Konzeption und Implementierung eines Gamification Services mit Ruby

Reinhard Buchinger

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interaktive Medien

in Hagenberg

im Dezember 2012

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 3. Dezember 2012

Reinhard Buchinger

Inhaltsverzeichnis

Kurzfassung

Gamification, die Nutzung von Spieledesigntechniken in einem Kontext außerhalb von Spielen, wurde in den letzten Jahren zu einem weitreichenden Thema in der Wirtschaft und der Forschung. Die Auswirkungen von *Gamification* wurden bereits ausführlich wissenschaftlich betrachtet, jedoch gab es bisher keine bekannten Forschungsarbeiten im Bereich der technischen Umsetzung dieser *Gamification* Techniken.

In dieser Arbeit wird beschrieben, wie ein *Gamification Service* mit der Programmiersprache *Ruby* konzipiert und implementiert werden kann. Dieser Ansatz ermöglicht die Integration und Umsetzung von *Gamification* Elementen wie *Erfolge* und *Rekorde* in eine bestehende Systemarchitektur, wie es an dem Anwendungsbeispiel von *runtastic* dargestellt wird. Bei dieser Integration wird eine bisherige Schnittstelle zu mobilen Endgeräten adaptiert, damit zusätzlich zu bestehenden Daten auch Informationen für *Gamification* ausgetauscht werden können.

Das System teilt sich in die Komponenten *Core*, *Service* und *Framework* auf. Der *Core* enthält alle Komponenten, die sowohl vom *Service* als auch vom *Framework* verwendet werden. Das *Service* stellt eine Schnittstelle zur Verfügung, die von anderen Systemen abgerufen werden kann. Anfragen an die Schnittstelle werden vom *Service* auf Abhängigkeiten analysiert, diese in Aufgaben aufgeteilt und dann so weit wie möglich parallelisiert an das *Framework* zur Verarbeitung weitergegeben. Innerhalb des *Frameworks* werden die zu den Aufgaben gehörigen Inhalte evaluiert, berechnet, gespeichert und die Ergebnisse über das *Service* an die Schnittstelle zurückgegeben. Inhalte können auch mithilfe einer dafür entwickelten *domänenspezifischen Sprache* definiert werden, wobei diese Sprache durch Erweiterungen mit zusätzlicher Funktionalität ausgestattet werden kann.

Das in der Arbeit beschriebene *Gamification Service* konnte erfolgreich im Produktiveinsatz für *runtastic* verwendet werden und erfüllte dabei alle definierten Anforderungen. Aufgrund der flexiblen Auslegung des Ansatzes könnte das *Service* außerdem für ähnliche Szenarien adaptiert und verwendet werden.

Abstract

There has been great interest in *Gamification*, the use of game design techniques in non-game contexts, by both researchers and businesses alike in recent years. The impact of using *Gamification* has already been broadly documented but research regarding the technical implementation of *Gamification* techniques seems to be as of yet missing.

Therefore a *Gamification Service* has been designed and implemented using the *Ruby* programming language in this thesis. It allows for the integration of *Gamification* techniques, such as *Badges* and *Records*, into a system architecture that already exists. Additionally, the selected approach has been proven successful by a case study with the company, *runtastic*. *runtastic* already had an existing *API* for mobile devices that was then adapted to transmit *Gamification* data, as well.

The described system uses 3 separate components which are named *Core*, *Service* and *Framework*. The *Core* component contains all shared parts that are both used by the *Service* and the *Framework*. An *API* allows the *Service* to be accessed by other systems. Requests sent to the *Service* will get analyzed to determine if they have any dependencies. All identified dependencies will get split into separate tasks, which are processed in parallel using the *Framework*. The tasks and their contents will get evaluated, calculated and saved. Thereafter the result is returned to the *API* again. Contents can be defined by using a specifically designed, domain-specific language. This language can be extended with additional functionality, if needed.

The *Gamification Services* described in this thesis was successfully used in production for *runtastic* and thereby met all specified requirements. It could certainly be used for similar or completely different scenarios in the future, with little extra work, due to its flexible design and implementation process.

Kapitel 1

Einleitung

1.1 Motivation und Zielsetzung

Gamification, die Nutzung von Spieledesigntechniken in einem Kontext außerhalb von Spielen, wurde in den letzten Jahren zu einem weitreichenden Thema, das in vielen Bereichen der Wirtschaft, aber auch in der Forschung eingesetzt bzw. behandelt wird [3]. Unternehmen setzen in ihren Produkten auf *Gamification* Techniken, in der Hoffnung, dadurch ein besseres Benutzererlebnis und ein höheres Nutzerengagement zu erzielen. Manche Unternehmen spezialisieren sich sogar auf das Thema und bieten Plattformen an, mit denen auf einfache Weise *Gamification* Elemente in Webseiten integriert werden können. Bis auf wenige Ausnahmen im Open-Source Bereich sind die dafür verwendeten Berechnungssysteme proprietär und die technischen Details nicht zugänglich.

In der Forschung werden wiederum hauptsächlich die Auswirkungen des Einsatzes von Gamification in verschiedensten Bereichen, wie in der Bildung, am Arbeitsplatz oder im Gesundheitswesen betrachtet [36]. Zum Entstehungszeitpunkt dieser Arbeit konnten jedoch keine Forschungsarbeiten zu der Konzeption und Implementierung von Systemen gefunden werden, die sich der Umsetzung und Berechnung von *Gamification* Elementen widmen.

Als Ziel für diese Arbeit wurde die Konzipierung und Implementierung eines Prototyps für ein *Gamification Service* in Ruby gesetzt. Weil *runtastic* (s. Abschn. 2.8) zudem als Auftraggeber für das *Gamification Service* gewonnen werden konnte, war im Rahmen des Masterprojektes und der Masterarbeit im Studiengang Interaktive Medien außer der Konzeption und Implementierung auch die reale Verwendung des Systems im Produktivbetrieb mit einer Vielzahl an Benutzern möglich.

1.2 Inhaltlicher Aufbau

Nach der Einleitung werden in Kapitel 2 die benötigten *Grundlagen* für den weiteren Verlauf der Arbeit vermittelt. Im darauf folgenden Kapitel 3 werden zuerst die verwandten Systeme (Abschn. 3.1) und danach die Anforderungen (Abschn. 3.2) diskutiert, die Konzeption und Implementierung zugrunde gelegen sind.

