Hasso-Plattner-Institut für IT-Systems Engineering

Bachelorarbeit

Eine Architektur für ein ereignisgesteuertes, webbasiertes Backend für Project-Zoom

Bachelorthesis

An eventdriven webbased backend architecture for Project-Zoom

Tom Bocklisch

tom.bocklisch@student.hpi.uni-potsdam.de

Betreut von Prof. Dr. Holger Giese,
Thomas Beyhl, M.Sc. und
Gregor Berg, M.Sc.

Systemanalyse und Modellierung

Potsdam, 24. Juni 2013

Zusammenfassung

Die deutsche Zusammenfassung der Arbeit.

Abstract

The English abstract.

Inhaltsverzeichnis

1 Motivation

Die School of Design Thinking (D-School) lehrt die kreative Herangehensweise an Probleme und die Entwicklung einfallsreicher Lösungen. Die Lehre findet dabei in verschiedenen Kursen statt. Studierende können sich die Grundlagen im Basic-Track aneignen und nach erfolgreicher Teilnahme ihr Wissen im Advanced-Track vertiefen. Neben den studentischen Kursen bietet die D-School auch Weiterbildungskurse für Unternehmen an.

In der Regel besteht ein D-School Kurs aus mehreren Kleingruppen von ca. sechs Teilnehmern, die gemeinsam an einem Projekt arbeiten. Das Projektthema wird dabei entweder von der D-School selbst oder, vor allem bei längeren Projekten, von externen Projektpartnern vorgeschlagen.

Bei der Arbeit an den Projekten durchlaufen die Teilnehmenden verschiedene Phasen: Während dabei zunächst das Verstehen und Beobachten des Problemfeldes im Vordergrund stehen, wird anschließend ein Standpunkt definiert. Es folgt die Phase der Ideenfindung, an die die Erstellung eines Prototypen anknüpft. Dieser Prototyp wird zum Schluss mit geeigneten Versuchspersonen getestet. Im Verlauf dieser Phasen entstehen unterschiedlichste Dokumente, welche die Lösungsfindung dokumentieren. (vgl. [PMW09])

Im Verlauf des Projektes kommt es durchaus vor, dass ein Team eine Phase mehrmals durchläuft oder in eine vorherige Phase zurückkehrt. Dies erschwert die Organisation der Dokumente, z.B. in einer einfachen hierarchischen Struktur. Weiterhin ist es schwierig allein aus den Dokumenten deren Entstehungsreihenfolge und Bedeutung zu erfassen. Meist entstehen während eines drei Monate andauernden Projektes verschiedenste Präsentationen, Zusammenfassungen, Prototypen, Bilder von Whiteboards und Interviewdokumentationen. Diese werden in der Regel in der von den Studierenden bevorzugten Art und Weise gespeichert und verwaltet, beispielsweise mit Hilfe von Dropbox[1], Google Docs[2] oder Box[3].

Das Verständnis der Dokumentation ist sowohl für das Projekt selbst, zum Verstehen und Erlernen des Prozesses, als auch als Ideenquelle für zukünftige Projekte wichtig. Ferner sind die erstellten Artefakte nützlich um neue Projektpartner zum werben, welche die Projekte betreuen.

[1] `http://www.dropbox.com`
[2] `http://drive.google.com`
[3] `http://box.com`

1.1 Zielsetzung von Project-Zoom

Project-Zoom soll der Verbesserung der Dokumentation dienen. Dazu sollen die von den Studierenden erzeugten Artefakte durch manuelles Anordnen in eine Form gebracht werden, welche den Prozess der Gruppe visualisiert. Die Mitarbeiter der D-School können diese entstandenen Graphen anschließend nutzen um die Projektverläufe zu analysieren und gegebenenfalls den D-School Prozess anpassen.

Für eine reibungslose Integration des Systems ist vor allem die Anbindung an bereits existierende IT-Systeme und die von den Studierenden für das Projekt verwendete Software wichtig. Ebendiese muss dabei mit moderatem Aufwand angepasst werden können, um andere externe Dienste anbinden zu können.

1.2 Abgrenzung

Die Arbeit beschreibt und bezieht sich auf das Bachelorprojekt „From Creative Ideas to Well-Founded Engineering“ und das umgesetzte Softwaresystem Project-Zoom. Insgesamt haben sechs Studierende an dem Projekt gearbeitet, die in ihren Bachelorarbeiten aus verschiedenen Blickwinkeln und mit verschiedenen Schwerpunkten Project-Zoom beschreiben.

Tom Herolds Arbeit [Her13] beinhaltet die Interaktion mit kontextsensitiven Graphen. Dabei geht es darum, den Umgang der Studierenden mit der Nutzeroberfläche so intuitiv wie möglich zu gestallten und den Nutzer bei der Erfassung dokumentationsrelevanter Eigenschaften zu unterstützen.

Die Ausführungen von Anita Diekhoff [Die13] beschäftigen sich mit der ...

Norman Rzepkas Bachelorarbeit [Rze13] thematisiert die webbasierte, eventgesteuerte, clientseitige Architektur von Project-Zoom. Hier wird näher darauf eingegangen, wie die Daten von der Datenbank, über das Backend asynchron an den Client ausgeliefert werden.

Die Bachelorarbeit von Dominic Bräunlein [Brä13] erläutert das Generieren und Bereitstellen von semantischen Advanced-Track, um dem Nutzer das Erkennen der Dokumente seines Projektes zu erleichtern und somit selbst bei wenig verfügbarem Platz so viele Informationen eines Dokumentes anzeigen zu können wie möglich.

Thomas Werkmeister befasst sich in seiner Arbeit [Wer13] mit der Anbindung externer Systeme zur Integration von Daten. Diese aggregierte Datenbasis ist die Grundlage für die Wissensbasis und die einzelnen Projekte.

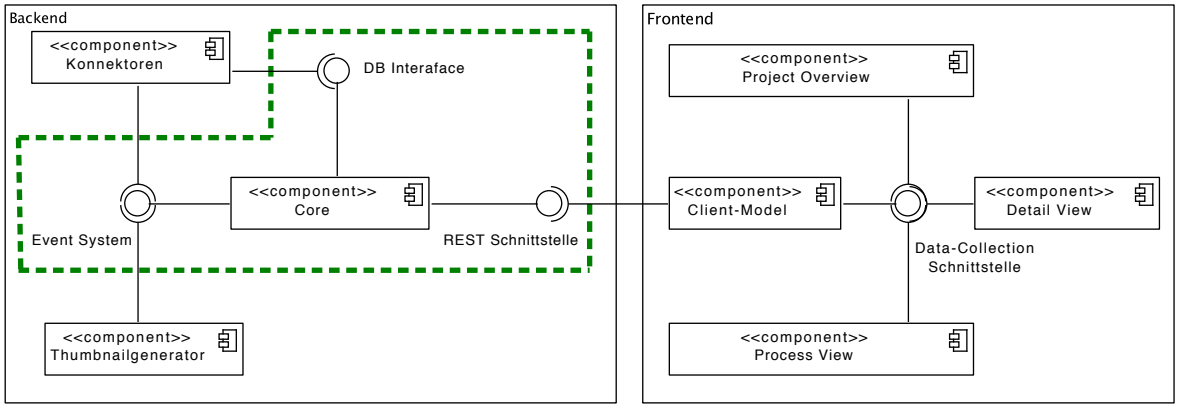

Abbildung 1.1 – Überblick über die verschiedenen Packete und ihre Schnittstellen; In gestricheltem Rahmen der für diese Arbeit relevante Teil der Architektur

Die Abbildung 1.1 zeigt einen groben Überblick über die Architektur des Systems. Der für diese Arbeit relevante Teil ist dabei gestrichelt eingerahmt. Die Komponenten, die in den Arbeiten [Wer13] (Konnektoren) und [Brä13] (Thumbnailgenerator) beschrieben sind, sind mit dem hier erläuterten Systemteil mittels eines Eventsystems verbunden. Das Client-Frontend ist über eine REST-Anbindung[4] an das Server-Backend angeschlossen. Mit der clientseitigen Implementierung der REST-Schnittstelle beschäftigt sich [Rze13].

1.3 Gliederung

In dieser Arbeit wird zunächst ein Überblick über das Gesamtsystem gegeben. Dazu werden die Anforderungen der D-School an das Backend analysiert, welche als Grundlage für die Wahl der verwendeten Technologien dienen. Anschließend wird die Architektur des Backends näher erläutert. Hier liegt der Hauptfokus zunächst auf einer neuen Art und Weise, eine Datenbank an eine Webapplikation anzubinden. Im Anschluss werden einige Feinheiten und ausgewählte Stellen der Datenmodellierung von Project-Zoom beleuchtet. Den Abschluss bildet die architekturelle Grundlage zur Anbindung externer Systeme für die Erweiterung von Project-Zoom.

