen automatisiert zu erstellen. Dabei handelt es sich um die Verzeichnisse `apps/`, `recess/temp/` und `recess/sqlite/`.

3. In der Datei `recess-conf.php` muss die Datenbank konfiguriert werden. Dabei ist zu entscheiden, ob eine *mySQL* oder eine *Sqllite* Datenbank verwendet werden soll. Dementsprechend muss die jeweilige Zeile einkommentiert und konfiguriert werden.

4. Danach ist die Installation und die Grundkonfiguration abgeschlossen. Es kann nun über die Weboberfläche eine Applikation erstellt werden. Hierfür muss in einem Browser die Adresse des Webservers gefolgt vom relativen Installationspfad angegeben werden. Es sollte eine Seite mit dem Text *„Welcome to Recess!“* erscheinen.

C.1.2. Verwendung

Zur Erstellung und Konfiguration einer Anwendung mit *Recess* kann die mitgelieferte Weboberfläche *Recess! Tools!* verwendet werden. Die Klassen werden anhand von Vorlagen generiert, die nur noch mit konkreten Werten gefüllt werden. Dies geschieht mithilfe der *schrittbasierten* Weboberfläche sehr schnell und einfach. Nachteilig ist allerdings, dass diese bei Modellen und Ressourcen keine Flexibilität für komplexere Strukturen, wie beispielsweise Ressourcen, die sich aus mehr als einer Datenbanktabelle zusammensetzen, bietet. Die Weboberfläche sollte allerdings zum Anlegen einer neuen Anwendung verwendet werden.

[2] `http://httpd.apache.org`

Für die weitere Entwicklung und Anpassung der generierten Skripte empfiehlt es sich *Eclipse*[3] mit dem *PDT-Plugin*[4] zu verwenden. In den Anleitungen auf der Herstellerwebseite[5] gibt es ebenfalls eine gute Einweisung anhand eines durchgängigen Beispiels. Eine neue Anwendung lässt sich wie folgt erstellen:

1. Aufrufen der Weboberfläche durch Adresse des Webservers gefolgt mit dem relativen Installationspfad zum Webverzeichnis mit dem Zusatz `/recess`, also beispielsweise `http://localhost/recess`.

2. Den Punkt *Datenbank* auswählen und prüfen, ob die gewünschte Datenbank angezeigt wird. Sollten mehrere aufgeführt sein, so sollte die zu verwendende Datenbank ausgewählt werden.

3. Über den Menüpunkt *Apps* können die aktuell vorhandenen Anwendungen eingesehen und neue angelegt werden. Eine weitere Anwendung wird wie folgt angelegt:

   a) Gestartet wird der Vorgang durch den Menüpunkt *Start a New Application*.

   b) Im ersten Schritt wird die Anwendung benannt, indem der *Human name* und der *Programmatic name* vergeben wird. Ein Beispiel für den *Human name* wäre *My First Recess App* und für den *Programmatic name* wäre *FirstApp* denkbar. Weiter geht es dann mittels *Next Step*.

   c) Der *Url prefix* über den die Anwendung später erreichbar ist, wird nun festgelegt. Nachdem erneut auf *Next Step* geklickt wurde, werden die Dateien und Verzeichnisse für die erste Applikation automatisch generiert. Im nächsten Schritt muss die Applikation im Framework registriert werden.

   d) Damit die Applikation vom Framework erkannt wird, muss diese in der Konfigurationsdatei eingetragen werden. Dies ist einmalig pro Applikation durchzuführen. Die restlichen Informationen zu der Anwendung, wie beispielsweise die Ressourcen, werden automatisch erkannt. Ein beispielhafter Eintrag für eine Anwendung wird in Quelltext C.1 dargestellt.

```php
<?php
    // ...
    RecessConf::$applications = array(
        'recess.apps.tools.RecessToolsApplication',
        'FirstApp.FirstAppApplication', // <-- ADD THIS LINE
    );
    // ...
?>
```

Quelltext C.1: Recess Applikationskonfiguration

[3] `http://www.eclipse.org`
[4] `http://www.eclipse.org/pdt`
[5] `http://www.recessframework.org/page/starting-an-app-in-the-recess-php-framework`

Die Anwendung kann nun beliebig erweitert werden; beispielsweise durch das Anlegen von Ressourcen. Auf diese Schritte wird hier allerdings nicht eingegangen, weder über die Weboberfläche noch über *Eclipse*. Diese können mithilfe der Herstellerseite oder anderen Quellen vertieft werden.

C.1.3. Publizierung

Anwendungen für *Recess* müssen nicht direkt publiziert werden, da sie bereits auf dem Server entwickelt werden. Sollte die Anwendung auf einen anderen Server portiert werden, so muss das entsprechende Verzeichnis der Anwendung `<Install Dir>/recess/apps/<App Name>` kopiert und die zugehörigen Datenstrukturen, sowie die Einstellungen aus der Konfigurationsdatei übertragen werden.

C.1.4. Verwenden von XML für Content Negotiation

Um bei *Recess* für die *Content Negotiation* das XML-Format zu verwenden, muss eine zusätzliche *View* installiert werden, da das Framework nur eigene *Layouts* bestehend aus *Templates* oder JSON unterstützt. Um in den Vorlagen XML unterzubringen muss viel Aufwand betrieben werden, da diese nur als Templates dienen. Für die Installation der zusätzlichen View müssen die folgenden Schritte durchgeführt werden:

1. Die Datei `XmlView.class.php` muss aus dem Git-Repository von Ryan Day[6] heruntergeladen werden. Bei Git[7] handelt es sich um eine freie verteilte Versionsverwaltung.

2. Die im vorherigen Schritt heruntergeladene Datei muss in das Verzeichnis `<Install Dir>/recess/framework/views` kopiert werden.

3. Anschließend muss diese Datei dem Framework bekannt gemacht werden. Hierzu muss in der Datei `<Install Dir>/recess/framework/DefaultPolicy.class.php` der Eintrag von `Zeile 3` aus Quelltext C.2 eingetragen werden.

```
1 <?php
2     // ...
3     Library::import( 'recess.framework.views.XmlView' );
4     // ...
5 ?>
```

Quelltext C.2: Recess XMLView registrieren

[6] `git://github.com/rday/recess.git`
[7] `http://git-scm.com`

4. Um XML zu aktivieren, muss in der Ressource die entsprechende Annotation in `!RespondsWith Xml` umgeändert werden.

Damit auch das Empfangen und Deserialisieren von XML ermöglicht wird, muss ebenfalls wieder Änderungen im Framework durchgeführt werden. In der Datei `<Install Dir>/recess/framework/controllers/Controller.class.php` muss in der Methode `wrappedServe` dafür gesorgt werden, dass ein entsprechendes Datenmodell instantiiert und mit den Werten aus dem XML gefüllt wird. Der Quelltext C.3 zeigt eine beispielhafte Implementierung dieser Funktion.

```
1  <?php
2      if (    'POST' == $this->request->method
3          ||  'PUT'  == $this->request->method ) {
4          // Nur bei POST und PUT, da sonst keine Daten zum Server uebertragen werden...
5          if (    empty( $_SERVER[ 'CONTENT_TYPE' ] )
6              ||  false !== strpos( $_SERVER[ 'CONTENT_TYPE' ], 'xml' ) ) {
7              // Nur wenn kein Content-Type angegeben wurde oder es ein XML-Typ ist
8              $this->m_oInjectedModel = XmlModelDecoder::decode( $this->request->input );
9
10             if ( null == $this->m_oInjectedModel ) {
11                 throw new RecessException('Given Model is invalid.', array());
12             }
13         }
14     }
15 ?>
```

Quelltext C.3: Recess XML-Konsumierung

Die Implementierung des verwendeten `XmlModelDecoder` kann dem Quelltext der Anwendung entnommen werden.