In Kapitel 4 wird das Konzept für das *Gamification Service* anhand der Prinzipien und Funktionalitäten vorgestellt, die für den eigenen Ansatz festgelegt worden sind. Außerdem werden die für eine Implementierung vorgeschlagenen Systemkomponenten im Abschnitt Systemaufbau (4.2) geschildert.

Das Kapitel 5 widmet sich der Implementierung des im vorigen Kapitel vorgestellten Konzepts. Nach einem allgemeinen Überblick wird auf den *Core* (Abschn. 5.1), das *Service* (Abschn. 5.2) und das *Framework* (Abschn. 5.3) sowie die darin enthaltenen Komponenten detailliert eingegangen. Zum Abschluss des Kapitels werden noch die 2 Abläufe vorgestellt, die in der Regel innerhalb des gesamten *Gamification Services* ablaufen.

In Kapitel 6 werden die Erfahrungen und Resultate sowohl während der Konzeption und Implementierung als auch im Produktivbetrieb des *Gamification Services* behandelt. In den danach folgenden Schlussbemerkungen (Kap. 7) folgt das Fazit über die Ergebnisse sowie der Ausblick auf weitere Verwendungs- und Verbesserungsmöglichkeiten für das in dieser Arbeit vorgestellte System.

Kapitel 2

Grundlagen

2.1 Gamification

Gamification bezeichnet die Nutzung von Spieledesignelementen in einem nichtspielerischen Kontext, wie zum Beispiel am Arbeitsplatz, im Gesundheits-, Fitness- oder Bildungsbereich [3]. Als Begriff wurde *Gamification* das erste Mal im Jahre 2002 in diesem Zusammenhang von Nick Pelling verwendet [8].

Unter anderem durch die im Jahr 2009 gestartete ortsbasierte Smartphone-Applikation *Foursquare*[1], welche einige Spieletechniken verwendet, um seine Benutzer an sich zu binden und deren Engagement zu erhöhen, wurde *Gamification* einer breiten Masse zugänglich gemacht [67]. Aufgrund dieses Erfolges begannen weitere Unternehmen, Entwickler und Designer in spielefernen Bereichen damit, spielerische Aspekte in ihre Anwendungsbereiche einzubinden. Es wurden einige neue Unternehmen wie z. B. *Bunchball*[2], *badgeville.com*[3] oder *Bigdoor*[4] gegründet, die *Gamification* als Webservice zur Integration in Webseiten, Handy-Applikationen und anderer Software anbieten.

2.1.1 Verfolgte Ziele

Mithilfe der Nutzung von Designtechniken, die üblicherweise in Spielen zum Einsatz kommen, können verschiedene Ziele verfolgt werden, wie z. B.:

- Die Motivation eines Benutzers zu steigern, um eine Tätigkeit besser oder schneller auszuführen.
- Die Bindung mit Kunden aufzubauen oder zu verbessern.
- Menschen dazu zu bewegen, ihr Verhalten zu verändern oder Dinge zu

[1] http://www.foursquare.com/
[2] http://www.bunchball.com/
[3] http://www.badgeville.com/
[4] http://www.bigdoor.com/

tun, die sie sonst nicht machen würden.

Mehr über *verfolgte Ziele* in [15, Kap. 2].

2.1.2 Geläufige Spielemechanismen

Für *Gamification* werden u. a. folgende Spielemechanismen häufig angewendet:

Achievements: Ein Achievement (Errungenschaft) stellt das Erreichen eines bestimmten Zieles dar. Der Nutzer bekommt dafür üblicherweise eine Belohnung, die in Form eines Abzeichens (Badge), eines Meilensteins, eines Pokals, einer Medaille oder in einer anderen Art, dargestellt wird. Die Ziele zum Erreichen eines Achievements können sowohl offensichtlich als Anreiz für den Benutzer dargestellt werden, diese zu erfüllen, aber auch bis zum Abschluss der Aufgabe versteckt bleiben, um ihn mit der Belohnung zu überraschen [35]. In der *Foursquare* App werden *Badges* z. B. wie in Abb. 2.1 (a) dargestellt.

Punkte: Durch das Erledigen von bestimmten Aktionen kann ein Nutzer Punkte sammeln. Die Gesamtpunkteanzahl kann dann beispielsweise zum Vergleich mit anderen Benutzern dienen, wie es in Abb. 2.1 (b) zu sehen ist, aber auch als Indikator für den Status oder die Vertrauenswürdigkeit des Users herangezogen werden [15, Kap. 3].

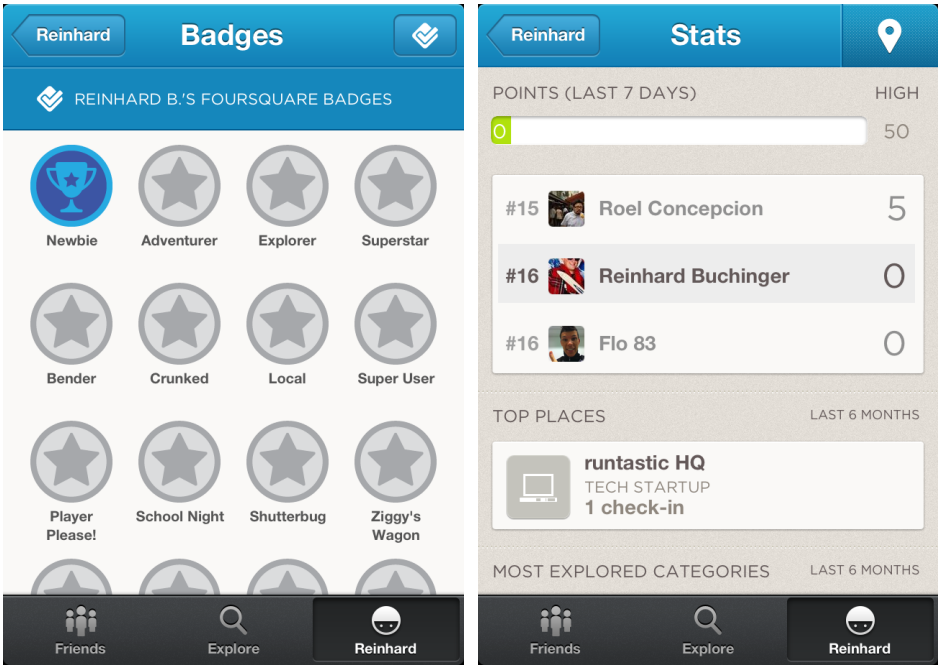
Leaderboards: Mithilfe von Ranglisten, wie sie z. B. in Abb. 2.2 dargestellt sind, können die besten Benutzer einer Anwendung hervorgehoben werden. Außerdem ermöglichen sie den gegenseitigen Vergleich zwischen Usern. Leaderboards können sowohl global für alle Benutzer einer Anwendung ausgelegt als auch nur lokal beschränkt auf den Freundeskreis des Nutzers sein. Als Vergleichswerte sind sowohl Punkte als auch andere aussagekräftige Werte verwendbar [34].

