



Technische Konzeption und Realisierung einer Multi-User Musiksammlungsverwaltung auf Basis aktueller Web Service- und Java-Techniken

Bachelorarbeit an der Universität Ulm

Vorgelegt von:

Max Käufer
max.kaeufer@uni-ulm.de

Gutachter:

Prof. Dr. Manfred Reichert

Betreuer:

Rüdiger Pryss

2014

Fassung 28. Februar 2014

© 2014 Max Käufer

This work is licensed under the Creative Commons. Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/de/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Satz: PDF- $\text{\LaTeX}$ 2_ε

Kurzfassung

In dieser Arbeit wird das Konzept einer mehrbenutzerfähigen Web-Applikation für die zentrale Verwaltung einer Musiksammlung und das Streamen der Musik entwickelt. Dieses Konzept wird mit der *Java Platform, Enterprise Edition 7 (Java EE 7)* auf der Server-Seite und dem *Ember.js* Framework auf der Client-Seite implementiert.

Danksagung

Ich bedanke mich bei meiner Familie und meinen Kommilitonen für ihre moralische Unterstützung. Weiterhin möchte ich dem Betreuer dieser Arbeit für seine Geduld und stets motivierenden Worte danken.

Inhaltsverzeichnis

1. Einleitung	1
2. Anforderungsanalyse	5
3. Verwendete Technologien	9
3.1. Server-Seite	9
3.1.1. Java Transaction API	13
3.1.2. Java Persistence API	19
3.1.3. Java EE Connector Architecture	28
3.1.4. Java API for JavaBean Validation	31
3.1.5. Java API for RESTful Web Services	32
3.1.6. Enterprise JavaBeans	34
3.1.7. Dependency Injection	38
3.2. Client-Seite	44
4. Architektur	47
4.1. Model-View-Controller	48
4.2. Ports and Adapters	53
4.3. Entity-Control-Boundary	58
4.4. Datenmodell	61
5. Implementierung und Implementierungsaspekte	67
5.1. Geschäftslogik	68
5.1.1. Entity	68

Inhaltsverzeichnis

5.1.2. Control	72
5.1.3. Boundary	74
5.2. REST Adapter	77
5.3. Transkodierung	82
6. Abgleich der Anforderungen	85
7. Fazit und Ausblick	87
A. Datenmodell	91

1

Einleitung

Für die meisten Menschen ist Musik heutzutage ein fester Bestandteil ihres Alltags. Sie hören sie während der Arbeit, während sie Sport treiben, während sie unterwegs oder zu Hause sind. Der Ort, an dem sie sich dabei befinden, soll keine Rolle mehr spielen, ihre Musik soll also für sie immer verfügbar sein.

Fast jeder Berufstätige oder Student besitzt mittlerweile einen Laptop, viele zusätzlich noch einen Desktop-PC. Selbst beim Besitz nur dieser zwei Geräte ergibt sich das Problem, dass ein Gerät für die hauptsächliche Haltung der Musik ausgewählt werden muss und die Musiksammlung in regelmäßigen Zeitintervallen mit dem anderen Gerät synchronisiert werden muss. Es lässt sich natürlich argumentieren, dass sich Musik auch in Form von simplen Audiodateien halten und in Ordnern verwalten lässt. Dann könnte die gesamte Musiksammlung vollständig auf beiden Geräten existieren. Diese Redundanz würde allerdings wieder zum gleichen Problem der Synchronisierung führen, sobald die

1. Einleitung

Sammlung auf einem der zwei Geräte erweitert würde. Das Problem multipliziert sich außerdem mit der Anzahl der Geräte.

Andererseits hat sich in den letzten Jahren der Gerätepark eines jeden vergrößert. Immer kleinere und portablere Geräte, wie Tablets und Smartphones, haben Einzug in unser Leben gehalten. Sie sollen den Wunsch erfüllen, dass immer mehr Daten und Informationen an immer mehr Orten verfügbar sind (siehe [PMLR14], [PLRH12], [SSP⁺13], [GPSR13], [GSP⁺14] und [RPR11]). Gerade bei diesen mobilen Geräten kommt jedoch das Problem hinzu, dass immer kleinere Geräte auch immer weniger Speicherplatz zur Verfügung haben (siehe [PMR13], [PTR10] und [PTKR10]).

Im Gegensatz dazu steht die Tatsache, dass es heute nicht mehr zur Debatte steht, Daten wieder zu verwerfen, und die Musiksammlungen deshalb immer weiter wachsen. Es wird also immer schwieriger, den Überblick über die Musiksammlung zu behalten oder die Musik gar nach bestimmten Kriterien zu durchsuchen oder zu filtern. Lösungen für dieses Problem bieten Systeme, die mit den Metadaten der Songs, wie z.B. ihren Künstlern oder Bewertungen des Benutzers, arbeiten.

Ein Lösungsansatz für das Problem der Synchronisierung ist die zentrale Speicherung der Musiksammlung. Die Songs werden dann automatisch, während sie angehört werden, von dem zentralen Ort abgerufen. Wenn die Metadaten der Musik auch noch zentral verwaltet werden, können auch automatisch Teile der Sammlung zu den einzelnen Endgeräten, auf denen die Musik gehört wird, synchronisiert werden. Dieser zentralisierte Ansatz bietet sich geradezu an, da gerade die mit wenig Speicherplatz ausgestatteten mobilen Endgeräte Verbindungen zu drahtlosen Netzwerken aufnehmen können.

Unternehmen wie *Google* oder *Amazon* haben sich dieses Problems mit ihren Cloud-basierten Diensten *Google Music* (siehe [Goo14]) und *Amazon Cloud Player* (siehe [Ama14b]) bereits angenommen. Für viele ist hier bereits eine Lösung gefunden. Doch wenn man diese Dienste genauer betrachtet, so erkennt man, dass das primäre Ziel dahinter das Verkaufen von Musik ist. Dementsprechend sind auch die Features dieser Applikationen auf genau dieses Ziel ausgerichtet. So werden beim Suchen nach Musik häufig Songs angeboten, die „einem gefallen könnten“. Man besitzt diese Songs natürlich noch nicht, kann sie jedoch direkt in der Applikation kostengünstig erwerben.

Zum Spannungsfeld der stets verfügbaren Musik kommt hinzu, dass nicht jeder Benutzer möchte, dass ein fremder Dienstleister weiß, welche Musik er zu welchem Zeitpunkt hört, und eventuell auch noch an welchem Ort. Beim Konsum nahezu aller Güter lassen sich Profile von Benutzern erstellen. Gerade bei immateriellen Gütern wie Büchern oder Musik, die Gedankengut enthalten, lassen sich verblüffende Schlüsse, wie z.B. die politische Gesinnung, über den Konsumenten ziehen.

Das Ziel dieser Arbeit ist es, eine Lösung, wie sie Google oder Amazon bereitstellt, zu entwickeln. Die Musiksammlung wird also zentral gespeichert, anhand der anfallenden Metadaten verwaltet und die gesamte Applikation nach dem *SaaS-Paradigma* (*Software as a Service*) in Form einer Web-Applikation bereitgestellt. Dem Benutzer wird dabei nur die eigene Musiksammlung – und keine Werbung – angezeigt und er hat genauere Kontrolle über die Features, die ihm zur Verfügung stehen. Da dem Benutzer die Software selbst und nicht nur der Dienst, den sie bereitstellt, zur Verfügung steht, kann er sie nach seinen eigenen Vorstellungen von Datenschutz und Privatsphäre an einem zentralen Ort seiner Wahl betreiben.

Abschließend wird in Abbildung 1.1 eine Übersicht dieser Arbeit mit den Zusammenhängen der einzelnen Kapitel gegeben.

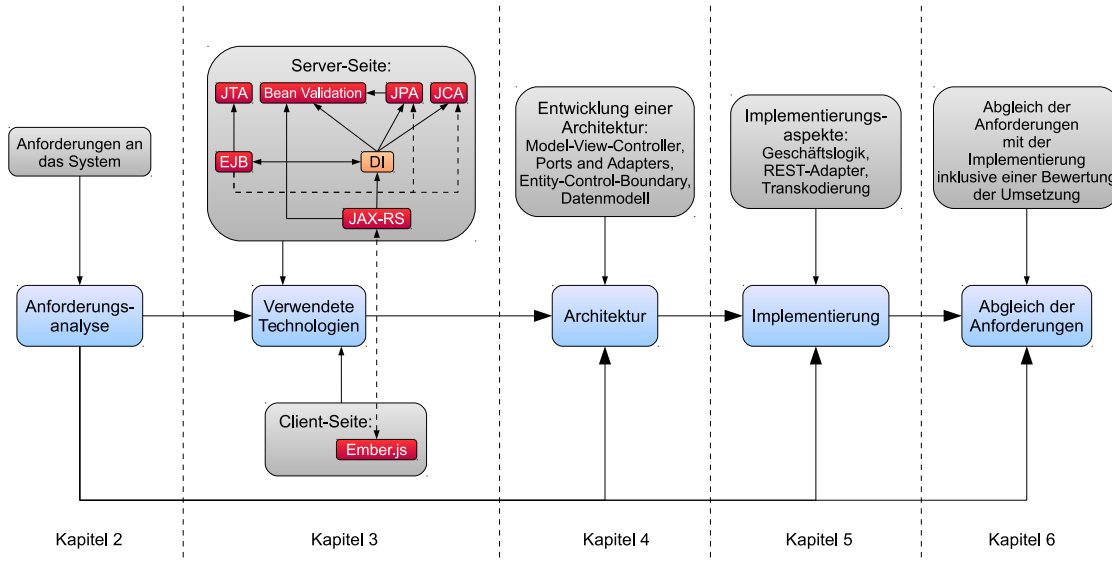


Abbildung 1.1.: Übersicht dieser Arbeit mit den Zusammenhängen der einzelnen Kapitel

2

Anforderungsanalyse

Im Folgenden werden die Anforderungen, die an das System gestellt werden, genauer vorgestellt.

- Das System soll in Form einer *Rich Internet Application* (kurz *RIA*, siehe [WF14]) implementiert werden. Dies impliziert eine Client-Server Architektur.
- Die Server-Seite soll ihre Funktionalität in Form einer *REST*-Schnittstelle (*Representational State Transfer*, siehe [Fie00]) bereitstellen.
- Die Server-Seite wird mithilfe der *Java Platform, Enterprise Edition* (siehe Abschnitt 3.1) implementiert.
- Die Client-Seite soll als *Single-Page Web-Application* implementiert werden, d.h. die Webseite wird nur einmalig beim ersten Aufruf der Web-Applikation vom Server mit den notwendigen Ressourcen geladen, danach werden nur noch asynchrone HTTP-Requests an den Server gestellt.

2. Anforderungsanalyse

- Die Client-Seite wird mithilfe des *Ember.js* Frameworks (siehe Abschnitt 3.2) implementiert.
- Die Architektur der Anwendung soll eine einfache Erweiterbarkeit sicherstellen.
- Auf die Songs, die in das System importiert worden sind, soll einfach „am System vorbei“ zugegriffen werden können. Das bedeutet, dass der Benutzer, ohne eine Export-Funktion des Systems zu verwenden, auf seine Musik zugreifen kann, z.B. wenn das System ausgefallen ist. Dies soll realisiert werden, indem die Songs als normale Dateien in einem Dateisystem auf der Server-Seite abgelegt werden.
- Die Architektur der Applikation soll sicherstellen, dass zukünftig möglichst leicht neue Formen der Speicherung von Daten hinzugefügt werden können. Insbesondere betrifft dies Orte, an denen Songs abgespeichert werden können, wie z.B. Cloud-Dienste.
- Das System soll auf der Client-Seite vollkommen ohne Browser-Plugins auskommen, insbesondere soll auf die Verwendung des *Adobe Flash Players* (siehe [Ado14]) verzichtet werden. Das Audio-Streaming erfolgt dabei mittels des *Audio-Tags* von *HTML5* (*Hypertext Markup Language*, siehe [W3C14b]).
- Weil die meisten Browser nur wenige Audio-Codecs unterstützen, muss die Musik *on-demand* beim Streaming transkodiert werden. Dies soll durch das Programm *FFmpeg* (siehe [Bel14]) erfolgen, welches ein Maximum an unterstützten Audio-Codecs bereitstellt.
- Es sollen die gleichen Semantiken gelten wie bei einer lokalen Applikation zum Abspielen von Musik. Zum Beispiel sollen Songs nur abgespielt werden können, wenn sie sich in einer Playlist befinden.
- Die Applikation soll mehrbenutzerfähig sein. Dabei sollen verschiedene Rollen existieren, die Benutzer inne haben können. Anhand dieser Rollen wird festgelegt, welche Aktionen der jeweilige Benutzer im System ausführen darf, und welche nicht.

- Jeder Benutzer, der dazu berechtigt ist, soll jeden Song im System bewerten können. Diese Bewertung ist nur für den Benutzer selbst und höchstens noch für Administratoren des Systems sichtbar.
- Das System soll automatisch festhalten, wann ein Benutzer einen Song zuletzt anhörte und wie oft er dies insgesamt tat.
- Weiterhin soll die Applikation dem Benutzer ermöglichen, die Musiksammlung nach eben diesen Kriterien zu sortieren und zu filtern.
- Jeder Song und jede Playlist gehören einem Benutzer. Zunächst ist dies jeweils der Benutzer, der den Song importierte bzw. die Playlist anlegte. Andere Benutzer, die dazu berechtigt sind, können Songs und Playlists übernehmen. Wenn ein Benutzer gelöscht wird, so werden auch alle von ihm besessenen Objekte gelöscht.

3

Verwendete Technologien

Im Folgenden werden die verwendeten Technologien beschrieben. Es wird dabei zunächst auf die Server-Seite eingegangen und anschließend auf die Client-Seite.

3.1. Server-Seite

Die einzige auf der Server-Seite eingesetzte Plattform ist die *Java Platform, Enterprise Edition* (kurz *Java EE*, siehe [Jav13a]). Sie ist eine Laufzeitumgebung und erweitert die *Java Platform, Standard Edition* (kurz *Java SE*, siehe [Rei11]) um *Cross-Cutting Concerns* (siehe [KLM⁺97]) wie *Transaktionsmanagement* (siehe Abschnitt 3.1.1), *Object-Relational Mapping* (siehe Abschnitt 3.1.2), die Anbindung an *Enterprise Information Systems* (siehe Abschnitt 3.1.3), *Dependency Injection* (siehe Abschnitt 3.1.7) und Sicherheitsaspekte wie *Authentifizierung* und *Autorisierung* (siehe 3.1.5 und 3.1.6).

3. Verwendete Technologien

In der Abbildung 3.1 wird ein Überblick über die Java EE-Architektur gegeben. Es werden dabei die logischen Beziehungen zwischen den einzelnen Komponenten gezeigt. Die Spezifikation legt ausdrücklich keine physikalische Partitionierung auf verschiedene Maschinen, Prozesse oder *Java Virtual Machines* (kurz *JVMs*) fest.

Zu den wichtigsten Konzepten von Java EE zählen die gezeigten *Container*. Sie stellen Laufzeitumgebungen dar und stellen den *Applikationskomponenten*, die in der Abbildung jeweils in den oberen Hälften der Container dargestellt werden, Dienste bereit. Diese werden in der Abbildung durch Rechtecke in den unteren Hälften der Container dargestellt. Die Container bieten den Applikationskomponenten, die in ihnen ausgeführt werden, eine gebündelte Sicht auf die zugrunde liegenden Java EE APIs (*Application Programming Interfaces*) an. Dienste, die von den Applikationskomponenten benötigt werden, wie z.B. deklaratives Transaktionsmanagement, Sicherheitsmanagement, Resource Pooling und Zustandsmanagement, können so von den Containern transparent in die Applikationskomponenten injiziert werden. Eine Voraussetzung an die Applikationskomponenten ist, dass sie **niemals** direkt miteinander interagieren, sondern stets die Protokolle und Methoden der Container verwenden. Die Container realisieren somit das *Inversion of Control*-Prinzip (siehe [GHJV11]). Sie benötigen selbst die Java SE Laufzeitumgebung, deren APIs den Applikationskomponenten innerhalb der Container zur Verfügung stehen.

Die Java EE Spezifikation definiert vier verschiedene Typen von Applikationskomponenten. Zwei davon sind die *Application clients* und *Applets*, welche im Kontext dieser Arbeit keine Rolle spielen und in der Abbildung ausgegraut wurden. Die relevanten Applikationskomponenten sind *Web-Komponenten* und *Enterprise JavaBeans* (kurz *EJBs*, siehe Abschnitt 3.1.6). Web-Komponenten können in Form von *Servlets* (siehe [Ser09]), *JSF Applikationen* (*JavaServer Faces*, siehe [JSF13]), *Filtern* (siehe [Ser09]) und *JAX-RS Ressourcen* (siehe Abschnitt 3.1.5) realisiert sein. Sie werden im Web-Container ausgeführt und antworten auf HTTP-Requests.

Die *Sicherheitsarchitektur* von Java EE folgt dem von Java SE bekannten „Write Once, Run Anywhere“ Prinzip. Demzufolge ist die Sicherheit transparent für Applikationskomponenten und deren Implementierer müssen nichts über Sicherheit wissen, um eine

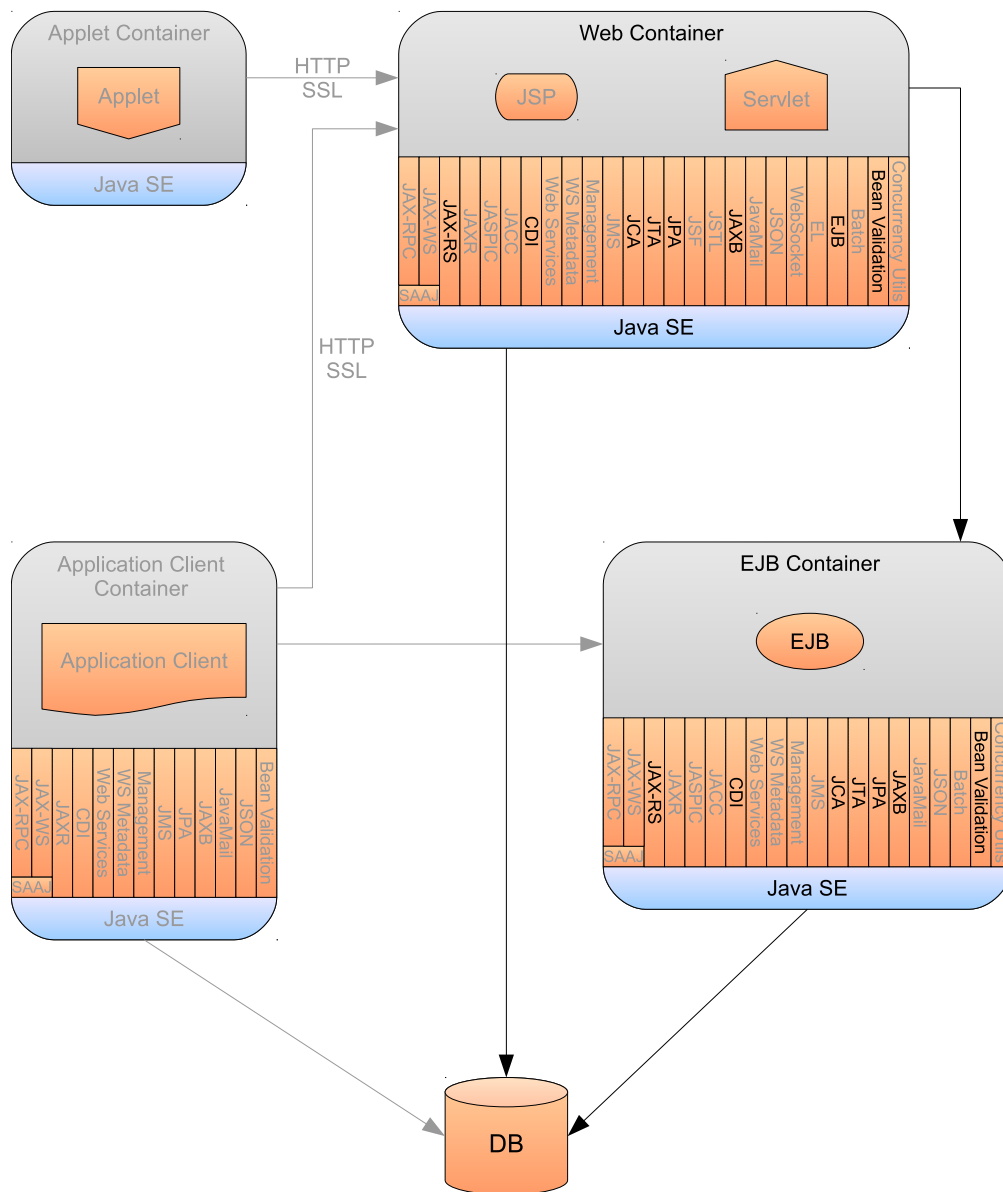


Abbildung 3.1.: Überblick über die *Java EE*-Architektur nach der *Java EE Spezifikation* (siehe [Jav13a])

3. Verwendete Technologien

zwischen Application Servern portable Applikation zu schreiben. Das *Autorisierungsmodell* basiert auf dem Konzept der *Sicherheitsrollen*. Eine Rolle ist dabei eine logische Gruppierung der Benutzer, die vom Applikationsentwickler definiert wird. Derjenige, der die Applikation im Application Server installiert (im Folgenden *Deployer* genannt), bildet die Rollen auf Identitäten in der Laufzeitumgebung ab. Dies können z.B. von ihm definierte Benutzergruppen sein. Der Zugriff auf EJB-Methoden und Web-Ressourcen wird über eben diese Rollen geregelt. Die Identität des Aufrufers von Methoden wird dabei vom Web-Container zum EJB-Container und zwischen den einzelnen EJBs einer Applikation propagiert.

Die Authentifizierung des Benutzers erfolgt durch Web Clients und Application Clients, es werden in dieser Arbeit jedoch nur die Web Clients betrachtet. Diesen stehen drei Methoden für die Authentifizierung zur Verfügung. Sie kann durch die *HTTP Basic Authentication* (siehe [BLF⁺99]) erfolgen, welche auf dem Benutzernamen und einem dazugehörigen Passwort basiert. Dieses Verfahren hat den Nachteil, dass es auf einer simplen *Base64*-Kodierung basiert und die Zugangsdaten des Benutzers bei der Übertragung somit nicht ausreichend gesichert werden. Es ist daher nötig, die Verbindung an sich per *HTTPS* zu verschlüsseln. Eine weitere Authentifizierungsmethode besteht in der *HTTPS Client Authentication*. Hierbei authentifiziert sich der Benutzer mit einem digitalen Zertifikat. Die letzte Möglichkeit ist die *Form Based Authentication*, welche auf Web-Formularen basiert. Diese Variante kann in statischen *HTML*-Seiten (*Hypertext Markup Language*, siehe [W3C14b]), Servlets, *JSPs* (*JavaServer Pages*, siehe [JSP13]) und *JSFs* verwendet werden. Die Zugangsdaten, die über einen Web-Login erlangt wurden, werden dabei in der Sitzung des Benutzers auf der Server-Seite abgelegt.

Sowohl die Authentifizierung, als auch die Autorisierung des Benutzers werden vom *Java Authentication and Authorization Service* (kurz *JAAS*) übernommen. Er erlaubt es Diensten, Benutzer zu authentifizieren und die Zugriffskontrolle durchzusetzen. Es handelt sich um eine Java-Version des *Pluggable Authentication Module Standards* (kurz *PAM*).

Bei der Verwendung der Sicherheitsmechanismen in einer Applikation wird zwischen den beiden Konzepten der *deklarativen Sicherheit* und der *programmatischen Sicherheit*

unterschieden. Die deklarative Sicherheit basiert auf *Annotationen* und Definitionen im *Deployment Deskriptor* der Anwendung, durch welche Anforderungen der Applikation an ihre Laufzeitumgebung ausgedrückt werden. Diese stellen einen Kontrakt zwischen dem *Application Component Provider*, also dem Implementierer einer Applikationskomponente, und dem Deployer dar. Der Deployer bildet beim Deployment der Anwendung die deklarativen Repräsentationen der Sicherheits-Policy der Applikation auf die Sicherheitsstrukturen der bestehenden Umgebung ab. Der Container verwendet diese Abbildung anschließend, um die Autorisierung von Benutzern durchzuführen. Die programmatische Sicherheit kann verwendet werden, wenn die deklarative Sicherheit nicht ausreicht, um die Sicherheits-Policy der Anwendung auszudrücken. In diesem Fall können beliebige Entscheidungen auf Basis der Rollen eines Benutzers getroffen werden.

Ein weiteres wichtiges Konzept der Java EE Plattform sind Annotationen und Deployment Deskriptoren. Durch sie werden Abhängigkeiten der Applikationskomponenten zu externen Ressourcen, Sicherheitsanforderungen und sonstige Anforderungen an die Laufzeitumgebung der Applikationskomponenten mit dem Deployer kommuniziert. Sie werden beim Deployment von dem Application Server interpretiert und auf dessen Fähigkeiten und Funktionalitäten abgebildet.

Weiterhin ist Java EE eine *Middleware*, die für die Entwicklung von verteilten Anwendungen verwendet werden kann, die z.B. auf Rechen-Clustern ausgeführt werden. Dieser durchaus wichtige Aspekt von Java EE wird im Weiteren jedoch nicht betrachtet.

In den folgenden Unterkapiteln werden die einzelnen für diese Arbeit relevanten Technologien der Java EE Plattform vorgestellt. Die Java EE Spezifikation verweist auf die jeweils vorgestellten anderen Spezifikationen und definiert deren Zusammenspiel.

3.1.1. Java Transaction API

Die *Java Transaction API* (kurz *JTA*, siehe [JTA13]) spezifiziert die Java Schnittstellen zwischen einem *Transaktionsmanager* und den Parteien, die in einem verteilten Transaktionssystem beteiligt sind, also der Applikation, verschiedenen *Ressourcenmanagern* (siehe Abschnitt 3.1.3) und dem Application Server. Sie besteht daher aus drei Teilen.

3. Verwendete Technologien

Der erste ist das *Demarcation Interface*, welches eine API darstellt, die es einer transaktionalen Applikation erlaubt, die Grenzen einer Transaktion abzustecken. Ein weiterer Teil besteht in einer Java-Abbildung des Industriestandards *X/Open XA Protocol*, der es einem transaktionalen Ressourcenmanager erlaubt, sich an einer globalen Transaktion zu beteiligen, die von einem externen Transaktionsmanager kontrolliert wird. Der letzte Teil von JTA besteht aus einer Schnittstelle zum Transaktionsmanager, die es einem Application Server erlaubt, die Abgrenzung von Transaktionen einer Applikation zu kontrollieren. Es wird in diesem Kontext auch von einem Java EE *Service Provider Interface* (kurz *SPI*) gesprochen, da der Application Server den Dienst verwendet, der diese Schnittstelle implementiert.

Im Folgenden wird beschrieben, wo und wie JTA in der Java EE Plattform eingeordnet werden kann. Diese Einordnung kann auch in Abbildung 3.2 nachvollzogen werden.

An den Transaktionen von JTA sind fünf Komponenten beteiligt. Der Dreh- und Angelpunkt ist der Transaktionsmanager. Er stellt die Services und Managementfunktionen bereit, die benötigt werden, um die Transaktionsdemarkation, das transaktionale Ressourcenmanagement, die Synchronisierung der Vorgänge innerhalb einer Transaktion und die Propagierung des Transaktionskontextes zwischen den Containern und Applikationskomponenten zu realisieren. Weiterhin ist er fähig, *Zwei-Phasen Commit Operationen* nach dem *X/Open XA Protocol* zu koordinieren, die eine oder mehrere XA-fähige JDBC Datenbanken, eine oder mehrere EJBs (siehe Abschnitt 3.1.6) und einen oder mehrere Ressourcenadapter (siehe Abschnitt 3.1.3) umfassen. Die Transaktionen sind somit transparent für die Applikation und können mehrere Applikationskomponenten und transaktionale Ressourcen umfassen. Vom Transaktionsmanager werden ausschließlich Schnittstellen verwendet, um mit Transaktionsservices zu kommunizieren. Die Schnittstelle des `TransactionManager` selbst erlaubt es dem Application Server, einer weiteren der fünf beteiligten Komponenten, Transaktionsgrenzen im Auftrag der Applikation zu kontrollieren. So verwaltet beispielsweise der EJB-Container die Transaktionszustände von transaktionalen EJBs und verwendet dabei das `TransactionManager` Interface, um Transaktionen abzugrenzen. Der Application Server wird in diesem Kontext auch als *Transaction Processing Monitor* (kurz *TP-Monitor*) bezeichnet. Der Transaktionsmanager führt dabei die Assoziationen zwischen dem Transaktionskontext und den Threads

der Applikation, in denen die Vorgänge einer Transaktion angestoßen werden, wobei mehrere Threads den gleichen Transaktionskontext besitzen können.