[4] vgl. Konzept des Representational state transfer in [Fie00]

2 Überblick über die Backend-Architektur

Im Verlauf des Projektes wurden in Zusammenarbeit mit Studierenden und den Mitarbeitern der D-School verschiedene Anforderungen an Project-Zoom als Projektverwaltungs- und Dokumentationstool entwickelt [BBD+13]. Aus diesen leiten sich dann die verwendeten Technologien und die umgesetzte Architektur ab. Hier aufgeführt sind nur die für diesen Teil des Projektes wichtigsten und relevanten Anforderungen. Diese beziehen sich hauptsächlich auf die Erweiterbarkeit des Systems.

2.1 Systemanforderungen

Eine wichtige Anforderung ist die übersichtliche Verwaltung der Artefakte. Hierbei sollen die von den Teams erstellten Dokumente aus der jeweiligen Box[1] für die Studierenden in Project-Zoom zur Verfügung stehen. Diese Verfügbarkeit soll schnellstmöglich nach Hinzufügen einer neuen Datei zur Box gegeben sein. Hierfür ist es sinnvoll ein asynchrones System zu bauen. Neben der Anbindung von Box ist für die Zukunft auch der Zugriff auf facebook, Dropbox und Fileshare vorgesehen. Es zeigt sich, dass ein paralleles, unabhängiges Abfragen der einzelnen Dienste notwendig ist.

Die Studierenden sollen in der Lage sein, die Anwendung auch außerhalb der D-School verwenden zu können. Dies ist wichtig, da die Studierenden meist nur zwei Tage in der Woche in der D-School verbringen. Zum Teil treffen sich die Projekt-Teilnehmer außerhalb der D-School mit Interviewpartnern und Projektpartnern, welche am Stand des Projektes interessiert sind.

Über die Zeit verändern sich die von den Studierenden verwendeten Werkzeuge zur Dokumentation, deshalb muss das System erweitert, als auch gewartet werden können. Es ist sehr wahrscheinlich, dass ein Folgeprojekt an diesem Projekt weiterarbeiten wird. Aus diesem Grund soll das System verständlich sein und nach einer Einarbeitung von anderen Entwicklern erweitert werden können.

Neben diesen priorisierten Anforderungen gibt es noch eine Reihe weiterer Punkte, die beim Architekturentwurf berücksichtigt werden müssen:

- Es muss sichergestellt werden, dass kein unautorisierter Nutzer auf Daten Zugriff hat, die geschützt sind.

[1] `http://box.com`

- Es sollen 100 parallele Anfragen an das Backend pro Sekunde möglich sein.
- Aggregierte Daten sollen nicht gelöscht werden. Es kann durchaus vorkommen, dass Daten in aggregierten Quellen nach einiger Zeit gelöscht werden müssen. Project-Zoom soll die Daten dann weiterhin vorhalten können.

2.2 Technologieauswahl

2.2.1 Webanwendung vs. native Applikation

Bevor die Implementierung des Projektes beginnen konnte, galt es zu klären, welche Technologien verwendet werden. Eine der grundlegenden Entscheidungen war, ob eine Webanwendung oder eine native Anwendung entwickelt wird.

Auf Grund der Anforderung der D-School, dass die Anwendung auch außerhalb der Gebäude der D-School verwendbar sein muss, liegt eine Webanwendung nahe. Hinzu kommt, dass die Projekt-Mitglieder bereits mit dem Umgang von Webseiten und deren Navigation vertraut sind. Für die Erweiterbarkeit stellt dies ebenfalls einen enormen Vorteil dar, da ein zentrales System gewartet werden kann und neue Funktionen einfach eingespielt werden können. Zudem dreht sich das Projekt um das Thema Dokumentation, bei der oft dazu geneigt wird, sie aufzuschieben. Eine Webseite senkt hier die Hemmschwelle und umgeht die Notwendigkeit einer Verteilung und Installation des Programms.

Eine Trennung der Applikation in Frontend und Backend erlaubt eine klare Funktionstrennung. Für Webapplikationen befindet sich das Backend auf Serverseite und das Frontend im Browser auf Clientseite. Die klassischen Aufgaben der beiden Teile sind:

- Backend-Aufgaben:
  - Sammeln und zur Verfügung stellen von Daten
  - Kommunikation mit anderen Servern
  - Validierung eingehender Daten vom Client
  - Speicherung von Daten
  - Business-Logik
- Frontend-Aufgaben:
  - Kommunikation mit dem Backend zur Daten-Synchronisation
  - Visualisierung der Daten
  - Interaktion mit dem Nutzer

	Java + Spring	Groovy + Grails	Scala + Play
Authentifizierung	Ja mit Spring-WS	Ja mit Authentication Plugin	Ja mit SecureSocial
Json Unterstützung	Ja mit Jersey	Ja	Ja
Vorhandene Erfahrung im Entwicklerteam	Gering	Gut	Sehr Gut
Dokumentation	Gut	Gut	Gut
Asynchron	Nein	Nein	Ja
Zustandslos	Nein	Nein	Ja
Produktiv Einsatz	Einsatz	Trial	Trial
Wichtigkeit	Am wichtigsten	Wichtig	Wichtiger

Tabelle 2.1 – Vergleich von verschiedenen Webframeworks miteinander

– Feedback an den Nutzer

2.2.2 Scala im Web: Play Framework

Die Auswahl der Programmiersprache wurde vor allem von dem Gedanken geprägt, eine Sprache zu finden, die einerseits auf benötigte Bibliotheken zugreifen kann und andererseits eine kurze Einarbeitungszeit benötigt. Für die Arbeit mit Thumbnails hat sich die Bibliothek Apache Tika als besonders wertvoll erwiesen. Nähere Informationen zu dieser Bibliothek finden sich in [Brä13]. Da diese Bibliothek in Java geschrieben wurde, musste die Wahl auf eine Sprache fallen, welche in der Lage ist, Java-Bibliotheken anzusprechen.

Um die verschiedenen Sprachen und ihr jeweiliges Webframework zu evaluieren, wurde sich mit den in Tabelle 2.1 aufgelisteten Kombinationen näher beschäftigt. Dazu wurde von den Backend-Entwicklern jeweils eine kurze Hello-World-Anwendung aufgesetzt. Dadurch konnte festgestellt werden, ob der Arbeitsablauf, der Aufwand und der zu erzeugende Quellcode angemessen sind. Neben dieser praktischen Kurzevaluierung wurden einige Fakten zusammengetragen, um die Entscheidung zu erleichtern.

In die nähere Auswahl kamen Java, Groovy und Scala mit den jeweiligen Webframeworks Spring MVC[2], Grails[3] und Play[4]. Im Gegensatz zu Spring und Grails ist Play ein zustandsloses, schlankes Framework, was eine ideale Voraussetzung für ein REST Backend darstellt. Zusätzlich spielten die persönlichen Präferenzen der Entwickler eine Rolle, die bereits Erfahrung in der Kombination Scala und Play hatten.

[2] `http://www.springsource.org/features/modern-web`
[3] `http://grails.org`
[4] `http://playframework.com`

Scala ist eine Programmiersprache, deren Syntax stark an Java orientiert ist. Programme welche in Scala geschrieben sind, können bidirektional mit Java-Code interagieren und so auf den vollen Funktionsumfang der Java-Bibliotheken zurückgreifen. Dabei werden in der Sprache Konzepte der funktionalen mit gewohnten Elementen der objektorientierten Programmierung verknüpft. Scala eignet sich somit gut für einen allmählichen Einstieg in die funktionale Programmierung. Die Lernkurve hängt stark von den Vorkenntnissen in Java ab. Durch die starke Ähnlichkeit ist ein Umstieg einfach. Die Verwendung der hinzugekommenen, funktionalen Aspekte der Programmiersprache erfolgen nach und nach. Hier empfiehlt es sich, frei verfügbare Bücher, wie [Ode11], zu nutzen.