2.1.3 Frühere Formen

Ähnliche Ansätze zur Nutzung von Gamedesignelementen, die sich auch unter dem Begriff *Gamification* eingliedern lassen, gibt es bereits länger, z. B.:

Frequent Flyer Programme bei denen Kunden von Airlines für ihre Flüge Punkte sammeln und diese dann in Vergünstigungen für Flüge oder andere Produkte einlösen können [14, Kap. 1].

Bonuskartenprogramme mit denen Kunden z. B. von Supermärkten bessere Angebote bekommen und ebenfalls Punkte sammeln können [14, Kap. 2].



(a) (b)

Abbildung 2.1: Bildschirmfotos entnommen aus der *Foursquare* iPhone App [33]. Darstellung der Badges (a). Darstellung der Punkte mit Freundeleaderboard (b).

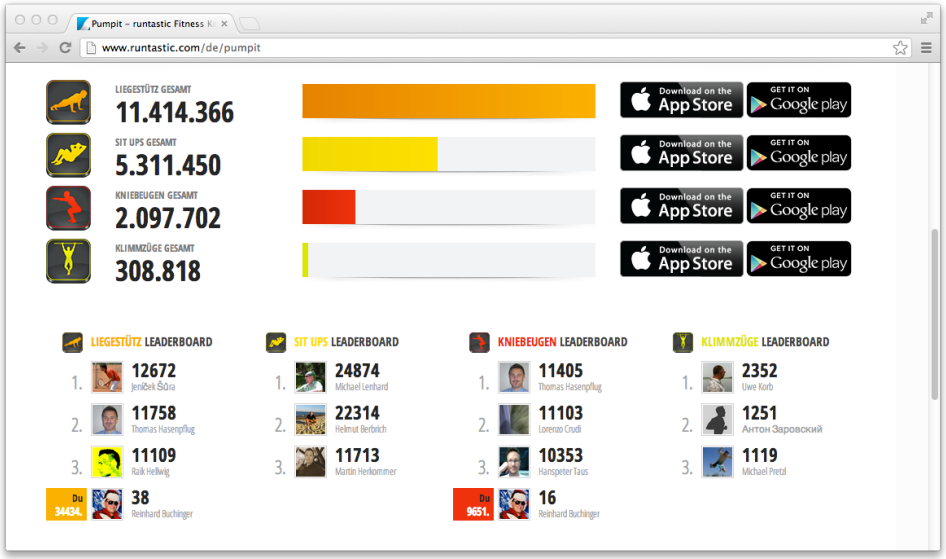


Abbildung 2.2: Leaderboards, wie sie bei *runtastic* verwendet werden [59].

2.2 Apache Cassandra

Apache Cassandra ist eine *NoSQL*[5] Datenbank, die ursprünglich bei *Facebook*[6] entwickelt wurde und dort für die Nachrichtensuche im Nachrichtensystem verwendet wurde [25]. Nachdem *Facebook* die Datenbank im Jahr 2008 unter einer Open Source Lizenz veröffentlicht hat, wurde *Cassandra* als Projekt unter dem Dach der *Apache Software Foundation*[7] integriert [32]. Außerdem gründeten *Cassandra*-Unterstützer die Firma *DataStax*[8], um sich an der Weiterentwicklung zu beteiligen sowie kommerzielle Unterstützung für Nutzer von *Cassandra* anbieten zu können. Seit der Veröffentlichung als Open Source Projekt ist *Cassandra* bei einigen namhaften Unternehmen wie z. B. *Netflix*, *Twitter* und *Adobe* in Verwendung. Zum Zeitpunkt des Schreibens war als aktuellste die Version 1.1.6 verfügbar.

2.2.1 Datenmodell im Vergleich zu RDBMS

Das Datenmodell von *Cassandra* unterscheidet sich grundlegend von *relationalen Datenbanken* (*RDBMS*). In der bei *RDBMS* üblichen Modellierung mit Entitäten und Relationen werden Daten in Tabellen gespeichert, die durch Fremdschlüssel miteinander verbunden sind. Einträge werden so weit wie möglich ohne Datenredundanzen erstellt und in Abfragen mittels der jeweiligen Schlüssel zusammengeführt [7].

Bei *Cassandra* hingegen werden die Daten so modelliert, dass sie für die entsprechenden Abfragen in der Anwendung ausgelegt sind, weshalb hier auch Daten mehrmals redundant gespeichert werden, wenn dies nötig ist. Außerdem gibt es keine bindenden Fremdschlüsselbeziehungen, wie dies bei *relationalen Datenbanken* möglich ist [66].

Daten werden in so genannten *Column Families* innerhalb eines *Keyspace* gespeichert, ähnlich wie Tabellen innerhalb einer Datenbank bei einer *relationalen Datenbank*. Sowohl in *Cassandra* als in *RDBMS* werden *Column Families*/Tabellen mit einem Schema beschrieben, in dem die enthaltenen Spalten definiert sind. *Column Families* sind *Schema-optional*, weshalb nicht alle Spalten einer Zeile vorhanden sein müssen. Deshalb speichert *Cassandra* in jeder Zeile die Spaltennamen zur jeweiligen Spalte [61].