Jeder Transaktionskontext wird dabei durch ein `Transaction`-Objekt gekapselt. Objekte, die diese Schnittstelle implementieren, erfüllen mehrere Aufgaben. Der Application Server stellt der Applikation das transaktionale Ressourcenmanagement bereit. Damit ein externer Transaktionsmanager die transaktionale Arbeit der Ressourcenmanager koordinieren kann, muss der Application Server die Ressourcen, die in einer Transaktion verwendet werden, beim Transaktionsmanager ein- und austragen. Der Transaktionsmanager wird so über die Teilnahme an einer globalen Transaktion informiert. Er informiert daraufhin die teilnehmenden Ressourcenmanager über die Assoziation mit der Transaktion. Dabei werden die involvierten Ressourcen von ihm nach ihren Typen gruppiert, um das *Two-Phase Commit Transaction Protocol* nach der X/Open XA Spezifikation zwischen dem Transaktionsmanager und den Ressourcenmanagern durchzuführen. Weiterhin wird mithilfe von `Transaction`-Objekten die *Transaktionssynchronisation* realisiert. Der Application Server wird hierbei vom Transaktionsmanager durch einen *Callback* benachrichtigt bevor der *Two-Phase Commit Process* beginnt und nachdem die Transaktion abgeschlossen wurde. Zuletzt erlaubt die `Transaction`-Schnittstelle den *Commit* und das *Rollback* einer Transaktion.

Die dritte Komponente, die an den Transaktionen beteiligt ist, besteht aus einem oder mehreren Ressourcenmanagern. Er stellt der Applikation den Zugriff auf die benötigten Ressourcen zur Verfügung. Zum Beispiel wird der Zugriff auf ein *relationales Datenbankmanagementsystem* über die *JDBC-API* durch einen transaktionalen Ressourcenmanager realisiert. Durch die Implementierung der `XAResource`-Schnittstelle nimmt ein Ressourcenmanager an einer verteilten Transaktion teil. Der Transaktionsmanager verwendet diese Schnittstelle, um die Verbindung des Ressourcenmanagers mit der Transaktion, den Abschluss der Transaktion und ihr Rollback zu kommunizieren. Er bezieht ein `XAResource`-Objekt für jede transaktionale Ressource, die an einer globalen Transaktion teilnimmt. Die `XAResource`-Schnittstelle ist eine Java-Abbildung des XA-Interface basierend auf der *X/Open Common Application Environment* Spezifikation (kurz *X/Open CAE*). Sie definiert einen Kontrakt zwischen einem Ressourcen- und einem Transaktionsmanager in einer *Distributed Transaction Processing* (kurz *DTP*)

3. Verwendete Technologien

Umgebung. Ein Ressourcenadapter für einen Ressourcenmanager implementiert das `XAResource`-Interface, um eine globale Transaktion mit der transaktionalen Ressource zu assoziieren und diese Assoziation zu lösen. Eine globale Transaktion ist hierbei eine Arbeitseinheit, die von einem oder mehreren Ressourcenmanagern verrichtet wird. Die Assoziation und deren Lösung geschehen über die `start`- bzw. die `end`-Methoden des `XAResource`-Interface. Der Ressourcenmanager assoziiert dabei die Transaktion mit der gesamten Arbeit, die zwischen den Aufrufen dieser beiden Methoden verrichtet wird. Am Ende einer Transaktion werden die Ressourcenmanager dann vom Transaktionsmanager nach dem Two-Phase Commit Protocol darüber informiert, ob ein Prepare, Commit oder Rollback ausgeführt werden soll. Die Java Transaction API stützt sich bei allen diesen Vorgängen auf einen externen Transaktionsmanager wie den *Java Transaction Service* (kurz *JTS*), um globale Transaktionen zu koordinieren.

Wenn eine Transaktion nur einen einzigen Ressourcenmanager involviert, kann sie aus Gründen der Performanz zu einer lokalen Transaktion optimiert werden, die ausschließlich von diesem Ressourcenmanager verwaltet wird. Hingegen sind von anderen Containern importierte Transaktionen niemals lokal. Für die Applikation ist die Art der Transaktion, also ob sie global oder lokal ist, dabei transparent. Eine weitere Optimierung besteht im *Connection Sharing*: Hat eine Applikation mehrere Verbindungen erlangt, die den gleichen Ressourcenmanager verwenden, können ihr die Container ein und dieselbe Verbindung mehrmals innerhalb des gleichen Transaktionskontextes zur Verfügung stellen.

Eine weitere Komponente, die an den Transaktionen beteiligt ist, besteht in einer komponentenbasierten transaktionalen Applikation. Sie ist darauf angewiesen, dass der Application Server das Transaktionsmanagement über deklarative Transaktionsattribute bereitstellt. Innerhalb der Applikation existieren für den Zugriff auf das beschriebene Transaktionsmanagement im Kontext dieser Arbeit zwei Möglichkeiten: Zum einen in Web-Komponenten, die im Web-Container ausgeführt werden, und zum anderen in EJBs, die im EJB-Container ausgeführt werden. Eine bereits in einer Web-Komponente gestartete Transaktion kann auf mehrere Ressourcenmanager zugreifen und mehrere EJBs aufrufen, wobei der Transaktionskontext automatisch zu den EJBs und transaktionalen Ressourcenmanagern propagiert wird. Eine Transaktion darf dabei nicht mehrere

HTTP-Requests des Clients umfassen. Sie muss beendet sein bevor die Methode, die die HTTP-Response generiert, verlassen wird, ansonsten wird vom Web-Container ein Fehler geworfen und ein Rollback der Transaktion durchgeführt.

Die letzte an den Transaktionen beteiligte Komponente ist der *Communication Resource Manager*. Er realisiert die Transaktionskontextpropagierung zwischen dem Transaktionservice und den externen Ressourcenmanagern und erlaubt den Zugriff auf den Transaktionservice über eingehende und ausgehende Requests von und zu den externen Ressourcenmanagern.

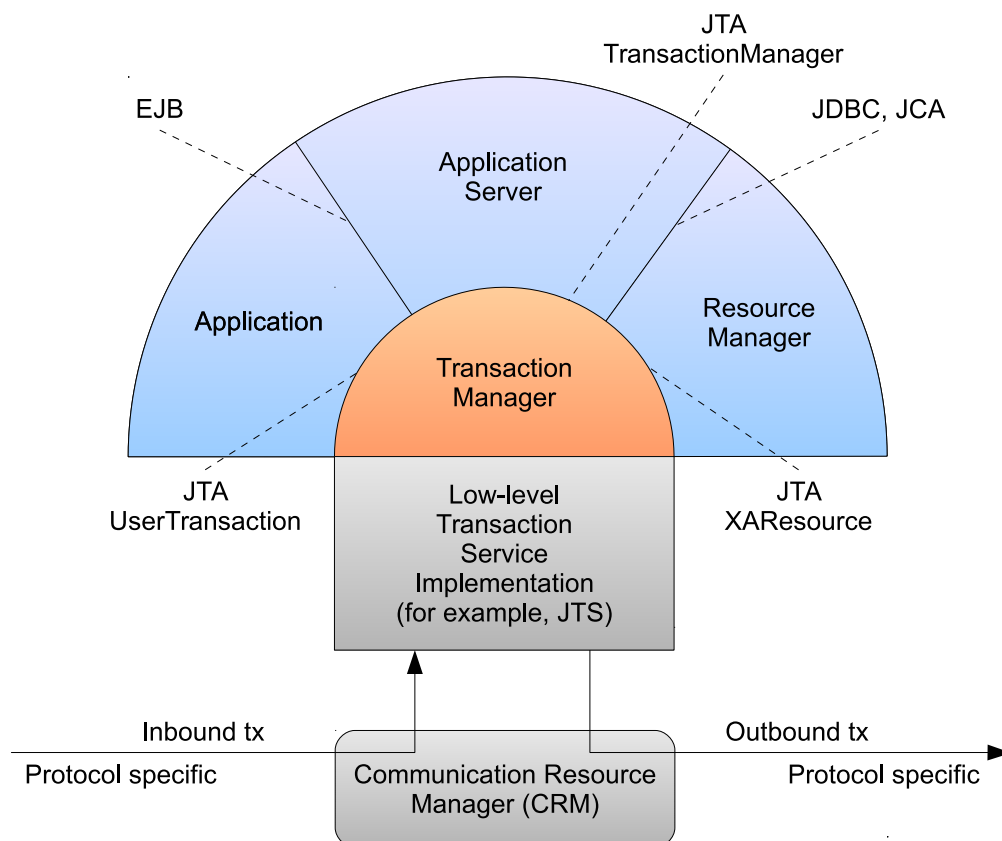


Abbildung 3.2.: Einordnung der Java Transaction API in Java EE nach der *JTA Spezifikation* (siehe [JTA13])

3. Verwendete Technologien

Im Folgenden wird die Verwendung des Transaktionsmanagements in der Applikation und der damit verbundene Ablauf zwischen den beteiligten Komponenten näher beschrieben. Beides wird auch in der Abbildung 3.3 illustriert.

1. Es wird angenommen, dass ein Ressourcenadapter eine verbindungsbasierte Ressourcen-API namens `TransactionalResource` bereitstellt, um auf den zugrunde liegenden Ressourcenmanager zuzugreifen. Die Applikation fordert nun die bereitgestellte Ressource per *Dependency Injection* oder *JNDI-Lookup* (siehe jeweils Abschnitt 3.1.7) an.
2. Der Application Server ruft die *Resource Factory* des Ressourcenadapters, um ein `TransactionalResource`-Objekt zu erzeugen. Der Ressourcenadapter assoziiert dieses erzeugte Objekt intern mit einem zweiten Objekt, das ein für den Ressourcenadapter spezifisches `Connection`-Interface implementiert und einem dritten Objekt, das das `XAResource`-Interface implementiert.
3. Der Application Server ruft die `getXAResource`-Methode auf dem erhaltenen `TransactionalResource`-Objekt auf und erhält so ein `XAResource`-Objekt.
4. Der Application Server trägt das erhaltene `XAResource`-Objekt beim Transaktionsmanager über die Methode `Transaction.enlistResource` als benötigte Ressource ein.
5. Der Transaktionsmanager informiert den beteiligten Ressourcenmanager durch die Methode `XAResource.start` darüber, dass der Ressourcenmanager die über diese Verbindung anfallende Arbeit mit der aktuellen Transaktion assoziieren soll.
6. Der Application Server ruft die `getConnection`-Methode auf dem zuvor erhaltenen `TransactionalResource`-Objekt auf, um das für den Ressourcenadapter spezifische `Connection`-Objekt zu erhalten.
7. Der Application Server gibt das erhaltene `Connection`-Objekt an die Applikation zurück. Das `Connection`-Interface wird vom Ressourcenadapter implementiert und ist spezifisch für den zugrunde liegenden Ressourcenmanager.

Es ist anzumerken, dass das `XAResource`-Interface bei diesem Ablauf transparent für die Applikation und das `Connection`-Interface transparent für den Transaktionsmanager

ist. Ferner ist der Application Server die einzige an der Transaktion beteiligte Partei, die eine Referenz auf das `TransactionalResource`-Objekt hält.

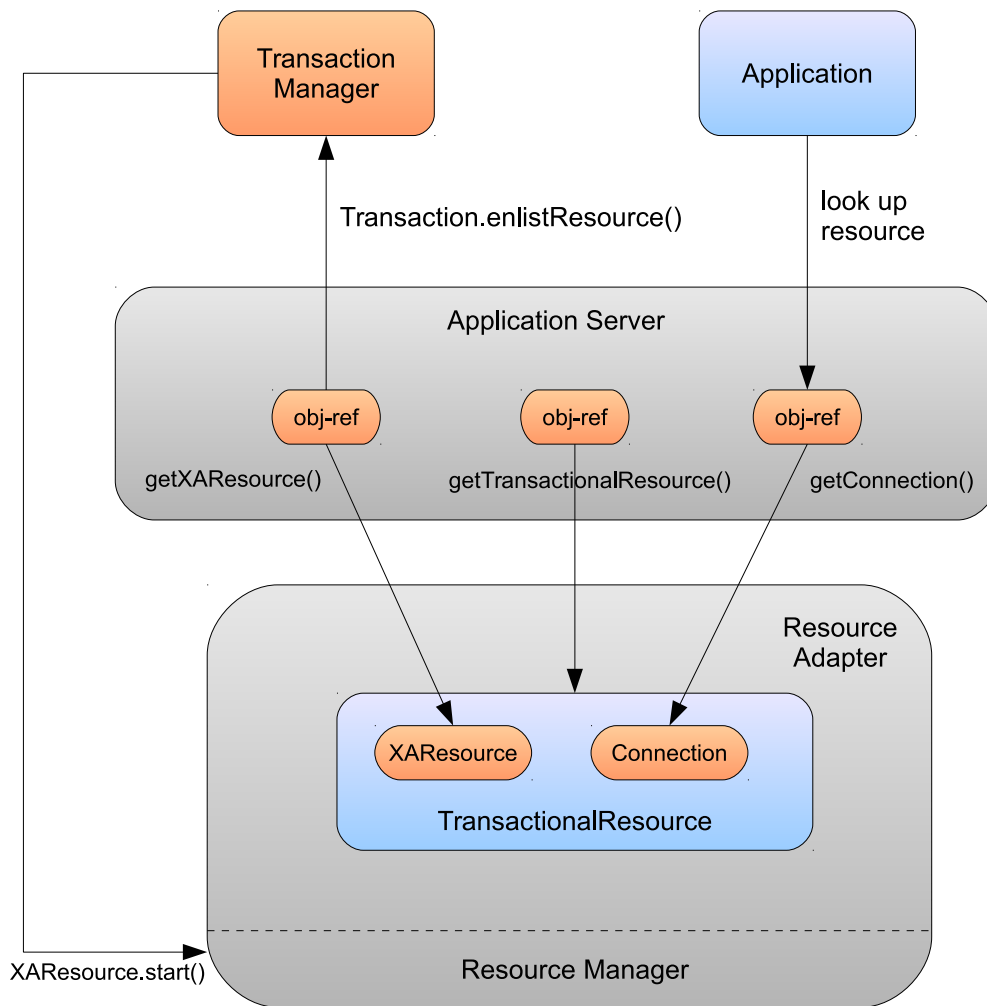


Abbildung 3.3.: Abläufe zwischen Application Server, Applikation, Resource Adapter und Transaktionsmanager nach der *JTA Spezifikation* (siehe [JTA13])

3.1.2. Java Persistence API

Die JDBC-API realisiert die Konnektivität mit einem relationalen Datenbankmanagementsystem. Sie besteht aus einer API, die den Applikationskomponenten den Zugriff auf die Datenbank ermöglicht und einem SPI, das vom Application Server verwendet

3. Verwendete Technologien

wird, um einen JDBC-Treiber anzusprechen und diesen so in die Java EE Plattform zu integrieren.

Dieses SPI wird auch von der *Java Persistence API* (kurz *JPA*, siehe [Jav13b]) verwendet, welche die Standard-API für Persistenz in Java EE ist. JPA ist weiterhin ein *Object-Relational-Mapper* (kurz *ORM*), d.h. sie bietet den Applikationsentwicklern einen Mechanismus, um eine relationale Datenbank mithilfe eines Java Datenmodells zu verwalten.

Ein wichtiges Konzept von JPA ist das der *Entity*, welche ein leichtgewichtiges persistentes Domänenobjekt repräsentiert. Eine Entity ist ein *Plain Old Java Object* (kurz *POJO*), das mit `@Entity` annotiert ist. Ihr persistenter Zustand wird durch ihre Attribute repräsentiert, auf die ausschließlich durch Methoden der Entity selbst zugegriffen werden darf. Insbesondere darf auf die Attribute nicht von Klienten des Objektes zugegriffen werden, d.h. sein Zustand ist ausschließlich über *Accessor Methoden* (auch bekannt als *Getter* und *Setter*) zugänglich. Eine Implementierung von JPA wird *Persistence Provider* genannt und kann entweder über Accessor Methoden oder direkt auf die Attribute zugreifen. Im Kontext dieser Arbeit wird ausschließlich der direkte Zugriff auf die Attribute verwendet. Persistente Attribute, die *Collections* sind, müssen dabei mit den Interfaces `Collection`, `Set`, `List` oder `Map` typisiert werden. Die Applikation kann jedoch eine jeweilige Implementierung einer dieser Schnittstellen für die Initialisierung vor der Persistierung verwenden. Alle Attribute, die nicht `transient` sind oder mit der JPA-Annotation `@Transient` versehen sind, sind persistent. Weiterhin dürfen Sub-Klassen einer Entity die objekt-relationalen Abbildungsmetadaten, die auf die persistenten Attribute angewendet werden, nicht überschreiben. Eine Entity muss einen *Primärschlüssel* besitzen. Dieser kann aus mehreren Attributen bestehen, welche jeweils mit `@Id` annotiert werden. Der Wert des Primärschlüssels identifiziert eine Entity-Instanz eindeutig innerhalb eines *Persistenzkontextes* und in Operationen des `EntityManager`s, weshalb dieser Wert nicht von der Applikation verändert werden darf. Ein Persistenzkontext ist eine Menge von Entity-Instanzen, in welcher für jede persistente Entity-Identität eine zugehörige eindeutige Entity-Instanz existiert.

Da Entities die Strukturen einer relationalen Datenbank in Java repräsentieren, besitzen auch sie Relationen. Hierfür kann ein Attribut einer Entity mit einer der Annotationen `@OneToOne`, `@OneToMany`, `@ManyToOne` oder `@ManyToMany` versehen werden, welche eine 1:1 Beziehung, eine 1:n Beziehung, eine n:1 Beziehung, bzw. eine m:n Beziehung ausdrückt. Die m:n Beziehung wird dabei mithilfe einer *Join-Tabelle* realisiert. Eine Relation kann *unidirektional* oder *bidirektional* sein, wobei eine bidirektionale Relation eine *besitzende Seite* und eine *inverse (nicht-besitzende)* Seite enthält, während eine unidirektionale Relation ausschließlich eine besitzende Seite enthält. Die besitzende Seite einer Relation bestimmt bei der Persistierung die Updates der Relation in der Datenbank und wird über das `mappedBy` Attribut der Annotation auf der **inversen** Seite angegeben. Der Persistence Provider verwaltet das *Object/Relational Mapping* der Relationen inklusive des Ladens und des Speicherns von und in der Datenbank wie sie in den Metadaten der Entities spezifiziert sind. Dabei sichert er die *referenzielle Integrität* der Relationen beim Persistieren. Allerdings ist die Applikation selbst für die Wahrung der Konsistenz der Relationen zur Laufzeit verantwortlich. So muss die Applikation beispielsweise dafür sorgen, dass die „One“- und „Many“-Seiten einer bidirektionalen Relation zueinander konsistent sind, wenn die Relation zur Laufzeit verändert wird. Wird hierauf nicht geachtet, können beim Persistieren Informationen verloren gehen. Weiterhin ist der Persistence Provider dafür verantwortlich, eine leere Collection als Wert einer Relation zurückzuliefern, falls keine assoziierten Entities für diese Relation existieren.

Die Entities werden von einem `EntityManager`-Objekt verwaltet, wobei jeweils ein solches Objekt mit einem Persistenzkontext assoziiert ist. Das `EntityManager`-Interface definiert Methoden, um mit eben diesem Persistenzkontext zu interagieren. Es wird dafür verwendet, um persistente Entity-Instanzen zu erstellen und zu löschen. Weiterhin können mit dieser Schnittstelle persistente Entities anhand ihres Primärschlüssels gefunden und Queries auf ihnen ausgeführt werden. Die Menge der Entities, die von einem `EntityManager` verwaltet werden können, wird durch eine zugehörige *Persistenzeinheit* festgelegt. Sie definiert die Menge aller Klassen, die laut der Applikation in Beziehung zueinander stehen oder zu einer gemeinsamen Gruppe gehören und deren

3. Verwendete Technologien

Objekte in ihrer Abbildung zusammen in einer einzigen Datenbank untergebracht werden müssen.

Ein `EntityManager` kann unter bestimmten Umständen ohne einen transaktionsgebundenen Persistenzkontext verwendet werden. In dieser Arbeit wird ein solcher Persistenzkontext allerdings nicht näher betrachtet, es werden ausschließlich *Container-Managed EntityManager* verwendet, deren Lebenszyklen vollständig von einem Container der Java EE Plattform verwaltet werden. Diese Verwaltung, und insbesondere das Erstellen und Schließen eines `EntityManager`s, findet transparent für die Applikation statt. Ein Persistenzkontext eines solchen `EntityManager`s wird automatisch an die aktuelle *JTA* Transaktion (siehe Abschnitt 3.1.1) gebunden, wenn er *injiziert* wird (siehe Abschnitt 3.1.7) oder über den *JNDI Naming Context* nachgeschlagen wird (siehe Abschnitt 3.1.7). In diesem Fall spricht man auch von einem *JTA EntityManager* und einem *Container-Managed Transaction-Scoped Persistence Context*. Falls aktuell keine *JTA* Transaktion existiert, so wird eine neue gestartet und der Persistenzkontext an sie gebunden.

Der Lebenszyklus einer Entity wird in Abbildung 3.4 grafisch dargestellt und von dem `EntityManager` verwaltet, der zu ihrer Persistenzeinheit gehört. Im Folgenden werden die einzelnen Zustände dieses Lebenszyklus und die Operationen des `EntityManager`s, die zu den Zustandsübergängen führen, näher beschrieben. Es ist dabei zu beachten, dass diese Lebenszyklus-Operationen mittels der `cascade`-Attribute der Annotationen, die eine Relation definieren, *kaskadiert* werden können, um den Effekt der entsprechenden Operation zu assoziierten Entities zu propagieren. Dieses Attribut wird typischerweise in *Parent-Child*-Relationen verwendet. Neben den beschriebenen Operationen können auch diese Kaskadierungen auf den Lebenszyklus einer Entity Einfluss nehmen.

Eine neu erstellte, aber noch nicht persistierte Entity befindet sich im Zustand *new*. Sie besitzt daher noch keine persistente Identität und ist mit keinem Persistenzkontext assoziiert. Entity-Instanzen werden dabei mit dem `new`-Operator von Java erzeugt.

Eine Entity, die bereits persistiert wurde, befindet sich im Zustand *managed*. Sie besitzt eine persistente Identität, die momentan mit einem Persistenzkontext assoziiert ist.

Eine Entity, die im Zustand *new* oder *removed* ist, wird durch den Aufruf der `persist`-Methode des `EntityManagers` in den Zustand *managed* überführt und beim Abschluss der aktuellen Transaktion persistent. Sofern eine Relation entsprechend annotiert wurde, wird die `persist`-Operation in jedem Fall kaskadiert, auch wenn die Entity, für die sie aufgerufen wurde, bereits *managed* ist. Weiterhin sind über den `EntityManager` aus der Datenbank erhaltene Entities im Zustand *managed*. Befindet sich eine Entity in diesem Zustand, so werden Veränderungen an ihrem sonstigen Zustand und an ihren Relationen automatisch beim Abschluss der aktuellen Transaktion zur Datenbank synchronisiert. Eine bidirektionale Relation wird dabei auf der Basis der Referenzen der besitzenden Seite persistiert, es ist also insbesondere wichtig die Referenzen zwischen den beiden Seiten der Relation synchron zu halten, damit die Veränderungen nicht verloren gehen. Der Zustand einer *managed* Entity kann durch den Aufruf der `refresh`-Methode des `EntityManagers` von der Datenbank erneuert werden. Die Veränderungen die zwischenzeitlich an der Entity durchgeführt wurden, werden dabei jedoch überschrieben. Zuletzt darf eine Entity nur in einem einzigen Persistenzkontext *managed* sein. Der Entwickler einer Applikation muss dies beachten.

Eine Entity befindet sich im Zustand *detached*, wenn sie eine persistente Identität besitzt, die jedoch nicht (mehr) mit einem Persistenzkontext assoziiert ist. Eine solche Entity kann nicht persistiert oder gelöscht werden. Eine Entity, die sich im Zustand *managed* oder *removed* befindet, wird durch den Aufruf der `detach`-Methode des `EntityManagers` oder durch den Commit oder das Rollback der aktuellen Transaktion in den Zustand *detached* überführt. Die Referenzen von anderen Entities auf eine *detached* Entity bleiben bei dieser Operation erhalten. Außerdem wird der weitere Zustand einer Entity durch die `detach`-Operation nicht verändert. JPA beherrscht jedoch das Prinzip des *Lazy Loading*, d.h. Attribute und Relationen einer Entity können bei Bedarf, also erst beim Zugriff auf sie, nachgeladen werden. Eine Konsequenz davon ist, dass beim Aufruf der `detach`-Operation unter Umständen noch nicht alle Attribute und Relationen der entsprechenden Entity geladen wurden. Sofern eine Relation entsprechend annotiert wurde, wird die `detach`-Operation in jedem Fall kaskadiert, auch wenn die Entity, für die sie aufgerufen wurde, bereits *detached* ist. Befindet sich eine Entity in diesem Zustand, so werden Veränderungen an ihrem sonstigen Zustand, sowie das Entfernen

3. Verwendete Technologien

der Entity, nicht zur Datenbank synchronisiert. Der Zustand einer *detached* Entity kann zu anderen persistenten Entities mit der gleichen Identität durch den Aufruf der `merge`-Operation des `EntityManagers` wieder propagiert werden, woraufhin dieser Zustand beim Abschluss der aktuellen Transaktion wieder persistiert wird. Falls keine solche Entity existiert, wird eine neue erzeugt. Sofern eine Relation entsprechend annotiert wurde, wird die `merge`-Operation in jedem Fall kaskadiert, auch wenn die Entity, für die sie aufgerufen wurde, bereits *managed* ist. Dabei werden Attribute, die dem Lazy Loading unterliegen und noch nicht geladen wurden, ignoriert.

Eine Entity befindet sich im Zustand *removed*, wenn sie eine persistente Identität besitzt, die mit einem Persistenzkontext assoziiert ist, jedoch beim Commit der aktuellen Transaktion aus der Datenbank entfernt wird. Eine Entity, die sich im Zustand *managed* befindet, wird durch den Aufruf der `remove`-Methode des `EntityManagers` in den Zustand *removed* überführt und beim Abschluss der aktuellen Transaktion aus der Datenbank gelöscht. Sofern eine Relation entsprechend annotiert wurde, wird die `remove`-Operation in jedem Fall kaskadiert, auch wenn die Entity, für die sie aufgerufen wurde, *new* ist. Der sonstige Zustand einer Entity wird durch diese Operation nicht verändert, d.h. er ist gleich wie beim Aufruf der Methode.

Mithilfe von sogenannten *Lifecycle Callback Methods* ist es möglich, auf Zustandsänderungen, die den Lebenszyklus von Entities betreffen, zu reagieren. Hierfür können Methoden in der jeweiligen Entity-Klasse mit Annotationen versehen werden. Die Notifizierung über ein *Lifecycle-Event* erfolgt dann über den Aufruf dieser Methoden beim Übergang der entsprechenden Entity in den entsprechenden Zustand. Die Methoden, die mit `@PrePersist` annotiert sind, werden vor `persist`- und `merge`-Operationen und deren Kaskadierung ausgeführt. Die Methoden, die mit `@PostPersist` annotiert sind, werden nach `persist`- und `merge`-Operationen, deren Kaskadierung und den dazugehörigen `insert`-Operationen in der Datenbank ausgeführt. Dabei sind von der Datenbank generierte Primärschlüssel bereits verfügbar. Die Methoden, die mit `@PreRemove` annotiert sind, werden vor `remove`-Operationen und deren Kaskadierung ausgeführt. Die Methoden, die mit `@PostRemove` annotiert sind, werden nach `remove`-Operationen, deren Kaskadierung und den dazugehörigen `delete`-Operationen in der Datenbank ausgeführt. Die Methoden, die mit `@PreUpdate` annotiert sind, werden

vor den `update`-Operationen in der Datenbank ausgeführt. Die Methoden, die mit `@PostUpdate` annotiert sind, werden nach den `update`-Operationen in der Datenbank ausgeführt. Die genauen Zeitpunkte der Ausführung stehen bei den Update-Callbacks nicht fest. Ein Persistence Provider kann sie beispielsweise bereits beim Update des Entity-Zustandes aufrufen oder erst am Ende der aktuellen Transaktion. Eine Applikation sollte sich also bei diesen Callbacks nicht auf einen genauen Zeitpunkt verlassen. Die Methoden, die mit `@PostLoad` annotiert sind, werden aufgerufen, nachdem die entsprechende Entity aus der Datenbank in den aktuellen Persistenzkontext geladen wurde und nach einer entsprechenden `refresh`-Operation. Dies geschieht bevor das Abfrageergebnis zurückgeliefert und bevor die Relationen traversiert werden. Innerhalb der Methoden, die mit den genannten Annotationen versehen sind, können *Unchecked Exceptions* geworfen werden, die zu einem Rollback der aktuellen Transaktion führen. Die Methoden sollten selbst keine Operationen des `EntityManagers` aufrufen, Datenbankabfragen ausführen oder auf andere Entity-Instanzen zugreifen. Insbesondere sollten innerhalb dieser Methoden keine Relationen verändert werden, die Veränderung des restlichen Zustandes der Entity, auf der sie ausgeführt werden, ist jedoch erlaubt. Weiterhin sollten sich Applikationen nicht darauf verlassen, ob diese Methoden vor oder nach einer Kaskadierung aufgerufen werden. Der Persistence Provider führt standardmäßig eine automatische *Bean Validation* (siehe Abschnitt 3.1.4) der entsprechenden Entities nach den `PrePersist`- und `PreUpdate`-Lifecycle-Events durch.