2.2.3 MongoDB als Datenbanksystem

Während der Prototypen-Phasen, in der sich die Idee der Umsetzung verfeinerte, wurde ein Datenmodell aufgestellt. Dieses Modell wird im späteren Verlauf der Arbeit näher beleuchtet. Ein wichtiger Aspekt, der für die Wahl der Datenbank entscheidend ist, sind die verschiedenen anbindbaren, externen Systeme. Das Datenmodell muss hier sicherstellen, dass Informationen, die aus allen Datenquellen verfügbar sind, einfach zugänglich sind. Gleichzeitig muss es dafür Sorge tragen, dass keine Informationen verloren gehen. Aus diesen Gründen fiel die Entscheidung auf eine NoSQL-Datenbank, welche die benötigte Flexibilität ohne großen Aufwand ermöglicht.

Neben der Open-Source Datenbank MongoDB[5] stand Apache CouchDB[6] als Alternative zur Verfügung. Die Wahl von MongoDB als persistenten Speicher für Project-Zoom fiel vorrangig aufgrund des sehr guten Datenbank-Treibers ReactiveMongo[7] für Scala. Dieser erlaubt komplett asynchrone Datenbankzugriffe und gliedert sich deshalb perfekt in das asynchrone Play Framework ein.

Als dokumentorientierter Datenspeicher passt MongoDB sehr gut zu einer REST-Architektur. Die Daten, welche in der Datenbank abgelegt werden, werden im Binary JSON (BSON)-Format gespeichert. Dieses BSON-Format ist eine binär encodierte Serialisierung von JSON-ähnlichen Dokumenten. BSON unterstützt die repräsentation aller Datentypen, welche auch von JSON unterstützt werden, und erlaubt zusätzlich weitere Datentypen. So wurden die JSON-Datentypen unter anderem um die Unterstützung von Binärdaten und Datumsangaben erweitert [Dir10]. Somit ähnelt die Datenform, die auf Clientseite verarbeitet wird, dem Format der Datenspeicherung in der Datenbank. Diese Konstellation erlaubt die Implementierung einer schlanken Schicht des Datenmodells, wie sie im Abschnitt Data-centric Design 3 erklärt wird.

[5] `http://mongodb.org`
[6] `http://couchdb.apache.org`
[7] `http://reactivemongo.org`

2.2.4 Verwendete Bibliotheken

Die umfangreichsten Bibliotheken, welche im Project-Zoom Backend Core verwendet werden, sind Akka[8], Play und SecureSocial[9]. Nicht hier aufgeführt ist ReactiveMongo als Datenbanktreiber. Diese Bibliothek wird im Abschnitt 3.1.2 näher erläutert.

Akka stellt die Grundlage für das Play Framework dar. Die Bibliothek ermöglicht die Nutzung verschiedenester Konzepte des asynchronen Programmierens. Das Ziel ist, das Schreiben von parallelen, fehlertoleranten und skalierbaren Anwendungen zu vereinfachen [Akk12].

Die sogenannten *Actors* der Bibliothek sind für dieses Projekt am relevantesten. Sie stellen eine Implementierung des Aktorenmodells dar. Aktoren sind abgeschlossene Einheiten, welche nur über Nachrichten kommunizieren. Dabei erfolgt die Abarbeitung der Nachrichten eines Aktors sequenziell, die Kommunikation mit anderen Aktoren hingegen asynchron. Dadurch können mehrere Nachrichten in unterschiedlichen Aktoren gleichzeitig abgearbeitet werden. Verschiedene Aktoren teilen sich nur über Nachrichten ausgetauschte Variablen. Damit das System also threadsicher arbeitet, müssen diese Nachrichten threadsicher sein. In Scala ist es deshalb üblich, für Nachrichten *Case Classes*[10] zu verwenden. Diese sind von vornherein unveränderbar und somit threadsicher.

Neben Actors finden auch *Agents* in Project-Zoom Verwendung. Ein Agent bildet eine Kapselung um einen Status. Dieser Status kann unverzüglich synchron gelesen und asynchron überschrieben werden. Bei einem Update wird dem Agent eine Funktion übergeben, welche den neuen Status des Agents berechnet. Einen Agent kann man zum Beispiel verwenden um Status zwischen verschiedenen Actors zu teilen.

Play Framework 2.0 ist die Weiterentwicklung und Portierung eines früheren Java Web Frameworks nach Scala. Es ist zwar in Scala programmiert, kann aber sowohl mit Java als auch Scala als Backend Sprache verwendet werden. Play liegt ein Model-View-Controller (MVC) Architektur zugrunde. Hierbei werden in jedem *Controller*[11] sogenannte *Actions* definiert und im *View* verschiedene *Templates* angelegt. Der Ablauf einer Anfrage an den Webserver verläuft wie folgt:

1. Der standartmäßige HTTP-Router leitet die Anfrage an eine Action weiter. Diese Weiterleitung basiert auf den Routen, die in der Datei conf/routes definiert sind.

[8] `http://akka.io`
[9] `http://securesocial.ws`
[10] `http://www.scala-lang.org/node/107`
[11] Aufgrund der allgemein üblichen Bezeichnung Model, View und Controller für die Teile einer MVC-Architekur werden diese anstatt der entsprechenden Begriffe Modell, Präsentation und Steuerung in dieser Arbeit verwendet.

```
GET /projects/:id controllers.ProjectController.read(id: String)
```

Eine Route besteht dabei immer aus der HTTP-Methode (GET, POST, HEAD, PUT, DELETE)[Pla13], einem URI-Pattern und einer Action, die den Request beantwortet.

2. Die Action im Controller ist für die Beantwortung zuständig. Dazu können Informationen im Model abgefragt werden. Die Templates können benutzt werden, um ein dynamisches Ergebnis für den Client zu erzeugen.

SecureSocial umfasst den User-Login. Dieses Paket ist ein Authentifizierungsmodul mit Support für OAuth, OAuth2, OpenID und Username/Passwort-Authentifizierung.

2.2.5 Modularisierung

Play erlaubt die Modularisierung des Codes in sogenannte Subprojekte. Ein solches Subprojekt ist dabei eine abgeschlossene Einheit, welche allein kompiliert, getestet und ausgeführt werden kann. Dabei können neben Play-Projekten auch Java- oder Scala-Projekte als Subprojekte verwendet werden. Die einzelnen Projekte und deren Abhängigkeiten werden in der *Build.scala*-Datei angegeben. Project-Zoom besteht aus drei verschiedenen Subprojekten:

common enthält den Quelltext der sowohl von den Projekten *main* als auch *admin* benötigt wird. In diesem Projekt sind die Datenmodelle definiert. Weiterhin befinden sich in diesem Modul das Eventsystem, die Erweiterungen zum Datenaggregieren und zum Thumbnailgenerieren sowid die Authentifizierung.

main schließt alle Controller ein, die für den normalen Nutzer ansprechbar sind. Es wurden verschiedenen Actions definiert und für die jeweiligen Sichten Templates angelegt. Der Frontend-Code, welcher näher in den Arbeiten [Rze13], [Her13] und [Die13] beschrieben ist, wird ebenfalls in diesem Projekt verwaltet.

admin definiert jede Interaktion, die nur für privilegierte Nutzer sichtbar sein soll. Dies sorgt für eine klare Trennung zwischen User- und Admin-Anfragen und gewährleistet mehr Sicherheit. Ein weiterer Vorteil ist, dass durch das Abschalten dieses Subprojektes jedwede Admin-Aktion unterbunden werden kann.

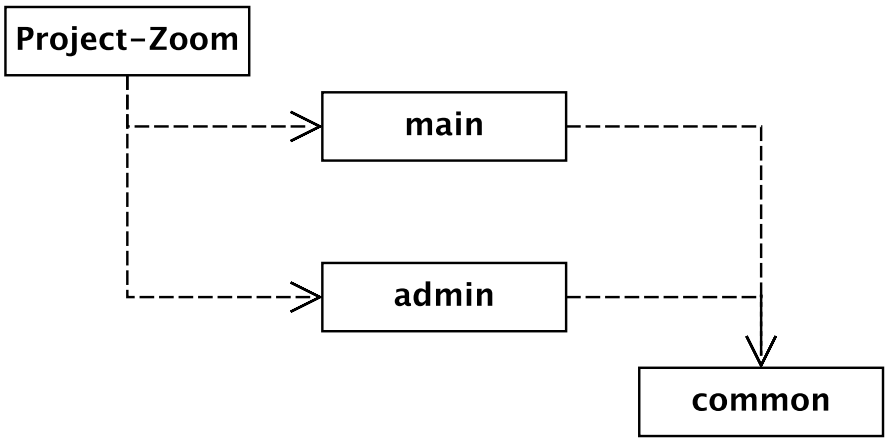

Abbildung 2.1 – Projekte und deren Abhängigkeiten

In der Grafik 2.1 sind die Abhängigkeiten der Pakte voneinander dargestellt. Das Anlegen eines Hauptprojektes, in diesem Fall Project-Zoom, erleichtert die Arbeit mit dem Gesamtprojekt. Durch die Abhängigkeit zu *main* und *admin* muss nur noch das Hauptprojekt kompiliert werden, denn die Abhängigkeiten werden bei Source-Code Änderungen automatisch mit kompiliert.