Für Abfragen der Daten kommt bei *Cassandra* nicht die bei Datenbanken übliche *Structured Query Language*[9] (*SQL*) zum Einsatz. Stattdessen wurde sie anfangs nur über eine *Thrift*[10]-basierte Schnittstelle angesprochen. Diese ermöglicht den Aufruf von entfernten Methoden (*Remote Procedure Call*) aus beliebigen Programmiersprachen, für die *Thrift* implementiert wurde.

[5] http://en.wikipedia.org/wiki/NoSQL
[6] http://www.facebook.com/
[7] http://www.apache.org/
[8] http://www.datastax.org/
[9] http://en.wikipedia.org/wiki/SQL
[10] http://thrift.apache.org/

Mit der Weiterentwicklung von *Cassandra* wurde dann eine eigene, an *SQL* angelehnte *Cassandra Query Language* (*CQL*) entwickelt, die Abfragen mit einer verständlichen Sprache ermöglichen. Außerdem lässt diese die Implementierung neuer Funktionen in der Datenbank zu, ohne dass Datenbanktreiber verändert werden müssen, weil die Verarbeitung der Sprache direkt am Datenbankserver stattfindet.

Im Gegensatz zu *relationalen Datenbanken* können mit *Cassandra* keine Abfragen getätigt werden, die über mehrere Tabellen/*Column Families* verknüpft sind. Selbst die Filterung von Daten, sowie die Sortierung von Daten innerhalb einer *Column Family*, ist stark eingeschränkt. Grundsätzlich unterstützt *Cassandra* nur Abfragen über den Primärschlüssel sowie über etwaige im Schema definierte Indizes. Primärschlüssel können sowohl aus einer Spalte, als auch aus mehreren Spalten bestehen (*Composite Primary Keys*). Bei Abfragen über mehrspaltige Primärschlüssel ist jedoch zu beachten, dass die Einschränkungen immer von der ersten Spalte ausgehend sein müssen. Daher kann nicht einfach nur z. B. die zweite Spalte des *Composite Primary Keys* verwendet werden, wie es in einer *relationalen Datenbank* möglich wäre. Deshalb ist die Art der Abfrage bei der Definition des *Column Family* Schemas unbedingt zu beachten, wie eingangs bereits in Bezug auf die Datenmodellierung erwähnt wurde [40].

2.2.2 Vorteile im Clusterbetrieb

Im Clusterbetrieb auf mehreren Servern zeichnet sich *Cassandra* durch folgende Vorteile im Vergleich zu herkömmlichen *relationalen Datenbanksystemen* wie z. B. *MySQL*[11] aus:

- Keine *Single Points of Failure*[12] aufgrund dezentraler Struktur, wobei kein Knoten eine Sonderstellung einnimmt.
- Der Lese- und Schreibdurchsatz erhöht sich linear mit dem Hinzufügen neuer Server zu einem Cluster, ohne eine *Downtime*[13] der Anwendung in Kauf nehmen zu müssen.
- Fehlertoleranz aufgrund von automatischer Datenreplikation auf mehrere Knoten und sogar zwischen mehreren Datencentern. Fehlerhafte Knoten können jederzeit ohne Unterbrechung ersetzt werden [17].

2.3 Apache ZooKeeper

Apache ZooKeeper ist ein hochverfügbares und hochzuverlässiges Service zur Koordination von verteilten Systemen. Diese als Open Source zur Verfügung gestellte *Java*-basierte Software ermöglicht es, Konfigurationsdaten,

[11] http://www.mysql.com/
[12] http://en.wikipedia.org/wiki/Single_point_of_failure
[13] Zeitdauer, die ein System nicht verfügbar ist: http://en.wikipedia.org/wiki/Downtime

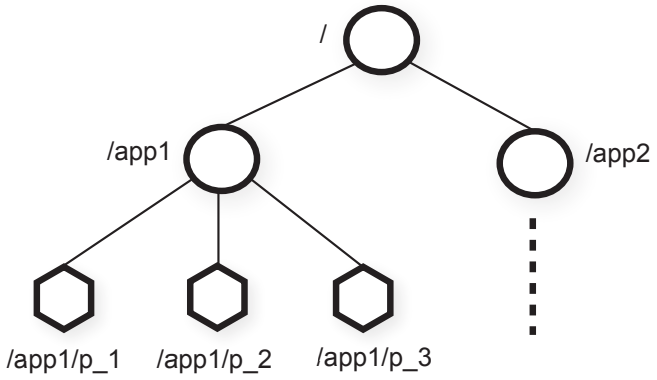


Abbildung 2.3: Darstellung der Baumstruktur in ZooKeeper [6, Kap. 2.1].

Synchronisationsmechnismen und Gruppen für verteilte Anwendungen und Services zentralisiert zu verwalten. Üblicherweise wird *ZooKeeper* als Cluster bestehend aus mehreren Servern ausgeführt, damit ein Ausfall eines Servers nicht zum *Single Point of Failure* wird [22].

Sämtliche Daten werden im RAM-Speicher des Servers gehalten, zusätzlich werden sie auch auf der Festplatte mithilfe eines Binärlogs persistiert, um bei einem Ausfall Datenverluste soweit wie möglich zu minimieren. Diese Architektur ermöglicht besonders schnelle Antwortzeiten bei gleichzeitiger Datenkonsistenz, wodurch *ZooKeeper* mit einem Verhältnis von 10:1 bei Lese- und Schreiboperationen die beste Performance liefert. Außerdem werden alle Operationen strikt in der Eingangsreihenfolge nach Zeitstempel ausgeführt, wodurch sich das Service für die Abbildung von Synchronisierungsmechanismen sehr gut eignet [23].

Die Daten werden mithilfe einer dateisystemähnlichen Baumstruktur abgebildet, wie sie in Abb. 2.3 dargestellt ist, deren Knoten als *znodes* bezeichnet werden. Jeder Knoten kann selbst Daten und beliebig viele Kinderknoten beinhalten. Knoten können entweder *persistent* oder *ephemeral* angelegt werden, wobei *ephemerale* Knoten nur Kinderknoten beinhalten können, die ebenfalls als *ephemeral* markiert sind. *Ephemeral* bedeutet, dass die Knoten sofort gelöscht werden, wenn die Verbindung zwischen einem Client und dem *ZooKeeper* Server abbricht [6, Kap. 2.1]. Dieses Vorgehen ist für die Abbildung von Synchronisierungsmechanismen sehr von Vorteil, da z. B. *Locks* sofort gelöscht werden, wenn der Client abstürzt und so ein anderer Client die Arbeit übernehmen kann.