Bezüglich der Nebenläufigkeit von Transaktionen unterstützt die Java Persistence API sowohl *optimistisches*, als auch *pessimistisches Locking*, wobei standardmäßig die Verwendung von optimistischem Locking angenommen wird. Hierbei werden Updates der Daten in der Datenbank, die mit dem Zustand einer entsprechenden Entity korrespondieren, nur dann durchgeführt, wenn keine zwischenzeitlich durchgeführte Transaktion diese Daten verändert hat, seit der Entity-Zustand aus der Datenbank gelesen wurde. Diese Strategie sorgt dafür, dass Updates oder Deletes dieser Daten konsistent mit dem aktuellen Zustand der Datenbank sind und verhindert, dass zwischenzeitliche Updates verloren gehen (sogenannte *Lost Updates*). Beim Abschluss von Transaktionen, die diese Bedingungen verletzen, wird vom Persistence Provider eine `OptimisticLockException` geworfen und es folgt ein Rollback der Transaktion.

3. Verwendete Technologien

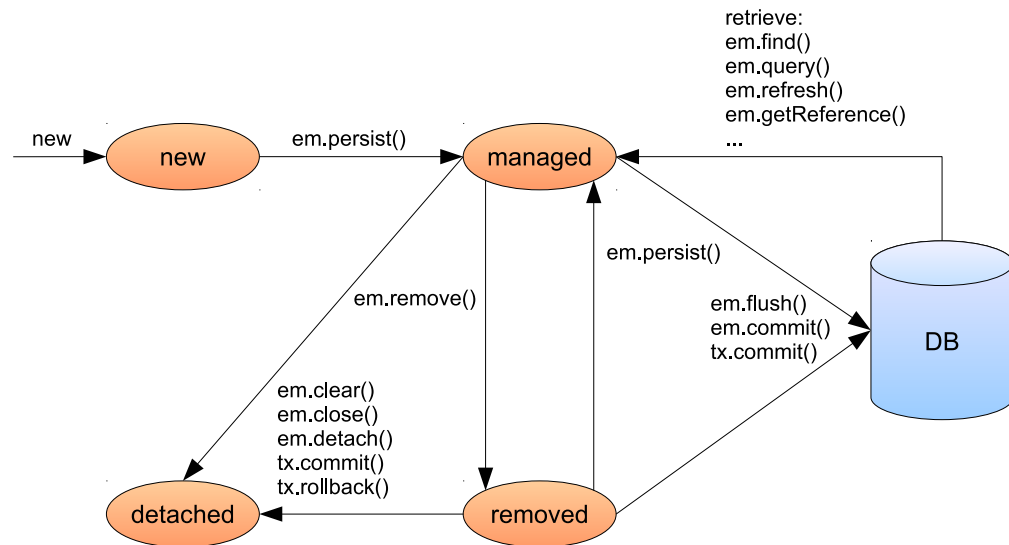


Abbildung 3.4.: Lebenszyklus einer JPA-Entity unter einem `EntityManager` mit einem *transaction-scoped persistence context* nach den Beschreibungen der *JPA Spezifikation* (siehe [Jav13b])

Um das optimistische Locking für eine Entity zu aktivieren, muss eines ihrer Attribute vom Typ `int` oder `java.sql.Date` mit `@Version` annotiert werden. Dieses Attribut wird dann vom Persistence Provider beim Ausführen von Lifecycle-Operationen ausgewertet und gesetzt. Außerdem wird das Attribut beim Schreiben in die Datenbank aktualisiert und während einer `merge`-Operation und beim Commit einer Transaktion überprüft. Die Applikation darf deswegen ein solches Attribut ausschließlich lesen, jedoch nicht verändern. Zuletzt wird standardmäßig angenommen, dass auf Datenbanken mittels der *Read Committed Isolation* zugegriffen wird, es können jedoch auch strikere *Transaktionsisolationsebenen* gewählt werden.

JPA bietet für die Formulierung von Datenbankabfragen mehrere Möglichkeiten. Im Kontext dieser Arbeit sind jedoch nur Abfragen über die *Java Persistence Query Language* (kurz *JPQL*) und die *Java Persistence Criteria API* relevant. Beide Varianten sind portabel und erlauben die Formulierung von Abfragen, die unabhängig von der verwendeten Datenbank sind und vom Persistence Provider in die Zielsprache der Datenbank kompiliert werden. Ihre Ausführung findet anschließend in der Datenbank unter der Verwendung ihrer Möglichkeiten statt. Das `EntityManager`-Interface bietet dabei die

Methoden an, mit deren Hilfe die Abfragen konstruiert und ihre Ausführung angegossen werden kann.

JPQL ist eine `String`-basierte Abfragesprache, die eine Syntax ähnlich der *Structured Query Language* (kurz *SQL*) verwendet. Mit ihr werden Abfragen über Entities und ihren persistenten Zustand definiert, nicht aber über datenbankspezifische Konstrukte. Weiterhin ist sie sowohl für die Ausführung von *statischen*, als auch von *dynamischen Abfragen* geeignet und kann für *Prepared Statements* verwendet werden. Standardmäßig sind hierbei alle Abfragen *polymorph*, d.h. die Abfragen liefern nicht nur Instanzen der konkreten Entity-Klassen zurück, auf die sie verweisen, sondern auch Instanzen ihrer Subklassen. Die `EntityManager`-Schnittstelle bietet Methoden für die Formulierung und Ausführung von *untypisierten* und *typisierten* Abfragen in JPQL. Dafür werden jeweils Objekte mit der `Query`- bzw. `TypedQuery`-Schnittstelle zurückgeliefert. Des Weiteren bietet JPA für die Formulierung von statischen Queries mithilfe von JPQL die Verwendung sogenannter *Named Queries* an. Dies sind Abfragen, die über Annotationen der entsprechenden Entities definiert werden, wodurch eine Art *Kapselung* realisiert wird. Die Abfragen sind so folglich an zentralen Orten in der Applikation definiert und nicht chaotisch über die ganze Applikation verteilt. Weiterhin sind sie auch bezüglich der Semantik an dem Ort definiert, für den sie wichtig sind: den Entities. Als statische Abfragen werden Named Queries einmalig in die Zielsprache der Datenbank kompiliert und diese Kompilate während der gesamten Laufzeit der Anwendung wiederverwendet.

Die Criteria API ist hingegen hauptsächlich für die Formulierung von dynamischen Abfragen konzipiert. Die Abfragen werden in diesem Fall durch die Konstruktion von *Query-Definitions-Objekten* mithilfe des *Builder Patterns* (siehe [GHJV11]) definiert. Es entsteht so ein *Query-Graph*, dessen Knoten den Elementen der später kompilierten Abfrage entsprechen. Die gesamte Criteria API bietet hierbei eine *starke Typisierung*. Um den Bezug zu Entities und ihren Attributen herzustellen, wird die *Metamodel API* verwendet. Mit ihrer Hilfe kann ein abstraktes Persistenzschema der Entities und ihrer Relationen definiert werden, also ein Modell des Datenmodells. Dies geschieht in Form von Metamodel-Objekten, die die Interfaces der Metamodel-API implementieren. Das so konstruierte Modell ist dabei statisch und kann entweder vom Applikationsentwickler ge-

3. Verwendete Technologien

geschrieben oder durch einen *Annotationsprozessor* bei der Kompilierung der Anwendung generiert werden.

3.1.3. Java EE Connector Architecture

Mithilfe der *Java EE Connector Architecture* (kurz *JCA*, siehe [Con13]) können die Funktionalität eines Application Servers und die Dienste, die dieser einer Applikation bereitstellt, erweitert werden. Hierfür definiert die Connector Architecture ein Java EE SPI, das von einem Application Server dafür genutzt werden kann, sich mit einem *Enterprise Information System* (kurz *EIS*) zu verbinden. Ein EIS stellt typischerweise die Informationsstruktur eines Unternehmens bereit und bietet seinen Clients Dienste in Form von lokalen oder entfernten Schnittstellen an. Beispiele hierfür sind *Enterprise Resource Planning Systeme* (kurz *ERP*), *Mainframe Transaction Processing Systeme* (kurz *TP*) oder *Legacy Datenbanksysteme*.

Der Application Server verwendet einen sogenannten *Ressourcenadapter*, um mit dem EIS zu kommunizieren. Ein Ressourcenadapter ist eine Komponente auf der Systemebene, die die Netzwerk-Konnektivität zu einem externen *Ressourcenmanager* implementiert. Er kann die Funktionalität der Java EE Plattform erweitern, indem er entweder eine *Java EE Standard Service API*, wie z.B. JDBC, implementiert oder, indem er einen spezifischen *Connector* zu einem externen System definiert und implementiert. Zusätzlich implementiert er ein Java EE SPI, wodurch er mit der Java EE Plattform verbunden werden kann.

Ein Ressourcenmanager ist eine Softwarekomponente, die eine Menge an gemeinsamen *EIS-Ressourcen* verwaltet. Er wird typischerweise auf einem anderen Rechner als seine Clients ausgeführt, kann in einer anderen Sprache als in Java geschrieben sein und unter einem anderen Betriebssystem als seine Clients ausgeführt werden. Eine EIS-Ressource, wie z.B. ein Geschäftsobjekt eines ERP-Systems, bietet hierbei den Clients, die auf sie zugreifen, eine spezifische Funktionalität des EIS an. Ein Client, in diesem Fall der Application Server selbst, greift auf einen solchen Ressourcenmanager zu, um die von ihm verwalteten Ressourcen zu verwenden. Ein *transaktionaler Ressourcenmanager* kann zusätzlich an Transaktionen teilnehmen, die von einem externen

Transaktionsmanager (siehe Abschnitt 3.1.1) koordiniert werden. Die Kommunikation des Ressourcenadapters mit dem dazugehörigen Ressourcenmanager erfolgt dabei *verbindungsbasiert*. Eine solche *Verbindung* ermöglicht einer Applikation, Transaktionen durchzuführen und auf Dienste zuzugreifen, die ein Ressourcenmanager anbietet.

Die Zusammenhänge der für die Connector Architecture relevanten Komponenten werden in Abbildung 3.5 illustriert. Ein Ressourcenadapter implementiert das generische *Common Client Interface* (kurz *CCI*), das es im Wesentlichen einer Applikationskomponente erlaubt, über ihn eine Verbindung zu einem externen EIS aufzubauen. Ist eine solche Verbindung aufgebaut, verwendet die Applikationskomponente eine spezifische *Client API* des Ressourcenadapters, um über ihn mit dem EIS zu kommunizieren. Hierfür verwendet der Ressourcenadapter wiederum eine für das EIS spezifische Schnittstelle. Weiterhin basiert das SPI, das der Application Server für die Interaktion mit dem Ressourcenadapter verwendet, auf einem Standardsatz aus Kontrakten auf der Systemebene, welche im Folgenden kurz erläutert werden. Zum einen besteht zwischen beiden Komponenten der *Connection Management Contract*. Er erlaubt es dem Application Server, die Verbindungen zum EIS zu poolen und den Applikationskomponenten, sich mit dem EIS zu verbinden. Dieses Pooling fördert eine effiziente und gut skalierende Integration des EIS. Weiterhin besteht zwischen dem Transaktionsmanager und dem Ressourcenmanager des EIS der *Transaction Management Contract*. Er erlaubt den transaktionalen Zugriff auf das EIS und damit dem Java EE Server, einen Transaktionsmanager zu verwenden, der Transaktionen verwaltet, die mehrere Ressourcenmanager umfassen. Andersherum wird aber auch die Propagierung eines Transaktionskontext vom EIS zum Application Server ermöglicht, der diesen Transaktionskontext anschließend importiert. Alle weiteren Interaktionen mit dem Server und Applikationskomponenten werden dann Teil dieser Transaktion. Die *ACID*-Eigenschaften (*Atomicity, Consistency, Isolation, Durability*) einer importierten Transaktion werden dabei erhalten. Außerdem existiert ein *Security Contract*, der den sicheren Zugriff auf das EIS ermöglicht, d.h. Aspekte wie Authentifizierung und Autorisierung regelt. Der *Thread Management Contract* erlaubt es dem Ressourcenadapter, seine Arbeit an andere Threads zu delegieren. Diese Threads können andererseits vom Application Server in einem *Thread-Pool* verwaltet werden, wodurch eine effiziente und gut skalierende Integration des externen EIS erreicht wird.

3. Verwendete Technologien

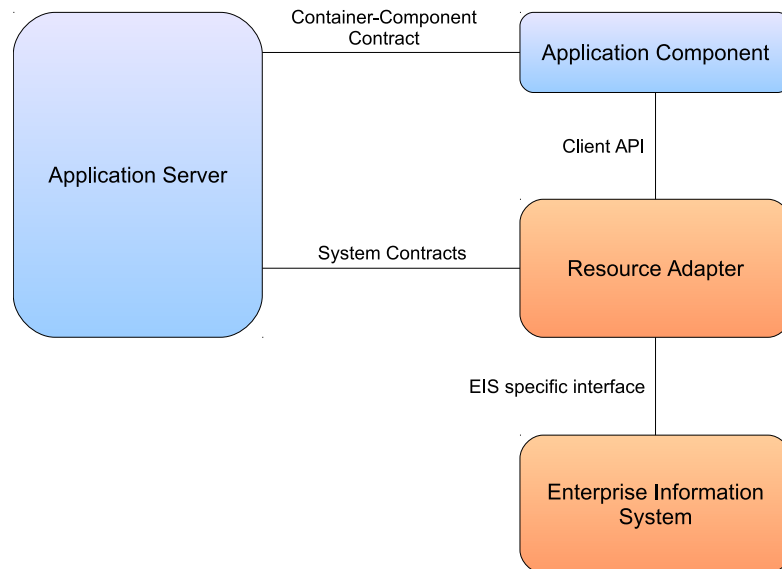


Abbildung 3.5.: Überblick über die Connector Architecture nach der *JCA Spezifikation* (siehe [Con13])

JCA definiert damit einen Standard, mit dessen Hilfe unterschiedliche Enterprise Information Systems mit einem Application Server verbunden werden können. Ein EIS muss somit nicht für jeden Application Server erneut angepasst werden. Auf der anderen Seite muss ein Application Server nicht spezifischen Code für jedes EIS enthalten. Es genügt, einmalig einen Ressourcenadapter für ein spezifisches EIS zu entwickeln, der in jedem Application Server lauffähig ist, der zur Java EE Spezifikation konform ist. Sobald ein solcher Ressourcenadapter in einem Application Server deployed wurde, können Applikationskomponenten auf das EIS zugreifen. Dabei sind alle Mechanismen, wie z.B. Transaktionen, Sicherheit und Verbindungsverwaltung, auf der Systemebene realisiert und damit transparent für die Applikationskomponenten.

Ressourcenadapter können jedoch entgegen der bisherigen Beschreibungen auch Dienste bereitstellen, die vollständig lokal ablaufen und z.B. mit nativen Ressourcen interagieren. Ein solcher Ressourcenadapter, der in der Implementierung dieser Arbeit verwendet wird, wird von *XADisk* (siehe [Ver14]) realisiert. Er erlaubt den transaktionalen Zugriff auf das Dateisystem, wobei er sowohl lokale, als auch verteilte (XA-) Transaktionen unterstützt. Für den Fall eines Ausfalls bietet der XADisk-Adapter eine

Ausfallerholung, d.h. die Datenintegrität wird nach Ausfällen und Re-Deployments erhalten. Weiterhin werden gleichzeitige Transaktionen durch Locking isoliert und besitzen die Atomicity-Semantik, d.h. es werden entweder alle Operationen einer Transaktion ausgeführt oder im Fehlerfall keine. Zuletzt ist XADisk durch die Verwendung der Java *New I/O* APIs (kurz *NIO*) für den Dateizugriff performant.

3.1.4. Java API for JavaBean Validation

Die Validierung von Daten, insbesondere derer, die von Eingaben des Benutzers stammen, ist ein Cross-Cutting Concern, der in allen Schichten einer Applikation benötigt werden kann. Die *Java API for JavaBean Validation* (kurz *Bean Validation*, siehe [Bea13]) ist ein Framework mit dessen Hilfe *Constraints* auf der Objektebene definiert und diese dann validiert werden können.

Die Constraints werden dabei deklarativ mit Annotations angegeben und repräsentieren Bedingungen, die der Entwickler an die annotierten Elemente stellt. Dieser Ansatz ist gegenüber einer programmatischen Definition von Bedingungen übersichtlicher, weniger fehlerträchtig und verringert die Menge an Code, die geschrieben und gewartet werden muss. Die Constraints können auf Objektattribute, Methodenparameter und Rückgabewerte von Methoden angewandt werden. Die Annotation von Methodenparametern entspricht dann den *Vorbedingungen*, die vom Aufrufer erfüllt werden müssen und die Annotation von Rückgabewerten entspricht den *Nachbedingungen*, die dem Aufrufer garantiert werden. Insgesamt wird so die Beschreibung und Validierung eines *Kontraktes* bestehend aus den Vor- und Nachbedingungen einer Methode, also *Programming by Contract*, ermöglicht. Alle Annotationen werden hierbei automatisch in der *JavaDoc*-Dokumentation mit eingeschlossen, wodurch Redundanzen verringert und Inkonsistenzen zwischen der Implementierung und der Dokumentation vermieden werden.

Bei der Validierung mittels der Bean Validation API wird ein `Set von Constraint Violation`-Objekten zurückgegeben. Jedes dieser Objekte enthält jeweils ein Attribut bzw. einen Methodenparameter, der ein Constraint nicht erfüllt. Weiterhin wird von *JPA* (siehe Abschnitt 3.1.2) automatisch eine Validierung der Constraints vor der Persistierung

3. Verwendete Technologien

von Entities ausgeführt und vom jeweiligen Container, in dem eine Methode ausgeführt wird, vor und nach deren Ausführung ein *Interceptor* (siehe [EJB09] und [CDI13]) gerufen, der die Methodenparameter bzw. den Rückgabewert validiert. Bei diesen drei automatischen Validierungen wird jeweils eine `ConstraintViolationException` geworfen, falls die Constraints nicht erfüllt werden. Diese Exception enthält dann das Set aus `ConstraintViolations`.

Die Bean Validation Spezifikation definiert einen Standardsatz von Constraints, die alle Implementierungen der Spezifikation mitbringen müssen. Es steht diesen Implementierungen jedoch frei, zusätzliche Constraints mitzuliefern. Die Referenzimplementierung *Hibernate Validator* (siehe [Red14]) bringt beispielsweise nahezu doppelt so viele Constraints mit wie die Spezifikation fordert. Die Verwendung dieser implementierungsspezifischen Constraints führt jedoch dazu, dass eine Applikation nicht mehr portabel ist. Deshalb wurde im Rahmen dieser Arbeit eine Constraint-Bibliothek entwickelt, die genau die Constraints des Hibernate Validator enthält, die nicht zu den Standard-Constraints der Spezifikation gehören. Sie kann dem Archiv der Applikation hinzugefügt und so mit ihr deployed werden, wodurch die Applikation portabel bleibt. Die Bibliothek enthält insbesondere eine `@Email`-Annotation, die überprüft, ob ein `String` eine gültige Email-Adresse nach *RFC 2822* (siehe [QUA01]) enthält und eine `@URL`-Annotation, die überprüft, ob ein `String` eine gültige *URL* (*Uniform Resource Locator*, siehe [BLF⁺98]) nach *RFC 2396* enthält. Weiterhin enthält sie die Annotation `@NotEmpty`, die überprüft, ob ein `String` oder eine `Collection` weder `null`, noch leer ist.

3.1.5. Java API for RESTful Web Services

Die *Java API for RESTful Web Services* (kurz *JAX-RS*, siehe [JAX13]) definiert Schnittstellen für die Entwicklung von *Web-Services* nach dem REST Architekturstil. Sie ist POJO-basiert und definiert Annotationen, die auf die Objekte einer Applikation angewendet werden können.

Eine JAX-RS Applikation enthält eine Klasse, die von der `Application`-Klasse von JAX-RS abgeleitet ist und die *Wurzel-URL* der REST Schnittstelle definiert. Die JAX-RS Implementierung sucht beim Deployment nach solch einer Klasse und registriert alle

weiteren *Web-Ressourcen* der Applikation unter URLs, die sich unterhalb der URL dieser Klasse befinden.

Eine Web-Ressource wird als Java-Klasse implementiert und die eintreffenden HTTP-Requests werden von den Methoden dieser Klasse verarbeitet. Standardmäßig wird für jeden Request eine neue Instanz der Klasse erstellt. Hierbei findet auch eine *Dependency Injection* (siehe Abschnitt 3.1.7) statt. Die URL der Ressource wird dabei durch die `@Path`-Annotation definiert. Die Methoden der Ressource können auch mit dieser Annotation versehen werden. Sie werden dann zu *Subressourcen* und ihre URLs sind unterhalb derer ihrer zugehörigen Ressource. Weiterhin müssen die Methoden, die zur REST Schnittstelle der Applikation gehören sollen, mit einem *Request Method Designator* wie `@GET`, `@POST`, `@PUT` oder `@DELETE` annotiert werden, welcher auf die entsprechende *HTTP-Methode* (siehe [BLF⁺99]) abgebildet wird.

Die Methodenparameter müssen mit Annotationen versehen werden, die die entsprechenden Werte aus den korrespondierenden Teilen des Requests extrahieren. Die wichtigsten davon werden im Folgenden kurz erläutert. Durch die Annotation eines Parameters mit `@QueryParam` wird der Wert des Methodenparameters aus dem *Query-String* der URL (siehe [BLF⁺99] und [BLF⁺98]) extrahiert. Die Annotation `@PathParam` extrahiert den Wert des Methodenparameters aus einem *Vorlagenparameter* in der URL, welcher in der `@Path`-Annotation der dazugehörigen Methode definiert wird. Die Annotation einer Methode mit `@Path("foo/{id}/bar")` definiert beispielsweise den Vorlagenparameter `id`. Wird nun ein Parameter der Methode mit `@QueryParam("id")` annotiert, so wird der Wert dieses Parameters aus der Stelle der URL extrahiert, an der der Vorlagenparameter definiert ist. Zuletzt extrahiert die Annotation `@HeaderParam` den Wert eines Parameters aus einem *HTTP-Header* (siehe [BLF⁺99]) des Requests.

Ein Parameter der Methode kann jedoch ohne eine dieser Annotationen verbleiben. Dieser wird dann *Entity Parameter* genannt und sein Wert wird von einem *Entity Provider* aus dem *Entity Body* (siehe [BLF⁺99]) des HTTP-Requests extrahiert. Die Entity Provider bilden Java-Klassen auf Repräsentationen eines bestimmten *Medientyps* (siehe [BLF⁺99]) des HTTP-Requests bzw. der HTTP-Response ab. Dabei werten sie die Annotationen der *Java Architecture for XML Binding* (kurz *JAXB*, siehe [JAX09]) in den

3. Verwendete Technologien

entsprechenden Klassen aus. Die Klassen und Methoden, die zur REST Schnittstelle der Applikation gehören, können für die Abbildung auf Medientypen mit `@Consumes` und `@Produces` annotiert werden. Damit wird definiert, welche Medientypen sie in Requests und Responses unterstützen. Die Unterstützung des *JSON*-Medientyps (*JavaScript Object Notation*, siehe [Cro06]) wird zwar in der JAX-RS Spezifikation nicht explizit gefordert, die beiden Java EE 7 Application Server bringen jedoch Entity Provider mit, die das *Unmarshalling* aus und das *Marshalling* in das JSON-Format erlauben. Die Spezifikation fordert jedoch explizit, dass mit der Applikation mitgelieferte Entity Provider von der JAX-RS Implementierung des Application Servers verwendet werden müssen. Somit kann für die Unterstützung künftiger Application Server, die einen solchen Entity Provider nicht mitliefern, ein entsprechender Entity Provider in das Archiv der Applikation integriert werden. Nach der Extraktion der Werte der Methodenparameter wird automatisch eine *Bean Validation* (siehe Abschnitt 3.1.4) durchgeführt.

Der Rückgabetypp einer Methode kann eine beliebige Java-Klasse sein. Das zurückgegebene Objekt wird automatisch vom Entity Provider in ein Format des definierten Medientyps der Methode überführt. JAX-RS definiert außerdem eine *Response*-Klasse, die als Rückgabetypp verwendet werden kann. Ihre Schnittstelle erlaubt den Zugriff auf die wichtigsten Eigenschaften der HTTP-Response. Weiterhin kann über diese Schnittstelle auch ein Objekt als Entity Body gesetzt werden, welches wiederum vom Entity Provider in das entsprechende Format überführt wird.

3.1.6. Enterprise JavaBeans

Enterprise JavaBeans (kurz *EJBs*, siehe [EJB09]) stellen ein Framework für komponentenbasierte transaktionsorientierte Unternehmensanwendungen bereit. Sie enthalten typischerweise die *Geschäftslogik* einer Applikation und aktualisieren somit die Daten in der zugrunde liegenden Datenbank, wobei sie selbst keine solchen Daten repräsentieren. Sie sind also nicht persistent und existieren nur innerhalb des EJB-Containers.

Die EJB-Instanzen einer Applikation, und insbesondere ihre Lebenszyklen, werden zur Laufzeit vom EJB-Container verwaltet. Der Zugriff ihrer Clients auf sie erfolgt ausschließlich über die Vermittlung durch den Container mithilfe von *JNDI-Lookups* und

Dependency Injection (siehe jeweils Abschnitt 3.1.7). Dabei wird eine EJB im Auftrag eines einzigen Clients ausgeführt und sie existiert aus der Sicht dieses Clients, sobald dieser eine Referenz auf sie erhalten hat. Dieses Vorgehen erlaubt dem Container, die Funktionalität für Sicherheit, Nebenläufigkeit und Transaktionen bereitzustellen. Weiterhin kann so das Verhalten einer EJB zum Zeitpunkt des Deployments, ohne ihren Quellcode zu verändern, angepasst werden, indem die von ihr verwendeten *Umgebungseinträge* (siehe Abschnitt 3.1.7) bearbeitet werden.

Die EJB Spezifikation definiert verschiedene Typen von EJBs. Dies sind einerseits sogenannte *Message-Driven Beans* (kurz *MDBs*), die für die asynchrone, nachrichtenbasierte Kommunikation verwendet werden können und in dieser Arbeit keine Rolle spielen. Andererseits definiert die Spezifikation sogenannte *Session Beans*, die – wie ihr Name bereits ausdrückt – sitzungsbasiert sind, also an eine Sitzung des Benutzers mit der Applikation gebunden sind. Dabei wird zwischen *Stateful Session Beans*, *Stateless Session Beans* und *Singleton Session Beans* unterschieden, letztere finden im Kontext dieser Arbeit keine Anwendung.

Die *Stateful Session Beans* besitzen einen inneren Zustand und sind an die Sitzung und den jeweiligen Client gebunden. Falls mehrere Clients gleichzeitig auf eine Instanz einer solchen Session Bean zugreifen wollen, werden diese Zugriffe *serialisiert*, wodurch diese Session Beans *thread-safe* sind. Typischerweise existiert in der Schnittstelle einer solchen Bean eine applikationsspezifische Methode, die ihren Zustand initialisiert und eine Methode, die ihren Zustand nach ihrer Verwendung wieder freigibt. Dabei wird die Bean-Instanz aus der Sicht des Clients beim Aufrufen der freigebenden Methode entfernt, tatsächlich wird sie jedoch vom Container für andere Clients wiederverwendet. Dieses Vorgehen wird auch *Pooling* genannt. Um eine Klasse als *Stateful Session Bean* zu kennzeichnen, wird sie mit `@Stateful` annotiert.