C.2. Java - Jersey

Dieser Abschnitt befasst sich mit der Installation und der Verwendung des Frameworks *Jersey* für Java.

C.2.1. Installation

Für das Framework *Jersey* muss außer Java nichts installiert werden. Es müssen nur die entsprechenden Bibliotheken, in diesem Fall Java Archive (JAR) Dateien heruntergeladen und entweder im Projekt eingebettet, oder im Bibliotheksverzeichnis des Webservers abgelegt werden. Es handelt sich dabei um die folgenden Bibliotheken:

- asm-3.1.jar
- jersey-core-1.4.jar

- jersey-server-1.4.jar *(nur serverseitig)*
- jersey-client-1.4.jar *(nur clientseitig)*

Eine Installation im klassischen Sinne ist daher nicht notwendig. Wie die Bibliotheken in ein entsprechendes Web-Projekt eingebunden werden können, zeigt der nächste Abschnitt.

C.2.2. Verwendung

Unter *Eclipse* kann ein entsprechendes Projekt durch *New > Web > Dynamic Web Project* erstellt werden. Die notwendigen Bibliotheken werden dann, sofern diese eingebettet werden sollen, in das Verzeichnis `WebContent/WEB-INF/lib` kopiert. Diese JAR-Dateien werden automatisch beim Erstellen einer WAR-Datei eingebettet.

Damit *Jersey* für das Projekt aktiviert wird, muss die Beschreibungsdatei, der so genannte *Web Descriptor* `web.xml`, wie im Quelltext C.4 dargestellt, editiert werden. Mit dieser Einstellung sucht *Jersey* beim Laden der Anwendung die Ressourcen anhand der vergebenen Annotation. Es wird dadurch also festgelegt, für welche URLs der Webcontainer das Framework ansprechen soll.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
       http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
3
4      <!-- Anzeigename der Anwendung -->
5      <display-name>pixLibJavaServer</display-name>
6
7      <!-- Jersey-Servlet deklarieren -->
8      <servlet>
9          <servlet-name>pixLib</servlet-name>
10         <servlet-class>
11             com.sun.jersey.spi.container.servlet.ServletContainer
12         </servlet-class>
13         <load-on-startup>1</load-on-startup>
14     </servlet>
15
16     <!-- Jersey-Servlet fuer alle URLs verwenden -->
17     <servlet-mapping>
18         <servlet-name>pixLib</servlet-name>
19         <url-pattern>/*</url-pattern>
20     </servlet-mapping>
21 </web-app>
```

Quelltext C.4: Webdescriptor für Jersey

Im Quelltext C.4 wird ab *Zeile 8* ein Servlet namens *pixLib* definiert, welches aus der Klasse `ServletContainer` des Frameworks besteht. Die Option `load-on-startup` sorgt dafür, dass beim Starten der Webanwendung dieses *Servlet* geladen wird. Bei diesem Ladevorgang werden die vorhandenen Ressourcen identifiziert und im Framework entsprechend registriert. Im `servlet-mapping` wird durch das `url-pattern` eine oder mehrere URLs

auf ein *Servlet* gemapped. Dadurch weiss der Webcontainer, wohin eine Anfrage weitergeleitet werden soll. Im Normalfall wird bei *Jersey* als Muster `/*` angegeben, so dass das dort definierte *Servlet* als *Front-Controller* eingesetzt wird, welcher sich selber um die Weiterleitung der Anfragen an die Ressourcen kümmert. Weitere Informationen für das Mapping von *Servlets* und URLs bietet Kulandai [2008].

Konfiguriert wird *Jersey* über Annotationen. In Olakara [2009] können die vom Framework unterstützten Annotationen eingesehen werden. Ressourcen werden als reine POJO-Klassen implementiert.

C.2.3. Publizierung

Sollte die Anwendung, wie zuvor beschrieben, mittels *Eclipse* entwickelt werden, so kann mittels eines Rechtsklick auf *Deployment Descriptor: <appName>* und *Export > WAR File* die Anwendung exportiert werden. Die daraus entstandene WAR Datei kann anschließend über den *Tomcat Application Manager* installiert werden. Dieser lässt sich über einen Browser unter dem relativen Pfad `/manager/html` erreichen. Existiert die Anwendung bereits, ist diese zunächst über den zugehörigen Link *Undeploy* zu entfernen. Hochgeladen wird die Datei über das Formular unterhalb der Anwendungsliste, in dem die WAR Datei auswählt und anschließend auf den Knopf *Deploy* geklickt wird.

Alternativ kann das Erstellen der WAR Datei und das Installieren im Webserver auch automatisiert über ein Skript, beispielsweise über *Apache ANT*[8], erfolgen.

C.3. C# - Windows Communication Foundation

Dieser Abschnitt befasst sich mit der Installation und der Verwendung des Frameworks *Windows Communication Foundation* für C#.

C.3.1. Installation

Bei *Windows Vista*, *Windows 7* und *Windows Server 2008* ist das *WCF* bereits integriert. Bei älteren Betriebssystemen wird dies mit dem *.NET*-Framework installiert. Bei einer vorhandenen Installation von *Visual Studio 2010* ist dies ebenfalls bereits integriert.

[8] `http://ant.apache.org`

C.3.2. Verwendung

Um ein *WCF*-Projekt zu erstellen, wird empfohlen eine vorhandene Vorlage zu verwenden. Solch eine Vorlage erstellt eine Grundstruktur und Beispielklassen für das Projekt. Über das Menü in *Visual Studio* kann durch *Neues Projekt => Online Vorlagen => WCF => WCF REST Service Template 40(CS)* die empfohlene Vorlage ausgewählt werden. Das Auswahlfenster wird in Abbildung C.1 gezeigt.

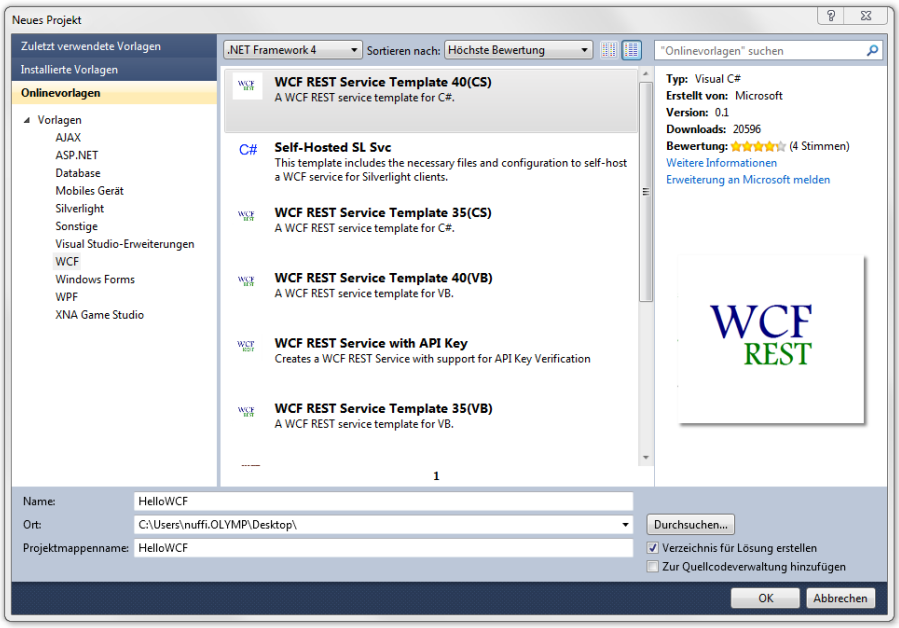

Abbildung C.1.: WCF-Projekt Vorlage

Gesteuert wird *WCF* ausschließlich über *Attribute*. Ressourcen werden als reine POCO-Klassen implementiert. Es ist nur zu beachten, dass implementierte Ressourcen im Gegensatz zu *Jersey* und *Recess* nicht automatisch erkannt werden, sondern explizit registriert werden müssen. Dafür muss der relative Pfad und die Klasse, die die Ressource implementiert, zur Routingtabelle hinzugefügt werden. Dies wird im Quelltext C.5 dargestellt.