Neben der Unterscheidung zwischen *persistent* und *ephemeral* gibt es mit *sequential* eine weitere Option, mit der definiert werden kann, ob Knoten eine fortlaufende Sequenznummer bekommen [6, Kap. 2.1]. Diese ermöglicht es, Knoten mit demselben Namen zu erstellen und danach herauszufinden,

ob der vom aktuellen Client erstellte Knoten die höchste oder niedrigste Nummer aller Knoten im selben Baumabschnitt hat, um daraus Schlüsse für das weitere Vorgehen im Algorithmus zu ziehen.

Sämtliche *znodes* können außerdem mit einem so genannten *Watch* beobachtet werden. Das bedeutet, dass der Client, der den *Watch* setzt, über alle oder nur spezifische Änderungen des beobachteten Knotens informiert wird und dadurch auf Veränderungen des Knotens bzw. seiner Kinderknoten oder der darin enthaltenen Daten reagieren kann.

Die Schnittstelle, mit der auf *ZooKeeper* zugegriffen werden kann, ist neben den bereits erwähnten *Watches* mit simplen Lese- und Schreiboperationen absichtlich einfach gehalten. Client-Bibliotheken, die für sämtliche Programmiersprachen zur Verfügung stehen, kümmern sich deshalb um die eigentliche Implementierung der Rezepte für Abstraktionen wie z. B. *Locks* [6, Kap. 2.4].

2.4 RabbitMQ

RabbitMQ ist ein Service für *Message Queues*, das auf dem offenen *Advanced Message Queuing Protocol* (*AMQP*) Standard aufbaut [2]. Die als Open Source verfügbare Erlang[14]-basierte Software fungiert als *Broker* zwischen Writer-Clients, die Nachrichten über einen *Exchange* je nach gewählter Option in einer oder mehreren *Queues* veröffentlichen, und Reader-Clients, die aus einer *Queue* lesen [48].

Um nicht als Single Point of Failure zu fungieren, kann *RabbitMQ* in einer speziellen Hochverfügbarkeitskonfiguration auf mehreren Servern betrieben werden. Üblicherweise werden Nachrichten nur im RAM-Speicher des Servers gehalten, um die bestmögliche Performance zu bieten. Allerdings erlaubt das *AMQP* Protokoll, und damit *RabbitMQ*, jede einzelne Nachricht ebenfalls auch auf der Serverfestplatte zu persistieren, wodurch etwaigen Datenverlusten durch Systemausfälle vorgebeugt werden können [46].

Das *AMQP* Protokoll ermöglicht mithilfe seiner vielseitigen Optionen einige für verteilte Systeme relevante Szenarien:

Work Queues: Dabei entnehmen mehrere *Arbeiter Clients* ihre Arbeit aus einer gemeinsamen *Work Queue*, die von Auftraggebern über einen *Exchange* gespeist wird. Damit können auf einfache Art und Weise mehrere Server gleichmäßig mit Aufgaben versorgt werden, wobei jede Aufgabe nur an einen Server mitgeteilt wird. Mithilfe des *Message Acknowledgement* Features, bei dessen Nutzung ein *Queue Client* nach getaner Arbeit bestätigen muss, ob die Nachricht erfolgreich oder nicht erfolgreich bearbeitet worden ist, kann verhindert werden, dass Nachrichten einmal nicht verarbeitet werden und im Fehlerfall an einen anderen *Arbeiter Client* weitergeleitet werden können [52].

[14] http://www.erlang.org/

Publish/Subscribe: Im Gegensatz zu *Work Queues* wird bei diesem Prinzip jedem Client die Nachricht mitgeteilt. Dazu verfügt jeder *Reader Client* über eine eigene *Queue*, die von dem gemeinsamen *Exchange* gespeist wird, worüber *Writer Clients* ihre Nachrichten veröffentlichen [47].

Routing: Dabei verwendet jeder Client seine eigene *Queue* und registriert sich bei einem gemeinsamen Exchange nur für die Schlüsselwörter, zu denen er Nachrichten empfangen möchte [49].

Topics: Topics erweitert das Routing-Prinzip dahingehend, dass anhand eines mit *Wildcards*[15] (* für ein Wort, # für 0 oder mehrere Wörter) gespickten Musters entschieden wird, ob das Thema einer Nachricht für eine *Client Queue* relevant ist oder nicht. Dabei werden einzelne Wörter in einem String durch Punkte getrennt zur Themenbeschreibung vom Nachrichten-Sender überliefert und vom Exchange mit den abonnierten Mustern der *Client Queues* verglichen und eingeordnet [51].

RPC: Das *Remote Procedure Call* Prinzip ermöglicht es, die Arbeitsverteilung für einen Prozess über eine *Queue* abzuwickeln und danach das Ergebnis wieder an den korrekten Absender der Arbeitsaufgabe zu senden. Dies ist z. B. hilfreich, um eine blockierende Schnittstelle für andere Systeme darzustellen und interne Abläufe asynchron zu handhaben. Um die Antwort an den korrekten Absender zurückzusenden, kann in den Metadaten der Nachricht eine für die Nachricht eindeutige Identifizierung angegeben werden, weiters wird festgelegt, an welche *Antwort Queue* die Ergebnisse schlussendlich geliefert werden sollen. Aufgrund dieser Informationen weiß der Empfänger der Nachricht sofort, auf welche Anfrage gerade eine Antwort eingetroffen ist [50].

2.5 Memcached

Memcached ist ein ursprünglich von *LiveJournal*[16] entwickeltes Service, das zur Zwischenspeicherung von jeglichen Daten verwendet werden kann und als Open-Source Software zur Verfügung steht. Die Dateninhalte werden im RAM-Speicher des Servers gehalten, wodurch mit *Memcached* sehr schnelle Antwortzeiten ermöglicht werden [43].