Die *Stateless Session Beans* besitzen hingegen keinen inneren Zustand. Dadurch können sich Clients insbesondere nicht darauf verlassen, dass ein Zustand zwischen den Methodenaufrufen an einer solchen Bean behalten wird oder existiert. Es darf trotzdem immer nur ein Client gleichzeitig auf eine solche Bean-Instanz zugreifen. Durch die Tatsache, dass kein innerer Zustand existiert, gleichen sich jedoch alle Instanzen

3. Verwendete Technologien

einer Stateless Session Bean Klasse. Das vom Container durchgeführte Pooling besteht in diesem Fall darin, dass ein Client beim Aufruf einer Methode einfach eine andere Instanz von ihm erhält, wodurch auch der Zugriff auf diese Art der Session Beans thread-safe wird. Um eine Klasse als Stateless Session Bean zu kennzeichnen, wird sie mit `@Stateless` annotiert.

Der EJB-Container stellt des Weiteren eine *Middleware* bereit, die die Realisierung der Server-Seite als *verteilte Anwendung* ermöglicht. Die EJBs werden hierbei auf einem *Rechencluster*, also mehreren verschiedenen Rechnern und JVMs, ausgeführt. Zu diesem Zweck kann jede EJB mehrere sogenannte *Views* besitzen, die jeweils durch Java-Interfaces ausgedrückt werden. Ihr Client spricht eine solche EJB dann ausschließlich über eines dieser Interfaces an. Die *Remote View* erlaubt dabei den Zugriff von außerhalb der JVM. Um diese Sicht auf eine Session Bean zu definieren, wird die entsprechende Klasse mit `@Remote` annotiert und als Parameter der Annotation die Klasse des entsprechenden Interfaces übergeben. Die *Local View* erlaubt hingegen ausschließlich den Zugriff von Objekten innerhalb derselben JVM und derselben Applikation. Ein lokaler Client kann hierbei eine andere EJB oder eine Web-Komponente sein. Um diese Sicht auf eine Session Bean zu definieren, wird die entsprechende Klasse mit `@Local` annotiert und als Parameter der Annotation die Klasse des entsprechenden Interfaces übergeben. Zuletzt existiert die sogenannte *No-Interface View*. Sie enthält alle `public`-Methoden der EJB, welche hierfür mit `@LocalBean` annotiert wird. Der Client einer solchen EJB spricht sie ohne Interface direkt über ihre Klasse an.

Jeder Aufruf einer Methode einer Session Bean wird vom Container abgefangen, welcher mit ihr verknüpfte *Interceptors* aufruft. Diese Objekte besitzen den gleichen Lebenszyklus wie die Objekte der Zielklasse. Sie werden dabei nach den Verwaltungsaufgaben des Containers, wie z.B. dem Starten einer Transaktion, aber vor der aufzurufenden Bean-Methode aufgerufen. Folglich nehmen sie auch an der entsprechenden Transaktion des Objektes der Zielklasse teil. Mithilfe der Interceptors können Cross-Cutting Concerns wie z.B. *Logging* effektiv realisiert werden. Dafür muss der entsprechende Code nur einmal als Interceptor entwickelt werden, welcher anschließend als solcher mit der `@Interceptors` Annotation in den EJBs deklariert und bei jedem nachfolgenden Aufruf der entsprechenden Bean-Methode aufgerufen wird.

Ein Schlüssel-Feature der EJBs ist die Unterstützung *verteilter Transaktionen* (siehe Abschnitt 3.1.1). Im Gegensatz zu den *Low-Level APIs* von JTA, wird die Komplexität des Transaktionsmanagements bei der Verwendung von EJBs versteckt. Der EJB-Container selbst verwendet jedoch weiterhin die Java Transaction API. Bei der Verwendung von EJBs mit Transaktionen wird zwischen *Bean-Managed Transactions* (kurz *BMTs*) und *Container-Managed Transactions* (kurz *CMTs*) unterschieden. Bean-Managed Transactions erlauben eine *programmatische Transaktionsdemarkation*, die von den Applikationskomponenten selbst verwaltet wird. Sie bieten eine ähnliche Low-Level API wie die von den EJBs verwendete Java Transaction API und werden vor allem in Stateful Session Beans verwendet, wenn Transaktionen mehrere Methodenaufrufe des Clients an einer EJB umfassen müssen. Container-Managed Transactions sind hingegen vom Container verwaltete Transaktionen. Sie werden durch die Methoden einer EJB abgegrenzt und das Verhalten des Containers kann *deklarativ* durch die Annotation `@TransactionAttribute` angegeben werden. Diese Annotation kann sowohl auf der Klassenebene angegeben werden, wodurch sie sich auf alle Methoden der EJB bezieht, als auch auf der Methodenebene, wodurch sie sich auf die jeweilige Methode bezieht. Die Annotation auf der Methodenebene überschreibt eine etwaige Annotation auf der Klassenebene. Der Container schaltet sich dann zwischen den Client und den von ihm angestoßenen Methodenaufruf der EJB und handelt entsprechend den geltenden Annotationen. Er startet also bei Bedarf eine Transaktion bevor die Methode der EJB ausgeführt wird und beendet die Transaktion danach wieder oder führt beim Werfen einer Exception innerhalb der EJB-Methode ein Rollback der Transaktion durch. Der `@TransactionAttribute`-Annotation wird ein Parameter übergeben, der das Verhalten des Containers bezüglich den Transaktionen definiert. Standardmäßig, also wenn die EJB nicht mit dieser Annotation versehen ist, gilt die Strategie `REQUIRED`. In diesem Fall wird eine bereits laufende Transaktion weiter verwendet oder eine neue gestartet, falls noch keine existiert. Die Aktionen innerhalb der EJB-Methode werden somit auf jeden Fall innerhalb einer Transaktion ausgeführt. Mithilfe des Transaktionsattributs `REQUIRES_NEW` wird der Container angewiesen, auf jeden Fall eine neue Transaktion zu starten. Eine bereits laufende Transaktion wird dann ausgesetzt, bevor die neue Transaktion gestartet wird, und nach dem Abschluss der neuen Transaktion wieder

3. Verwendete Technologien

aufgenommen. Mithilfe des Transaktionsattributes `MANDATORY` wird beim Aufruf der EJB-Methode eine bereits bestehende Transaktion vorausgesetzt. Existiert eine solche Transaktion nicht, so wird eine `TransactionRequiredException` vom Container geworfen. Es existieren weitere Strategien, die jedoch im Kontext dieser Arbeit nicht relevant sind. Wird innerhalb einer EJB-Methode eine andere Methode an einer anderen EJB gerufen, so wird der Transaktionskontext der rufenden EJB unabhängig von ihrem Transaktionsattribut zur gerufenen EJB propagiert.

Der EJB-Container bietet außerdem ein rollenbasiertes Sicherheitsmanagement. In der EJB werden hierfür die Rollen *deklarativ* mittels der Annotation `@RolesAllowed` definiert, die zum Ausführen der jeweiligen Methoden benötigt werden. Es kann wieder die Klasse annotiert werden, woraufhin sich die Annotation auf alle Methoden bezieht, und/oder einzelne Methoden. Weiterhin sind auch *programmatische* Überprüfungen möglich, z.B. bezüglich des Benutzernamens. Werden auch die Rollen programmatisch überprüft, beispielsweise wenn die rein deklarative Überprüfung mit Annotations die Sicherheits-Policy der Applikation nicht abbilden kann, so müssen dem Container alle in dieser Überprüfung verwendeten Rollen bekannt gemacht werden. Hierfür wird die EJB auf der Klassenebene mit `@DeclareRoles` annotiert und ihr werden als Parameter alle für die Überprüfung relevanten Rollen übergeben. Es ist jedoch eine *Best Practice* die Annotation `@DeclareRoles` auch bei der deklarativen Rollenüberprüfung mittels `@RolesAllowed` zu verwenden. Ist die von der EJB-Methode benötigte Rolle nicht mit dem Sicherheitskontext des Aufrufers assoziiert, so lehnt der EJB-Container den Aufruf der Methode ab und wirft eine Exception. Beim Deployment werden die in den EJBs einer Applikation verwendeten Rollen auf Benutzer oder Gruppen der Laufzeitumgebung abgebildet.

3.1.7. Dependency Injection

Die Spezifikationen der Java EE Plattform unterstützen und empfehlen die Entwicklung von komponentenbasierten Applikationen. Wenn sich der Entwickler der Applikation dabei an Design-Prinzipien wie das *Single-Responsibility-Prinzip* (siehe [Mar02]) und *Separation of Concerns* (siehe [Dij82]) hält, entstehen viele kleine Komponenten, die

sich gegenseitig verwenden, also Abhängigkeiten zueinander besitzen. Eine Komponente sollte dabei ausschließlich die für die Erfüllung ihrer Aufgabe notwendigen Details der anderen verwendeten Komponenten kennen, nicht jedoch Details, die mit deren Implementierungen zusammenhängen. Dieses Prinzip wird *Information Hiding* genannt und führt zwischen den Komponenten zu einer *losen Kopplung*. Beides wird in Java mithilfe von Interfaces realisiert. Die Komponenten definieren hierfür Interfaces und implementieren sie. Andere Komponenten greifen dann ausschließlich über diese Interfaces auf sie zu.

Irgendwo in der Applikation müssen jedoch die verwendeten Objekte, also Instanzen konkreter Klassen, erzeugt werden. In Java sind, wie in den meisten objektorientierten Programmiersprachen, die Objekte selbst dafür verantwortlich, die von ihnen verwendeten Objekte zu erzeugen. Dadurch entstehen jedoch wieder Abhängigkeiten zu konkreten Klassen. Es existieren Design-Patterns, die das Problem der Erzeugung von Objekten lösen und daher zur Klasse der *Creational Patterns* (siehe [GHJV11]) gehören. Hierzu gehören beispielsweise das *Abstract Factory Pattern*, das *Factory Method Pattern* und das *Singleton Pattern* (siehe jeweils [GHJV11]). Sie besitzen jedoch den Nachteil, dass der Entwickler sie bei jeder Klasse anwenden und damit für jede Klasse erneut die Infrastruktur für die Erzeugung von Objekten entwickeln muss. Es muss also der immer gleiche *Boilerplate Code* in vielen Klassen wiederholt werden, wobei er gleichzeitig nichts mit den eigentlichen Aufgaben der Klassen zu tun hat. Dies widerspricht dem *Don't Repeat Yourself-Prinzip* (kurz *DRY-Prinzip*) der Softwareentwicklung, vermindert die Übersichtlichkeit des Codes und verhindert, dass der Entwickler einer Klasse sich auf ihre eigentliche Aufgabe konzentrieren kann. Weiterhin muss ein Objekt die beschriebene Infrastruktur einer verwendeten Klasse kennen, um eine Referenz eines ihrer Objekte zu erhalten. Damit bindet es sich jedoch wieder an ein Detail einer spezifischen Implementierung, nämlich der des Design-Patterns, das für die Erzeugung der Objekte verwendet wird.

Wie nahezu jedes andere Framework verwendet auch die Java EE Plattform das *Inversion of Control*-Prinzip. Hierbei liegt die Kontrolle nicht wie üblich bei den Applikationskomponenten, die die Funktionalität des Frameworks durch Aufrufe verwenden. Die Applikationskomponenten implementieren stattdessen Schnittstellen des Frameworks

3. Verwendete Technologien

und werden bei ihm registriert. Sie erweitern also in gewisser Hinsicht das Framework. Die Kontrolle liegt dann bei dem Framework, das die Applikationskomponenten zum gegebenen Zeitpunkt ruft. Dieses Konzept lässt sich auch auf die Erstellung von Objekten anwenden.

Martin Fowler konkretisierte dieses Konzept in [Fow14] und prägte den Begriff *Dependency Injection* für die Verwendung des Inversion of Control-Prinzips bei der Erstellung von Objekten. Hierbei teilen die Objekte dem Dependency Injection Framework einerseits mit, welche Objekte sie verwenden, also von welchen sie abhängig sind. Dies kann auf Basis der Interfaces geschehen, die die zu verwendenden Objekte implementieren. Andererseits werden die Klassen, deren Objekte von dem Framework verwaltet werden sollen, bei dem selbigen registriert. Anschließend erstellt ein sogenannter *Assembler* oder *Injektor* des Frameworks die benötigten Objekte der registrierten Klassen und *injiziert* deren Referenzen in die Objekte, die sie verwenden. Die Mechanismen, die diese Injektionen realisieren und die Zeitpunkte zu denen sie stattfinden, variieren zwischen den Frameworks. Sie finden jedoch auf jeden Fall vor dem Aufruf einer Methode des Objektes statt, in das die Referenzen injiziert werden. Somit wird die Infrastruktur für die Erstellung von Objekten nur ein einziges Mal durch das Dependency Injection Framework realisiert. Weiterhin besitzen die Objekte ausschließlich Abhängigkeiten zu diesem Framework und nicht zu konkreten Klassen. Die Entscheidung, welche konkreten Klassen instanziiert und injiziert werden, wird an einer anderen Stelle vom Entwickler getroffen, beispielsweise in einer Konfigurationsdatei des Frameworks.

Die *Java Community* (siehe [Ora14b]) geht diesbezüglich noch einen Schritt weiter. So gehört die Spezifikation *Dependency Injection for Java 1.0* (siehe [Dep09]) bereits seit Version 5 zu den Spezifikationen von Java SE und wird damit von allen Java SE Implementierungen seit Version 5 unterstützt. Sie enthält rudimentäre Annotationen und dazugehörige Anforderungen, die von Dependency Injection Frameworks verwendet werden können. Insbesondere enthält sie die Annotation `@Inject`, mit deren Hilfe sogenannte *Injection Points* definiert werden können. Ein Injection Point definiert dabei die Abhängigkeit zu einem anderen Objekt und gibt an, wie und wo die Injektion stattfinden soll. Die Spezifikation sieht hierfür die drei Mechanismen *Field Injection*, *Method Injection* und *Constructor Injection* vor. Um die Field Injection zu verwenden, wird ein Attribut

mit der Annotation `@Inject` versehen, woraufhin das verwendete Framework diesem Attribut die zu injizierende Referenz zuweist. Um die Method Injection zu verwenden, wird eine entsprechende Setter-Methode mit der Annotation versehen, woraufhin das verwendete Framework diese Methode aufruft, um die Referenz zu injizieren. Zuletzt können Referenzen durch die Constructor Injection injiziert werden. Hierfür wird ein Konstruktor der Klasse mit `@Inject` annotiert, woraufhin das Framework diesen Konstruktor zur Erstellung eines entsprechenden Objektes verwendet und als Parameter entsprechende Referenzen übergibt. Diese mit Java SE mitgelieferten Annotationen stellen eine Schnittstelle bereit, die Applikationen unabhängig vom verwendeten Dependency Injection Framework verwenden können. Somit werden die Abhängigkeiten zwischen diesen beiden Softwarekomponenten noch weiter minimiert und das verwendete Framework kann leichter ausgetauscht werden.

Aufbauend auf diesen Annotationen definiert die Spezifikation *Contexts and Dependency Injection for the Java EE Platform* (kurz *CDI*, siehe [CDI13]) das Dependency Injection Framework der Java EE Plattform. Die von ihm automatisch verwalteten und damit injizierbaren Objekte werden *Managed Beans* genannt. Alle *EJB Session Beans* (siehe Abschnitt 3.1.6) und alle Klassen, die mit `@ManagedBean` annotiert wurden, sind Managed Beans. Weiterhin ist jede Klasse, die den Bedingungen der CDI-Spezifikation genügt, automatisch eine Managed Bean. Dies sind insbesondere alle *konkreten Top-Level Klassen*, die einen No-Arg Konstruktor oder einen Konstruktor, der mit `@Inject` annotiert wurde, enthalten. Jede Managed Bean ist an ein *Scope* gebunden, das den Lebenszyklus seiner Instanzen und ihre Sichtbarkeit für die Instanzen anderer Managed Beans bestimmt. Um das Scope einer Managed Bean zu definieren, wird ihre Klasse mit einer entsprechenden Annotation versehen. Es existieren unter anderem die von den *Servlets* bekannten Scopes *Request Scope*, *Session Scope* und *Application Scope* mit ihren zugehörigen Semantiken. Eine Managed Bean wird entsprechend mit der Annotation `@RequestScoped`, `@SessionScoped` oder `@ApplicationScoped` versehen. Jeder Typ von Scope ist dabei an einen von dem entsprechenden Container verwalteten *Kontext* gebunden, der alle Instanzen mit diesem Scope enthält. Dieser Kontext wird dann vom Container mit dem aktuellen Thread assoziiert, wobei immer nur ein Kontext pro Thread aktiv ist. Weiterhin wird der Kontext zwischen den Containern propagiert.

3. Verwendete Technologien

Dieser Mechanismus sorgt dafür, dass stets die korrekten Instanzen injiziert werden. Sofern eine Klasse nicht mit einer Annotation versehen ist, die ihr Scope definiert, oder mit `@Dependent` annotiert wurde, ist sie standardmäßig an das *Pseudo-Scope Dependent* gebunden. Die Instanzen, die an dieses Pseudo-Scope gebunden sind, werden niemals unter verschiedenen Injection Points geteilt und erben als abhängige Objekte den Lebenszyklus des Objektes, in das sie injiziert werden. Weiterhin ist das Pseudo-Scope Dependent immer aktiv.

Seit Version 7 der Java EE Plattform ist CDI standardmäßig aktiviert. Seine Konfiguration erfolgt über den Deployment Deskriptor `beans.xml`. Dort kann vom Entwickler definiert werden, welche Klassen CDI Managed Beans sind, bzw. welcher *Bean Discovery Mode* aktiviert ist, d.h. welche Klassen automatisch CDI Managed Beans sind. Standardmäßig, also auch wenn keine `beans.xml` im Archiv der Applikation existiert, ist der Bean Discovery Mode `annotated` aktiviert. In diesem Modus werden nur Klassen zu Managed Beans, die mit einer CDI-Annotation versehen wurden. Damit alle Klassen, die den genannten Kriterien von CDI entsprechen, automatisch Managed Beans werden, muss der Bean Discovery Mode `all` aktiviert werden. Es reicht also im Regelfall aus, eine `beans.xml` anzulegen, die diesen Modus aktiviert und ansonsten leer ist.

Über die Funktionalität von CDI hinaus unterstützt Java EE die Injektion von externen Ressourcen, die von dem Application Server verwaltet werden. Einerseits wird dem Deployer dadurch erlaubt, das Verhalten der Geschäftslogik einer Applikation zu verändern ohne auf deren Source-Code zuzugreifen. Andererseits kann eine Applikation so auf externe Ressourcen zugreifen ohne zu wissen, wie diese heißen, wie sie organisiert sind und wo sie sich befinden. Die Anforderungen einer Applikation an ihre Laufzeitumgebung können hierfür durch Annotationen und Deployment Deskriptoren ausgedrückt werden.

Die Injektion von externen Ressourcen basiert auf dem *Java Naming and Directory Interface* (kurz *JNDI*, siehe [Ora14a]). Der jeweilige Container, in dem eine Applikationskomponente ausgeführt wird, implementiert ihre Umgebung und stellt ihr diese als *JNDI Naming Context* (auch *JNDI Naming Environment*, siehe [Ora14a]) zur Laufzeit zur Verfügung. Die Geschäftsmethoden einer Applikationskomponente können die Einträge dieser Umgebung verwenden. Dies kann durch die JNDI-Interfaces, *Lookup-Methoden*

komponenten- bzw. containerspezifischer *Kontext*-Objekte oder durch die Injektion in die Felder oder Methoden der Applikationskomponente geschehen. Eine Injektion erlaubt dabei den deklarativen – und damit einfachsten – Zugang zu JNDI. Der Entwickler einer Applikationskomponente deklariert hierfür im Deployment Deskriptor oder mit den entsprechenden Annotationen alle zur Laufzeit erwarteten *Umgebungseinträge*. Der Container führt für diese Einträge dann zur Laufzeit einen *JNDI-Lookup* durch und injiziert die so erhaltenen Referenzen in die Felder oder Methoden der Applikationskomponente. Weiterhin stellt der Application Server Werkzeuge bereit, mit denen diese Umgebungseinträge beim Deployment initialisiert werden können. Im Folgenden werden die wichtigsten Typen von externen Ressourcen und die Annotationen für deren Injektion vorgestellt.

Mithilfe *einfacher Umgebungseinträge* können Konfigurationsparameter der Applikation realisiert werden. Sie können beispielsweise genutzt werden, um die Geschäftslogik einer Applikation anzupassen und sind mit einer der Basisklassen, `Class` oder `enum` typisiert. Ihre Injektion erfolgt durch die Annotation `@Resource`. Dabei findet das von Java SE bekannte *Auto-Boxing* bzw. *Auto-Unboxing* statt.

Weiterhin erlaubt Java EE die Injektion von *Resource Manager Connection Factory* Referenzen. Dies sind Objekte, die verwendet werden, um Verbindungen zu *Ressourcenmanagern* (siehe Abschnitt 3.1.3) aufzubauen. Zum Beispiel ist ein Objekt, das das `javax.sql.DataSource`-Interface implementiert, eine Resource Manager Connection Factory für `java.sql.Connection`-Objekte, die Verbindungen zu einem *Datenbankmanagementsystem* (kurz *DBMS*) implementieren. Dieser Mechanismus wird auch dafür verwendet, *Ressourcenadapter* (siehe Abschnitt 3.1.3) in einer Applikation zu verwenden. Die Injektion einer solchen Factory erfolgt durch die Annotation `@Resource`. Anschließend muss in einer Geschäftsmethode der Applikationskomponente eine entsprechende Methode der injizierten Factory aufgerufen werden, um eine Verbindung zu dem entsprechenden Ressourcenmanager zu erhalten. Diese Methode ist spezifisch für den Ressourcentyp bzw. den Ressourcenadapter. Die so erlangten Verbindungen werden *Managed Resources* genannt, da der Container so Einfluss auf das Ressourcenmanagement nehmen kann, z.B. durch *Connection Pooling* und das automatische Eintragen der Verbindungen in Transaktionen. Beim Deployment der Applikation bindet

3. Verwendete Technologien

der Deployer die Resource Manager Connection Factory Referenzen an tatsächliche Resource Manager Connection Factories, die in der anvisierten Laufzeitumgebung existieren.

Zuletzt unterstützt Java EE die Injektion von *Persistenzkontexten* (siehe Abschnitt 3.1.2). Eine Applikationskomponente erhält dadurch eine Referenz auf einen *Container-Managed EntityManager* (siehe Abschnitt 3.1.2). Die Injektion erfolgt mithilfe der Annotation `@PersistenceContext`, deren optionaler Parameter `unitName` den Namen der zu verwendenden *Persistenzeinheit* (siehe Abschnitt 3.1.2) spezifiziert, die im Deployment Deskriptor `persistence.xml` definiert wird. Dieser Parameter muss nicht angegeben werden, falls nur eine Persistenzeinheit in der Applikation existiert. Über den optionalen Parameter `type` kann außerdem der Typ des Persistenzkontextes spezifiziert werden. Sein Wert kann entweder `TRANSACTION` oder `EXTENDED` sein, wodurch entsprechend ein *transaction-scoped* oder *erweiterter* Persistenzkontext (siehe Abschnitt 3.1.2) injiziert wird. Der Standardwert dieses Parameters ist hierbei `TRANSACTION` und ein erweiterter Persistenzkontext kann nur in *Stateful Session Beans* (siehe Abschnitt 3.1.6) injiziert werden. Über den optionalen Parameter `synchronization` kann spezifiziert werden, ob der Persistenzkontext immer automatisch mit der aktuellen Transaktion synchronisiert wird oder, ob er explizit in die Transaktion eingetragen werden muss. Standardmäßig wird ein injizierter Persistenzkontext automatisch synchronisiert. Wenn also alle optionalen Parameter weggelassen werden, wird ein *Container-Managed Transaction-Scoped Persistence Context* (siehe Abschnitt 3.1.2) injiziert.

3.2. Client-Seite

Auf der Client-Seite wird das *Ember.js* (siehe [Til14]) verwendet. Es handelt sich um ein *JavaScript* Web-Applikations Framework, das auf dem *Model-View-Controller Pattern* (kurz *MVC*, siehe Abschnitt 4.1) basiert. Das primäre Ziel dieses Frameworks ist es, die Entwicklung von gut skalierenden *Single-Page Applikationen* zu ermöglichen. Das Prinzip einer solchen Applikation besteht darin, dass der Browser nur beim ersten Aufruf der Applikation von der Server-Seite eine vollständige Webseite und die dazugehörigen Res-

sourcen lädt. Jede darauf folgende Kommunikation mit dem Server wird programmatisch durch JavaScript mittels asynchroner Requests durchgeführt.

Die URL wird von den Entwicklern von Ember.js als Schlüssel-Feature des *World Wide Webs* und als Grund für dessen Erfolg gesehen. Daher ist ein Kernkonzept des Frameworks der sogenannte *Router*. Er verbindet alle weiteren Komponenten des Frameworks und der mit ihm entwickelten Applikation. Dabei bildet er eine URL auf eine Menge an verschachtelten *Templates* ab, wobei jedes dieser Templates von einem *Model* gestützt wird. Während dem User ein Template präsentiert wird, wird vom Framework automatisch die momentane URL im Browser entsprechend aktualisiert. Der aktuelle Zustand der Applikation steckt somit teilweise in der URL. Insbesondere besitzt die URL wieder die gleiche Semantik wie in einer „normalen“ Web-Applikation, die auf der Generierung von vollständigen Webseiten auf der Server-Seite basiert. URLs können also wieder als Lesezeichen gesetzt und weiter gegeben werden und beim Aufruf eines solchen Lesezeichens wird wieder der gleiche Inhalt angezeigt wie beim Setzen des Lesezeichens. Weiterhin verwaltet der Router sogenannte *Routes*. Dies sind Objekte, die dem jeweiligen Template mitteilen, welches Model angezeigt werden soll.

Ein *Template* beschreibt das *User Interface* (kurz *UI*) der Applikation. Jedes Template wird durch ein Model unterstützt. Ein Template besteht aus HTML und kann Inhalte von drei weiteren Typen enthalten. Zum einen können dies *Expressions* sein, die Informationen aus dem anzuzeigenden Model extrahieren. Diese Expressions werden dann im Template durch diese Informationen ersetzt. Weiterhin kann ein Template *Outlets* enthalten. Dies sind Platzhalter, die auf andere Templates verweisen, und durch deren Inhalt ersetzt werden. Zuletzt kann ein Template *Components* enthalten. Es handelt sich hierbei um applikationsspezifische, wiederverwendbare Komponenten.

Ein *Model* ist ein Objekt, das einen persistenten Zustand verwaltet. Folglich sind diese Objekte auch für die Datenbindung zum Server verantwortlich. In Ember.js existiert bereits ein REST Adapter, der sich nach den Konventionen von REST mit einem Server verbinden kann.

3. Verwendete Technologien

Ein *Controller* ist ein Objekt, das den Applikationszustand auf der Client-Seite verwaltet. Es kann von einem Template optional verwendet werden, um auch Eigenschaften des Zustands von der Client-Seite anzuzeigen.

Eine *Component* ist ein applikationsspezifisches HTML-Element, dessen Verhalten mit JavaScript implementiert wird. Sein Aussehen wird mit Templates beschrieben. Dieses Konzept dient vornehmlich der Kapselung und Wiederverwendung von Code, der sich ansonsten häufig in der Applikation wiederholen würde.

4

Architektur

Im Folgenden wird die Architektur der Applikation erläutert. Es wird dabei auf die Entwicklung des ersten Entwurfs, dessen Schwächen und deren Lösungen eingegangen. Zuletzt wird die tatsächlich umgesetzte Architektur erklärt.

Die Tatsache, dass es sich bei dem umgesetzten System um eine Web-Applikation handelt, impliziert eine *Client-Server Architektur*, wobei *Client* und *Server* das *Hypertext Transfer Protocol* (kurz *HTTP*, siehe [BLF⁺99]) verwenden. Dieses Protokoll sieht vor, dass Nachrichten, wie in Abbildung 4.1 gezeigt, nach dem *Request-Response Verfahren* ausgetauscht werden. Der Client sendet zunächst einen *Request* an den Server, welcher diesen bearbeitet und anschließend mit einer *Response* beantwortet.

4. Architektur

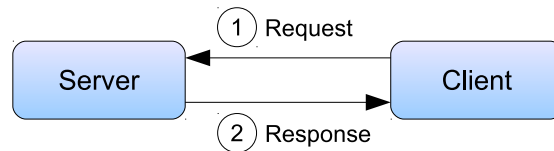


Abbildung 4.1.: Client-Server Architektur mit Request-Response Verfahren

4.1. Model-View-Controller

In „normalen“ Web-Applikationen besteht der Client ausschließlich aus dem Browser, welcher die Requests direkt an den Server schickt. Die Responses wiederum bestehen aus vollständigen HTML-Seiten und nicht nur Fragmenten, weshalb auf der Client-Seite mit einem vollständigen Neuladen der Seite reagiert wird.