```
1 RouteTable.Routes.Add(
2                       new ServiceRoute(
3                                        "relativeURL",
4                                        new WebServiceHostFactory(),
5                                        typeof( RessourceClass )
6                       )
7 );
```

Quelltext C.5: Ressourcenregistrierung bei WCF

C.3.3. Publizierung

Für die Publizierung einer *WCF*-Webanwendung gibt es verschiedene Möglichkeiten. Es wird hier die Möglichkeit der *Remote-Publizierung* über das *Visual Studio* beschrieben. Zunächst muss für die Webseite das richtige *.NET*-Framework eingestellt sein. Dies kann unter den Eigenschaften der entsprechenden Webseite unter dem Reiter *ASP.NET* kontrolliert und eingestellt werden. Weiterhin muss der Dienst *Web Deploy*[9] installiert und gestartet sein. Eine Anleitung zur Installation und Konfiguration dieses Dienstes findet sich auf der offiziellen Webseite des IIS[10].

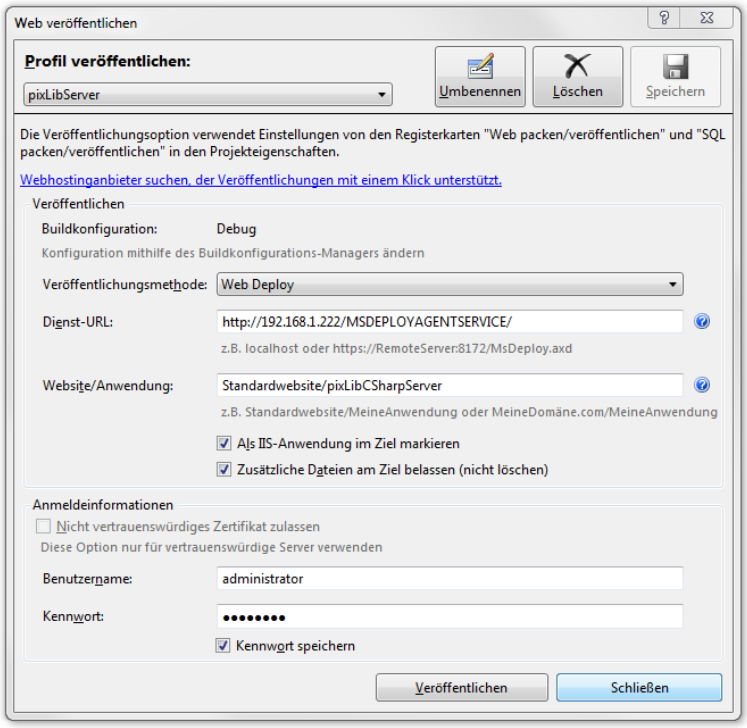

Abbildung C.2.: WCF-Publizierung

Sind die zuvor erwähnten Vorbedingungen erfüllt, kann die Anwendung über einen Rechtsklick auf das Projekt und der Auswahl des Menüpunkts *Veröffentlichen* publiziert werden. Die Abbildung C.2 zeigt den Dialog für die Veröffentlichung der Anwendung.

[9] `http://www.iis.net/download/webdeploy`
[10] `http://learn.iis.net/page.aspx/421/installing-web-deploy`

D

Testen der Anwendung

Eine große Herausforderung bestand darin, alle drei Server-Varianten einheitlich zu testen, ohne eine große Menge an redundanten Testfällen schreiben zu müssen. Dieses Vorgehen hätte für eine möglichst gute Testabdeckung unverhältnismäßig viel Zeit in Anspruch genommen. Aus diesem Grund wurde ein kleines *Testframework* entworfen, welches durch einfache Parameter viele Tests durchführen kann.

Für die Realisierung dieses Frameworks wurde *JUnit*[1] in der Version `4.5` zusammen mit der Client-Bibliothek von *Jersey* eingesetzt. Dafür wurde das Prinzip eines so genannten *Data Providers* [siehe Bergmann, 2009] eingesetzt, bei dem eine Sammlung von Daten als Eingabeparameter verwendet wird. Unter *JUnit* wird dieses Verfahren auch *parametrisiertes Testen*[2] genannt.

Um die drei Server-Varianten sowohl gesammelt, als auch einzeln testen zu können, wurde eine *Testsuite*[3] angelegt, welche weitere *Testsuiten* zusammenfasst. Diese Testsuite wird im Quelltext D.1 für alle drei Server-Implementierungen dargestellt.