Daten können mithilfe eines Schlüsselwertes eingefügt, geändert und abgefragt werden. Zusätzlich lässt sich *Memcached* zur höheren Verfügbarkeit und Skalierbarkeit auf mehreren Servern installieren und ansprechen. Dazu dient ein Hashing-Algorithmus in der Zugriffsbibliothek, der den richtigen Speicherort für einen bestimmten Schlüssel auswählt und auch beim Ausfall eines Servers für die Umleitung auf einen anderen Server sorgt [5].

[15] http://en.wikipedia.org/wiki/Wildcard_character
[16] http://www.livejournal.com/

2.6 Ruby

Ruby ist eine dynamische Open Source Programmiersprache, die ursprünglich von Yukihiro Matsumoto entwickelt und im Jahr 1995 das erste Mal der Weltöffentlichkeit vorgestellt wurde [18, 30]. Seither erfreut sich die Sprache großer Beliebtheit bei Programmierern, weil sie einige Ansätze verfolgt, die in Kombination sehr vorteilhaft sind [13].

Ruby ist eine dynamische, objektorientierte Programmiersprache, die außerdem umfangreiche Möglichkeiten der Metaprogrammierung besitzt. Mit dem Prinzip der Metaprogrammierung lassen sich Bereiche der *Ruby* Sprache flexibel an benötigte Anwendungsfälle anpassen, ohne dafür aufwendigen bzw. umfangreichen Quellcode schreiben zu müssen, da die dafür benötigte Funktionalität bereits in der Sprache integriert ist [11].

2.6.1 JRuby

Ruby gibt es neben den Referenzimplementierungen *MRI* (*Matz Ruby Interpreter*) und *YARV* auch in weiteren Ausführungen. Dazu zählt neben *Rubinius*[17] und *MacRuby*[18] unter anderem auch *JRuby*. *JRuby* läuft in der *Java Virtual Machine* und wandelt den *Ruby* Code zur Laufzeit in Bytecode um, damit dieser in der *JVM* ausgeführt werden kann [41].

Durch die Verwendung der *Java Virtual Machine* fällt die Limitierung der Standardinterpreter weg, welche aufgrund des *Global Interpreter Lock* immer nur einen Thread gleichzeitig abarbeiten können, obwohl die Sprache selbst keine Hindernisse zu Multithreading in den Weg stellt. Mit *JRuby* werden alle Threads, wie auch bei in *Java* programmierter Software, parallel und auf mehreren Prozessorkernen ausgeführt [58]. Deshalb wird mit *JRuby* auch die mit dem *MRI* übliche Vorgehensweise, für jede CPU einen eigenen Prozess zu starten, nicht mehr benötigt, wodurch der Speicherverbrauch um einiges geringer ausfällt, da der Speicherbereich zwischen den Threads gemeinsam genutzt wird [45]. Außerdem kann Dank der Verwendung der *JVM* auch auf Bibliotheken zugegriffen werden, die nicht in *Ruby*, sondern z. B. in *Java* oder *Clojure*[19], entwickelt wurden. Dadurch lässt sich oft ein Ersatz für *Ruby* Bibliotheken finden, die von in *C* programmierten Erweiterungen abhängig sind.

Ein Nachteil von *JRuby* ist, dass die Sprachenkompatibilität den Versionen von *Ruby* hinterherhinkt. Hier wurde mit der aktuellen Version 1.7 immerhin die Standardversionsunterstützung auf die Version 1.9.3 angehoben. Bei darunterliegenden *JRuby* Versionen war als standardmäßiger Kompatibilitätsmodus *Ruby* 1.8 aktiviert und nur optional eine Unterstützung von 1.9 verwendbar [42]. Geschwindigkeitstechnisch ist *JRuby* in Benchmarks je

[17] http://rubini.us/
[18] http://macruby.org/
[19] http://clojure.org/

nach Anwendungsszenario viel schneller oder deutlich langsamer als der Standardinterpreter, weshalb in jedem Fall die Wahl der Plattform ausführlich getestet werden sollte, bevor eine Anwendung in Produktion geht [57].

2.6.2 Gems

Erweiterungsbibliotheken für *Ruby* werden als *Gems* (Edelsteine) bezeichnet. *Gems* sind unter anderem über typische Anlaufstellen wie *RubyGems*[20], *The Ruby Toolbox*[21] oder *Github*[22] erhältlich. *Gems* stehen üblicherweise unter einer Open-Source Lizenz und sind deswegen in der Regel als Quellcode verfügbar. Viele der Projekte werden über *Github* abgewickelt, wo über ein Webinterface direkt durch den Quellcode geschmökert werden kann, um ein besseres Verständnis für die Funktionsweise eines *Gems* zu erhalten. *Gems* können mithilfe eines *Gemfiles* durch den *Bundler* in eigene Anwendungen eingebunden werden [37].

Mithilfe des *Ruby Version Managers* können mehrere verschiedene *Ruby* Versionen und Interpreter einfach nebeneinander installiert und verwendet werden. Außerdem ermöglicht der *RVM*, dass für jedes Projekt automatisch das richtige *Gemset* (Ansammlung an *Gems*) geladen wird.

Celluloid

Celluloid ist eine Bibliothek, die es erleichtert, Applikationen mit Nebenläufigkeiten in *Ruby* zu implementieren [26]. *Celluloid* setzt dabei auf das *Actor Model*, bei dem Programmeinheiten in eigenen Threads ausgeführt werden. Die Kommunikation zwischen den Akteuren erfolgt durch den Austausch von Nachrichten. Mehr über das *Actor Model* findet sich in [1].

Die Bibliothek ermöglicht bei jedem Methodenaufruf zu unterscheiden, ob dieser nebenläufig (asynchron) oder sequentiell (synchron) abgearbeitet werden soll. Bei einem asynchronen Aufruf wird an den eigentlichen Methodennamen ein Rufzeichen angehängt. Weil bei einem asynchronen Methodenaufruf das Abrufen des Ergebnisses nicht möglich ist, unterstützt *Celluloid* zu diesem Zweck die Nutzung von *Futures*. Der Aufrufende erhält sofort ein *Future Objekt* als Antwort, weshalb der Akteur sofort weiterarbeiten kann, ohne auf die Fertigstellung der Methode warten zu müssen. Mithilfe des *Future Objektes* kann jederzeit überprüft werden, ob der Methodenaufruf bereits abgeschlossen wurde. Sobald dies der Fall ist, enthält der *Future* auch das Ergebnis des asynchronen Methodenaufrufs, welches danach zur Weiterverarbeitung durch den aufrufenden Akteur herangezogen werden kann [27].