3 Data-centric Design

In diesem Kapitel wird die grundlegende Architektur der Anbindung eines persistenten Speichers im Core von Project-Zoom näher erläutert. Das gewählte Design hat vorrangig Auswirkungen auf die Art und Weise, wie der Code für Datenmodelle geschrieben wird, erstreckt sich aber als Betrachtungsweise über die gesamte Architektur. Zuerst wird das verwendete Data Centric Design näher beschrieben und mit alternativen etablierten Formen der Datenbankanbindung verglichen. Anschließend wird näher auf die Umsetzung des Designs in Project-Zoom eingegangen

3.1 Motivation

Das JSON Coast-To-Coast Design als Umsetzung des data-centric Designs wurde erstmals zusammenhängend und mit Beispielen unterlegt von Pascal Voitet dargestellt [Voi13]. Voitet ist selbst Mitwirkender am Open-Source-Projekt Play und Autor von Scala-Bibliotheken (*play-reactivemongo*[1], *play-autosource*[2]).

Ein Backend für eine Webapplikation ist zunehmend eine Verbindung zwischen verschiedenen anderen Backends und Frontends. Diese Entwicklung geht auf die Bereitstellung von APIs für viele große und kleine im Web verfügbare Dienste zurück. Anwendungen, welche Daten vorrangig aus anderen Quellen aggregieren, sind zum großen Teil mit der Datenmanipulation beschäftigt. Hier bietet sich ein data-centric Design an. Grundlegend für das Design sind dabei drei Entwicklungen: Zum einen das Aufstreben von NoSQL-Datenbanken und zum anderen asynchrone Datenbanktreiber mit einfachen Methoden zur JSON-Manipulation.

3.1.1 NoSQL

Die Entwicklung der Datenbankmanagementsysteme bestand viele Jahre lang in der Optimierung und Verbesserung bestehender relationaler Datenbankmodelle. Im Jahr 1998 kam der Term Not-only SQL (NoSQL) auf [LM10]. Heute gibt es verschiedene etablierte Datenbanken, die keine reinen SQL Datenbanken mehr sind, wie beispielsweise MongoDB, Apache CouchDB und Apache Cassandra[3].

[1] `https://github.com/zenexity/Play-ReactiveMongo`
[2] `https://github.com/mandubian/play-autosource`
[3] `http://cassandra.apache.org`

3.1.2 Datenbanktreiber

Für die Anbindung von relationalen Datenbanken gibt es in Java die Java Database Connectivity (JDBC). Diese Schnittstelle abstrahiert über Datenbanken und deren Treiber, indem eine einheitliche API angeboten wird. Ausgerichtet ist JDBC auf relationale Datenbanken [Ree00].

Um NoSQL-Datenbanken anzubinden, benötigt man, wie bei JDBC, einen eigenen Treiber. Der Unterschied ist, dass es hier keine Abstraktionsebene über verschiedene NoSQL-Datenbanken gibt. Dies liegt vorrangig an der sich stark unterscheidenden Struktur der einzelnen Speichersysteme. Die unterschiedlichen NoSQL-Datenbanken sind jeweils speziell auf eine bestimmte Aufgabe ausgerichtet, wie z.B. auf Durchsatz, verteilte Umgebungen oder flexible Datenschemas.

ReactiveMongo ist ein asynchroner Datenbanktreiber für MongoDB und die Programmiersprache Scala. Die Vorteile eines asynchronen Treibers liegen auf der Hand: Für jede synchrone Datenbankabfrage wird normalerweise ein Thread verwendet, der bis zur Antwort blockiert ist. Bei mehreren Datenbankabfragen pro Request werden bei Last viele Threads benötigt, um Datenbankabfragen auszuführen. Asynchrone Treiber umgehen dieses Problem, indem sie Threads, welche Datenbankabfragen ausführen, nicht blockieren. Für dieses Konzept ist es notwendig, Platzhalter einzuführen. Diese ersetzen das Ergebnis, solange es noch nicht vorhanden ist.

Quelltext 3.1 – Funktionssignatur für asynchronen Datenbankzugriff

```
def findOneById(bid: BSONObjectID): Future[Option[Graph]]
```

Im Quelltext 3.1 zeigt die Funktionssignatur den Rückgabetyp *Future[Option[Graph]]*. *Future* ist hierbei ebendieser Platzhalter für ein Ergebnis, welches noch nicht existiert. Um auf das Ergebnis eines *Futures* zu reagieren, können verschiedene *Callbacks* festgelegt werden.

3.1.3 Manipulation von JSON

Das data-centric Design basiert auf der Idee der direkten Manipulation von Daten. Für eine Umsetzung des data-centric Designs mit Hilfe von JSON benötigt man somit eine Bibliothek, welche die Möglichkeit bietet, JSON zu transformieren und zu validieren.

Eine solche Bibliothek, welche das Validieren, Transformieren und Serialisieren erlaubt, ist Play-JSON. Ein Beispiel, wie solch eine Transformation und Validierung aussieht, ist im Anhang im Quelltextauszug A.1 zu sehen. Dort wird ein JSON-Objekt erzeugt, welches anschließend zuerst transformiert wird, indem das Attribut *role* angehangen wird, und danach validiert wird. Bei der Validierung wird überprüft, ob der Nutzer über 18 Jahre alt ist.

3.2 Idee hinter data-centric Design

Das Ziel im data-centric Design ist die direkte Manipulation von Daten. Der Fokus des Designs liegt dabei sowohl auf der Datenmanipulation als auch auf dem Datenfluss. Wie bereits im Abschnitt 3.1 beschrieben, sind Webapplikation Backends immer Häufiger Knotenpunkte in einem mehrere Server durchlaufenden Datenfluss. Ein Teil dieses Datenflusses, die Kommunikation des Backends mit der Datenbank und dem Client, ist in Abbildung 3.1 dargestellt.

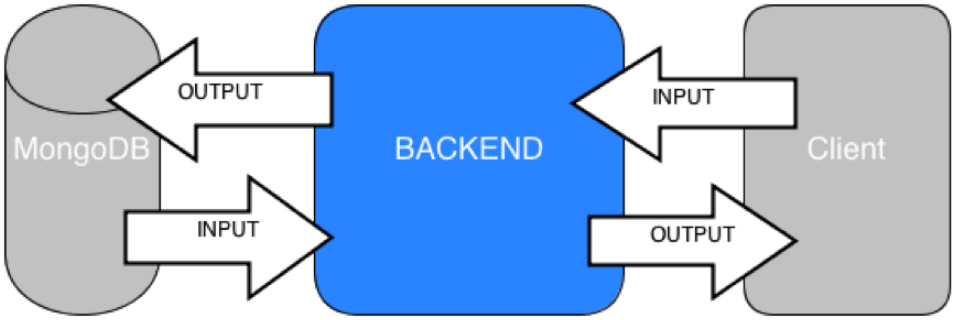

Abbildung 3.1 – Datenfluss zwischen Datenbank, Backend und Client

In diesem Datenfluss soll es keine implizite Umwandlung der Daten in Objekte der Programmiersprache[4] geben. Für ein einfaches Durchleiten der Daten aus der Datenbank zum Client oder von einer anderen Webressource zum Client ist eine Umwandlung nicht notwendig oder sinnvoll.

Im Gegensatz zu anderen Modellen ist die Abstraktion von der Datenquelle kein Ziel des data-centric Designs. Das Festlegen auf eine Datenquelle bei der Entwicklung eines Systems erlaubt die Verwendung aller Funktionen, die diese Datenquelle zur Verfügung stellt. Beispielsweise erlaubt MongoDB die Verwendung sogenannter *Capped Collections*[5], welche auf hohen Durchsatz optimiert sind. Diese verhalten sich wie eine zirkuläre Queue mit begrenzter Größe. Viele Datenbanken besitzen spezielle, stark angepasste Funktionen, auf die jedoch bei der Verwendung von allgemeinen Datenbank APIs (z.B. JDBC) meist nicht zugegriffen werden kann. Ein Nachteil dieser engen Kopplung ist, dass der Austausch der Datenquelle dadurch schwieriger ist.