```
1  /**
2   * Testsuite um alle Webservices zu testen
3   *
4   * @author Guido Kaiser
5   * @version 1.0
6   */
7  @RunWith(Suite.class)
8  @SuiteClasses( { TS_JavaWebservice.class, TS_CSharpWebservice.class, TS_PHPWebservice.class } )
9  public class TS_AllWebservices {
10
11     /**
12      * Vor den Tests aufräumen
13      */
14     @BeforeClass
15     public static void setUp() {
```

[1] `http://www.junit.org`
[2] `http://www.mkyong.com/unittest/junit-4-tutorial-6-parameterized-test`
[3] Ansammlungen von Testfällen und weiteren Testsuiten.

```
16        System.out.println( "cleaning up before complete test..." );
17      }
18
19      /**
20       * Nach den Tests aufräumen
21       */
22      @AfterClass
23      public static void tearDown() {
24        System.out.println( "cleaning up after complete test..." );
25      }
26  }
```

Quelltext D.1: Testsuite für alle Webservices

Jeder dieser drei *Testsuiten* dient eigentlich nur dazu einem selbst geschriebenen `TestController` mitzuteilen, welche Server-Variante getestet werden soll. Diese *TestSuiten* nutzen die von *JUnit* bereitgestellten Annotationen `@BeforeClass` und `@AfterClass`, die jeweils am Anfang bzw. am Ende des gesamten Testvorgangs, der von dieser Klasse ausgeht, ausgeführt werden. Solch eine *TestSuite* ist im Quelltext D.2 dargestellt.

```
1  /**
2   * Testsuite um den Java Webservice zu testen
3   *
4   * @author Guido Kaiser
5   * @version 1.0
6   */
7  @RunWith(Suite.class)
8  @SuiteClasses( { TS_AllTests.class } )
9  public class TS_JavaWebservice {
10
11      /**
12       * Webservice konfigurieren
13       */
14      @BeforeClass
15      public static void setUp() {
16        TestController.getInstance().setCurrentWebservice( EWebservice.JAVA );
17      }
18
19      /**
20       * Webservice zurücksetzen
21       */
22      @AfterClass
23      public static void tearDown() {
24        TestController.getInstance().setCurrentWebservice( EWebservice.UNDEFINED );
25      }
26
27 }
```

Quelltext D.2: Testsuite für Java-Webservice

Der `TestController` ist dafür zuständig, anhand der aktuellen Einstellung, den richtigen Basis-URI der zu testenden Server-Variante zu liefern. Aufgrund dieses Mechanismus können alle Testfälle für alle Server-Varianten eingesetzt werden. Jede dieser drei *TestSuites* beinhaltet wiederum eine *TestSuite*, die alle *Testfälle* für eine Server-Variante sammelt. Die zuletzt erwähnte *TestSuite* wird im Quelltext D.3 dargestellt.

```
1  /**
2   * Testsuite um alle Tests für einen Webservice auszuführen
3   *
4   * @author Guido Kaiser
5   * @version 1.0
6   */
7  @RunWith(Suite.class)
8  @SuiteClasses( {
9    TC_HttpGet.class,
10   TC_HttpPost.class,
11   TC_HttpPut.class,
12   TC_HttpDelete.class,
```

```
13   TC_HttpOptions.class
14 } )
15 public class TS_AllTests {
16
17     /**
18      * Vor den Tests aufräumen
19      */
20     @BeforeClass
21     public static void setUp() {
22       System.out.println( "cleaning up before partial test..." );
23       System.out.println( "URL:  " + TestController.getInstance().getBaseURL());
24     }
25
26     /**
27      * Nach den Tests aufräumen
28      */
29     @AfterClass
30     public static void tearDown() {
31       System.out.println( "cleaning up after partial test..." );
32     }
33
34 }
```

Quelltext D.3: Testsuite für alle Tests

Die Testfälle werden in die fünf HTTP-Methoden `GET`, `PUT`, `POST`, `DELETE` und `OPTIONS` aufgeteilt, damit es eine semantische Trennung der Testfälle gibt. Weiterhin benötigen die verschiedenen Methoden unterschiedliche Eingabewerte. Eine Testsammlung wird in vereinfachter Form im Quelltext D.4 dargestellt. Es handelt sich hierbei um eine beispielhafte Testsammlung für die Methode `GET`.

```
1  /**
2   * Automatisierte Testklasse für HTTP-GET-Methoden
3   *
4   * @author Guido Kaiser
5   * @version 1.0
6   */
7  @RunWith(Parameterized.class)
8  public class TC_HttpGet extends WebRessourceTestCase {
9
10
11   /**
12    * Parameter fuer Tests:
13    *  1. Testfall-Name
14    *  2. Relative-URL der Ressource
15    *  3. Zusaetzlicher Pfad (bspw. ID) oder NULL
16    *  4. Art der Authentifizierung
17    *  5. Erwarteter Medientyp oder NULL
18    *  6. Erwarter Teil-Inhalt oder NULL
19    *  7. Erwartete Stati oder NULL
20    *
21    * @return Daten-Sets für die Testfälle
22    */
23   @Parameters
24   public static List<Object[]> data() {
25     Object[][] data = new Object[][] {
26       { "001",
27         RESSOURCE_USER,                          // URL: user/
28         null,
29         EAuthenticationType.NONE,                // Keine Authentifizierung
30         null,
31         null,
32         new Status[] {                           // Kompatibilitaet zwischen Recess und Jersey
33           Status.METHOD_NOT_ALLOWED, Status.NOT_FOUND } },
34       // ----------------------------------------
35       { "002",
36         RESSOURCE_USER,                          // URL: user/
37         RESSOURCE_USER_INVALIDID,                // 999
38         EAuthenticationType.NONE,                // Keine Authentifizierung
39         null,
40         null,
41         new Status[] { Status.NOT_FOUND } },
42       // ----------------------------------------
43       { "003",
44         RESSOURCE_USER,                          // URL: user/
```