Zur Verwendung von mehreren gleichen Akteuren bietet *Celluloid* einen

[20] http://www.rubygems.org/
[21] https://www.ruby-toolbox.com/
[22] https://www.github.com/

Thread Pool an. Dieser ermöglicht es, transparent auf mehrere parallele Akteure zurückzugreifen, welche jedoch für den Aufrufenden wie ein einzelner Akteur aussehen. Außerdem kümmert sich *Celluloid* um den Neustart von Akteuren im Fehlerfall, wodurch für eine starke Fehlertoleranz innerhalb einer Applikation gesorgt wird [26].

Rack

Das *Rack Gem* wird von vielen *Ruby* Webapplikationen und -frameworks verwendet, um mit einer einheitlichen Schnittstelle an verschiedene Webserver angebunden und damit ausgeführt werden zu können [39]. Mehr dazu findet sich in [53].

Anfragen an den Webserver werden mit *Rack* üblicherweise synchron abgearbeitet. Um eine asynchrone Abarbeitung zu ermöglichen, wird einerseits ein Webserver benötigt, der eine asynchrone Bearbeitung von Anfragen unterstützt, andererseits muss ein ergänzendes *Gem* wie z. B. *Async Rack* (für *Ruby*) oder *Mizuno* (für *JRuby*) verwendet werden [24, 44]. Diese *Gems* ermöglichen eine Mitteilung an den Webserver, dass dieser die erwartete Antwort der Webapplikation nicht über den Anwortinhalt der aufgerufenen Methode, sondern über einen von ihm definierten asynchronen Callback zurückgeliefert bekommt.

Grape

Grape ist ein Framework zur Implementierung von REST-Schnittstellen, das mit einer einfach gehaltenen *domänenspezifische Sprache* ermöglicht, API-Endpunkte zu definieren. Außerdem kümmert sich das *Framework* um das Parsen und Validieren sämtlicher Parameter, die bei einer Anfrage an die API gestellt werden. Falls dabei ein Fehler auftritt, übernimmt *Grape* die Beantwortung der Anfrage mit einer Fehlermeldung. Das Framework kann sowohl mit Anfragen in *XML* als auch in *JSON* umgehen und abstrahiert damit die Formate von der eigentlichen Applikationslogik der API-Anwendung. Mithilfe der DSL können u. a. unterschiedliche API-Versionen zur Verfügung gestellt, diese in Namespaces gegliedert und modular in andere *Rack*-Anwendungen integriert werden. Mehr dazu findet sich in [38].

AMQP

AMQP[23] ist eine Bibliothek, die den Zugriff auf *Message Queues* ermöglicht, die das *AMQP* Protokoll unterstützen [20]. Die Bibliothek arbeitet mit einem eventbasierten Callback Prinzip, weshalb sie nicht ohne weiteres in bestehende sequentielle Anwendungen integriert werden kann. Intern verwendet das *AMQP Gem* die *EventMachine* Library, welche eventbasierte Abarbeitung

[23] http://rubyamqp.info/

von I/O auf Basis des *Reactor Patterns* ermöglicht [70]. *EventMachine* läuft in einem eigenen Thread und ruft in einer Endlosschleife für sämtliche eingehende Daten die dazu registrierten Callbacks auf. Mehr über *EventMachine* in [31].

ZK

ZK ist eine Bibliothek, mit der auf den *Apache ZooKeeper* Server zugegriffen werden kann. Mehr zu *ZK* findet sich in [71].

Dalli

Dalli ist eine Bibliothek, mit der auf den *Memcached* Server zugegriffen werden kann. Mehr zu *Dalli* findet sich in [29].

Connection Pool

Connection Pool ist eine Bibliothek, mit der ein Pool von Verbindungen zu externen Services erstellt werden kann. Dieser Pool erleichtert das Teilen von Verbindungen über die Grenzen von Threads hinweg. Mehr zu *Connection Pool* findet sich in [28].

2.7 Domänenspezifische Sprachen

Eine *domänenspezifische Sprache* ist eine Sprache, die nur für einen gewissen Zweck ausgelegt ist. Sie hilft dabei, an einen spezifischen Problembereich mit dafür maßgeschneiderten Befehlen und Werkzeugen heranzugehen und diesen zu meistern. Beispiele für eine *DSL* sind *HTML*, *XML*, *SQL*, *LaTeX* oder *UML*. *DSLs* können sowohl intern (eingebettet), als auch extern ausgelegt werden. Eine *interne DSL* ist eine domänenspezifische Sprache, die direkt innerhalb einer Programmiersprache abgebildet wird und dabei auf bestehende Sprachelemente zurückgreift. Eine *externe DSL* dagegen beschreibt eine eigens für den Problembereich entwickelte Sprache, die mit einem Parser eingelesen und ausführbar gemacht wird [4].

2.7.1 Vorteile

Eine *DSL* erlaubt es Personen aus dem Fachbereich, für den die Sprache geschaffen wurde, Programme selbst zu schreiben und zu modifizieren. Außerdem sind die Sprachen zu einem Großteil selbstbeschreibend, weshalb keine eigene Dokumentation benötigt wird. Eine *DSL* erlaubt die Validierung und Optimierung auf Basis des Fachbereichs und ist deswegen produktiver, zuverlässiger, wartbarer und portabler als allgemeine Sprachen [4].

2.7.2 Nachteile

Der Aufwand für die Implementierung und die Wartung der Sprache, sowie der Aufwand für die Schulung der User, können sich nachteilig auswirken. Außerdem kann es schwierig sein, den passenden Anwendungsbereich für die Sprache zu finden. In weiterer Folge kann die Effizienz durch die Abstraktion einer *DSL* im Vergleich zu normal programmierter Software verloren gehen [4].