Eine Deserialisierung der Daten in Objekte ist für komplexe Business-Logik von Vorteil. Statische Datenmodelle sollten immer dann verwendet werden, wenn die Manipulation der Daten kompliziert wird oder wenn die Verwendung externer Bibliotheken eine Objektrepräsentation benötigt.

[4] Es ist bekannt, dass die Repräsentation der Daten des Datenflusses ebenfalls in Objekten der Programmiersprache erfolgt (beispielsweise JSON-Objekte). Mit dem Ausdruck ist die Überführung der Daten aus einem allgemeinen Modell (z.B. JSON) in eine spezifische Objektdarstellung (z.B. ein Objekt einer User-Klasse) gemeint.

[5] Mehr Informationen zu Capped Collections z.B. auf `http://docs.mongodb.org/manual/core/capped-collections/`

Nach Voitet ist die Frage also nicht, statische Datenmodelle zu vergessen, sondern sie nur dann zu benutzen, wenn sie notwendig sind. Einfache und dynamische Strukturen sollten so oft es geht erhalten bleiben [Voi13].

3.3 Abgrenzung zur objektrelationalen Abbildung

Das *Object-relational Mapping* (ORM) bezeichnet man auch als *All-Model-Approach*. Hierbei erfolgt die Kommunikation mit der Datenbankschnittstelle über Objekte der Programmiersprache. Abfragen an die Datenbank resultieren in Objekten der Programmiersprache. Der Fokus der objektrelationalen Abbildung liegt in der Überführung von Objekten der objektorientierten Programmierung in ein relationales, tabellenorientiertes Schema [Amb03].

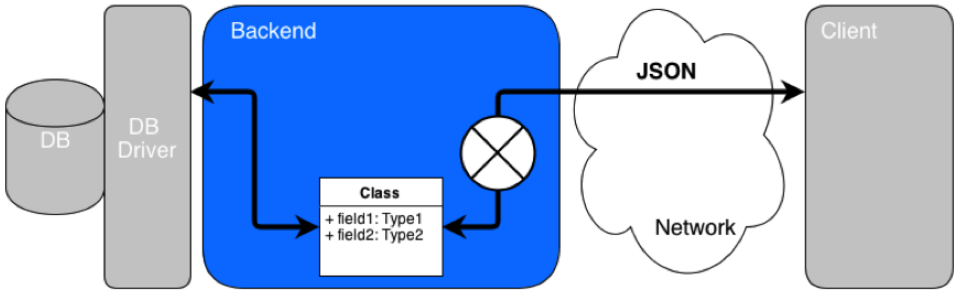

Abbildung 3.2 – Datenfluss einer Applikation bei Verwendung eines ORM Frameworks

Der wichtigste Vorteil eines Frameworks für die objektrelationale Abbildung liegt in der Abstraktion von der Datenbank. Für die Business-Logik gibt es keinen Unterschied zwischen Objekten aus der Datenbank und Objekten der Programmiersprache. Die Datenbank kann meist ohne Änderungen an der Applikation ausgetauscht werden.

3.3.1 Objektrelationale Unverträglichkeit

Ein ORM bringt konzeptionelle Probleme mit sich. Diese werden in der Literatur in der Regel als *Object-relational Impedance Mismatch* (ORIM) bezeichnet. Die objektrelationale Unverträglichkeit resultiert aus den Unterschieden im zugrundeliegenden Konzept, sowie aus Differenzen in der Darstellung und den Erwartungen des Entwicklers [Bow10].

Folgende Herausforderungen ergeben sich für die praktische Arbeit mit einem ORM:

- Die Abbildung der Relationen zwischen den Objekten ist komplex. Es müssen One-to-One, One-to-Many und Many-to-Many Abhängigkeiten ins relationale Schema übertragen werden.

- Ein weiterer zentraler Punkt ist die Abbildung von Hierarchien und Vererbungen aus der objektorientierten Programmierung in das physische Datenmodell.
- Innerhalb des ORM muss es ein Objekt-Caching geben. Dieses ist notwendig um die Illusion zu erzeugen, dass es sich um ein Objekt der Programmiersprache handelt. Denn wird ein Objekt mehrmals aus der Datenbank abgefragt, so muss ein referenzgleiches Objekt zurückgegeben werden [HVE03].
- Der Entwickler muss die ORM-Limitierungen, die sich aus dem ORIM ergeben, akzeptieren. Die Probleme müssen somit bekannt sein, um sie bereits bei der Datenmodellierung zu umgehen [Atw06].

Ted Neward, Autor von „The Busy Java Developers Guide to Scala“, macht den frühen Erfolg von objektrelationalen Abbildungen und die Erleichterungen, welche die ersten ORMs mit sich brachten, dafür verantwortlich, dass zum Teil ORMs verwendet werden, ohne deren Limitationen zu beachten. Das wiederum führt zu Mehraufwand an Zeit und Energie für die Erhaltung und Anpassung an alle Nutzungsfälle [Atw06].

3.3.2 Unterschiede zum data-centric Design

Das data-centric Design ist viel stärker an die Datenbank gebunden und deren Austausch ist deutlich schwieriger. Dies sorgt dafür, dass keine Illusion einer abstrahierten Datenbank entsteht. Da das data-centric Design ein Ansatz mit funktionalem Hintergrund ist, sind die Datenstrukturen in der Regel unveränderbar. Ein Caching zur Sicherung der Objektidendität ist daher nicht notwendig. Die Objektgleichheit ist in diesem Fall über Wertgleichheit definiert (vgl. Scala Case Klassen [Sca08]). Im Gegensatz zum ORM muss der Entwickler beim data-centric Design selbst um die Darstellung und Umwandlung seiner Daten in das Datenbankformat kümmern. Durch die Verwendung einer objekt- oder dokumentenbasierten Datenbank ist diese Umwandlung einfacher und mit weniger Problemen behaftet, als eine Konvertierung in ein relationales Schema.

3.4 Abgrenzung zu Datenzugriffsobjekten

Implementierungen von Datenzugriffsobjekten, in der englischsprachigen Literatur *Data Access Object* (DAO) oder *Data Access Procedure*, basieren auf dem *Data Access Object* Pattern [ACM03]. Das Pattern hat das Ziel, den Datenzugriff und die Datenmanipulation in einen separaten Layer auszulagern. Daraus ergibt sich eine Kapselung der Datenbankzugriffe an einem Ort der Applikation. Bei der Umsetzung gibt es die Möglichkeit, ein DAO zur Abstraktion der Datenbank als Ganzes zu verwenden oder für jede einzelne Datenbankstruktur ein DAO anzulegen.

Der Hauptfokus liegt in der Abstrahierung der Datenbank. Bei Änderungen an der Datenquelle müssen in der Regel nur die Datenzugriffsobjekte angepasst werden, denn der Anwendungscode bleibt in der Regel unberührt. Die Serialisierung der Daten von den Strukturen der Anwendung in Strukturen der Datenbank ist nicht Aufgabe des DAO, sondern des *Data Transfer Object* (DTO) [6].

Im DAO findet über einen Datenbanktreiber direkter Zugriff auf die Datenbank statt. Man kann das Datenzugriffsobjekt somit als Bindeglied zwischen Datenbank und Applikation sehen. Durch die direkte Kommunikation mit der Datenquelle können alle unterstützten Funktionen ausgenutzt werden.

Sowohl DAO als auch ORM haben die Abstrahierung der Datenbank als Ziel. Im Gegensatz zum DAO geht das ORM jedoch noch einen Schritt weiter und abstrahiert nicht nur über die Funktionen, welche durch die Datenbank zur Verfügung gestellt werden, sondern auch über die Innere Struktur der Datenquelle. In der Regel bauen ORM-Implementierungen auf dem Data Access Object Pattern und dem Data Transfer Object Pattern auf.

3.4.1 Unterschiede zum data-centric Design

Das DAO-Pattern enthält, ähnlich dem data-centric Design, keine Serialisierung oder Deserialisierung. Beide Designs können sehr gut zusammen verwendet werden. Der Datenfluss kann so durch ein DAO realisiert werden. Dadurch wird eine deutlich stärkere Kapselung erreicht. Voitet benutzt in seinen Beispielen kein DAO, sodass sich die Datenbankabfragen über den kompletten Quellcode verteilen. Während das bei Beispielapplikationen oder kleinen Anwendungen kein Problem darstellt, geht bei größeren Anwendungen mit vielen verschiedenen Datenmodellen schnell die Übersichtlichkeit verloren. Die Verwendung eines Datenzugriffsobjektes führt zu einem *Single point of responsibility*[7] und stärkt somit die Struktur des gesamten Projektes.