```
45          RESSOURCE_USER_VALIDID,                    // 1
46          EAuthenticationType.NONE,                  // Keine Authentifizierung
47          MediaType.APPLICATION_XML,                 // Antwortformat muss XML sein
48          "<id>" + RESSOURCE_USER_VALIDID + "</id>", // Angeforderte ID muss enthalten sein
49          null },
50        // ---------------------------------------
51        { "004",
52          RESSOURCE_USER,                            // URL: user/
53          RESSOURCE_USER_VALIDID,                    // 1
54          EAuthenticationType.NONE,                  // Keine Authentifizierung
55          MediaType.TEXT_PLAIN,                      // Antwortformat muss Plaintext sein
56          null,
57          new Status[] { Status.NOT_ACCEPTABLE } },  // Darf nicht akzeptiert werden
58      };
59
60       return Arrays.asList(data);
61    }
62
63    // ------------------------------------------------------------------
64
65
66    /**
67     * Konstruktor
68     */
69    public TC_HttpGet( String strTestName, String strRessourceName, String strInstanceID,
          EAuthenticationType eAuthType, String strTypeToAccept, String strExpectedContent, Status
          eExpectedStatus ) {
70      super( strTestName, strRessourceName, strInstanceID, eAuthType, strTypeToAccept, strExpectedContent,
            eExpectedStatus );
71    }
72
73    /**
74     * Initialisieren
75     */
76    @BeforeClass
77    public static void prepare() {
78      DatabaseCleaner.cleanDatabase();
79    }
80
81
82    /**
83     * Aufräumen
84     */
85    @AfterClass
86    public static void cleanup() {
87      DatabaseCleaner.cleanDatabase();
88    }
89
90
91    /**
92     * Testen von HTTP-GET
93     */
94    @Test
95    public void testGET() {
96      System.out.println( "Testing: " + this );
97
98      try {
99        RessourceController.testGet( getRessourceName(), getInstanceID(), getAuthType(),
              getTypeToAccept(), getExpectedContent(), getExpectedStatus() );
100       System.out.println( "Result: PASS" );
101     } catch ( TestFailedException e ) {
102       System.err.println( "Result: FAIL - " + e.getMessage() );
103       fail( e.getMessage() );
104     }
105   }
106
107 }
```

Quelltext D.4: Beispiel-Tests für HTTP-GET

Werden diese Tests durchgeführt, so wird in `stdout` der aktuelle Test inklusive seiner Parameter, der zugehörige URI und das Ergebnis ausgegeben. Eine stark gekürzte Beispielausgabe kann dem Quelltext D.5 entnommen werden.

```
1 cleaning up before complete test...
2 Switching current webservice from <undefined> to <java>.
3 cleaning up before partial test...
```