Eine Anforderung an das System besteht jedoch darin (siehe Abschnitt 2), dass es eine *Rich Internet Application* sein soll. Die Anfragen solcher Web-Applikationen werden zwar immer noch vom Browser an den Server geschickt, es wird jedoch nicht zum Beispiel durch Klicken eines Links eine komplett neue Seite angefordert, sondern die Anfrage wird dynamisch im Browser durch *Javascript* angestoßen. Solche Anfragen werden als *Asynchronous Javascript and XML* (kurz *AJAX*, siehe [Gar14]) Requests bezeichnet. Obwohl der Name es suggeriert, müssen die ausgetauschten Nachrichten nicht zwangsläufig *XML* beinhalten. Üblich ist es auch Nachrichten im *JSON*-Format auszutauschen. Dieses Format besitzt weitaus weniger Overhead (alle Daten, die keine Nutzdaten im jeweiligen Kontext sind) und damit kodierte Nachrichten sind in *Javascript* direkt als Objekte ansprechbar, weshalb sich der Autor im Rahmen dieses Systems für dieses Format entschied. Die Server-Seite bietet für diese Art des Nachrichtenaustausches eine *REST* Schnittstelle an.

Die Konsequenz dieser Vorgehensweise ist, dass nur noch Objekte, also höchstens Fragmente von Webseiten zwischen Client und Server ausgetauscht werden. Es werden also nicht, wie bei „normalen“ Web-Applikationen üblich, auf der Server-Seite vollständige HTML-Seiten erzeugt, an die Client-Seite geschickt und dort vom Browser direkt grafisch dargestellt, sondern Objekte oder Fragmente von HTML-Seiten auf der Server-Seite

erzeugt, an den Client geschickt, vom Browser interpretiert und diese Interpretation der Daten in die aktuell angezeigte Webseite integriert und grafisch dargestellt. Um diese Aufgabe zu vereinfachen, verwendet man typischerweise ein Javascript Framework. Wie in Abschnitt 3.2 beschrieben, entschied sich der Autor für das *Ember.js* Framework, welches der Applikation auf Client-Seite eine *MVC-Architektur* (*Model-View-Controller*, siehe [GHJV11]) auferlegt. Die Interaktionen zwischen den drei Komponenten *Model*, *View* und *Controller* sind in Abbildung 4.2 ersichtlich. Der User interagiert ausschließlich mit der View-Komponente. Sie zeigt den aktuellen Zustand der jeweiligen System-Komponente an und leitet Interaktionen des Users durch zum Beispiel einen Klick an den jeweiligen Controller weiter. Dieser manipuliert das Datenmodell der System-Komponente, welches wiederum die View-Komponente aktualisiert. Wichtig hierbei ist, dass es in einer System-Komponente mehrere Views und dazugehörige Controller zu einem einzigen Model geben kann. Wird das Model von einem Controller manipuliert, so aktualisiert es alle bei ihm registrierten Views.

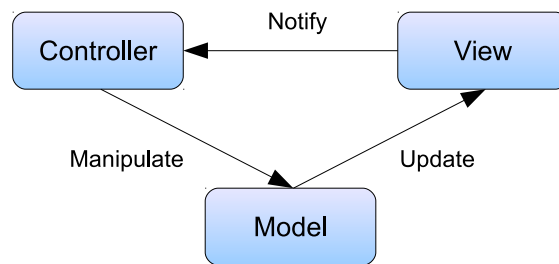


Abbildung 4.2.: Model-View-Controller Architektur

Wenn die Applikation auf der Client-Seite eine MVC-Architektur verwenden muss, liegt es nahe, auch auf der Server-Seite eine solche zu verwenden. Die Server-Seite kann sich mit der Client-Seite jedoch nur aufgrund einer vom Client zuvor gestellten Anfrage in Verbindung setzen. Insbesondere ist es nicht möglich, dass das Model mehrere Views aktualisiert. Beim Eingehen eines Requests wird stattdessen auf Server-Seite der entsprechende Controller aufgerufen, welcher daraufhin das Datenmodell manipuliert und anschließend die entsprechende View aufruft, welche die Antwort für den Client generiert. Die MVC-Architektur muss daher für die Server-Seite angepasst werden.

4. Architektur

Dafür etablierte sich insbesondere für Java-basierte Server die *MVC-2* Architektur (auch *Model-2* Architektur genannt, siehe [Ses14]).

Die gesamte bisherige System-Architektur ist in Abbildung 4.3 zu sehen. Die MVC-Architektur wurde auf der Client-Seite am Controller aufgebrochen, welcher nicht mehr direkt das Model auf Client-Seite manipuliert, sondern stattdessen einen Request an den Server schickt. In der darauf folgenden Response ist ein aktualisiertes Model enthalten, und die View wird mit eben diesem aktualisiert.

Im Folgenden werden die Interaktionen der einzelnen Komponenten genauer erläutert:

1. Der Benutzer sieht eine View-Komponente, die den aktuellen Zustand des Systems wiedergibt. Diese View-Komponente ist mit *HTML5*, *CSS3* (siehe [W3C14a]) und *HandleBars*, einer Komponente von *Ember.js* (siehe Abschnitt 3.2) realisiert. Der Benutzer interagiert mit der Webseite, indem er zum Beispiel auf einen Link klickt.
2. Es wird der zur aktuell dargestellten View-Komponente zugehörige Controller aufgerufen, wodurch die zur Interaktion passende Aktion des Systems ausgelöst wird. Der Controller ist mithilfe von *Ember.js* (siehe Abschnitt 3.2) realisiert.
3. Der Controller schickt einen entsprechenden Request an den Server. Dieser ist *RESTful* und falls Daten geändert werden sollen, sind diese im Body des Requests im JSON Format kodiert.
4. Der Server-seitige *FrontController* empfängt den Request und bearbeitet ihn. Während der Bearbeitung werden Cross-Cutting Concerns wie das Unmarshalling der im Body des Requests befindlichen JSON Daten zu Java Objekten durchgeführt. Hierbei wird auch direkt eine Fehler-Response zurückgeschickt, falls die Daten im Request „fehlerhaft“ sind. Am Ende ruft der FrontController den zum Request passenden Controller auf. Der FrontController ist eine generische Komponente des verwendeten Frameworks, in diesem Falle von *JAX-RS* (siehe Abschnitt 3.1.5).
5. Der gerufene Controller enthält die Geschäftslogik. Er manipuliert das Datenmodell des Systems. Die Controller sind mit *EJB* (siehe Abschnitt 3.1.6) realisiert und führen Cross-Cutting Concerns wie Authentifizierung und Autorisierung durch. Weiterhin sind alle Methoden der Controller transaktional, d.h. jede Methode

öffnet beim Betreten eine neue Transaktion und schließt dieselbe beim Verlassen wieder. Wenn es zu einem Fehler kommt, wird ein Rollback der veränderten Daten durchgeführt, ansonsten ein Commit. Die Models mit denen die Controller interagieren, sind *JPA Entities* (siehe Abschnitt 3.1.2). JPA kommuniziert dabei mit der Datenbank.

6. Nach der Manipulation des Datenmodells erhält der Controller die geänderten Daten in Form von Java-Objekten zurück. Dies können *Detached Entities* (siehe Abschnitt 3.1.2) oder *Data Transfer Objects* (kurz *DTOs*) sein.
7. Nachdem alle Aktionen der Geschäftslogik im Controller ausgeführt wurden, ruft der Controller die zum Request und Kontext passende View auf. Diese View ist mit *JAX-RS* (siehe Abschnitt 3.1.5) realisiert. Hier findet das Marshalling der Java-Objekte in das JSON Format statt.
8. Am Ende der Server-seitigen Verarbeitungskette wird die in der View generierte Antwort an den FrontController übergeben.
9. Der FrontController baut schließlich die *REST Response* zusammen und schickt sie zurück an den Client. Auf der Client-seite wird die Antwort wiederum vom JSON Format in Javascript Objekte transformiert.
10. Das nun geänderte Model aktualisiert die dazugehörige View. Das Model ist auch mit *Ember.js* (siehe Abschnitt 3.2) realisiert.
11. Die aktualisierte View-Komponente wird grafisch dargestellt und der veränderte Zustand des Systems somit dem Benutzer präsentiert.

4. Architektur

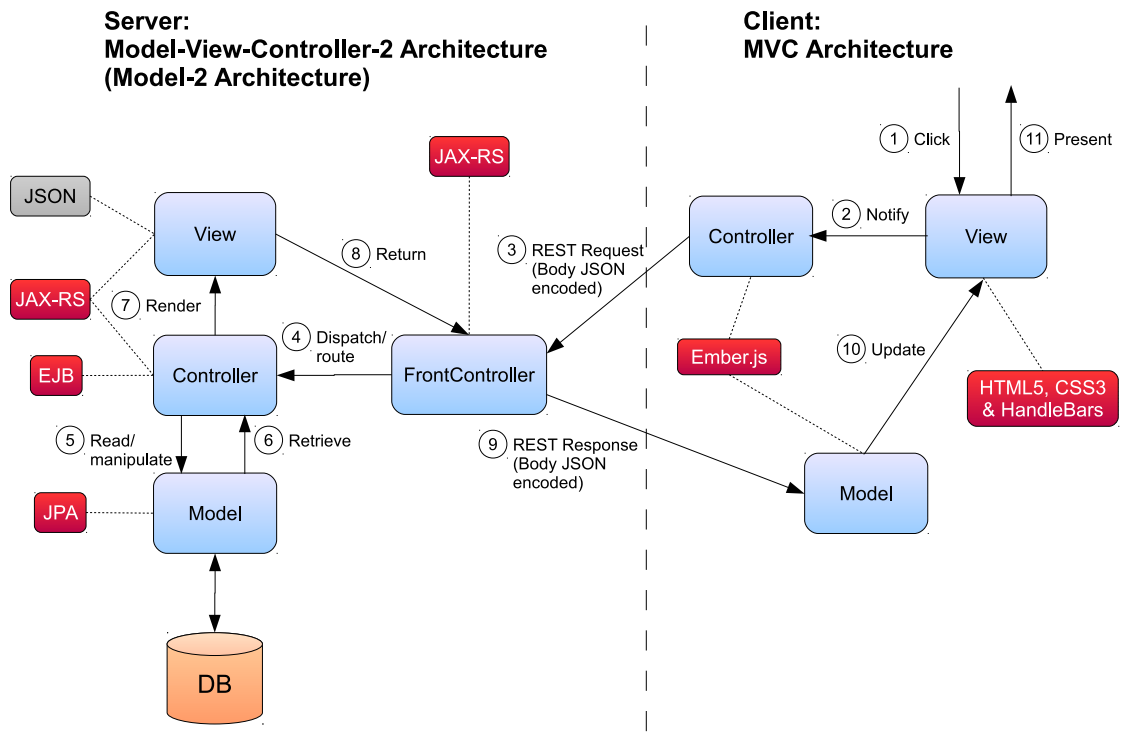


Abbildung 4.3.: System-Architektur basierend auf den MVC- und Model-2 Architekturen

4.2. Ports and Adapters

Im Laufe der Recherchen zu den verwendeten Technologien traten jedoch einige Schwächen auf der Server-Seite der vorgestellten Architektur zutage.

Zunächst gibt es eine semantische Diskrepanz zwischen einer View-Komponente in einer Model-2 Architektur und dessen was eine View-Komponente in der vorgestellten Architektur in Abbildung 4.3 darstellt. Die Views werden in einer klassischen Model-2 Architektur als dynamische Webseiten, zum Beispiel in Form von JSPs, dargestellt. Das Ergebnis der Ausführung einer solchen dynamischen Webseite ist eine Webseite im HTML Format, welche so auch zum Client geschickt wird. Diese HTML Seite ist quasi direkt vom Benutzer lesbar, weil sie nur noch von seinem Browser grafisch dargestellt werden muss. Die Bedeutung der View-Komponente in der Model-2 Architektur ist somit identisch zur Bedeutung der View-Komponente in einer klassischen MVC-Architektur: Sie stellt die Daten in einer vom Benutzer lesbaren Form dar. Das Ergebnis der Ausführung einer View in der vorgestellten Architektur ist zwar von Menschen lesbar, aber nicht dafür gedacht. Es ist vielmehr in einem Format das auf Maschinenlesbarkeit optimiert wurde. Weiterhin existieren die Views in der vorgestellten Architektur nur konzeptionell, da das Erzeugen der Antwort im JSON Format zwar im Controller angestoßen wird, aber vollständig von JAX-RS (siehe Abschnitt 3.1.5) übernommen wird.

Eine Anforderung an das System ist, dass es erweiterbar sein soll. Eine dieser Erweiterungen wird eine Administrationsoberfläche sein, die mit JSF realisiert werden wird. Es wird also auch eine Sicht auf das System geben, die Views im klassischen Sinne der Model-2 Architektur verwendet.

Die Model-2 Architektur erbt ein weiteres Problem der MVC-Architektur. Es ist hinreichend bekannt, dass bei der Anwendung der MVC-Architektur im Controller Code der Geschäftslogik mit Verwaltungs-Code für die jeweiligen Views vermischt wird. Dieses Problem tritt bei Desktop-Anwendungen, die zum Beispiel mit *Swing* (siehe [Rei11]) geschrieben wurden, genauso auf wie bei Web-Applikationen, die mit der Model-2 Architektur entwickelt wurden.

4. Architektur

Außerdem ziehen Einschränkungen in der *EJB Spezifikation* (siehe Abschnitt 3.1.6) weitere Probleme nach sich. Die Spezifikation verbietet explizit den Zugriff auf das Dateisystem. Da eine Anforderung an das System ist, dass der Benutzer auch am System „vorbei“ seine Dateien aus dem System heraus kopieren kann (siehe Abschnitt 2), kommt die Speicherung der Audiodateien als *Binary Large Object* (kurz *BLOB*) in der Datenbank nicht infrage. Die Lösung des Problems besteht darin, einen Connector, der über *JCA* (siehe Abschnitt 3.1.3) erreichbar ist, für den Dateizugriff zu verwenden. Dies hat zudem den Vorteil, dass jegliche Dateizugriffe transaktional ablaufen. Diese Abhängigkeit zu einer externen Ressource – selbst wenn der direkte Zugriff auf Dateien erlaubt wäre – ist durch eine reine Model-2 Architektur nicht abbildbar.

Die EJB Spezifikation verbietet weiterhin die manuelle Verwaltung von Threads und Prozessen. Somit ist auch das Erzeugen von neuen Prozessen verboten. Dies ist jedoch nötig, um eine weitere Anforderung an das System zu erfüllen: Die Transkodierung des Audiomaterials soll durch das externe Programm FFmpeg erfolgen. Doch selbst wenn es erlaubt wäre, Prozesse innerhalb des *EJB-Containers* (siehe Abschnitt 3.1.6) zu erzeugen, spricht dagegen, dass unter sämtlichen *Unix*-basierten Betriebssystemen Prozesse nur durch den *System-Call* `fork()` erzeugt werden können. Dieser System-Call kopiert den aktuellen Prozess in einen neuen Kind-Prozess. Erst anschließend wird das neue *Executable* als *Speicherabbild* in den neuen Kind-Prozess geladen. Daraus folgt, dass für einen kurzen Moment der gesamte aktuelle Prozess mit seinem Speicherabbild geklont im Speicher vorliegt. Folglich verdoppelt sich auch der Speicherverbrauch für diesen Moment. Der Speicherverbrauch eines Application Servers kann jedoch sehr hoch sein und dieses Verhalten multipliziert sich mit der Anzahl der Songs, die gleichzeitig gestreamed werden. Die Lösung des Autors besteht darin, die Erzeugung des Prozesses und das Ausführen von FFmpeg in eine separate JVM auszulagern. Diese wird vom Application Server aus per *Socket* angesprochen. Das Verwenden von Client-Sockets wird in der EJB Spezifikation explizit erlaubt. Daraus folgt wiederum das gleiche Problem wie beim Dateizugriff: Die Abhängigkeit zu einer externen Ressource kann durch eine reine Model-2 Architektur nicht abgebildet werden.

Insgesamt gibt es also Ressourcen, die von der Geschäftslogik benötigt werden, von einer Model-2 Architektur aber nicht abgebildet werden können. Andererseits gibt es

verschiedene Sichten auf das System, von denen nicht alle vom Benutzer lesbare Views einer klassischen Model-2 Architektur sind.

Eine Lösung für die genannten Probleme stellt die in Abbildung 4.4 gezeigte *Ports and Adapters* Architektur (auch bekannt als *Hexagonal Architecture*, siehe [Coc14]) dar. Der Fokus dieser Architektur liegt auf der *Geschäftslogik*, welche vollständig gekapselt wird und den Kern der Applikation bildet. Die Geschäftslogik unterliegt also keiner *Leaky Abstraction* (siehe [Spo14]), die zur Folge hätte, dass Code der Geschäftslogik mit dem für Views zuständigen Code vermischt wird und es entsteht somit eine **harte Grenze**. Die Grenze der kompletten Applikation wird durch das äußere Sechseck dargestellt.

Der Name *Ports and Adapters* impliziert bereits die Funktionsweise der Architektur. Die Geschäftslogik stellt ihre Dienste ausschließlich als API zur Verfügung, welche in für die unterschiedlichen Sichten auf die Applikation sinnvolle Teile aufgeteilt wird. Diese Teile werden *primäre Ports* genannt. Innerhalb der Applikation, aber außerhalb der Geschäftslogik, verwenden *primäre Adapter* diese Ports, um die jeweilige Sicht für die Außenwelt freizugeben. Die primären Adapter sind die treibende Kraft, indem sie die Geschäftslogik aufrufen. In eben diesen primären Adaptern steckt zum Beispiel der für das User Interface relevante Code und dort werden die dafür notwendigen Transformationen der Daten durchgeführt. Dieses Vorgehen hat den weiteren Vorteil, dass statt den UI-Adaptern einfach Test-Adapter mit den jeweiligen Ports kommunizieren können, um eine Sicht auf die Geschäftslogik zu testen.

Andererseits verwendet die Geschäftslogik selbst sogenannte *sekundäre Adapter* für die Kommunikation mit externen Ressourcen. Die API, die die Geschäftslogik von dem jeweiligen Adapter verwendet, wird *sekundärer Port* genannt. Die treibende Kraft ist hier die Geschäftslogik, die die sekundären Adapter aufruft.

Sowohl die primären, als auch die sekundären Adapter, sind Adapter im Sinne des *Adapter Design-Pattern* (siehe [GHJV11]).

Die Ports and Adapters Architektur stellt die neue Gesamtarchitektur des Systems dar. Sie macht jedoch keine Vorgaben wie die Geschäftslogik selbst realisiert sein muss, insbesondere werden keine Aussagen über deren Architektur getroffen. Ob die nun gewählte System-Architektur im Widerspruch zur vorherigen Model-2 basierten

4. Architektur

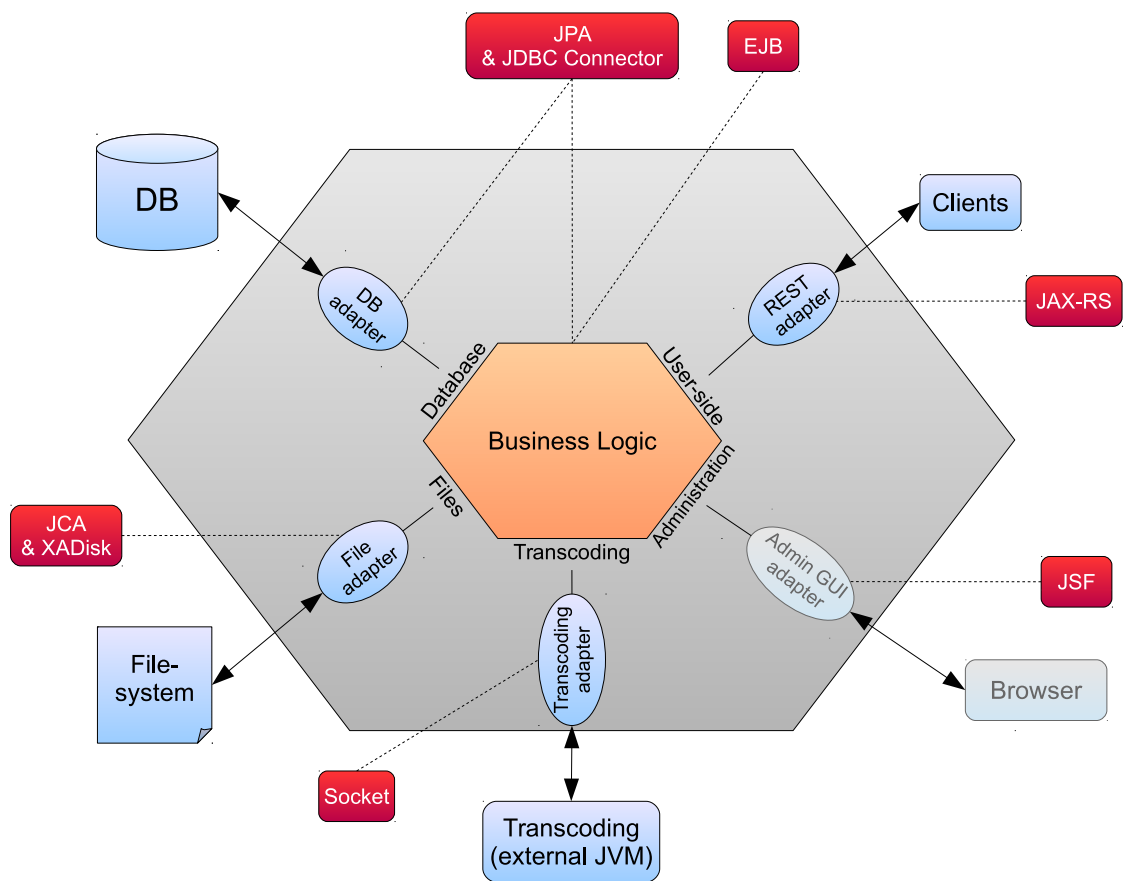


Abbildung 4.4.: System-Architektur basierend auf der Ports and Adapters Architektur

Architektur steht, ist daher maßgeblich von der Architektur der Geschäftslogik abhängig. Die Geschäftslogik enthält in der Ports and Adapters Architektur den Code, der in der Model-2 Architektur im Controller wäre. Weiterhin enthält sie das Datenmodell und somit den Code, der in der Model-2 Architektur im Model wäre. Im Falle eines UI-Adapters enthält dieser die Views der Model-2 Architektur und den dazugehörigen Verwaltungs-Code, der sich in der Model-2 Architektur fälschlicherweise auch im Controller befindet. Andersherum betrachtet ist die Controller-Komponente einer Model-2 Architektur in einer Ports and Adapters Architektur aufgetrennt, und auf die Geschäftslogik und den UI-Adapter verteilt. Zusammen mit einem geeigneten Datenbank-Adapter sind somit alle Komponenten vorhanden, die eine Model-2 Architektur beinhaltet. Es können somit beide Sichten auf das System, sowohl die Ports and Adapters Architektur, als auch die Model-2 Architektur, gleichzeitig korrekt sein. Die Ports and Adapters Architektur stellt in diesem Fall eine Abstraktion der Model-2 Architektur dar. Die Model-2 Architektur ist einerseits korrekt, kann andererseits aber nicht vollständig das System darstellen. Sie stellt viel mehr einen Querschnitt einer Ports and Adapters Architektur bestehend aus **einem** UI-Adapter, der Geschäftslogik und **einem** Datenbank-Adapter dar. Diese möglichen Zusammenhänge sind noch einmal in Tabelle 4.1 dargestellt.

Komponente in der Ports and Adapters Architektur	Komponente in der MVC-Architektur
Datenbank-Adapter	Model
Geschäftslogik	Model
Geschäftslogik	Controller
UI-Adapter	Controller
UI-Adapter	View

Tabelle 4.1.: Mögliche Zusammenhänge zwischen den Komponenten einer Ports and Adapters Architektur und jenen einer MVC-Architektur

Im Folgenden wird die in Abbildung 4.4 dargestellte System-Architektur genauer beschrieben. Die Geschäftslogik beinhaltet den zentralen Code der Applikation, welcher mit *EJB* (siehe Abschnitt 3.1.6) realisiert ist, und das Datenmodell in Form von *JPA Entities* (siehe Abschnitt 3.1.2).

4. Architektur

Ein sekundärer Adapter dient der Datenbankanbindung und wird mithilfe von JPA realisiert, welches unter Verwendung eines geeigneten *JDBC Connectors* (siehe Abschnitt 3.1.2) mit der Datenbank kommuniziert. Ein weiterer sekundärer Adapter wird für den Dateizugriff verwendet und löst das Problem, dass die EJB Spezifikation den direkten Zugriff auf das Dateisystem verbietet. Er ist durch *JCA* und den *XADisk Connector* realisiert (siehe jeweils Abschnitt 3.1.3). Zuletzt wird ein sekundärer Adapter für die Transkodierung des Audiomaterials benötigt, welcher die Problematik der Erstellung neuer Prozesse löst. Dieser ist durch einen Socket realisiert, über den eine externe JVM angesprochen wird. In ihr wird der FFmpeg Prozess gestartet, welcher über den Socket das zu transkodierende Audiomaterial als Datenstrom entgegen nimmt, es in das gewünschte Format transkodiert und das transkodierte Audiomaterial als Datenstrom über den Socket zurück an die Geschäftslogik schickt.

Die Benutzersicht auf die Geschäftslogik der Applikation wird über einen REST Adapter für die Außenwelt freigegeben. Teile der Geschäftslogik selbst werden dabei mithilfe von *JAX-RS* (siehe Abschnitt 3.1.5) auf REST Ressourcen abgebildet und die von ihr zurück gelieferten Java-Objekte in JSON Daten transformiert. Andersherum werden eingehende JSON Daten in für die Geschäftslogik verständliche Java-Objekte transformiert. Die Administrationsoberfläche wird außerhalb dieser Arbeit in einem Administrations UI-Adapter realisiert werden. Dieser wird JSF verwenden.

4.3. Entity-Control-Boundary

Die Geschäftslogik ist wie in [Bie09] beschrieben, als *Service Oriented Architecture* (kurz SOA, siehe [BBP09]) realisiert und besteht aus einer Menge von *Geschäftskomponenten*. Eine Geschäftskomponente ist eine in sich abgeschlossene, domänenspezifische und hoch-kohäsive Einheit. In ihr werden in der Anwendungsdomäne zusammenhängende Objekte und mit diesen ausführbare Aktionen gekapselt. So gibt es beispielsweise eine Geschäftskomponente, die für die Verwaltung von Usern zuständig ist und alle dafür notwendigen Objekte, wie zum Beispiel ein User-Objekt, enthält. Die Hauptaufgaben

einer Geschäftskomponente sind die Veröffentlichung ihrer *Spezifikation*, zumeist in Form einer *Schnittstelle*, und das Verbergen ihrer *Realisierung*.

Die Geschäftskomponenten verwenden die *Entity-Control-Boundary* (kurz *ECB*) Architektur, welche mit einer vereinfachten Form der *Unified Modeling Language* (kurz *UML*) beschrieben wird. Die dazugehörigen Diagramme werden *Robustness Diagrams* genannt und ein solches ist in Abbildung 4.5 zu sehen.