3.5 Grenzen des data-centric Designs

Das data-centric Design eignet sich sehr gut für *Create-Read-Update-Delete* (CRUD)-Anwendungen. Dort liegt der Fokus auf der Datenmanipulation, was zum Grundsatz des Designs passt.

Es erfolgt eine explizite Bindung an das Datenmodell. Durch die Nähe zur Datenquelle können spezifische Fähigkeiten einer Datenquelle genutzt und Anfragen effektiver gestaltet werden. Die dynamischen Strukturen des Datenmodells ermöglichen eine leichte Veränderbarkeit des Modells sowie einfache Verknüpfungen zwischen Daten.

[6] vgl. J2EE Patterns – Data Transfer Object `http://www.oracle.com/technetwork/java/transferobject-139757.html`

[7] Single responsibility prinziple (SRP), vgl. [FFBS04, p. 339]

Grenzen sind dem data-centric Design vor allem durch die Business-Logik gesetzt. Durch die dynamischen Strukturen im Design ist der Code meist komplizierter und länger.

Ohne die Verwendung eines DAO gibt es in diesem Design keinen Singe point of responsibility für Datenbankzugriffe. Das erschwert wiederum Änderungen am Schema, da diese zu unvorhergesehenen Problemen an anderen Stellen im Anwendungsquelltext führen können. Aus diesem Grund sollten bei diesem Design ein oder mehrere Datenzugriffsobjekte verwendet werden.

3.6 Umsetzung in Project-Zoom

Die REST-Schnittstelle von Project-Zoom entspricht die einer klassischen CRUD-Anwendung. Aufgrund der geringen Business-Logik liegt es nahe, das data-centric Design zu verwenden. Um die Wartung des Systems zu erleichtern, werden Datenzugriffsobjekte für jedes Datenmodell verwendet.

Quelltext 3.2 – Macro Verwendung zur Generierung einer JSON Konvertierung

```
case class User(firstName: String, lastName: String)

val userFormat = Json.format[User]
```

Für einige Teile der Anwendung ist es notwendig die JSON-Strukturen der Datenbank in Scala-Objekte umzuwandeln. Dies ist zum Beispiel für User Objekte der Fall, welche von der Authentifizierungsbibliothek SecureSocial verwendet werden. Play erlaubt die Generierung der Transformationsfunktionen mit Hilfe von Macros[8]. Das bedeutet, dass die Anwendung die Datenklassen des Modells definiert und die Funktionen zur Konvertierung von und nach JSON während der Kompilierung vom Framework übernommen werden. Ein Beispiel für solch eine Transformation ist in 3.2 mit der Variablen *userFormat* gegeben. Diese ist in der Lage, zwischen den beispielhaften Repräsentationen

```
{
  "firstName" : "Max",
  "lastName"  : "Mustermann"
}
```

und

```
User("Max", "Mustermann")
```

[8] JSON Macro incepetion `http://www.playframework.com/documentation/2.1.1/ScalaJsonInception`

zu konvertieren. Diese Bedarfskonvertierung ist durch die Verwendung von unveränderbaren Case-Klassen typsicher und entspricht den Konzepten der funktionalen Programmierung [For11].

Die CRUD-Operationen *list* und *read* wurden implementiert, ohne Datenbankobjekte zu konvertieren. Die Daten werden aus der Datenbank gelesen, transformiert und anschließend an den anfragenden Clienten zurückgegeben. Bei der Transformation werden z.B. Passwort-Hashes und Login-Informationen, die sich im User-Objekt befinden, entfernt. Die Funktionen *create*, *update* und *delete* sind nicht für alle Datenbankobjekte implementiert. Dies liegt an der Anforderung, dass die Datenhaltung in der von der D-School bereits verwendeten Datenbank geschehen soll. Project-Zoom soll diese Daten nur aggregieren.

4 Datenmodellierung

In diesem Abschnitt sollen die Besonderheiten der in Project-Zoom verwendeten Datenmodelle erläutert werden. An ausgewählten Modellen werden die Besonderheiten der Umsetzung erklärt. Ein Teil dieses Kapitels beschäftigt sich ferner mit dem Zugriffsschutz für Objekte in der Datenbank.

4.1 Überblick

In der Abbildung 4.1 ist ein Auszug aus dem Datenmodell für Project-Zoom gegeben. Gezeigt sind die verschiedenen Daten Klassen und ihre Assoziationen.

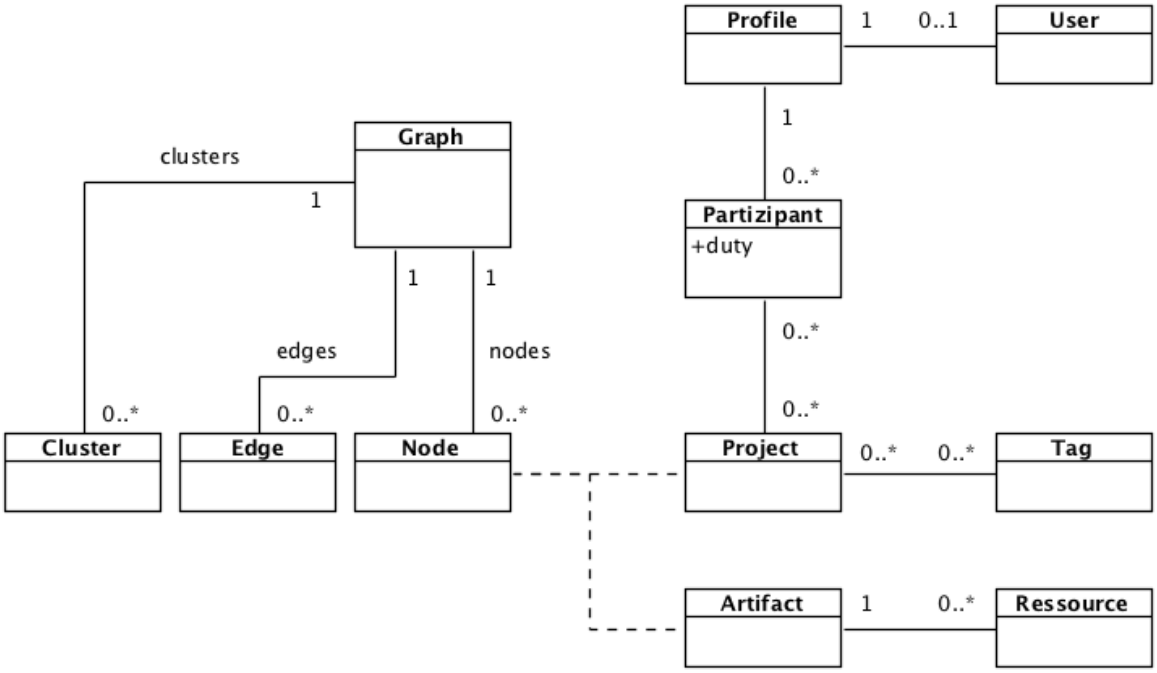

Abbildung 4.1 – Ausgewählte Datenmodelle und ihre Assoziationen

4.2 Daten Modell für Graphen

Ein Graph wurde als gerichteter Graph mit jeweils einer Liste für Knoten, Kanten und Cluster modelliert. Knoten und Kanten entsprechen denen aus der Graphentheorie bekannten Struktu-

ren[1]. Cluster wurden eingeführt, um Nutzern die Möglichkeit zu geben, Knoten zu gruppieren und *Tags* hinzuzufügen.

In der Datenbank sind für den *Payload* eines Knotens nur Referenzen abgelegt. Der Payload beschreibt dabei den eigentlichen Inhalt eines Knotens, z.B. ein Projekt oder ein Artefakt. Wenn der Graph über die REST-Schnittstelle ausgeliefert wird, dann wird diese Payload-Referenz mit dem eigentlichen Inhalt des Knotens ersetzt. Dies verringert den Speicherbedarf des Graphs und verhindert das mehrfache Speichern des Payloads in der Datenbank.

4.2.1 Versionierung

Um die Veränderungen an einem Graph nachverfolgen zu können, wurde eine Versionierung für Graphen eingeführt. Dadurch, dass nur Referenzen auf den Inhalt eines Knotens gespeichert werden, ist die Größe eines Graphens vergleichsweise gering. Aus diesem Grund ist es möglich, alle Versionen eines Graphens in der Datenbank abzulegen. Zur Identifizierung bekommt ein Graph eine Gruppe (*group*) zugewiesen. In einer Gruppe liegen die verschiedenen Versionen eines Graphen. Zusammen mit der Versionsnummer *version*, welche bei jedem Update inkrementiert wird, bildet die Gruppe einen Schlüssel.