```
 4 URL:  http://192.168.1.222:8080/pixLibJavaServer/
 5 Testing: TC_HttpGet(001)[ ressource=user, instanceID=null, authentication=Authentication: none,
       accept-mimetype=null, expected-status=405: Method Not Allowed, expected-content=null ]
 6 URL: http://192.168.1.222:8080/pixLibJavaServer/user
 7 Result: PASS
 8 Testing: TC_HttpGet(002)[ ressource=user, instanceID=null, authentication=Authentication: ok,
       accept-mimetype=null, expected-status=405: Method Not Allowed, expected-content=null ]
 9 URL: http://192.168.1.222:8080/pixLibJavaServer/user
10 Result: PASS
11 Testing: TC_HttpGet(004)[ ressource=user, instanceID=999, authentication=Authentication: none,
       accept-mimetype=null, expected-status=404: Not Found, expected-content=null ]
12 URL: http://192.168.1.222:8080/pixLibJavaServer/user/999
13 Result: PASS
14
15 ...
16
17 cleaning up after complete test...
```

Quelltext D.5: Unit-Test Konsolenausgabe

In Abbildung D.1 wird die Struktur des Frameworks übersichtlich in einer der UML ähnlichen Notation dargestellt. Der `TestController` wurde hier aus Gründen der Übersichtlichkeit nicht mit aufgeführt.

Aufgrund der Tatsache, dass die Frameworks unterschiedliche HTTP-Status-Codes bei verschiedenen Aufrufen liefern, wurde eine Unterstützung von mehreren Status-Codes für Testfälle implementiert. Dabei muss ein `Array` von Status-Codes an den Testfall übergeben werden. Wie in Kapitel 5.6.2 bereits kurz erwähnt, liefert das Java-Framework *Jersey* einen `405 Not allowed` und das PHP-Framework *Recess* einen `404 Not Found` bei Ressourcen die vorhanden sind, aber nicht richtig angesprochen werden. Damit ist gemeint, dass eine Ressource, deren einziger spezifizierter URI beispielsweise als `<RessourceName>` `/\protect\T1\textbraceleftid\protect\T1\textbraceright` definiert ist, und diese Ressource fälschlicherweise durch `<RessourceName>` angesprochen wird.

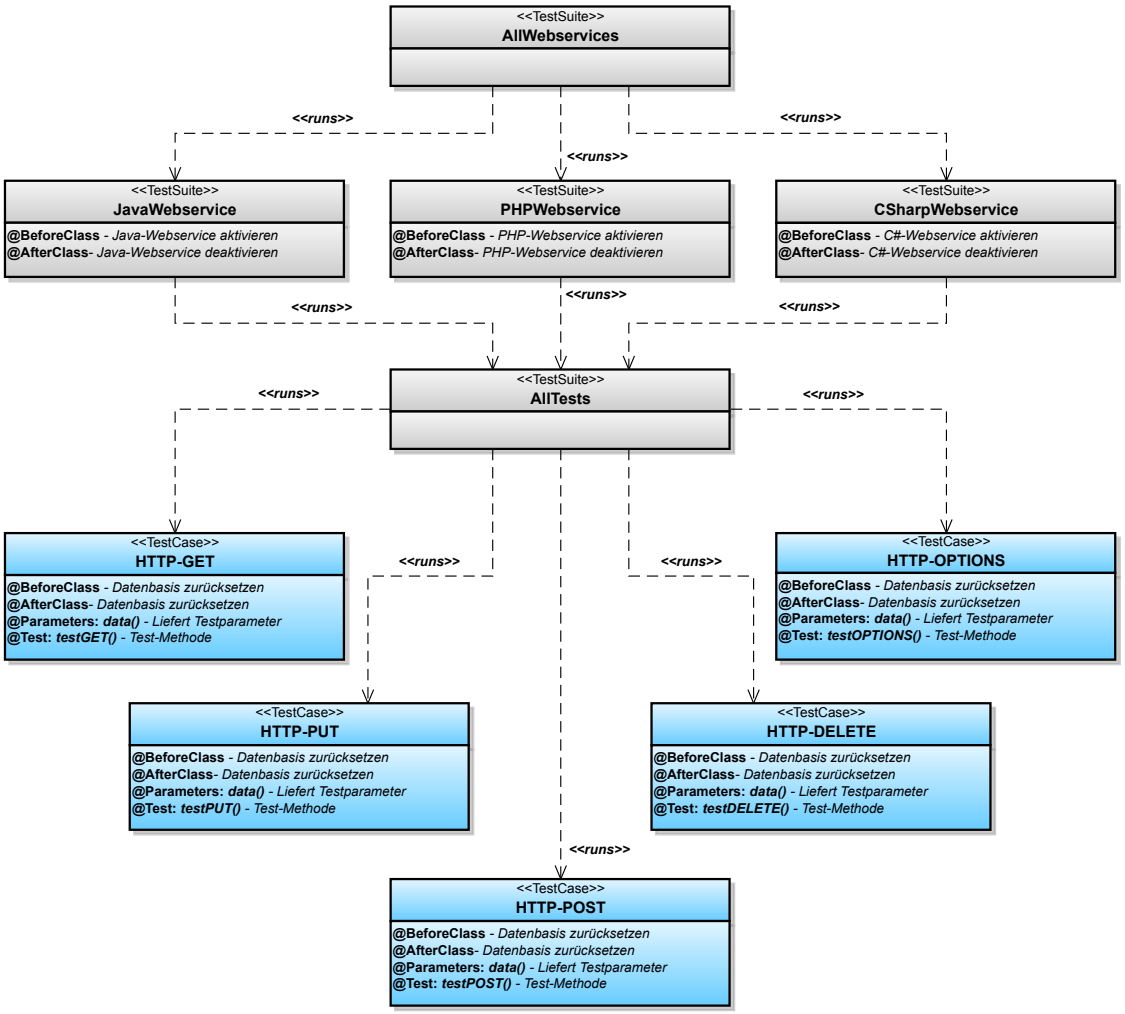

Abbildung D.1.: Struktur Test-Framework

E

Inhalt der CD

Die beigefügte CD enthält die Quelltexte der verschiedenen Implementierungen, Quelltextdokumentationen, kompilierte Versionen der Implementierungen und Entwurfs-Diagramme als Vektorgrafiken. Eine ausführliche Auflistung der enthaltenen Daten kann der Abbildung E.1 entnommen werden.

Dokument
Masterarbeit in digitaler Form als PDF.

Installationsdateien
Installationsdateien der verwendeten Programme und Werkzeuge.

Quelltext
Quelltexte der „Hello World"-Beispiele, der pixLib-Clients und -Server.

Releases
Kompilierte Dateien der „Hello World"-Beispiele, der pixLib-Clients und -Server.

Vektorgrafiken
Vektorgrafiken der Diagramme und Zeichnungen.

Abbildung E.1.: Inhalt der beigelegten CD

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
ASF	Atom Syndication Format
ASP	Active Server Pages
BMP	Windows Bitmap
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
DLL	Dynamic Link Library
DTD	Document Type Definition
GIF	Graphics Interchange Format
HATEOAS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IIS	Microsoft Internet Information Services
IP	Internet Protocol
JAR	Java Archive
JAX-RS	Java API for RESTful Web Services
JDBC	Java Database Connectivity
JEE	Java Enterprise Edition
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
JSP	JavaServer Pages
JSR	Java Specification Request
MVC	Model-View-Controller
PDF	Portable Document Format
PDO	PHP Data Objects
PHP	Personal Home Page Tools / Hypertext Preprocessor
PNG	Portable Network Graphics
POCO	Plain Old Common Language Runtime Object
POJO	Plain Old Java Object
POPO	Plain Old PHP Object
RDBMS	Relational Database Management System
REST	Representational State Transfer
RMI	Remote Method Invocation
ROA	Resource-oriented Architecture

RPC	Remote Procedure Call
RSS	Really Simple Syndication
SMTP	Simple Mail Transfer Protocol
SOA	Service-oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SRB	Storage Resource Broker
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TIFF	Tagged Image File Format
TLS	Transport Layer Security
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WADL	Web Application Description Language
WAR	Web Application Archive
WCF	Windows Communication Foundation
WebDAV	Web-based Distributed Authoring and Versioning
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extensionable Markup Language
XOP	XML-binary Optimized Packaging
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformation
YAML	YAML Ain't Markup Language

Literaturverzeichnis

[Adobe Systems 2008] Adobe Systems (Hrsg.): *Document management - Portable document format - Part 1 - PDF 1.7*. Juli 2008. – URL `http://www.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf`. – Online Ressource - Abruf: 16.12.2010, 14:23

[Bayer 2002] Bayer, Thomas: *REST Web Services - Eine Einführung*. November 2002. – URL `http://www.oio.de/public/xml/rest-webservices.pdf`. – Online Ressource - Abruf: 16.12.2010, 19:30

[Bayer und Sohn 2007] Bayer, Thomas ; Sohn, Dirk M.: *REST Web Services - Eine Einführung*. 2007

[Bergmann 2009] Bergmann, Sebastian: *PHPUnit Manual*. 2009. – URL `http://www.phpunit.de/manual/3.5/en/index.html`. – Online Ressource - Abruf: 18.12.2010, 11:25

[Berners-Lee 1994] Berners-Lee, T.: *RFC1630 - Universal Resource Identifiers in WWW*. Juni 1994. – URL `http://tools.ietf.org/rfc/rfc1630.txt`. – Online Ressource - Abruf: 16.12.2010, 13:06

[Berners-Lee u. a. 2005] Berners-Lee, T. ; Fielding, Roy T. ; Masinter, L.: *RFC3986 - Uniform Resource Identifiers (URI): Generic Syntax*. Januar 2005. – URL `http://tools.ietf.org/rfc/rfc3986.txt`. – Online Ressource - Abruf: 16.12.2010, 13:04

[Birrel und Nelson 1983] Birrel, Andrew D. ; Nelson, Bruce J.: *Implementing Remote Procedure Calls*. März 1983. – URL `http://www.cs.yale.edu/homes/arvind/cs422/doc/rpc.pdf`. – Online Ressource - Abruf: 16.12.2010, 19:22