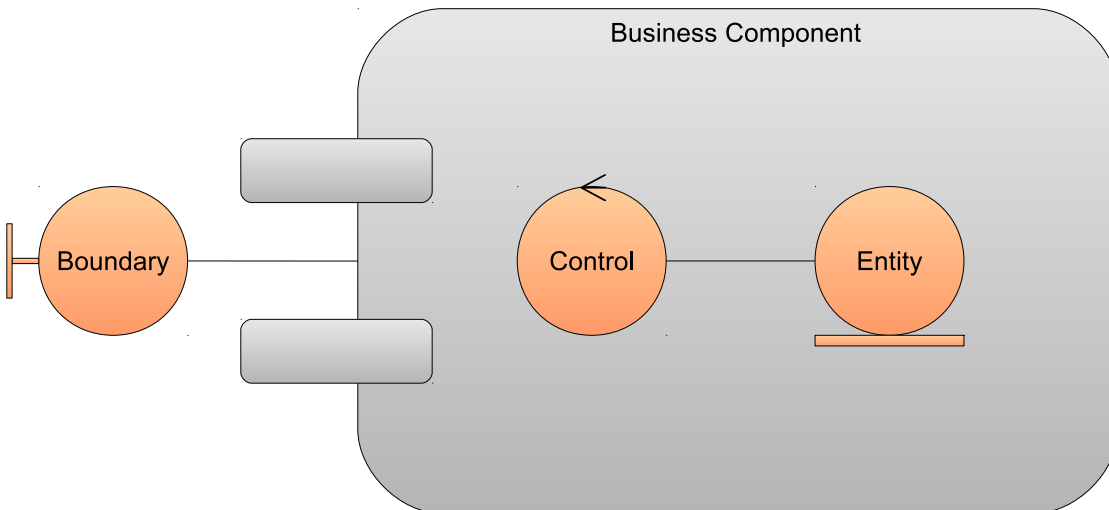


Abbildung 4.5.: Architektur der Geschäftslogik basierend auf dem Entity-Control-Boundary Design-Pattern

Im Folgenden werden die einzelnen Elemente der ECB Architektur im Kontext von *Java EE* (siehe Abschnitt 3.1) beschrieben. Zum *Entity* Element gehören alle *domänen-spezifischen* Objekte, die für die Geschäftskomponente relevant sind. In den meisten Fällen sind diese Objekte *persistent* und werden deshalb durch *JPA Entities* (siehe Abschnitt 3.1.2) abgebildet. Das *Control* Element repräsentiert einen Prozess oder eine Aktivität und wird mit *EJBs* (siehe Abschnitt 3.1.6) realisiert. Zu ihm gehören wiederverwendbare *fein-granulare Services*, welche jeweils eine Menge von *kohäsiven* Aktionen bereitstellen. Das *Boundary* Element ist eine *Fassade* (siehe [GHJV11]), die die Funktionalität einer Geschäftskomponente offen legt und direkt mit dem User Interface interagiert, sofern eines vorhanden ist. Dieses Element besteht aus *Service Façades* und ist somit der zentrale und einzige Einstiegspunkt für Komponenten au-

4. Architektur

Berhalb der Geschäftslogik in eine Geschäftskomponente. Zu diesem Zweck besteht die Boundary aus einem *Interface* und dessen *Implementierung*, welche mit EJBs realisiert wird. Die Fassaden in einer Boundary stellen unter Verwendung der Services aus Control Elementen wieder jeweils einige kohäsive Aktionen bereit, diese sind jedoch im Abstraktionsgrad höherwertiger als die der Services. In den Boundaries werden somit ausschließlich Services aus Control Elementen *komponiert*.

Weil die ECB Architektur eine SOA ist, sind die von den Services und Service Façades bereitgestellten Methoden von *prozeduraler* Natur. Dies bedeutet insbesondere, dass zwischen zwei Methodenaufrufen **kein** Zustand zwischengespeichert wird. Weiterhin finden Cross-Cutting Concerns in den Service Façades ihre Anwendung, es wird zum Beispiel bei jedem Aufruf einer Methode einer Service Façade die Autorisierung des Benutzers überprüft und eine neue Transaktion gestartet. In den Services der Controls hingegen wird in jeder Methode eine laufende Transaktion vorausgesetzt.

Aus der *Ports and Adapters* Architektur (siehe Abschnitt 4.2) als Gesamtarchitektur zusammen mit der ECB-Architektur der Geschäftslogik ergibt sich insgesamt eine Architektur, die nicht im Widerspruch zu einer Model-2 Architektur (siehe Abschnitt 4.1) steht. Es gelten weiterhin die bereits beschriebenen Zusammenhänge zwischen einer Ports and Adapters Architektur und einer Model-2 Architektur, die Zusammenhänge zwischen der dort eingeführten Geschäftslogik und der Model-2 Architektur lassen sich weiter verfeinern: Das Entity-Element der ECB-Architektur enthält den Code, der in einer Model-2 Architektur in der Model-Komponente wäre und die Control- und Boundary-Elemente enthalten den Code, der in einer Model-2 Architektur in der Controller-Komponente wäre. Diese Zusammenhänge werden noch einmal in Tabelle 4.2 dargestellt.

Komponente in der Ports and Adapters Architektur bzw. Entity-Control-Boundary Architektur	Komponente in der MVC-Architektur
Datenbank-Adapter	Model
Geschäftslogik – Entity	Model
Geschäftslogik – Control	Controller
Geschäftslogik – Boundary	Controller
UI-Adapter	Controller
UI-Adapter	View

Tabelle 4.2.: Mögliche Zusammenhänge zwischen den Komponenten einer Ports and Adapters Architektur zusammen mit der Entity-Control-Boundary Architektur für die Geschäftslogik und jenen einer MVC-Architektur

4.4. Datenmodell

Das der Anwendung zugrunde liegende Datenmodell basiert auf einer relationalen Datenbank und wird in Abbildung 4.6 in Form eines *Entity-Relationship Diagramms* (kurz *ER-Diagramm*) gezeigt. Im Folgenden werden die im System vorkommenden Objekte, deren wesentliche Eigenschaften, und die dazugehörigen Datenbanktabellen beschrieben. Eine detaillierte Beschreibung der Spalten der einzelnen Datenbanktabellen ist Tabelle A.1 zu entnehmen, wobei in keiner Spalte ein `NULL`-Wert gespeichert werden darf, in deren Beschreibung nicht explizit darauf hingewiesen wird, dass der Wert **optional** ist.

Nahezu alle dem System bekannten Objekte besitzen eine eindeutig identifizierende, automatisch generierte Nummer `id` und die Zeitstempel `created_at` und `updated_at`, die wiedergeben, wann die Objekte angelegt und wann sie zuletzt verändert wurden. Das zentrale Element der Applikation ist der *Song*, welcher in der Tabelle `song` gespeichert wird. Er besitzt die von den Metadaten von Audiodateien bekannten Felder wie Titel, Titelnummer, Kommentar, Copyright, Encoder und CD-Nummer, welche in den Spalten `title`, `track_number`, `comment`, `copyright`, `encoder` und `cd_number` gespeichert werden. Die Tabelle enthält außerdem verschiedene Verwaltungsinformationen über den Song, wie zum Beispiel den Pfad der zugehörigen Audiodatei,

4. Architektur

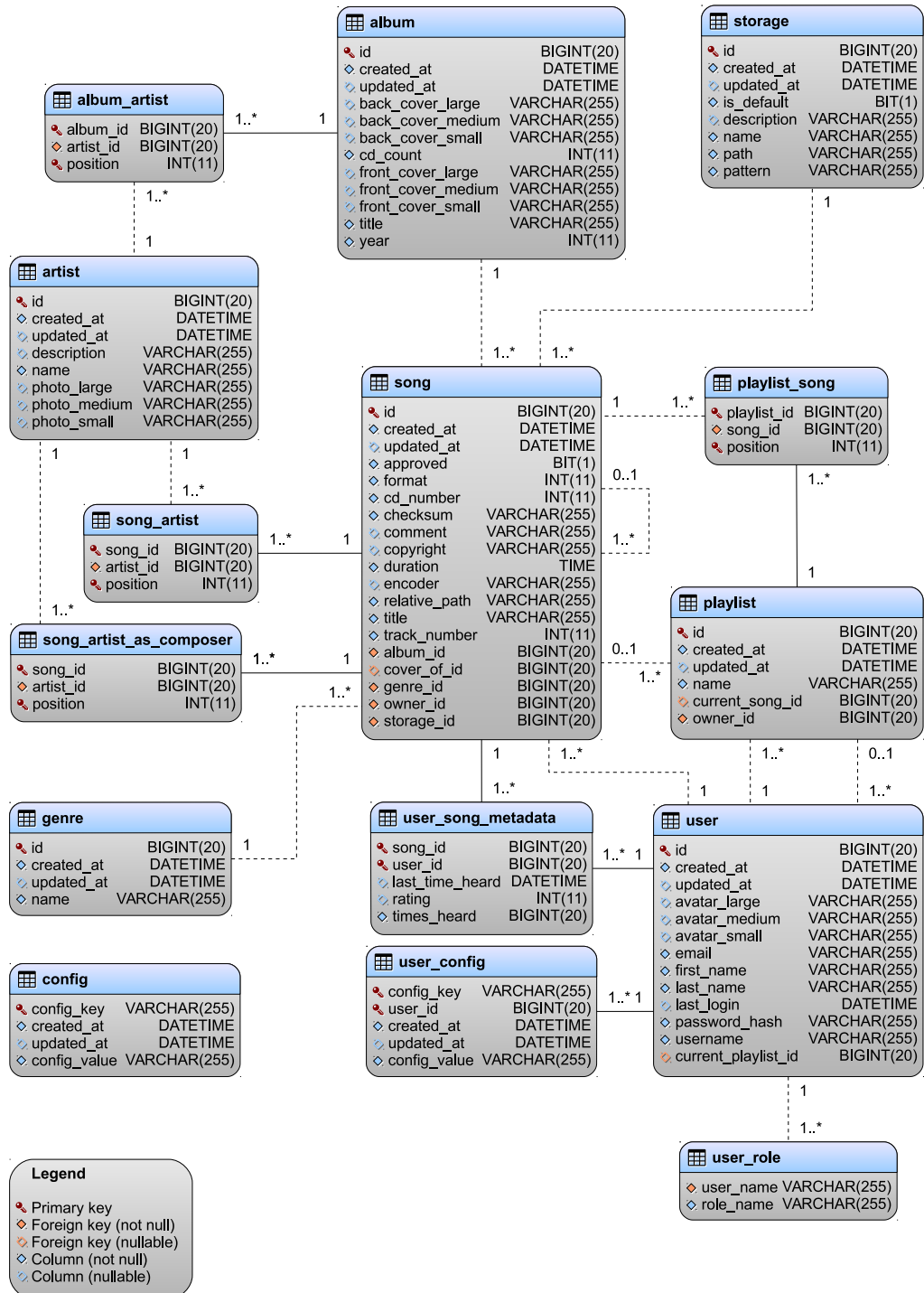


Abbildung 4.6.: Datenmodell der Applikation in Form eines Entity-Relationship Diagramms

deren Format, ihre Checksumme und die Dauer des Songs, welche in den Spalten `relative_path`, `format`, `checksum` und `duration` gespeichert werden. Der Pfad der Audiodatei ist dabei ein relativer Pfad unterhalb des referenzierten *Storage*-Objektes, welches durch die Spalte `storage_id` referenziert wird. Ein Storage-Objekt ist ein Ort, an dem Audiodateien gespeichert sind, und es wird in der Tabelle `storage` gespeichert. Dieses Konstrukt wurde eingeführt, um der Anforderung zu genügen, dass das System auch in Bezug auf den Zugriff auf Audiodateien erweiterbar sein soll (siehe Abschnitt 2). Momentan können hiermit ausschließlich verschiedene Ordner, in denen sich Audiodateien befinden, von der Applikation verwaltet werden, zukünftig könnten jedoch auch andere Dienste wie der *Amazon Simple Storage Service* (kurz *Amazon S3*, siehe [Ama14a]) angesprochen werden. Je nach Wahl des Betriebssystems, unter dem die Applikation ausgeführt wird, können aber bereits jetzt Internetdienste und Netzwerkfreigaben in das Dateisystem eingehängt und der Applikation zugänglich gemacht werden. Die Storage-Objekte besitzen jeweils eine Beschreibung, einen Namen, einen Pfad und ein Muster, nachdem die Applikation in dem Storage-Objekt abzulegende Dateien benennt. Diese Eigenschaften werden in den Spalten `description`, `name`, `path` und `pattern` gespeichert. Weiterhin existiert in dieser Tabelle eine Spalte namens `is_default`, deren Wert bestimmt, ob es sich um das Standard-Storage-Objekt handelt, in welchem Songs standardmäßig abgespeichert werden, falls kein anderes Storage-Objekt spezifiziert wurde. Nachdem ein Song in das System importiert wurde, befindet er sich zunächst in einer *Staging-Area*, d.h. er ist noch nicht vollständig in die Musiksammlung importiert. Insbesondere erscheint er nicht beim Browsen durch die Sammlung und beim Suchen in der Sammlung. Da Musiksammlungen mitunter sehr groß werden können, soll durch dieses Vorgehen verhindert werden, dass inkorrekte Metadaten von importierten Songs Chaos in die Musiksammlung bringen. Die Metadaten von Songs in der Staging-Area können bearbeitet werden, und wenn diese vom Benutzer als korrekt erachtet werden, kann der Song vom Benutzer bestätigt werden, wodurch er vollständig in die Musiksammlung importiert wird. Ob dies geschehen ist, wird in der Spalte `approved` der `song`-Tabelle gespeichert. Weiterhin kann ein Song ein *Cover-Song* eines anderen Songs sein, welcher durch die Spalte `cover_of_id` referenziert wird. Ein Song wird immer von einem *User* gehalten, welcher durch die

4. Architektur

Spalte `owner_id` referenziert wird. Dies ist nach dem Importieren eines Songs derjenige Benutzer, der den Song importierte. Andere User, die in der Applikation das Recht dazu besitzen, können den Song jedoch übernehmen. Die noch fehlenden Metadaten wie das Album, auf dem sich der Song befindet, und das Genre zu dem der Song gehört, werden nicht direkt im Song, sondern in den eigenständigen Objekten *Album* und *Genre* gespeichert und durch die Spalten `album_id` und `genre_id` referenziert. Ein Song kann außerdem von mehreren Künstlern aufgenommen und von mehreren Komponisten komponiert worden sein. Sowohl ein Künstler, als auch ein Komponist, werden in der Applikation als *Artist* abgebildet. Dieser Ansatz ist der Tatsache geschuldet, dass in der modernen Musik die Künstler der Songs meistens auch deren Komponisten sind. Andersherum hat dieses Vorgehen bei klassischer Musik den Vorteil, dass ein Musiktitel von ein und demselben Komponist in der Applikation unterschiedliche Interpretationen von unterschiedlichen Künstlern in Form von unterschiedlichen Songs besitzen kann, die dennoch unter demselben Komponist geführt werden. Diese Beziehungen zwischen Songs und Artists werden durch die *Join-Tabellen* `song_artist` und `song_artist_as_composer` ausgedrückt. In beiden wird durch die Spalten `song_id` und `artist_id` ein Song und ein Artist referenziert. Die Künstler und Komponisten eines Songs sind dabei geordnet und diese Ordnung wird jeweils in der `position`-Spalte der jeweiligen Tabelle gespeichert.

Ein Album wird in der Tabelle `album` gespeichert. Es besitzt einen Titel, ein vorderseitiges Cover und ein rückseitiges Cover, in jeweils drei Größen als Eigenschaften, welche in den Spalten `title`, `back_cover_small`, `back_cover_medium`, `back_cover_large`, `front_cover_small`, `front_cover_medium` und `front_cover_large` gespeichert werden. Die Spalten der Cover beinhalten hierbei die Pfade zu den Bilddateien. Weiterhin werden das Veröffentlichungsjahr und die Anzahl der enthaltenen CDs in den Spalten `year` und `cd_count` gespeichert. Ein Album kann wie ein Song auch von mehreren Künstlern aufgenommen worden sein. Diese Beziehungen werden durch die Join-Tabelle `album_artist` realisiert. Der Aufbau ist identisch mit den Join-Tabellen zwischen Song und Artist, wobei in diesem Fall durch die Spalte `album_id` ein Album referenziert wird.

Ein Künstler wird in der Tabelle `artist` gespeichert. Er besitzt einen Namen, eine Beschreibung und ein Foto in drei Größen, welche in den Spalten `name`, `description`, `photo_small`, `photo_medium` und `photo_large` gespeichert werden. Die Spalten der Fotos beinhalten hierbei die Pfade zu den Bilddateien.

Ein Genre wird in der Tabelle `genre` gespeichert. Es besitzt nur einen Namen, der in der Spalte `name` gespeichert wird.

Ein User wird in der Tabelle `user` gespeichert. Er besitzt einen Vornamen, einen Nachnamen, einen Benutzernamen, eine E-Mail Adresse und einen Avatar in drei Größen als Eigenschaften, welche in den Spalten `first_name`, `last_name`, `username`, `email`, `avatar_small`, `avatar_medium` und `avatar_large` gespeichert werden. Die Spalten der Avatare beinhalten hierbei die Pfade zu den Bilddateien. Sein Passwort wird als *Hash* in der Spalte `password_hash` abgelegt und der Zeitstempel seines letzten Logins in der Spalte `last_login` gespeichert. Die von ihm aktuell oder zuletzt gehörte *Playlist* wird durch die Spalte `current_playlist_id` referenziert. Wird ein User gelöscht, so werden auch alle von ihm besessenen Songs und Playlists gelöscht. Ein Benutzer kann mehrere *Rollen* inne haben, welche in der Join-Tabelle `user_role` gespeichert werden. Eine Rolle entspricht dabei einer Berechtigung, bestimmte Aktionen der Applikation ausführen zu dürfen. Durch die Spalte `user_name` wird der jeweilige User durch seinen Usernamen referenziert und die Spalte `role_name` beinhaltet die jeweilige Rolle. Diese Form der Speicherung wird von den Application Servern gefordert. Weiterhin werden die benutzerspezifischen Applikationseinstellungen in der Tabelle `user_config` gespeichert. Ein Eintrag in dieser Tabelle entspricht dabei einem einfachen *Key-Value Paar*, welches in den Spalten `config_key` und `config_value` gespeichert wird. Der jeweilige User, zu dem der Eintrag gehört, wird durch die Spalte `user_id` referenziert. Diese generische Form der Speicherung ermöglicht zukünftigen Clients der Applikation, ihre Einstellungen auch zentral auf dem Server abzulegen. Die benutzerspezifischen Metadaten eines Songs werden in der Tabelle `user_song_metadata` abgelegt. Zu diesen Metadaten gehören der Zeitpunkt, zu dem der User den Song zuletzt hörte, in Form eines Zeitstempels, die Bewertung, die der User zu dem Song abgegeben hat, und, wie oft der User den Song insgesamt schon anhörte. Sie werden in den Spalten `last_time_heard`, `rating` und `times_heard` gespeichert. Da eine Anforderung (siehe

4. Architektur

Abschnitt 2) an das System ist, dass es die Rangordnung der Songs aus Sicht des Benutzers automatisch erstellt, hat der Benutzer über die Bewertung nur die Möglichkeit festzuhalten, ob er den Song *mag* (*like*) oder *nicht mag* (*dislike*). Die neutrale Bewertung wird durch `NULL` abgebildet. Eine genauere manuelle Bewertungsskala ist nicht vorgesehen, es werden aber die anderen beiden Metadaten automatisch erfasst. Die Applikation kann so beispielsweise über die Zähler, die festhalten, wie oft Songs gehört wurden, und die Information darüber, welche Songs der Benutzer mag, automatisch eine Rangordnung für den jeweiligen Benutzer erstellen. Die Spalten `song_id` und `user_id` referenzieren jeweils den Song und den Benutzer zu dem ein Metadatum gehört.

Eine Playlist wird in der Tabelle `playlist` gespeichert. Sie besitzt nur einen Namen, der in der Spalte `name` gespeichert wird. Sie wird immer von einem User gehalten, welcher durch die Spalte `owner_id` referenziert wird. Dies ist zunächst immer der erstellende User, andere User, die in der Applikation das Recht dazu, besitzen können jedoch die Playlist übernehmen. Weiterhin referenziert die Spalte `current_song_id` den aktuellen oder zuletzt gehörten Song in der Playlist. Eine Playlist kann mehrere Songs beinhalten. Diese Beziehungen werden in der Tabelle `playlist_song` abgebildet. Die Spalte `playlist_id` referenziert dabei die jeweilige Playlist und die Spalte `song_id` den jeweiligen Song. Eine Playlist ist geordnet und diese Ordnung wird durch die Spalte `position` abgebildet.

Die applikationsweiten und vom Administrator gesetzten Konfigurationseinträge werden in der Tabelle `config` gespeichert. Sie ist identisch mit der Tabelle `user_config` für benutzerspezifische Konfigurationseinträge, besitzt jedoch keine Referenzen auf User.

5

Implementierung und Implementierungsaspekte

Im Folgenden werden exemplarisch einige ausgewählte Aspekte der Implementierung beschrieben. Zunächst werden Standardfälle genauer beleuchtet, die repräsentativ für einen Großteil der Applikation sind. Anschließend wird die Lösung des Problems der Transkodierung des Audiomaterials (siehe Abschnitt 4.2) vorgestellt. Die Architektur (siehe Abschnitt 4) wird durch simple Java-*Packages* realisiert. Auf `package`- und `import`-Anweisungen wird in den folgenden Listings aus Platzgründen verzichtet. Außerdem werden nur Auszüge der jeweiligen Quelltexte gezeigt.

5.1. Geschäftslogik

Die *Geschäftslogik* (siehe Abschnitt 4.3) befindet sich unterhalb des Hauptpackage der Anwendung im Package `business`. Unterhalb dieses Package befindet sich für jede *Geschäftskomponente* (siehe Abschnitt 4.3) ein Subpackage, welches das gesamte Interface der jeweiligen Boundary-Komponente enthält. Dazu gehören insbesondere auch die Interfaces von etwaigen DTOs.

5.1.1. Entity

Sofern eine Geschäftskomponente eine *Entity*-Komponente (siehe Abschnitt 4.3) besitzt, befindet sich diese unterhalb des Package der Geschäftskomponente im Subpackage `entity`.

In Listing 5.1 wird beispielhaft die Implementierung der *Artist*-Entity (siehe Abschnitt 4.4) gezeigt. Es handelt sich um eine POJO-Klasse, die mit Annotationen angereichert wurde. Die Klasse ist eine Subklasse von `CrudEntity`, in der allgemeine Attribute wie `id`, `createdAt` und `updatedAt` definiert sind. Diese Oberklasse ist mit `@MappedSuperclass` annotiert, wodurch alle dort definierten Attribute auf Spalten in der Tabelle der jeweiligen Unterklasse abgebildet werden. Zunächst wird die Klasse `Artist` mit der `@Entity`-Annotation als *Entity* (siehe Abschnitt 3.1.2) markiert, wodurch alle Attribute automatisch mit den Standardeinstellungen von JPA auf Tabellenspalten in der Datenbank abgebildet werden. Die folgenden Annotationen dienen dem Überschreiben dieser Standards.

Zunächst wird mit `@Table` der Name der Tabelle überschrieben, die zu dieser Klasse gehört. Hierdurch werden die Tabellennamen in allen Datenbankmanagementsystemen und unter der Verwendung aller JPA-Implementierungen vereinheitlicht. Durch das `name`-Attribut der `@Column`-Annotation geschieht das gleiche für die Spaltennamen. Weiterhin kann mit den Attributen `unique` und `nullable` dieser Annotation definiert werden, ob Zeilen in der dazugehörigen Tabellenspalte einen eindeutigen Wert aufweisen müssen oder den `NULL`-Wert enthalten dürfen. Das `name`-Attribut der `Artist`-Entity ist weiterhin mit der *Bean Validation Constraint* `@NotEmpty` (siehe Abschnitt 3.1.4) annotiert, welche

sicherstellt, dass der Name eines Künstlers nie `NULL` oder leer ist (d.h. nur Whitespace enthält), bevor die dazugehörige Entity in der Datenbank abgespeichert wird.

Durch `@ManyToMany` wird eine m:n Beziehung definiert und mit dem dazugehörigen Attribut `cascade` wird für die angegebenen Lifecycle-Events eine Kaskadierung ausgelöst. Durch `PERSIST` werden in diesem Fall alle Alben eines neuen Künstlers gespeichert, sobald der neue Künstler gespeichert wird, und durch `REMOVE` werden alle Alben eines Künstlers mit diesem gelöscht. Mithilfe des `mappedBy`-Attributes der Annotation wird der Besitzer der Relation auf die dazugehörige `Album`-Entity gesetzt. Weiterhin muss eine Entity-Klasse einen Default-Konstruktor (oder No-Arg-Konstruktor) besitzen, dieser kann laut der JPA-Spezifikation allerdings die `protected`-Sichtbarkeit aufweisen.

Laut der JPA-Spezifikation muss die Konsistenz zwischen Entity-Objekten in der Applikation selbst gesichert werden. Dieses Problem wurde durch die Implementierung der generischen Klasse `ObservableList` gelöst, welche das *Decorator Pattern* (siehe [GHJV11]) umsetzt. Dafür implementiert sie das `List`-Interface, wobei ihr beim Erstellen selbst ein Objekt übergeben wird, dessen Klasse dieses Interface implementiert, und an das alle Methodenaufrufe delegiert werden. Weiterhin wird beim Erstellen ein Objekt übergeben, das das `ListListener`-Interface implementiert. Objekte, die dieses Interface implementieren, dienen als Callback für `ObservableList`-Objekte. Dafür existieren die Methoden `afterAdd` und `afterRemove`, die gerufen werden, nachdem ein Objekt zur Liste hinzugefügt bzw. eines von der Liste entfernt wurde. Die Implementierung des Callback-Interfaces und die Erstellung eines solchen List-Decorators werden in der Methode `getAlbums` der `Artist`-Entity gezeigt. Wird ein Album zu den Alben eines Künstlers hinzugefügt, so wird dieser Künstler zu den Künstlern des Albums hinzugefügt und wird ein Album von den Alben eines Künstlers entfernt, so wird dieser Künstler von der Liste der Künstler des Albums entfernt. Dieses Verhalten wurde in die allgemeinen Methoden `addElementIfNotContains` und `removeElementIfContains` der Oberklasse `CrudEntity` ausgelagert. Um das beschriebene Verhalten zu forcieren, existiert für die Alben ausschließlich ein Getter in der `Artist`-Klasse und kein Setter. Ein solcher List-Decorator wird nur beim ersten Aufruf der `getAlbum`-Methode erzeugt und im Attribut `relationUpdatingAlbumList` gespeichert. Dieses Attribut ist mit

5. Implementierung und Implementierungsaspekte

`@Transient` annotiert, wodurch es nicht auf eine Tabellenspalte in der Datenbank abgebildet wird.

Listing 5.1: Artist-Entity

```
1 @Entity
2 @Table(name = "artist")
3 public class Artist extends CrudEntity {
4
5     @NotEmpty
6     @Column(name = "name", unique = true, nullable = false)
7     private String name;
8
9     @ManyToMany(cascade = { PERSIST, REMOVE },
10                mappedBy = "artists")
11     private List<Album> albums;
12
13     @Transient
14     private ObservableList<Album> relationUpdatingAlbumList;
15
16     // Other attributes...
17
18     protected Artist() {
19     }
20
21     // Other constructors...
22
23     public List<Album> getAlbums() {
24         if (relationUpdatingAlbumList == null) {
25             relationUpdatingAlbumList = new ObservableList<>(
26                 albums, new ListListener<Album>() {
27
28                 @Override
```

```

28         public void afterAdd(final Album album) {
29             addElementIfNotContains(album.getArtists(),
30                                     Artist.this);
31         }
32
33         @Override
34         public void afterRemove(final Album album) {
35             removeElementIfContains(album.getArtists(),
36                                     Artist.this);
37         }
38     });
39
40     return relationUpdatingAlbumList;
41 }
42
43 // ...
44 }

```

In Listing 5.2 wird die Definition der anderen Seite einer m:n Beziehung am Beispiel der Album-Entity gezeigt (siehe Abschnitt 4.4). Mithilfe der `@JoinTable`-Annotation wird eine Join-Tabelle definiert und durch die Attribute `joinColumns` und `inverseJoinColumns` werden die *Fremdschlüssel* zur Tabelle dieser Entity bzw. zur Tabelle der referenzierten Entity definiert. Weiterhin wird der Datenbanktabelle durch `@OrderColumn` eine Spalte hinzugefügt, in der die Reihenfolge der Beziehungen zwischen Alben und Künstlern festgehalten wird.

Listing 5.2: Album-Entity

```

1 public class Album extends CrudEntity {
2
3     // ...

```

5. Implementierung und Implementierungsaspekte

```
4
5     @ManyToMany(cascade = PERSIST)
6     @JoinTable(name = "album_artist",
7                 joinColumns = @JoinColumn(name = "album_id",
8                                           nullable = false, updatable = false),
9                 inverseJoinColumns = @JoinColumn(name = "
10                                                    artist_id", nullable = false, updatable =
11                                                    false)
12             )
13     @OrderColumn(name = "position", nullable = false)
14     private List<Artist> artists;
15
16     // ...
17
18 }
```

5.1.2. Control

Sofern eine Geschäftskomponente eine *Control*-Komponente (siehe Abschnitt 4.3) besitzt, befindet sich diese unterhalb des Package der Geschäftskomponente im Subpackage `control`.