4.2.2 Updaten eines Graphen

Um ein Graphen-Update ausführen zu können, wurde die JSON Patch-Spezifikation[2] zur Veränderung von JSON-Objekten implementiert. Diese definiert eine JSON-Struktur, um ein gegebenes Objekt durch die Anwendung einer geordneten Folge von Änderungen in einen neuen Zustand zu überführen. Das Patch-Objekt ist ein JSON-Array, dessen Elemente als Operationen auf das zu verändernde JSON-Objekt angewendet werden sollen. Die definierten Operationen sind das Hinzufügen, Testen, Entfernen, Ersetzen, Verschieben und Kopieren von Attributen.

Alternativ bestünde die Möglichkeit, bei jeder Änderung den kompletten, geänderten Graphen zu senden. Da es sich hier um kleine Objekte handelt, wäre der Mehrbedarf an Bandbreite vertretbar. Für den Mehrnutzerbetrieb ist allerdings die Verwendung von JSON-Patch vorteilhaft, denn durch das explizite Senden der Änderungen ist ein Zusammenführen gleichzeitiger Änderungen von unterschiedlichen Quellen einfacher. Im Hinblick auf ebendiesen Vorteil wurde in Project-Zoom JSON-Patch verwendet.

[1] Als konzeptionelle Referenz dient [CLRS01, p. 531]
[2] JavaScript Object Notation Patch `http://tools.ietf.org/html/rfc6902`

4.3 Datenmodell für Artefakte

Artefakte werden durch externe Konnektoren gefunden und anschließend durch den Backend-Core in der Datenbank abgelegt, wo alle Informationen enthalten sind, die charakteristisch für das jeweilige Artefakt sind. Eine Besonderheit zeigt sich im Feld *metadata*. Hier können alle Informationen eines Connetors gespeichert werden, die keinem allgemeinen Feld zugeordnet werden können. Durch diese Speicherung von Informationen, die zwar im Moment für die Anwendung irrelevant sind und die nicht von allen Connectoren geliefert werden können, vermeidet man einen Informationsverlust. Bei der Weiterentwicklung der Anwendung können dann Informationen aus diesen Metadaten extrahiert und als eigenständige Felder verwendet werden.

Zu einem Artefakt sind verschiedene *Ressourcen* abgelegt. Eine Ressource bezieht sich hierbei auf die physikalische Repräsentation eines Artefakts. Das heißt, dass es zu jedem Ressourcenobjekt in der Datenbank eine zugehörige Datei im Dateisystem gibt. Beispielsweise ist die Originale Datei eine Ressource, generierte Thumbnails sind weitere. Das Feld *typ* beschreibt die Art der Ressource und erlaubt einem Client die differenzierte Behandlung unterschiedlicher Typen.

4.4 Datenmodell für Nutzer

Neben Artefakten können durch einen externen Connector auch Profile gesammelt werden. Diese Profile bestimmen die Zugriffsrechte eines Nutzers. Der mit einem Profil verknüpfte User enthält die Anmeldeinformationen, welche für das Authentifizierungsmodul SecureSocial benötigt werden. Ein User wird erst dann an ein Profil angehangen, wenn sich ein Nutzer mit der im Profil angegebenen E-Mail-Adresse für Project-Zoom registriert. Aus diesem Grund ist es auch nur Nutzern erlaubt sich zu registrieren, für die bereits ein Profil existiert.

4.5 Zugriffsschutz für Datenbankobjekte

Eine Anforderung der D-School bestand im Schutz der Daten. Dabei soll verhindert werden, dass Informationen aus Projekten für Personen, die kein Geheimhaltungsvertrag (NDA) für dieses Projekt unterschrieben haben, zugänglich sind. Studierende haben folglich nur Lesezugriff auf Projekte, an denen sie selbst teilgenommen haben oder die veröffentlicht[3] wurden. Schreibzugriff erhält ein Student nur als in der Filemaker-Datenbank vermerkter Teilnehmer des Projektes.

[3] Aufgrund der fehlenden Einstellungsmöglichkeiten in der Filemaker Datenbank gibt es in der aktuellen Implementierung nicht die Möglichkeit ein Projekt als öffentlich zu kennzeichnen.

Bei der Betrachtung der Struktur von Zugriffsbeschränkungen fällt auf, dass diese abhängig von der zugriffssuchenden Person, der Art des Zugriffes und der Ressource sind. Basierend auf dieser Erkenntnis wurde ein sogenannter *DBAccessContext* eingeführt. Für jeden Datenbankzugriff wird ein solcher Kontext benötigt. Mit Hilfe dessen können dann auf unterster Zugriffsebene, in den Funktionen *insert*, *update*, *remove* und *find*, die Berechtigungen überprüft werden.

Das Datenzugriffsobjekt für das jeweilige Datenmodell kann dann festlegen, wie mit Hilfe des DBAccessContext der Zugriff auf diese Methoden reguliert wird. Jedes dieser Datenzugriffsobjekte erbt von *SecuredDAO*. In dieser Klasse findet die Ausführung der Datenbank Funktionen und die Zugriffsbeschränkung statt. Dadurch, dass die Zugriffsbeschränkungen für die jeweiligen Datenbankfunktionen einzeln festgelegt werden können, ist eine sehr feingranulare Differenzierung möglich.

Die Klasse *SecuredDAO* überprüft die Berechtigungen wie folgt:

insert Beim Hinzufügen wird direkt kontrolliert ob der User die Berechtigung hat, neue Elemente in das jeweilige Datenmodell einzufügen. Fehlt diese, wird die Aktion mit einem Fehler abgebrochen

update Ein Update Aktion benötigt immer eine Suchanfrage und das eigentliche Update was auf die gefundenen Ergebnisse angewandt wird. Soll ein User nicht alle Objekte updaten können, so wird an die Suchanfrage die jeweilige Einschränkung angehangen. Damit wird verhindert, dass Objekte gefunden werden, auf die der User keinen Zugriff hat. Objekte die nicht gefunden werden, können demnach auch nicht verändert werden.

find, remove Für das Abfragen und Löschen ist eine Suchanfrage notwendig. Diese wird genauso verändert wie dies für die *update*-Funktion beschrieben ist. Der Ergebnisraum wird somit schon während der Abfrage auf die dem User zugänglichen Objekte beschränkt.

5 Anbindung externer Komponenten

In diesem Kapitel wird zunächst der asynchrone Aufbau des Projektes näher erläutert. Im Anschluss daran wird die Anbindung von Erweiterungen an Project-Zoom betrachtet.

5.1 Eventsystem

Ein Eventsystem basiert auf dem Prinzip des Abonnierens und Veröffentlichens von Nachrichten und modelliert eine Eine-zu-Viele-Relation. Ein sogenannter *Publisher* sendet seine Events, welche anschließend vom *Dispatcher* an die *Subscriber* verteilt werden. Die Subscriber können dieses Event schließlich abarbeiten. Dabei entsteht eine Asynchronität, da der Sender beim Emittieren einer Nachricht sofort sequentiell weiterarbeiten kann und die Bearbeitung des Events nicht abwarten muss.

Die parallele Abarbeitung von verschiedenen Aufgaben ist ein wichtiger Bestandteil von Project-Zoom. Ein Beispiel hierfür ist das Erzeugen von Thumbnails, was bei größeren Dateien eine gewisse Zeit in Anspruch nehmen kann. Diese Thumbnails werden für jedes durch einen Konnektor gefundene Artefakt erzeugt. Während der Thumbnailerzeugung muss die Webapplikation weiterhin Client-Anfragen beantworten können, weswegen die Thumbnailgenerierung ausgelagert werden muss.

Ebenfalls von Bedeutung ist die Ausfallsicherheit. Der Ausfall einer einzelnen Komponente darf die Stabilität des Gesamtsystems nicht beeinträchtigen. Eine falsch konfigurierte OpenOffice-Anbindung, welche der Thumbnailgenerator benötigt, darf selbst wenn der Thumnailerzeuger nicht mehr in der Lage ist zu arbeiten, das System nicht zum Stillstand bringen.

Ein weiterer Vorteil ist die Flexibilität, die durch ein Publisher-Subscriber-Modell erreicht wird. Wenn eine neue Komponente eingebunden werden soll, so kann sich diese auf bereits emittierte Nachrichten abonnieren und somit auf Ereignisse im Gesamtsystem reagieren.