[Bray u. a. 2008] Bray, Tim ; Paoli, Jean ; Sperberg-McQueen, C. M. ; Maler, Eve ; Yergeau, François: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. November 2008. – URL `http://www.w3.org/TR/xml/`. – Online Ressource - Abruf: 16.12.2010, 13:50

[Cerf u. a. 1974] Cerf, Vinton ; Dalal, Yogen ; Sunshine, Carl: *RFC0675 - Specification of Internet TCP*. Dezember 1974. – URL `http://www.ietf.org/rfc/rfc0675.txt`. – Online Ressource - Abruf: 16.12.2010, 13:13

[Christensen u. a. 2001] Christensen, Erik ; Curbera, Francisco ; Meredith, Greg ; Weerawarana, Sanjiva: *Web Services Description Language (WSDL) 1.1.* März 2001. – URL `http://www.w3.org/TR/wsdl`. – Online Ressource - Abruf: 16.12.2010, 14:00

[Clark 1999] Clark, James: *XSL Transformations (XSLT) Version 1.0.* November 1999. – URL `http://www.w3.org/TR/xslt`. – Online Ressource - Abruf: 16.12.2010, 13:58

[Connolly und Masinter 2000] Connolly, D. ; Masinter, L.: *RFC2854 - The 'text/html' Media Type.* Juni 2000. – URL `http://www.ietf.org/rfc/rfc2854.txt`. – Online Ressource - Abruf: 13.12.2010, 20:05

[Costello 2003] Costello, Roger L.: *Building Web Services the REST Way.* Januar 2003. – URL `http://www.xfront.com/REST-Web-Services.html`. – Online Ressource - Abruf: 30.09.2010, 11:14

[Crockford 2006] Crockford, D.: *RFC4627 - The application/json Media Type for JavaScript Object Notation (JSON).* Juli 2006. – URL `http://www.ietf.org/rfc/rfc4627.txt`. – Online Ressource - Abruf: 16.12.2010, 14:19

[Delisle u. a. 2006] Delisle, Pierre ; Luehe, Jan ; Roth, Mark: *JSR-000245 - JavaServer PagesTM Specification 2.1.* Mai 2006. – URL `http://jcp.org/aboutJava/communityprocess/final/jsr245/index.html`. – Online Ressource - Abruf: 13.12.2010, 20:49

[Dusseault und Snell 2010] Dusseault, Lisa ; Snell, James M.: *RFC5789 - PATCH Method for HTTP.* März 2010. – URL `http://tools.ietf.org/rfc/rfc5789.txt`. – Online Ressource - Abruf: 14.12.2010, 09:14

[Fielding 2000] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Dissertation, 2000

[Fielding u. a. 1999] Fielding, Roy T. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T.: *RFC2616 - Hypertext Transfer Protocol – HTTP/1.1.* Juni 1999. – URL `http://www.w3.org/Protocols/rfc2616/rfc2616.html`. – Online Ressource - Abruf: 13.12.2010, 22:03

[Gabhart 2003] Gabhart, Kyle: *J2EE pathfinder: Create and manage stateful Web apps.* Juli 2003. – URL `http://www.ibm.com/developerworks/java/library/j-pj2ee6.html`. – Online Ressource - Abruf: 17.12.2010, 16:32

[Gamma u. a. 2009] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissi-

DES, John: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, München, Februar 2009. – ISBN 978-3827328243

[Goland u. a. 1999] GOLAND, Y. ; WHITEHEAD, E. ; FAIZI, A. ; CARTER, S. ; JENSEN, D.: *RFC2518 - HTTP Extensions for Distributed Authoring – WEBDAV*. Februar 1999. – URL `http://www.ietf.org/rfc/rfc2518.txt`. – Online Ressource - Abruf: 02.01.2011, 05:06

[Google] GOOGLE (Hrsg.): *Picasa Web Albums Data API*. – URL `http://code.google.com/intl/de-DE/apis/picasaweb/overview.html`. – Online Ressource - Abruf: 17.12.2010, 16:39

[Gudgin u. a. 2007] GUDGIN, Martin ; HADLEY, Marc ; MENDELSOHN, Noah ; MOREAU, Jean-Jacques ; FRYSTYK NIELSEN, Henrik ; KARMARKAR, Anish ; LAFON, Yves: *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. April 2007. – URL `http://www.w3.org/TR/soap12-part1/`. – Online Ressource - Abruf: 13.12.2010, 21:35

[Gudgin u. a. 2005] GUDGIN, Martin ; MENDELSOHN, Noah ; NOTTINGHAM, Mark ; RUELLAN, Hervé: *XML-binary Optimized Packaging*. Januar 2005. – URL `http://www.w3.org/TR/xop10/`. – Online Ressource - Abruf: 17.12.2010, 16:02

[Güvenç 2009] GÜVENÇ, Ersin: *Easyrest Rest Framework 1.0 Released (Client and Server Library)*. August 2009. – URL `http://develturk.com/2009/08/17/easyrest-rest-framework-10-released-client-and-server-library/`. – Online Ressource - Abruf: 26.09.2010, 08:35

[Hadley 2009] HADLEY, Marc: *Web Application Description Language*. August 2009. – URL `http://www.w3.org/Submission/wadl/`. – Online Ressource - Abruf: 02.01.2011, 14:08

[Hadley und Sandoz 2008] HADLEY, Marc ; SANDOZ, Paul: *JAX-RS: JavaTM API for RESTful Web Services*. September 2008. – URL `http://jcp.org/aboutJava/communityprocess/final/jsr311/index.html`. – Online Ressource - Abruf: 17.12.2010, 15:43

[Hüffmeyer 2010] HÜFFMEYER, Marc: *SOAP und REST - Ein Vergleich von service- und ressourcenorientierten Architekturen und deren Einsatz im VMA-Projekt*. April 2010. – URL `http://vma.web.fh-koeln.de/pdf/hueffmeyer_prae.pdf`. – Online Ressource - Abruf: 26.10.2010, 13:12

[Jordan 2008a] JORDAN, Kris: *Recess - Controlling the Controller*. Dezember 2008. – URL `http://www.recessframework.org/page/controlling-the-controller`. – Online Ressource - Abruf: 17.12.2010, 15:21

[Jordan 2008b] Jordan, Kris: *Recess - Routing in Recess!* Dezember 2008. – URL `http://www.recessframework.org/page/routing-in-recess-screencast`. – Online Ressource - Abruf: 17.12.2010, 15:26

[Jordan 2009a] Jordan, Kris: *The Book of Recess - Official Guide to the Recess PHP Framework.* Oktober 2009. – URL `http://www.recessframework.org/book/html/index.html`. – Online Ressource - Abruf: 15.11.2010, 20:48

[Jordan 2009b] Jordan, Kris: *Recess - Models and Relationships at a Glance.* Januar 2009. – URL `http://www.recessframework.org/page/recess-models-at-a-glance`. – Online Ressource - Abruf: 17.12.2010, 15:28

[van Kesteren 2010] Kesteren, Anne van: *XMLHttpRequest.* August 2010. – URL `http://www.w3.org/TR/XMLHttpRequest/`. – Online Ressource - Abruf: 16.12.2010, 14:33

[Kulandai 2008] Kulandai, Joseph: *What is servlet mapping?* Mai 2008. – URL `http://javapapers.com/servlet/what-is-servlet-mapping/`. – Online Ressource - Abruf: 16.12.2010, 20:58

[Lorenz und Six 2009] Lorenz, Alexander ; Six, Hans-Werner: *Software Engineering II - Methodische Entwicklung von Webapplikationen.* 2009