Da die meisten Services in den Boundary-Komponenten der Geschäftskomponenten zum größten Teil nur Objekte in der Datenbank speichern, lesen, aktualisieren und löschen (*Create, Read, Update, Destroy*, kurz *CRUD*), wurde dieser gemeinsame Code in einen generischen `CrudService` ausgelagert, der sich in der Control-Komponente einer *technischen Geschäftskomponente* im Package `technical.crud.control` unterhalb des `business`-Packages befindet. In den meisten Services der Geschäftskomponenten muss deswegen nicht mehr direkt der `EntityManager` von *JPA* (siehe Abschnitt 3.1.2) verwendet werden, sondern es reicht aus, den `CrudService` zu ver-

wenden. Dieser Service wird in Listing 5.3 auszugsweise vorgestellt und im Folgenden beschrieben.

Es handelt sich wieder um eine POJO-Klasse, die mit Annotationen angereichert wird. Mithilfe von `@Stateless` wird die Klasse zu einer EJB, genauer zu einer *Stateless Session Bean* (siehe Abschnitt 3.1.6). Da es sich um einen Service in einer Control-Komponente handelt, wird eine bestehende Transaktion vorausgesetzt. Dies wird durch die Annotation `@TransactionAttribute` mit dem Attribut `MANDATORY` ausgedrückt.

Durch `@PersistenceContext` wird eine `EntityManager`-Instanz von JPA als Resource *injiziert* (siehe Abschnitt 3.1.7). Mit der Methode `create` kann ein Entity-Objekt persistiert werden. Auf dem `EntityManager` wird `flush` aufgerufen, damit automatisch generierte Primärschlüssel der Entity erzeugt und gesetzt werden. Anschließend wird `refresh` aufgerufen, um eben diese Attribute zu aktualisieren und dem Aufrufer von `create` eine aktuelle Version der Entity zurückgeben zu können. Der Service enthält weiterhin die Methoden `findById`, `update` und `delete`, durch welche Entities nach ihrem Primärschlüssel aus der Datenbank geladen, aktualisiert und gelöscht werden können. Weiterhin befinden sich in diesem Service Methoden, um statische *Named-Queries* (siehe Abschnitt 3.1.2) ausführen zu können, und Methoden, die die *Criteria API* (siehe Abschnitt 3.1.2) für dynamische Queries verwenden, um alle Entities einer Klasse nach einer bestimmten Spalte sortiert aus der Datenbank zu laden.

Listing 5.3: CRUD-Service der generischen Control-Komponente

```

1 @Stateless
2 @TransactionAttribute(MANDATORY)
3 public class CrudService {
4
5     @PersistenceContext
6     private EntityManager em;
7
8     // ...
9
10    public <T> T create(final T entity){

```

5. Implementierung und Implementierungsaspekte

```
11         em.persist(entity);
12         em.flush();
13         return em.refresh(entity);
14     }
15
16     public <T> T findById(final Object id,
17                          final Class<T> type) {
18
19         return em.find(type, id);
20     }
21
22     public <T> T update(final T entity) {
23         return em.merge(entity);
24     }
25
26     public <T> void delete(final T entity) {
27         em.remove(entity);
28     }
29
30     // ...
31
32 }
```

5.1.3. Boundary

Die *Boundary*-Komponente (siehe Abschnitt 4.3) einer Geschäftskomponente befindet sich unterhalb des Package der Geschäftskomponente im Subpackage `boundary`.

In Listing 5.4 wird auszugsweise die Implementierung des `ArtistService` der Boundary-Komponente der Artist-Geschäftskomponente gezeigt. Es handelt sich um eine *Stateless Session Bean* (siehe Abschnitt 3.1.6), welche das Interface `ArtistService`

implementiert. Durch die Annotation `TransactionAttribute` und das dazugehörige Attribut `REQUIRES_NEW` wird ausgedrückt, dass beim Aufruf einer Methode eines Objekts dieser Klasse eine neue Transaktion gestartet wird. Weiterhin werden dem *EJB-Container* (siehe Abschnitt 3.1.6) die in dieser EJB verwendeten Sicherheitsrollen mittels `@DeclareRoles` bekannt gegeben. In diesem Auszug sind die Rollen auf `Role.ROLE_USER` beschränkt, welche jeder User der Applikation inne hat. Die Rollen werden in der Applikation durch den `enum Role` ausgedrückt, in den Annotationen können jedoch ausschließlich statische Werte verwendet werden, weshalb zu `Role.USER` die Konstante `Role.ROLE_USER` existiert und hier verwendet wird. Mithilfe von `@RolesAllowed` werden die Rollen festgelegt, die benötigt werden, um eine Methode auszuführen. Die Anwendung dieser Annotation auf Klassenebene legt die benötigten Rollen für alle Methoden der EJB fest, sie können jedoch durch die gleiche Annotation auf Methodenebene für eine Methode überschrieben werden. Durch die *CDI*-Annotation (siehe Abschnitt 3.1.7) `@Inject` wird eine `CrudService`-Instanz (siehe Abschnitt 5.1.2) injiziert.

Die Methode `findAll` gibt eine Liste aller Künstler zurück. Ihre Parameter sind mit *Bean Validation Constraints* (siehe Abschnitt 3.1.4) versehen, deren Gültigkeit beim Aufruf der Methode vor deren Ausführung durch einen *Interceptor* (siehe Abschnitt 3.1.6) des EJB-Containers überprüft wird. Das `ArtistSortCriterion` ist ein `enum`, das zum Interface dieser Geschäftskomponente gehört, und mit dessen Hilfe angegeben werden kann, nach welchem Attribut die Liste sortiert werden soll. Die `SortDirection` gibt an, ob aufsteigend oder absteigend sortiert werden soll. Über `offset` und `limit` kann angegeben werden, beim wievielten Eintrag in der Datenbanktabelle begonnen werden soll und wieviele Entities maximal zurückgegeben werden sollen. Diese Angaben werden insbesondere für die *Paginierung* verwendet. Die Abbildung zwischen einem `ArtistSortCriterion` und dem dazugehörigen Attribut des *Metamodels* (siehe Abschnitt 3.1.2) von JPA wird durch das `enum ArtistSortCrterionWithAttribute` und dessen Methode `withAttribute` hergestellt. Mithilfe des `CrudService` (siehe Abschnitt 5.1.2) wird nun die Liste aus der Datenbank geladen und abschließend in eine Liste von DTOs umgewandelt.

5. Implementierung und Implementierungsaspekte

Die Methode `findById` findet einen Künstler anhand seiner `id`. Es wird zunächst die `findById`-Methode des `CrudService` aufgerufen, um die entsprechende Entity aus der Datenbank zu laden. Abschließend wird, falls nicht `null` zurückgeliefert wurde, die Entity in ein entsprechendes DTO umgewandelt und zurückgegeben.

Listing 5.4: Artist-Service

```
1 @Stateless
2 @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
3 @DeclareRoles(Role.ROLE_USER)
4 @RolesAllowed(Role.ROLE_USER)
5 public class ArtistServiceBean implements ArtistService {
6
7     @Inject
8     private CrudService crudService;
9
10    @Override
11    public List<ArtistDto> findAll(@NotNull final
12                                   ArtistSortCriterion sortCriterion, @NotNull final
13                                   SortDirection sortDirection, @Min(0) final long offset,
14                                   @Min(0) final int limit) {
15
16        final ArtistSortCriterionWithAttribute
17            sortCriterionWithAttribute =
18            ArtistSortCriterionWithAttribute.withAttribute(
19                sortCriterion);
20
21        final List<Artist> artists = crudService.findAllSorted(
22            sortCriterionWithAttribute.getAttribute(),
23            sortDirection, offset, limit, Artist.class);
24
25        return ArtistDtoBean.from(artists);
26    }
27 }
```

```
19
20     @Override
21     public ArtistDto findById(final long artistId) {
22         final Artist artist = crudService.findById(artistId,
23             Artist.class);
24
25         if (artist == null) {
26             return null;
27         }
28
29         return ArtistDtoBean.from(artist);
30     }
31
32     // ...
33 }
```

5.2. REST Adapter

Der REST Adapter (siehe Abschnitt 4.2) befindet sich unterhalb des Hauptpackage der Anwendung im Package `adapter.rest`. Unterhalb dieses Package befindet sich für jede Geschäftskomponente der *Geschäftslogik* (siehe Abschnitt 4.3) ein entsprechendes Subpackage, welches das gesamte Interface der Geschäftskomponente auf REST Ressourcen und entsprechende HTTP-Methoden abbildet, die auf ihnen ausgeführt werden können.

Weil keine dem Autor bekannten *JAXB*-Implementierungen (siehe Abschnitt 3.1.5) in der Lage sind, ein Unmarshalling zu Interfaces durchzuführen, da diese nicht instanziiert werden können, werden im REST Adapter noch einmal DTOs eingeführt. Ein solches DTO wird in Listing 5.5 exemplarisch gezeigt. Es handelt sich um ein simples *Messenger*-Objekt, das außer den gezeigten Attributen ausschließlich Methoden für die

5. Implementierung und Implementierungsaspekte

Konvertierung von und zu DTOs der Geschäftslogik besitzt. Deswegen besitzen alle Attribute auch die Sichtbarkeit `private` und es gibt dennoch keine entsprechenden Getter und Setter. Das Marshalling und Unmarshalling von JAXB funktioniert auch auf Attributen mit `private`-Sichtbarkeit, es wird lediglich ein Default-Konstruktur vorausgesetzt, der jedoch auch `private` sein kann. Der JAXB-Implementierung wird die Klasse mittels der `@XmlRootElement` Annotation bekannt gegeben. Dies ist insbesondere beim Unmarshalling für die Zuordnung von JSON- oder XML- Daten zu Klassen wichtig. Alle Attribute, die für das Marshalling und Unmarshalling relevant sind, werden mit `@XmlElement` annotiert. Der zum jeweiligen Attribut gehörende Name in den JSON- und XML-Daten wird von JAXB aus dem Attributnamen abgeleitet. Er kann über das `name`-Attribut der Annotation überschrieben werden.

Listing 5.5: Artist REST DTO

```
1 @XmlRootElement(name = "artist")
2 public class ArtistRestDto extends CrudRestDto {
3
4     @XmlElement
5     private String name;
6
7     @XmlElement
8     private String description;
9
10    private ArtistRestDto() {
11        super();
12    }
13
14    // ...
15
16 }
```

Abschließend wird in Listing 5.6 exemplarisch die Implementierung einer *REST Resource* (siehe Abschnitt 3.1.5) gezeigt. Mithilfe der `@Path` Annotation wird die Res-

source unter einer URL unterhalb der URL der Applikation registriert, in diesem Fall `.../artists`. Dieselbe Annotation führt auf Methodenebene zur Registrierung der Methode unter einer URL unterhalb der URL der Ressource. In diesem Fall können auch Platzhalter für Pfad-Parameter angegeben werden, wie es bei der Methode `findById` der Fall ist. Diese Platzhalter können wiederum mittels der Annotation `@PathParam` auf Methoden-Parameter abgebildet werden. Weiterhin werden die von den *EJBs* (siehe Abschnitt 5.1.3) bekannten Annotationen für Sicherheitsrollen erneut verwendet. Diese sind insbesondere wichtig, um die *Credentials* des Users über seinen Browser entgegen zu nehmen. Der hier initialisierte Sicherheitskontext wird beim Aufrufen von Methoden der entsprechenden EJBs zum EJB-Container propagiert. Es ist anzumerken, dass derzeit nicht alle JAX-RS-Implementierungen diese Annotationen befolgen, weshalb die jeweils notwendigen Rollen zusätzlich im Deployment Deskriptor für den Web-Container (`web.xml`) definiert werden müssen. Mittels der *CDI*-Annotation (siehe Abschnitt 3.1.7) `@Inject` wird eine Instanz des `ArtistService` (siehe Abschnitt 5.1.3) injiziert. Die Annotationen `@POST`, `@GET`, `@PUT` und `@DELETE` geben an, für welche HTTP-Methode eine Methode der Ressource registriert wird. Mit den Annotationen `@Produces` und `@Consumes` kann angegeben werden, in welchem Format Daten mit der Client-Seite ausgetauscht werden. Mithilfe der Annotation `@QueryParam` können HTTP-Get-Parameter, die vom Client an die URL angehängt werden können, spezifiziert werden. Nach dem Aufrufen der Geschäftslogik, in Form von Methodenaufrufen der EJBs, müssen die Objekte noch in Wrapper-Objekte, die im Listing unten zu sehen sind, eingepackt werden, bevor sie zurückgegeben und von JAXB zu JSON-Objekten umgewandelt werden. Dieses Wrapping ist nötig, damit die JSON-Daten im von *Ember.js* (siehe Abschnitt 3.2) erwarteten Format sind. Für das Zurückgeben der Objekte wird die `Response`-Klasse von JAX-RS verwendet, welche das *Builder Pattern* (siehe [GHJV11]) verwendet. Dadurch ist es einfach möglich, zum Beispiel den Status-Code der HTTP-Response zu setzen.

Listing 5.6: Artist REST Ressource

```

1 @Path("artists")
2 @DeclareRoles(Role.ROLE_ADMIN)
3 @RolesAllowed(Role.ROLE_ADMIN)

```

5. Implementierung und Implementierungsaspekte

```
4 public class ArtistResource {
5
6     @Inject
7     private ArtistService artistService;
8
9     @GET
10    @Produces(MediaType.APPLICATION_JSON)
11    public Response findAll(
12        @QueryParam("sort") ArtistSortCriterion sortCriterion,
13        @QueryParam("dir") SortDirection sortDirection,
14        @QueryParam("offset") final long offset,
15        @QueryParam("limit") final int limit) {
16
17        if (sortCriterion == null) {
18            sortCriterion = ArtistSortCriterion.DEFAULT;
19        }
20        if (sortDirection == null) {
21            sortDirection = SortDirection.DEFAULT;
22        }
23
24        final List<ArtistDto> artists = artistService.findAll(
25            sortCriterion, sortDirection, offset, limit);
26
27        final ArtistsWrapper wrapper = new ArtistsWrapper();
28        wrapper.artists = ArtistRestDto.from(artists);
29
30        return Response.ok(wrapper).build();
31    }
32
33    @Path("{id}")
34    @GET
```

```

34 @Produces(MediaType.APPLICATION_JSON)
35 public Response findById(
36     @PathParam("id") final long artistId) {
37
38     final ArtistDto artist = artistService.findById(
39         artistId);
40
41     if (artist == null) {
42         return Response.status(SC_NOT_FOUND).build();
43     }
44
45     final ArtistWrapper wrapper = new ArtistWrapper();
46     wrapper.artist = ArtistRestDto.from(artist);
47
48     return Response.ok(wrapper).build();
49 }
50
51 // ...
52
53 @XmlRootElement
54 public static class ArtistWrapper {
55
56     @XmlElement
57     private ArtistRestDto artist;
58 }
59
60 public static class ArtistsWrapper {
61
62     @XmlElement
63     private List<ArtistRestDto> artists;

```

64
65
66

```
}  
}
```

5.3. Transkodierung

Im Folgenden wird die Implementierung des in Abschnitt 4.2 beschriebenen Ansatzes für die Transkodierung des Audiomaterials anhand des Ablaufes eines Request-Response-Zyklus beschrieben. Es wird dabei aus Platzmangel auf Code-Listings verzichtet. Die beschriebene Verarbeitungs-Pipeline wird in Abbildung 5.1 gezeigt.

1. Beim Eintreffen eines HTTP-Requests erhält die entsprechende Methode in der entsprechenden *REST Ressource* (siehe Abschnitt 3.1.5) Zugriff auf den `OutputStream` der dazugehörigen HTTP-Response und übergibt diesen an die `play-Methode` des `StreamService` in der Geschäftslogik.
2. Der `StreamService` erhält über den *XADisk Resource Adapter* (siehe Abschnitt 3.1.3) Zugriff auf den `InputStream` der entsprechenden Audiodatei.
3. Es wird ein Client-Socket zur Transkodierer-Komponente geöffnet, die sich außerhalb der JVM des Application Server in einer anderen JVM befindet und auf eingehende Verbindungen wartet. Diese Transkodierer-Komponente startet nun drei Threads und einen FFmpeg-Prozess.
4. Der `StreamService` beginnt, Daten aus dem `InputStream` der Audiodatei zu lesen und diese über den `OutputStream` des Sockets an die externe JVM zu schicken.
5. Der Input-Thread in der externen JVM liest die Daten von dem `InputStream` des Sockets und schreibt die Daten in den `OutputStream` des FFmpeg-Prozesses.
6. Der FFmpeg-Prozess nimmt die Daten über die *Standardeingabe* entgegen und transkodiert sie in ein anderes Audioformat.

7. Nach der Transkodierung der Daten gibt der FFmpeg-Prozess die transkodierten Daten über die *Standardausgabe* wieder aus. Dabei werden Statusmeldungen über die *Standardfehlerausgabe* ausgegeben, die konsumiert werden müssen, damit der FFmpeg-Prozess nicht blockiert.
8. Der Output-Thread der externen JVM liest die Daten von dem `InputStream` des FFmpeg-Prozesses und schickt sie über den `OutputStream` des Sockets an den `StreamService` der Geschäftslogik zurück. Der Error-Thread der externen JVM liest gleichzeitig die Daten von dem `ErrorStream` des FFmpeg-Prozesses, damit dieser nicht blockiert.
9. Der `StreamService` liest die Daten von dem `InputStream` des Sockets und schreibt sie in den `OutputStream` der HTTP-Response.

5. Implementierung und Implementierungsaspekte

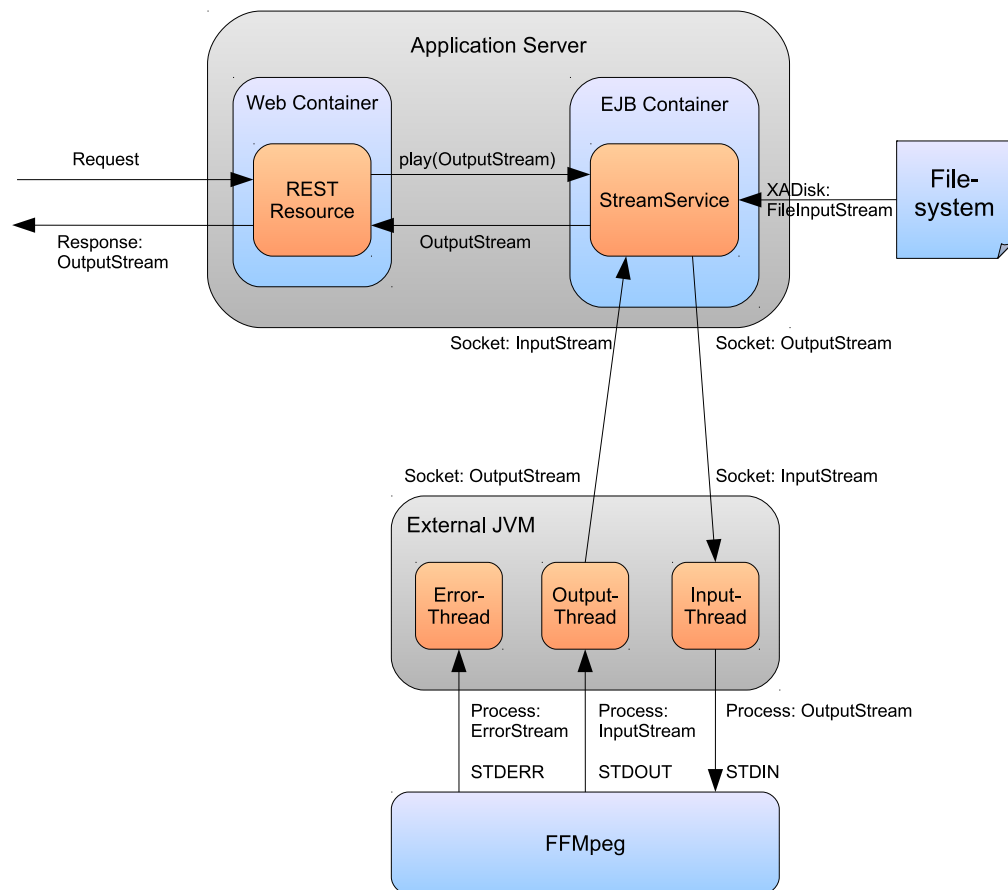


Abbildung 5.1.: Verarbeitungs-Pipeline während der Transkodierung

6

Abgleich der Anforderungen

Im Folgenden werden die definierten Anforderungen (siehe Abschnitt 2) mit der durchgeführten Implementierung (siehe Abschnitt 5) abgeglichen. Die einzelnen Anforderungen werden noch einmal in Tabelle 6.1 zusammen mit einer Bewertung, wie gut bzw. schlecht die jeweilige Anforderung umzusetzen war, aufgeführt. Die Skala der Bewertung reicht dabei von *gut* (+), über *neutral* (0), bis hin zu *schlecht* (–).

Anforderung	Bewertung
Das System ist in Form einer Rich Internet Application implementiert.	0
Die Server-Seite stellt ihre Funktionalität in Form einer REST Schnittstelle bereit.	+
Die Server-Seite ist mithilfe von Java EE implementiert.	0

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle 6.1.: Anforderungen mit Bewertungen, wie gut bzw. schlecht sie umsetzbar waren

6. Abgleich der Anforderungen

Anforderung	Bewertung
Die Client-Seite ist als Single-Page Web-Application implementiert.	+
Die Client-Seite ist mithilfe des Ember.js Frameworks implementiert.	+
Die Architektur der Anwendung stellt eine einfache Erweiterbarkeit sicher.	0
Auf importierte Songs kann „am System vorbei“ zugegriffen werden. Die Songs werden deshalb als Dateien auf der Server-Seite abgelegt.	–
Die Architektur stellt sicher, dass zukünftig möglichst leicht neue Orte für die Speicherung von Songs hinzugefügt werden können.	–
Das System kommt auf der Client-Seite ohne Browser-Plugins aus.	0
Die Applikation verwendet das externe Programm FFmpeg für die Transkodierung.	–
Es gelten die Semantiken, die auch bei einer lokalen Musikanwendung gelten.	+
Die Applikation ist mehrbenutzerfähig und es gibt Sicherheitsrollen.	0
Jeder Benutzer, der dazu berechtigt ist, kann jeden Song im System bewerten.	+
Das System hält automatisch fest, wann ein Benutzer einen Song zuletzt anhörte und wie oft er dies insgesamt tat.	+
Die Anwendung erlaubt dem Benutzer, die Musiksammlung nach diesen Kriterien zu sortieren und zu filtern.	0
Songs und Playlisten werden von Benutzern besessen. Benutzer, die dazu berechtigt sind, können Songs und Playlisten übernehmen. Wird ein Benutzer gelöscht, so werden auch alle von ihm besessenen Objekte gelöscht.	0

Tabelle 6.1.: Anforderungen mit Bewertungen, wie gut bzw. schlecht sie umsetzbar waren

7

Fazit und Ausblick

Es wurde zunächst die Motivation für die Entwicklung einer Web-Applikation gegeben, mit der es möglich ist, eine Musiksammlung zentral – inklusive ihrer Metadaten – zu speichern und zu verwalten (siehe Abschnitt 1). Anschließend wurden die Anforderungen an ein solches System spezifiziert (siehe Abschnitt 2) und die dafür ausgewählten Technologien erläutert (siehe Abschnitt 3). Weiterhin wurde eine geeignete Architektur diskutiert (siehe Abschnitt 4). Durch die Implementierung (siehe Abschnitt 5) eines Prototypen wurde gezeigt, dass die genannten Anforderungen mit den vorgestellten Technologien realisierbar sind (siehe Abschnitt 6).

Es bestehen jedoch ungelöste Probleme, die verhindern, dass das System bereits produktiv eingesetzt werden kann. Zum einen existiert noch keine Administrationsoberfläche, mit deren Hilfe das System verwaltet werden kann. Das Hinzufügen von Benutzern und das Anlegen von *Storage*-Objekten (siehe Abschnitt 4.4) ist momentan

7. Fazit und Ausblick

ausschließlich über den direkten Zugriff auf die Datenbank möglich. Weiterhin kann keine applikationsweite Konfiguration stattfinden. Die Administrationsoberfläche kann zukünftig mit JSF implementiert werden, da eine solche Oberfläche nur einmalig im System existieren muss und künftige Clients – z.B. in Form von Smartphone-Apps – keinen Zugriff auf diese Funktionalität über die REST Schnittstelle benötigen.

Weiterhin existiert noch keine Lösung für die Migration von bestehenden Daten in der Datenbank, falls sich in einer neuen Version des Systems das Datenmodell ändert und diese in einem Application Server, in dem eine frühere Version des Systems läuft, deployed wird. JPA bietet diesbezüglich nur zwei Optionen an. Es können einerseits Datenbankspalten zu bestehenden Datenbanktabellen hinzugefügt werden und neue Tabellen zur Datenbank während des Deployments hinzugefügt werden. Folglich ist in diesem Fall nur das Hinzufügen von neuen Entity-Klassen und das Hinzufügen von neuen Attributen zu bestehenden Entity-Klassen möglich. Weder Attribute, noch Entity-Klassen können entfernt oder verändert werden. Auf der anderen Seite gibt es die Option, alle Tabellen während des Deployments zu entfernen und neu zu erstellen. In diesem Fall sind den Veränderungen des Datenmodells zwar keine Grenzen gesetzt, andererseits gehen alle bestehenden Daten der Benutzer beim Deployment der neuen Version verloren. Da *JPQL* (siehe Abschnitt 3.1.2) keine Operationen für die Manipulation von Datenbankschemas bereitstellt, bietet Java EE außer den beiden genannten Konfigurationsoptionen von JPA keine weiteren Möglichkeiten, um auf die Initialisierung der Datenbank beim Deployment Einfluss zu nehmen.

Eine mögliche Lösung des Problems stellt das Datenbankmigrations-Framework *Flyway* (siehe [Fon14]) dar. Es speichert die Schemaversion der Datenbank in einer zusätzlichen Tabelle in der Datenbank und führt bei seiner Ausführung alle Migrationsskripte, die eine höhere Version aufweisen, aus. Dabei bringt es selbst keine Abstraktion der Datenbankabfragen, wie etwa *JPQL* oder die *Criteria API* (siehe Abschnitt 3.1.2) mit. Alle Migrationsskripte müssen in der jeweiligen nativen Abfragesprache vorliegen. Bei der Verwendung dieses Frameworks wird also die von JPA eingeführte Unabhängigkeit von Datenbankimplementierungen teilweise aufgegeben. Andererseits erlaubt es gerade dieser einfache Ansatz, beliebige Transformationen der bestehenden Daten durchzuführen.

Die Performanz der Applikation kann an einigen Stellen auch noch verbessert werden. Die applikationsweite Konfiguration könnte, unabhängig von eventuellen Caching-Mechanismen der JPA-Implementierung, während der Laufzeit der Applikation in einem Cache vorgehalten werden. Weiterhin durchläuft im bestehenden Prototyp ein Song jedes Mal, wenn er angehört wird, die gesamte beschriebene Verarbeitungs-Pipeline (siehe Abschnitt 5.3). Dies könnte auch durch einen Caching-Mechanismus optimiert werden.



Datenmodell

Tabelle	Spalte	Beschreibung
song	id	Dieser Primärschlüssel ist eine automatisch generierte Nummer, die den <i>Song</i> eindeutig identifiziert.
	created_at	Dies ist ein Zeitstempel, der wiedergibt, wann der <i>Song</i> vom Benutzer in das System importiert wurde.
	updated_at	Dies ist ein optionaler Zeitstempel, der wiedergibt, wann die Metadaten des <i>Songs</i> zuletzt verändert wurden.

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

A. Datenmodell

Tabelle	Spalte	Beschreibung
	approved	Dieser Wert bestimmt, ob der <i>Song</i> vom Benutzer bestätigt wurde. Alle Songs bei denen dieser Wert auf <code>false</code> steht, werden beim Browsen der Musiksamm- lung und beim Suchen nach Songs nicht berücksich- tigt.
	format	In diesem Wert wird das Format der in das System importierten Datei festgehalten. Er wird vom System intern durch einen <code>enum</code> abgebildet.
	cd_number	In diesem Wert wird gespeichert auf welcher CD eines <i>Albums</i> sich der <i>Song</i> befindet. <code>title</code> , <code>cd_number</code> und <code>album_id</code> müssen gemeinsam eindeutig sein.
	checksum	In diesem Wert wird die Checksumme der zu dem <i>Song</i> gehörenden Datei gespeichert. <code>checksum</code> und <code>storage_id</code> müssen gemeinsam eindeutig sein.
	comment	Dieser optionale Wert entspricht dem Kommentar-Feld des Metadaten-Teils der zugehörigen Audiodatei.
	copyright	Dieser optionale Wert entspricht dem Copyright-Feld des Metadaten-Teils der zugehörigen Audiodatei.
	duration	In diesem Wert wird die Dauer des <i>Songs</i> festgehal- ten.
	encoder	Dieser optionale Wert entspricht dem Encoder-Feld des Metadaten-Teils der zugehörigen Audiodatei.
	relative_ path	In dieser Spalte wird der relative Pfad der Audiodatei unterhalb des Pfades des <i>Storage</i> -Objektes gespei- chert. <code>relative_path</code> und <code>storage_id</code> müssen gemeinsam eindeutig sein.