5.1.1 Umsetzung in Project-Zoom

Die Implementierung in Project-Zoom basiert auf dem Akka-Framework. Die in Absatz 2.2.4 beschriebenen Aktoren eignen sich sehr gut zur Modellierung eines Subscribers. Ein Aktor ist eine abgeschlossene funktionelle Einheit, deren Kommunikation nur über unveränderbare

Nachrichten erfolgt. Jeder Aktor muss eine RECEIVE-Funktion definieren. In dieser Funktion beginnt die Abarbeitung einer Nachricht aus der Mailbox des Aktors.

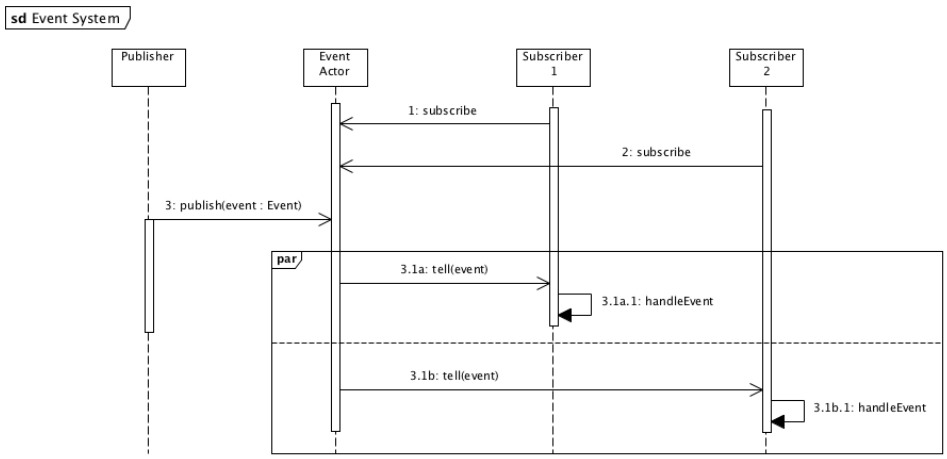

Abbildung 5.1 – Funktionsweise des Eventsystems an einem Beispiel

In Abbildung 5.1 ist der Aufbau des Eventsystems zu sehen. Der Event-Aktor übernimmt die Rolle des Dispatchers und verwaltet die Abonnements. Durch das Starten mehrerer Instanzen des Event-Aktors wird eine Lastverteilung erreicht.

In der Umsetzung erfolgt das Abonnieren einer Nachricht durch das Senden einer partiellen Funktion[1] an den Dispatcher. Diese Funktion bildet den Typ *Event* auf Unit ab und ist auf allen Events definiert, welche der Subscriber empfangen will. Der Dispatcher benutzt anschließend alle partiellen Funktionen, um eine eingehende Nachricht zu verteilen. Standartmäßig hat ein Subscriber alle Nachrichten abonniert, für die seine RECEIVE-Funktion definiert ist.

5.2 Anbindung der Konnektoren und des Thumbnailgenerators

[1]Eine partielle Funktion $f : A \to B$ ist im Gegensatz zu einer totalen Funktion nicht auf jedem Wert aus A definiert. Für die Anwendung in Scala siehe [Bru11])

6 Zusammenfassung und Ausblick

Literaturverzeichnis

[ACM03] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall / Sun Microsystems Press, USA, California, June 2003.

[Akk12] Akka. What is akka? Webseite, 2012. Erreichbar unter: `http://doc.akka.io/docs/akka/2.0/intro/what-is-akka.html`; besucht am 19. Juni 2013.

[Amb03] Scott W. Ambler. Mapping objects to relational databases. Webseite, 2003. Erreichbar unter: `http://www.agiledata.org/essays/mappingObjects.html`; besucht am 15. Mai 2013.

[Atw06] Jeff Atwood. Object-relational mapping is the vietnam of computer science. Webseite, 2006. Erreichbar unter: `http://www.codinghorror.com/blog/2006/06/object-relational-mapping-is-the-vietnam-of-computer-science.html`; besucht am 10. Juni 2013.

[BBD+13] Dominic Bräunlein, Tom Bocklisch, Anita Diekhoff, Norman Rzepka, Tom Herold, and Thomas Werkmeister. Software requirements specification. 2013.

[Bow10] David Bowers. Object relational database design – impedance mismatch or expectation mismatch? Webseite, 2010. Erreichbar unter: `http://www.dmsg.bcs.org/web/images/stories/PDFs/2010-10-13-david-bowers.pdf`; besucht am 23. Juni 2013.

[Brä13] Dominic Bräunlein. Generierung und bereitstellung von semantischen thumbnails aus heterogenen daten für project-zoom, 2013.

[Bru11] Erik Bruchez. Scala partial functions (without a phd). Webseite, 2011. Erreichbar unter: `http://blog.bruchez.name/2011/10/scala-partial-functions-without-phd.html`; besucht am 10. Juni 2013.

[CLRS01] Th. H. Corman, Ch. E. Leiserson, R. Rivest, and C. Stein. *Algorithmen - Eine Einführung*. MIT Press, 2001.

[Die13] Anita Diekhoff. Automatisches layouten in interaktiven graphen, 2013.

[Dir10] Mike Dirolf. Bson. Webseite, 2010. Erreichbar unter: `http://bsonspec.org`; besucht am 19. Juni 2013.

[FFBS04] Eric Freeman, Elisabeth Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly, Sebastopol, CA, October 2004.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, 2000.

[For11] Neal Ford. Functional thinking: Immutability - make java code more functional by changing less. Technical report, IBM Corporation, July 2011.

[Her13] Tom Herold. Kontextsensitver assistent für interaktive graphen, 2013.

[HVE03] Anders Hejlsberg, Bill Venners, and Bruce Eckel. Inappropriate abstractions - a conversation with anders hejlsberg, part vi. Webseite, 2003. Erreichbar unter: `http://www.artima.com/intv/abstract3.html`; besucht am 10. Juni 2013.

[LM10] Adam Lith and Jakob Mattsson. Investigating storage solutions for large data. Master's thesis, Chalmers University Of Technology, 2010.

[Ode11] Martin Odersky. *Scala By Example*. EPFL, Switzerland, May 2011.

[Pla13] Play. Http routing. Webseite, 2013. Erreichbar unter: `http://www.playframework.com/documentation/2.1.1/ScalaRouting`; besucht am 19. Juni 2013.

[PMW09] H. Plattner, C. Meinel, and U. Weinberg. *Design Thinking: Innovation lernen, Ideenwelten öffnen*. Mi-Wirtschaftsbuch, FinanzBuch-Verlag, 2009.

[Ree00] George Reese. *Database Programming with JDBC and JAVA*. O'Reilly Media, Inc., 2000.

[Rze13] Norman Rzepka. The design of an web-based event-driven client application architecture for project-zoom, 2013.

[Sca08] Scala. A tour of scala: Case classes. Webseite, 2008. Erreichbar unter: `http://www.scala-lang.org/node/107`; besucht am 17. Juni 2013.

[Voi13] Pascal Voitet. Json coast-to-coast. Webseite, 2013. Erreichbar unter: `http://mandubian.com/2013/01/13/JSON-Coast-to-Coast/`; besucht am 21. März 2013.

[Wer13] Thomas Werkmeister. Extensible backend plugin architecture for data aggregation of heterogeneous sources for project-zoom, 2013.

Glossar

ADVANCED-TRACK Cloud storage and collaboration solution for enterprises. 1, 2

BASIC-TRACK Cloud storage and collaboration solution for enterprises. 1

Box Cloud storage and collaboration solution for enterprises. 1, 4

Anhang A

Quelltext Beispiele

Quelltext A.1 – Beispiel für Play JSON -Transformation bzw. -Validierung

```
val json = Json.obj(
  "firstName" -> "max", "latsName" -> "Mustermann", "age" -> 17)

val addUserRole = JsPath.json.update(
  (JsPath \ "role").json.put(JsString("User")))
//> addUserRole : Reads[JsObject]

val isMajor =
  (JsPath \ "age").read[Int](min(18))
//> isMajor : Reads[Int]

json.transform(addUserRole)
//> res0: JsResult[JsObject] =
//     JsSuccess({
//       "firstName":"max",
//       "latsName":"Mustermann",
//       "age":17,
//       "role":"User"},)

json.validate(isMajor)
//> res1: JsResult[Int] =
//     JsError(List((
//       /age,List(
//         ValidationError(validate.error.min,WrappedArray(18))))))
```

Anhang B

Messdaten zur Evaluation

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und dafür keine anderen als die genannten Quellen und Hilfsmittel verwendet habe.

Tom Bocklisch

Potsdam 24. Juni 2013