[Marr 2006] Marr, Stefan: *RESTful Web Services.* 2006. – URL `http://www.stefan-marr.de/pages/restful-web-services/`. – Online Ressource - Abruf: 30.09.2010, 09:48

[Mordani 2009] Mordani, Rajiv: *JavaTM Servlet Specification 3.0.* April 2009. – URL `http://jcp.org/aboutJava/communityprocess/pfd/jsr315/index.html`. – Online Ressource - Abruf: 17.12.2010, 15:57

[msdna] msdn (Hrsg.): *Attribute (C#-Programmierhandbuch).* – URL `http://msdn.microsoft.com/de-de/library/z0w1kczw%28v=vs.80%29.aspx`. – Online Ressource - Abruf: 17.12.2010, 16:30

[msdnb] msdn (Hrsg.): *Using Type dynamic (C# Programming Guide).* – URL `http://msdn.microsoft.com/en-us/library/dd264736.aspx`. – Online Ressource - Abruf: 17.12.2010, 16:14

[Nottingham und Sayre 2005] Nottingham, M. ; Sayre, R.: *RFC4287 - The Atom Syndication Format.* Dezember 2005. – URL `http://tools.ietf.org/rfc/rfc4287.txt`. – Online Ressource - Abruf: 17.12.2010, 16:53

[Object Management Group 2004] Object Management Group (Hrsg.): *Common Object Request Broker Architecture: Core Specification*. März 2004. – URL `http://www.omg.org/cgi-bin/doc?formal/04-03-12.pdf`. – Online Ressource - Abruf: 16.12.2010, 13:29

[Object Management Group 2005] Object Management Group (Hrsg.): *Unified Modeling Language: Superstructure*. August 2005. – URL `http://www.omg.org/spec/UML/2.0/Superstructure/PDF/`. – Online Ressource - Abruf: 17.12.2010, 16:23

[Olakara 2009] Olakara, Abdel: *Jersey Annotations explained!* März 2009. – URL `http://technopaper.blogspot.com/2009/03/jersey-annotations-explained.html`. – Online Ressource - Abruf: 16.12.2010, 21:00

[Oracle] Oracle (Hrsg.): *Java Remote Method Invocation - Distributed Computing for Java*. – URL `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html`. – Online Ressource - Abruf: 16.12.2010, 13:46

[Oracle 2010] Oracle (Hrsg.): *MySQL 5.1 Referenzhandbuch*. Dezember 2010. – URL `http://downloads.mysql.com/docs/refman-5.1-de.a4.pdf`. – Online Ressource - Abruf: 02.01.2011, 18:34

[Postel 1982] Postel, Jonathan B.: *RFC821 - Simple Mail Transfer Protocol*. August 1982. – URL `http://www.faqs.org/rfcs/rfc821.html`. – Online Ressource - Abruf: 14.12.2010, 15:54

[Pree 1997] Pree, Wolfgang: *Komponentenbasierte Softwareentwicklung mit Frameworks*. dpunkt, 1997

[Richardson und Ruby 2007] Richardson, Leonard ; Ruby, Sam: *RESTful Web Services*. O'Reilly, 2007

[Rivest 1992] Rivest, R.: *RFC1321 - The MD5 Message-Digest Algorithm*. April 1992. – URL `http://tools.ietf.org/rfc/rfc1321.txt`. – Online Ressource - Abruf: 02.01.2011, 19:08

[Rodriguez 2008] Rodriguez, Alex: *RESTful Web services: The basics*. November 2008. – URL `http://www.ibm.com/developerworks/webservices/library/ws-restful/`. – Online Ressource - Abruf: 08.07.2010, 15:03

[RSS Advisory Board 2009] RSS Advisory Board (Hrsg.): *RSS 2.0 Specification*. März 2009. – URL `http://www.rssboard.org/rss-specification`. – Online Ressource - Abruf: 17.12.2010, 16:50

[Schneider 2010] Schneider, Andreas: *Vergleich und Beurteilung von Java Frameworks für Web Services mit REST*, FernUniversität in Hagen, Diplomarbeit, Februar 2010

[Schreier 2010] Schreier, Silvia: *On the Need for a REST Metamodel in Jahresbericht 2009 des Bereichs Elektrotechnik und Informationstechnik, FernUniversität Hagen.* April 2010

[Schwichtenberg 2010] Schwichtenberg, Holger: *Sinn und Unsinn der Windows Communication Foundation.* Februar 2010. – URL `http://www.heise.de/developer/artikel/Sinn-und-Unsinn-der-Windows-Communication-Foundation-934625.html`. – Online Ressource - Abruf: 17.12.2010, 16:17

[Shannon und Chinnici 2008] Shannon, Bill ; Chinnici, Roberto: *JSR 316: JavaTM Platform, Enterprise Edition 6 (Java EE 6) Specification.* 2008. – URL `http://www.jcp.org/en/jsr/detail?id=316`. – Online Ressource - Abruf: 08.12.2010, 22:13

[Shen u. a. 2001] Shen, Xiaohui ; Liao, Wei-keng ; Choudhary, Alok: *Remote I/O Optimization and Evaluation for Tertiary Storage Systems through Storage Resource Broker.* 2001. – URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.15.3218&rep=rep1&type=pdf`. – Online Ressource - Abruf: 16.12.2010, 20:23

[Sletten 2008] Sletten, Brian: *REST for Java developers.* Oktober 2008. – URL `http://www.javaworld.com/javaworld/jw-10-2008/jw-10-rest-series-1.html`. – Online Ressource - Abruf: 02.11.2010, 22:12

[Sun Microsystems 2002] Sun Microsystems (Hrsg.): *Core J2EE Patterns - Front Controller.* 2002. – URL `http://java.sun.com/blueprints/corej2eepatterns/Patterns/FrontController.html`. – Online Ressource - Abruf: 17.12.2010, 15:24

[Thompson u. a. 2004] Thompson, Henry S. ; Beech, David ; Maloney, Murray ; Mendelsohn, Noah: *XML Schema Part 1: Structures Second Edition.* Oktober 2004. – URL `http://www.w3.org/TR/xmlschema-1/`. – Online Ressource - Abruf: 16.12.2010, 13:54

[Tilkov 2009a] Tilkov, Stefan: *REST und HTTP: Einsatz der Architektur des Webs für Integrationsszenarien.* dpunkt.verlag, 2009

[Tilkov 2009b] Tilkov, Stefan: RESTful Web Services mit Java. In: *Javamagazin* 1 (2009)

[Tilkov und Ghadir 2006] Tilkov, Stefan ; Ghadir, Phillip: *REST: Die Architektur des Webs.* Mai 2006

[Vonhoegen 2004] Vonhoegen, Helmut: *Einstieg in JavaServer Pages 2.0*. Galileo Computing, 2004. – URL `http://www.galileocomputing.de/katalog/buecher/titel/gp/titelID-589?GalileoSession=21121889A4tfTx-oSw0`. – Online Ressource - Abruf: 01.10.2010, 10:00

[Wells 2000] Wells, Don: *Code the Unit Test First*. 2000. – URL `http://www.extremeprogramming.org/rules/testfirst.html`. – Online Ressource - Abruf: 17.12.2010, 16:25

[White 1976] White, James E.: *RFC0707 - A High-Level Framework for Network-Based Resource Sharing*. Januar 1976. – URL `http://tools.ietf.org/rfc/rfc707.txt`. – Online Ressource - Abruf: 16.12.2010, 13:17

[Yahoo!] Yahoo! (Hrsg.): *flickr - API-Dokumentation*. – URL `http://www.flickr.com/services/api/`. – Online Ressource - Abruf: 17.12.2010, 16:36