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

Tabelle	Spalte	Beschreibung
	title	In dieser Spalte wird der Titel des <i>Songs</i> gespeichert. <code>title</code> , <code>cd_number</code> und <code>album_id</code> müssen gemeinsam eindeutig sein.
	track_ number	In diesem Wert wird die Titelnummer des <i>Songs</i> auf der CD gespeichert.
	album_id	Dieser Fremdschlüssel referenziert das <i>Album</i> auf dem sich der <i>Song</i> befindet. <code>title</code> , <code>cd_number</code> und <code>album_id</code> müssen gemeinsam eindeutig sein.
	cover_of_ id	Falls der <i>Song</i> ein Cover-Song eines anderen <i>Songs</i> ist, referenziert dieser Fremdschlüssel eben jenen Song. Der Wert ist optional .
	genre_id	Dieser Fremdschlüssel referenziert das <i>Genre</i> des <i>Songs</i> .
	owner_id	Dieser Fremdschlüssel referenziert den <i>User</i> , dem der <i>Song</i> gehört. Sofern der Song nicht von einem anderen Benutzer übernommen wurde, ist dies der Benutzer, der den Song importierte.
	storage_id	Dieser Fremdschlüssel referenziert ein <i>Storage</i> -Objekt. Die zu dem <i>Song</i> gehörende Audiodatei befindet sich in einem Pfad unterhalb des Pfades des Storage-Objektes. <code>relative_path</code> und <code>storage_id</code> müssen gemeinsam eindeutig sein. <code>checksum</code> und <code>storage_id</code> müssen gemeinsam eindeutig sein.
album	id	Dieser Primärschlüssel ist eine automatisch generierte Nummer, die das <i>Album</i> eindeutig identifiziert.

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

A. Datenmodell

Tabelle	Spalte	Beschreibung
	created_at	Dies ist ein Zeitstempel, der wiedergibt, wann das <i>Album</i> im System angelegt wurde. Dies geschieht wenn der erste <i>Song</i> dieses Albums vom Benutzer in das System importiert wurde.
	updated_at	Dies ist ein optionaler Zeitstempel, der wiedergibt, wann das <i>Album</i> zuletzt verändert wurde.
	back_ cover_ large	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche das rückseitige Cover in großen Abmessungen beinhaltet.
	back_ cover_ medium	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche das rückseitige Cover in mittleren Abmessungen beinhaltet.
	back_ cover_ small	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche das rückseitige Cover in kleinen Abmessungen beinhaltet.
	cd_count	In diesem Wert wird die Anzahl der CDs des <i>Albums</i> gespeichert.
	front_ cover_ large	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche das vorderseitige Cover in großen Abmessungen beinhaltet.
	front_ cover_ medium	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche das vorderseitige Cover in mittleren Abmessungen beinhaltet.
	front_ cover_ small	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche das vorderseitige Cover in kleinen Abmessungen beinhaltet.

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

Tabelle	Spalte	Beschreibung
	title	In dieser Spalte wird der Titel des <i>Albums</i> festgehalten.
	year	In diesem Wert wird das Jahr der Veröffentlichung des <i>Albums</i> gespeichert.
album_ artist	album_id	Dieser Fremdschlüssel referenziert das <i>Album</i> der Verknüpfung zwischen einem Album und einem <i>Artist</i> . Er gehört zu dem Primärschlüssel der Verknüpfung.
	artist_id	Dieser Fremdschlüssel referenziert den <i>Artist</i> der Verknüpfung zwischen einem <i>Album</i> und einem Artist.
	position	Dieser Wert bestimmt die Position der Verknüpfung zwischen einem <i>Album</i> und einem <i>Artist</i> und damit die Position, an welcher der Künstler unter allen Künstlern steht, die das Album aufnehmen. Der Wert gehört zu dem Primärschlüssel der Verknüpfung.
artist	id	Dieser Primärschlüssel ist eine automatisch generierte Nummer, die den <i>Artist</i> eindeutig identifiziert.
	created_at	Dies ist ein Zeitstempel, der wiedergibt, wann der <i>Artist</i> im System angelegt wurde, d.h. der erste <i>Song</i> dieses Künstlers vom Benutzer in das System importiert wurde.
	updated_at	Dies ist ein optionaler Zeitstempel, der wiedergibt, wann der <i>Artist</i> zuletzt verändert wurde.
	description	In diesem optionalen Wert kann eine Beschreibung des <i>Artist</i> gespeichert werden.
	name	In dieser Spalte wird der eindeutige Name des <i>Artist</i> gespeichert.

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

A. Datenmodell

Tabelle	Spalte	Beschreibung
	photo_ large	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche das Bild des <i>Artist</i> in großen Abmessungen beinhaltet.
	photo_ medium	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche das Bild des <i>Artist</i> in mittleren Abmessungen beinhaltet.
	photo_ small	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche das Bild des <i>Artist</i> in kleinen Abmessungen beinhaltet.
song_ artist	song_id	Dieser Fremdschlüssel referenziert den <i>Song</i> der Verknüpfung zwischen einem Song und einem <i>Artist</i> . Er gehört zu dem Primärschlüssel der Verknüpfung.
	artist_id	Dieser Fremdschlüssel referenziert den <i>Artist</i> der Verknüpfung zwischen einem <i>Song</i> und einem Artist.
	position	Dieser Wert bestimmt die Position der Verknüpfung zwischen einem <i>Song</i> und einem <i>Artist</i> und damit die Position, an welcher der Künstler unter allen Künstlern steht, die den Song aufnehmen. Der Wert gehört zu dem Primärschlüssel der Verknüpfung.
song_ artist_ as_ composer	song_id	Dieser Fremdschlüssel referenziert den <i>Song</i> der Verknüpfung zwischen einem Song und einem Komponisten, welcher in Form eines <i>Artist</i> abgebildet wird. Er gehört zu dem Primärschlüssel der Verknüpfung.

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

Tabelle	Spalte	Beschreibung
	artist_id	Dieser Fremdschlüssel referenziert den Komponisten der Verknüpfung zwischen einem <i>Song</i> und einem Komponisten, welcher in Form eines <i>Artist</i> abgebildet wird.
	position	Dieser Wert bestimmt die Position der Verknüpfung zwischen einem <i>Song</i> und einem Komponisten, welcher in Form eines <i>Artist</i> abgebildet wird, und damit die Position, an welcher der Komponist unter allen Komponisten steht, die den Song komponierten. Der Wert gehört zu dem Primärschlüssel der Verknüpfung.
genre	id	Dieser Primärschlüssel ist eine automatisch generierte Nummer, die das <i>Genre</i> eindeutig identifiziert.
	created_at	Dies ist ein Zeitstempel, der wiedergibt, wann das <i>Genre</i> im System angelegt wurde, d.h. der erste <i>Song</i> dieses Genres vom Benutzer in das System importiert wurde.
	updated_at	Dies ist ein optionaler Zeitstempel, der wiedergibt, wann das <i>Genre</i> zuletzt verändert wurde.
	name	In dieser Spalte wird der eindeutige Name des <i>Genres</i> gespeichert.
storage	id	Dieser Primärschlüssel ist eine automatisch generierte Nummer, die das <i>Storage</i> -Objekt eindeutig identifiziert.
	created_at	Dies ist ein Zeitstempel, der wiedergibt, wann das <i>Storage</i> -Objekt im System angelegt wurde.

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

A. Datenmodell

Tabelle	Spalte	Beschreibung
	updated_at	Dies ist ein optionaler Zeitstempel, der wiedergibt, wann das <i>Storage</i> -Objekt zuletzt verändert wurde.
	is_default	In dieser Spalte wird gespeichert, ob es sich um das Default- <i>Storage</i> -Objekt handelt. Es kann nur ein <i>Storage</i> -Objekt geben, bei dem dieser Wert auf <code>true</code> gesetzt ist.
	description	In diesem optionalen Wert kann eine Beschreibung gespeichert werden.
	name	In dieser Spalte wird der eindeutige Name des <i>Storage</i> -Objektes gespeichert.
	path	In diesem Wert wird der eindeutige Pfad des <i>Storage</i> -Objektes gespeichert.
	pattern	In dieser Spalte wird das Muster gespeichert, nachdem Audiodateien, die unterhalb des Pfades des <i>Storage</i> -Objektes gespeichert werden, benannt werden.
user	id	Dieser Primärschlüssel ist eine automatisch generierte Nummer, die den <i>User</i> eindeutig identifiziert.
	created_at	Dies ist ein Zeitstempel, der wiedergibt, wann der <i>User</i> im System angelegt wurde.
	updated_at	Dies ist ein optionaler Zeitstempel, der wiedergibt, wann der <i>User</i> zuletzt verändert wurde.
	avatar_large	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche den Avatar des <i>Users</i> in großen Abmessungen beinhaltet.

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

Tabelle	Spalte	Beschreibung
	avatar_ medium	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche den Avatar des <i>Users</i> in mittleren Abmessungen beinhaltet.
	avatar_ small	In diesem optionalen und eindeutigen Wert wird der Pfad der Bilddatei gespeichert, welche den Avatar des <i>Users</i> in kleinen Abmessungen beinhaltet.
	email	In dieser Spalte wird die eindeutige E-Mail Adresse des <i>Users</i> gespeichert.
	first_name	In diesem Wert wird der Vorname des <i>Users</i> gespeichert. <code>first_name</code> und <code>last_name</code> müssen gemeinsam eindeutig sein.
	last_name	In dieser Spalte wird der Nachname des <i>Users</i> gespeichert. <code>first_name</code> und <code>last_name</code> müssen gemeinsam eindeutig sein.
	last_login	Dies ist ein optionaler Zeitstempel, der wiedergibt, wann sich der <i>User</i> zuletzt einloggte.
	password_ hash	In dieser Spalte wird der <i>Hash-Wert</i> des Passworts gespeichert.
	username	In diesem Wert wird der eindeutige Benutzername gespeichert.
	current_ playlist_ id	Dieser optionale Fremdschlüssel referenziert die aktuelle oder zuletzt gehörte <i>Playlist</i> des Benutzers.
user_ role	user_name	Dieser Fremdschlüssel referenziert den <i>User</i> in der Verknüpfung zwischen einem <i>User</i> und einer <i>Role</i> .

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

A. Datenmodell

Tabelle	Spalte	Beschreibung
	role_name	In dieser Spalte wird der Name der <i>Role</i> der Verknüpfung zwischen einem <i>User</i> und einer <i>Role</i> gespeichert. Es wird von den <i>Application Servern</i> verlangt, dass die Benutzerrollen in dieser Form abgespeichert werden.
user_ config	config_key	In dieser Spalte wird der Schlüssel des Konfigurationseintrages des <i>Users</i> gespeichert. Sie gehört zu dem Primärschlüssel .
	user_id	Dieser Wert referenziert den <i>User</i> , zu dem der Konfigurationseintrag gehört. Er ist Teil des Primärschlüssels .
	created_at	Dies ist ein Zeitstempel, der wiedergibt, wann der Konfigurationseintrag im System angelegt wurde.
	updated_at	Dies ist ein optionaler Zeitstempel, der wiedergibt, wann der Konfigurationseintrag zuletzt verändert wurde.
	config_ value	In dieser Spalte wird der Wert des Konfigurationseintrages gespeichert.
user_ song_ metadata	song_id	Dieser Wert referenziert den <i>Song</i> , auf den sich die Metadaten beziehen. Er gehört zu dem Primärschlüssel .
	user_id	Dieser Wert referenziert den <i>User</i> für den die Metadaten angelegt wurden. Er gehört zu dem Primärschlüssel .
	last_time_ heard	In dieser optionalen Spalte wird gespeichert, wann der <i>User</i> den <i>Song</i> zuletzt anhörte.

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

Tabelle	Spalte	Beschreibung
	rating	In diesem optionalen Wert wird gespeichert, wie der <i>User</i> den <i>Song</i> bewertet. Er wird vom System intern durch einen <code>enum</code> abgebildet.
	times_ heard	In dieser Spalte wird gespeichert, wie oft der <i>User</i> den <i>Song</i> bereits anhörte.
playlist	id	Dieser Primärschlüssel ist eine automatisch generierte Nummer, die die <i>Playlist</i> eindeutig identifiziert.
	created_at	Dies ist ein Zeitstempel, der wiedergibt, wann die <i>Playlist</i> im System angelegt wurde.
	updated_at	Dies ist ein optionaler Zeitstempel, der wiedergibt, wann die <i>Playlist</i> zuletzt verändert wurde.
	name	In dieser Spalte wird der Name der <i>Playlist</i> gespeichert. <code>name</code> und <code>owner_id</code> müssen gemeinsam eindeutig sein.
	current_ song_id	Dieser optionale Fremdschlüssel referenziert den aktuellen oder zuletzt gehörten <i>Song</i> der <i>Playlist</i> .
	owner_id	Dieser Fremdschlüssel referenziert den <i>User</i> , der die <i>Playlist</i> besitzt. <code>name</code> und <code>owner_id</code> müssen gemeinsam eindeutig sein.
playlist_ song	playlist_ id	Dieser Fremdschlüssel referenziert die <i>Playlist</i> in der Verknüpfung zwischen einer <i>Playlist</i> und einem <i>Song</i> . Er gehört zu dem Primärschlüssel der Verknüpfung.
	song_id	Dieser Fremdschlüssel referenziert den <i>Song</i> der Verknüpfung zwischen einer <i>Playlist</i> und einem <i>Song</i> .

Die Tabelle wird auf der nächsten Seite fortgesetzt.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

A. Datenmodell

Tabelle	Spalte	Beschreibung
	position	Dieser Wert bestimmt die Position der Verknüpfung zwischen einer <i>Playlist</i> und einem <i>Song</i> , und damit die Position, an welcher der Song in der Playlist steht. Der Wert gehört zu dem Primärschlüssel der Verknüpfung.
config	config_key	In dieser Spalte wird der Schlüssel des Konfigurationseintrages gespeichert. Sie gehört zu dem Primärschlüssel .
	created_at	Dies ist ein Zeitstempel, der wiedergibt, wann der Konfigurationseintrag im System angelegt wurde.
	updated_at	Dies ist ein optionaler Zeitstempel, der wiedergibt, wann der Konfigurationseintrag zuletzt verändert wurde.
	config_value	In dieser Spalte wird der Wert des Konfigurationseintrages gespeichert.

Tabelle A.1.: Beschreibung der Tabellenspalten des Datenmodells

Abbildungsverzeichnis

1.1. Übersicht dieser Arbeit mit den Zusammenhängen der einzelnen Kapitel	3
3.1. Überblick über die <i>Java EE</i> -Architektur nach der <i>Java EE Spezifikation</i> (siehe [Jav13a])	11
3.2. Einordnung der Java Transaction API in Java EE nach der <i>JTA Spezifikation</i> (siehe [JTA13])	17
3.3. Abläufe zwischen Application Server, Applikation, Resource Adapter und Transaktionsmanager nach der <i>JTA Spezifikation</i> (siehe [JTA13])	19
3.4. Lebenszyklus einer JPA-Entity unter einem <code>EntityManager</code> mit einem <i>transaction-scoped persistence context</i> nach den Beschreibungen der <i>JPA Spezifikation</i> (siehe [Jav13b])	26
3.5. Überblick über die Connector Architecture nach der <i>JCA Spezifikation</i>	30
4.1. Client-Server Architektur mit Request-Response Verfahren	48
4.2. Model-View-Controller Architektur	49
4.3. System-Architektur basierend auf den MVC- und Model-2 Architekturen	52
4.4. System-Architektur basierend auf der Ports and Adapters Architektur	56
4.5. Architektur der Geschäftslogik basierend auf dem Entity-Control-Boundary Design-Pattern	59
4.6. Datenmodell der Applikation in Form eines Entity-Relationship Diagramms	62
5.1. Verarbeitungs-Pipeline während der Transkodierung	84

Tabellenverzeichnis

4.1. Mögliche Zusammenhänge zwischen den Komponenten einer Ports and Adapters Architektur und jenen einer MVC-Architektur	57
4.2. Mögliche Zusammenhänge zwischen den Komponenten einer Ports and Adapters Architektur zusammen mit der Entity-Control-Boundary Archi- tektur für die Geschäftslogik und jenen einer MVC-Architektur	61
6.1. Anforderungen mit Bewertungen, wie gut bzw. schlecht sie umsetzbar waren	85
6.1. Anforderungen mit Bewertungen, wie gut bzw. schlecht sie umsetzbar waren	86
A.1. Beschreibung der Tabellenspalten des Datenmodells	91
A.1. Beschreibung der Tabellenspalten des Datenmodells	92
A.1. Beschreibung der Tabellenspalten des Datenmodells	93
A.1. Beschreibung der Tabellenspalten des Datenmodells	94
A.1. Beschreibung der Tabellenspalten des Datenmodells	95
A.1. Beschreibung der Tabellenspalten des Datenmodells	96
A.1. Beschreibung der Tabellenspalten des Datenmodells	97
A.1. Beschreibung der Tabellenspalten des Datenmodells	98
A.1. Beschreibung der Tabellenspalten des Datenmodells	99
A.1. Beschreibung der Tabellenspalten des Datenmodells	100
A.1. Beschreibung der Tabellenspalten des Datenmodells	101
A.1. Beschreibung der Tabellenspalten des Datenmodells	102

Literaturverzeichnis

- [Ado14] ADOBE SYSTEMS, INC. *Adobe Flash Player*. <http://www.adobe.com/de/products/flashplayer.html>, letzter Zugriff: 27.02.2014
- [Ama14a] AMAZON WEB SERVICES, INC. *Amazon Simple Storage Service (Amazon S3)*. <http://aws.amazon.com/s3>, letzter Zugriff: 20.02.2014
- [Ama14b] AMAZON.COM, INC. *Amazon Cloud Player*. <http://www.amazon.com/gp/dmusic/marketing/CloudPlayerLaunchPage>, letzter Zugriff: 08.02.2014
- [BBP09] BUCHWALD, Stephan ; BAUER, Thomas ; PRYSS, Rüdiger. IT-Infrastrukturen für flexible, service-orientierte Anwendungen – ein Rahmenwerk zur Bewertung. In *13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW'09)*, Koellen-Verlag, March 2009 (Lecture Notes in Informatics (LNI) P-144), 524–543
- [Bea13] BEAN VALIDATION EXPERT GROUP, SPECIFICATION LEAD: EMMANUEL BERNARD. *JSR349: Bean Validation specification, Version 1.1 Final*. Red Hat, Inc., 2013
- [Bel14] BELLARD, Fabrice. *FFmpeg*. <http://www.ffmpeg.org/>, letzter Zugriff: 08.02.2014
- [Bie09] BIEN, Adam. *Real World Java EE Patterns – Rethinking Best Practices*. press.adam-bien.com, 2009
- [BLF⁺98] BERNERS-LEE, Tim ; FIELDING, Roy T. u. a. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*. The Internet Society, 1998

- [BLF⁺99] BERNERS-LEE, Tim ; FIELDING, Roy T. u. a. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*. The Internet Society, 1999
- [CDI13] CDI EXPERT GROUP, SPECIFICATION LEAD: PETE MUIR. *JSR 346: Contexts and Dependency Injection for the Java EE platform*. Red Hat, Inc., 2013
- [Coc14] COCKBURN, Alistair. *Hexagonal architecture*. <http://alistair.cockburn.us/Hexagonal+architecture>, letzter Zugriff: 19.02.2014
- [Con13] CONNECTOR 1.7 EXPERT GROUP, SPECIFICATION LEAD: ORACLE AMERICA, INC. *JSR 322: Java EE Connector Architecture Specification, Version 1.7*. Oracle America, Inc., 2013
- [Cro06] CROCKFORD, D. *RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON)*. The Internet Society, 2006
- [Dep09] DEPENDENCY INJECTION FOR JAVA 1.0 EXPERT GROUP, SPECIFICATION LEAD: ROD JOHNSON, BOB LEE. *JSR 330: Dependency Injection for Java 1.0*. Oracle America, Inc., 2009
- [Dij82] DIJKSTRA, Edsger W. On the role of scientific thought. In *Selected writings on Computing: A Personal Perspective*. Springer-Verlag, 1982
- [EJB09] EJB 3.1 EXPERT GROUP, SPECIFICATION LEAD: KENNETH SAKS, SUN MICROSYSTEMS, INC. *JSR 318: Enterprise JavaBeans, Version 3.1 – EJB Core Contracts and Requirements*. Sun Microsystems, Inc., 2009
- [Fie00] FIELDING, Roy T. *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine, 2000
- [Fon14] FONTAINE, Axel. *Flyway: the agile database migration framework for Java*. <http://flywaydb.org/>, letzter Zugriff: 08.02.2014
- [Fow14] FOWLER, Martin. *Inversion of Control Containers and the Dependency Injection pattern*. <http://www.martinfowler.com/articles/injection.html>, letzter Zugriff: 08.02.2014

- [Gar14] GARRETT, Jesse J. *Ajax: A New Approach to Web Applications*. <http://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php>, letzter Zugriff: 19.02.2014
- [GHJV11] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 2011
- [Goo14] GOOGLE, INC. *Google Music*. <http://music.google.com>, letzter Zugriff: 08.02.2014
- [GPSR13] GEIGER, Philip ; PRYSS, Rüdiger ; SCHICKLER, Marc ; REICHERT, Manfred. Engineering an Advanced Location-Based Augmented Reality Engine for Smart Mobile Devices. University of Ulm, October 2013 (UIB-2013-09). – Technical Report
- [GSP⁺14] GEIGER, Philip ; SCHICKLER, Marc ; PRYSS, Rüdiger ; SCHOBEL, Johannes ; REICHERT, Manfred. Location-based Mobile Augmented Reality Applications: Challenges, Examples, Lessons Learned. In *10th Int'l Conference on Web Information Systems and Technologies (WEBIST 2014), Special Session on Business Apps*, 2014
- [Jav13a] JAVA EE 7 EXPERT GROUP, SPECIFICATION LEAD: LINDA DEMICHEL, BILL SHANNON. *JSR 342: Java Platform, Enterprise Edition (Java EE) Specification, Version 7*. Oracle America, Inc., 2013
- [Jav13b] JAVA PERSISTENCE 2.1 EXPERT GROUP, SPECIFICATION LEAD: LINDA DEMICHEL. *JSR 338: Java Persistence API, Version 2.1*. Oracle America, Inc., 2013
- [JAX09] JAXB 2.2 EXPERT GROUP, SPECIFICATION LEAD: MARTIN GREBÁČ. *JSR 222: The Java Architecture for XML Binding (JAXB) 2.2*. Sun Microsystems, Inc., 2009
- [JAX13] JAX-RS 2.0 EXPERT GROUP, SPECIFICATION LEAD: SANTIAGO PERICAS-GEERTSEN, MAREK POTOCIAR. *JSR 339: JAX-RS: Java API for RESTful*

- Web Services, Version 2.0 Final Release*. Oracle America, Inc., 2013
- [JSF13] JSF 2.2 EXPERT GROUP, SPECIFICATION LEAD: ORACLE AMERICA, INC. *JSR 344: JavaServer Faces Specification*. Oracle America, Inc., 2013
- [JSP13] JSP 2.3 EXPERT GROUP, SPECIFICATION LEAD: KIN-MAN CHUNG. *JSR 245: JavaServer Pages Specification Version 2.3 – Maintenance Release 3*. Oracle America, Inc., 2013
- [JTA13] JTA 1.2 EXPERT GROUP, SPECIFICATION LEAD: ORACLE AMERICA, INC. *JSR907: Java Transaction API (JTA), Version 1.2*. Oracle America, Inc., 2013
- [KLM⁺97] KICZALES, Gregor ; LAMPING, John ; MENDHEKAR, Anurag ; MAEDA, Chris u. a. *Aspect-Oriented Programming*. Springer-Verlag, 1997
- [Mar02] MARTIN, Robert C. *Agile Software Development – Principles, Patterns, and Practices*. Prentice Hall, 2002
- [Ora14a] ORACLE AMERICA, INC. *Java Naming and Directory Interface (JNDI)*. <http://docs.oracle.com/javase/7/docs/technotes/guides/jndi/>, letzter Zugriff: 05.01.2014
- [Ora14b] ORACLE AMERICA, INC. *The Java Community Process(SM) Program*. <https://www.jcp.org>, letzter Zugriff: 08.02.2014
- [PLRH12] PRYSS, Rüdiger ; LANGER, David ; REICHERT, Manfred ; HALLERBACH, Alena. Mobile Task Management for Medical Ward Rounds – The MEDo Approach. In *1st Int’l Workshop on Adaptive Case Management (ACM’12), BPM’12 Workshops*, Springer, September 2012 (LNBIP 132), 43–54
- [PMLR14] PRYSS, Rüdiger ; MUNDBROD, Nicolas ; LANGER, David ; REICHERT, Manfred. *Supporting medical ward rounds through mobile task and process management*. Springer Verlag, 2014
- [PMR13] PRYSS, Rüdiger ; MUSIOL, Steffen ; REICHERT, Manfred. Collaboration Support Through Mobile Processes and Entailment Constraints. In *9th IEEE Int’l Conference on Collaborative Computing: Networking, Applications and*

Worksharing (CollaborateCom'13), IEEE Computer Society Press, October 2013

- [PTKR10] PRYSS, Rüdiger ; TIEDEKEN, Julian ; KREHER, Ulrich ; REICHERT, Manfred. Towards Flexible Process Support on Mobile Devices. In *Proc. CAiSE'10 Forum – Information Systems Evolution*, Springer, 2010 (LNBIP 72), 150–165
- [PTR10] PRYSS, Rüdiger ; TIEDEKEN, Julian ; REICHERT, Manfred. Managing Processes on Mobile Devices: The MARPLE Approach. In *CAiSE'10 Demos*, 2010
- [QUA01] QUALCOMM INCORPORATED. *RFC 2822: Internet Message Format*. The Internet Society, 2001
- [Red14] RED HAT, INC. *Hibernate Validator*. <http://hibernate.org/validator/>, letzter Zugriff: 08.02.2014
- [Rei11] REINHOLD, Mark. *JSR 336: Java SE 7 – Final Release Specification*. Oracle America, Inc., 2011
- [RPR11] ROBECKE, Andreas ; PRYSS, Rüdiger ; REICHERT, Manfred. DBIScholar: An iPhone Application for Performing Citation Analyses. In *CAiSE Forum-2011*, CEUR Workshop Proceedings, June 2011
- [Ser09] SERVLET 3.0 EXPERT GROUP, SPECIFICATION LEAD: RAJIV MORDANI. *JSR 315: Java Servlet Specification, Version 3.0*. Sun Microsystems, Inc., 2009
- [Ses14] SESHADRI, Govind. *Understanding JavaServer Pages Model 2 architecture – Exploring the MVC design pattern*. <http://www.javaworld.com/article/2076557/java-web-development/understanding-javaserver-pages-model-2-architecture.html>, letzter Zugriff: 19.02.2014
- [Spo14] SPOLSKY, Joel. *The Law of Leaky Abstractions*. <http://www.joelonsoftware.com/articles/LeakyAbstractions.html>, letzter Zugriff: 19.02.2014

- [SSP⁺13] SCHOBEL, Johannes ; SCHICKLER, Marc ; PRYSS, Rüdiger ; NIENHAUS, Hans ; REICHERT, Manfred. Using Vital Sensors in Mobile Healthcare Business Applications: Challenges, Examples, Lessons Learned. In *9th Int'l Conference on Web Information Systems and Technologies (WEBIST 2013), Special Session on Business Apps*, 2013, 509–518
- [Til14] TILDE, Inc. *Ember.js – A framework for creating ambitious web applications*. <http://emberjs.com/>, letzter Zugriff: 08.02.2014
- [Ver14] VERMA, Nitin. *Transaction with File Systems in Java | XADisk Transactional File System*. <https://xadisk.java.net>, letzter Zugriff: 08.02.2014
- [W3C14a] W3C, STYLE ACTIVITY LEAD: BERT BOS. *CSS current work & how to participate*. <http://www.w3.org/Style/CSS/current-work>, letzter Zugriff: 20.02.2014
- [W3C14b] W3C, WHATWG. *HTML5 – A vocabulary and associated APIs for HTML and XHTML – W3C Candidate Recommendation 04 February 2014*. W3C (MIT, ERCIM, Keio, Beihang), 2014
- [WF14] WIKIMEDIA FOUNDATION, Inc. *Rich Internet Application*. http://en.wikipedia.org/wiki/Rich_Internet_application, letzter Zugriff: 27.02.2014