2.7.3 DSL in Ruby

Interne domänenspezifische Sprachen sind mit *Ruby* komfortabel umzusetzen, weil *Ruby* eine dynamische Sprache ist, welche umfangreiche Möglichkeiten zur Metaprogrammierung bietet. Außerdem können den Lesefluss störende Elemente wie Klammern in den meisten Fällen weggelassen werden, wodurch *Ruby DSLs* meistens an lesbare Anweisungen erinnern.

Bekannte *Gems* wie z. B. *Rails*[24], *Sinatra*[25] und *Grape*[26] verwenden *DSL-Sprachkonstrukte*, um ihre Funktionalität vereinfacht zur Verfügung zu stellen und die Lesbarkeit zu erhöhen. Anhand des Sinatra *Hello World* Beispiels kann man erkennen, wie selbsterklärend eine mit einer DSL definierte Anwendung sein kann [63]:

```
1 require 'sinatra'
2
3 get '/hi' do
4   "Hello World!"
5 end
```

Sobald die Anwendung ausgeführt wird, kann die Webseite mit der URL `/hi` aufgerufen werden, welche dann *Hello World!* als Antwort gibt.

2.8 runtastic

runtastic[27] ist ein Hersteller von Handy-Applikationen mit dem Hauptfokus auf die Bereiche Fitness und Gesundheit. Der Hauptsitz liegt in der *PlusCity* in Pasching bei Linz. Gleichzeitig war die Handyanwendung mit dem Namen *runtastic* auch das erste Produkt des Unternehmens. Mit dieser Anwendung kann ein Sportler seine Outdoor-Aktivitäten mithilfe des eingebauten GPS Modules seines Smartphones aufzeichnen. Bei Verwendung eines mit dem Mobiltelefon gekoppelten Herzfrequenzmessers wird der Herzschlag ebenfalls erfasst. Während der aktuellen Aktivität teilt ein virtueller Coach die aktuellen Daten, wie zum Beispiel Zeitdauer, Geschwindigkeit,

[24] http://rubyonrails.org/
[25] http://sinatrarb.org/
[26] https://github.com/intridea/grape
[27] http://www.runtastic.com/

Zeit pro Kilometer oder Herzfrequenz, per Sprachausgabe mit. Außerdem kann er über den aktuellen Stand eines Wettkampfes benachrichtigen, wenn man gegen frühere Aktivitäten oder Aktivitäten anderer Sportler mithilfe eines *Ghost-Runs* antritt.

Die aufgezeichneten Sportaktivitäten können dann auf das *runtastic* Webportal übertragen werden, um dort detaillierte Auswertungen und Statistiken betrachten zu können. Außerdem ist die Webseite auch ein soziales Netzwerk, wo sich Sportler mit anderen Sportlern unterhalten, motivieren und vergleichen können. Zur Motivation dient auch die Anfeuern-Funktion. Damit können Benutzer auf der Webseite Sportler, die gerade Live beim Sporteln unterwegs sind, per Mausklick mit einem vordefinierten Sound oder per Sprachaufzeichnung anfeuern. Diese Anfeuerungen werden dann direkt am Handy des Sportlers über Kopfhörer oder den eingebauten Lautsprecher ausgegeben. Außerdem kann der Webseitenbesucher, zusätzlich zu den aktuellen Statistikdaten, auf einer Landkarte genau sehen, wo sich der Sportler gerade befindet.

2.8.1 Produktpalette

Zusätzlich zu dem ursprünglichen namensgebenden Produkt gibt es inzwischen einige weitere Applikationen von *runtastic*. Neben speziell auf gewisse Sportarten optimierte Abwandlungen, wie zum Beispiel für Mountainbiking, Roadbiking und Walking, zählen dazu unter anderem auch eine Anwendung zum Schritte zählen, eine, mit der man sein Idealgewicht feststellen kann, und eine zur Messung der aktuellen Seehöhe für Wanderer. Außerdem können mithilfe einer Liegestütz-Anwendung innerhalb von wenigen Wochen 100 Liegestütze an einem Stück durch einen speziellen Trainingsplan erreicht werden. Sämtliche Anwendungen sind sowohl als kostenfreie, werbegestützte und im Funktionsumfang eingeschränkte Lite-Versionen sowie als kostenpflichtige Pro-Version verfügbar. Die Anwendungen sind für die Plattformen *iPhone*, *Android*, *Windows Phone* sowie *BlackBerry* in den jeweiligen Marktplätzen erhältlich.

Ausgehend von den Handyanwendungen wurde die Produktpalette in Richtung Hardware ausgeweitet, die das Nutzungserlebnis für Sportler ergänzen. Dazu gehören unter anderem ein Herzfrequenzmesser und ein Armband zur Befestigung des Mobiltelefons während einer Sportaktivität. Außerdem wurde eine eigene GPS-Uhr entwickelt, um die Nutzung der *runtastic* Plattform auch ohne den Besitz eines Smartphones zu ermöglichen.

2.8.2 Infrastruktur

runtastic zählt mit insgesamt über 14 Millionen Downloads zum Kreis der erfolgreichsten Hersteller für Smartphone-Anwendungen [19]. Mehr als sechs Millionen Menschen registrierten sich bereits als Benutzer.Diese laden täg-

lich mehr als 100.000 Sportaktivitäten auf das Webportal.[28] Um diese Anzahl an Anfragen zu bewältigen, bedarf es einer dementsprechend dimensionierten Infrastruktur. Sämtliche Server sind derzeit bei *Hetzner*[29] in Deutschland untergebracht, unterstützt wird die Auslieferung statischer Daten durch *Amazon CloudFront*. Die Webseite ist auf Basis von *Ruby on Rails*[30] implementiert, die mithilfe von *JRuby* in *Apache Tomcat*[31] auf mehreren *Linux-Servern* läuft und über *HAProxy*[32] lastverteilt ausgeliefert wird.

Als Datenbank wird hauptsächlich *MySQL*[33] verwendet, zusätzlich ist für spezielle Zwecke, wie zum Beispiel die Speicherung der GPS-Aufzeichnungen, *Cassandra* im Einsatz. Die Verbindung zwischen der Webplattform und den mobilen Endgeräten übernimmt eine *Java Enterprise Anwendung*, die auf einem *Java Glassfish*[34] Servercluster läuft. In Abb. 2.4 ist eine schematische Darstellung der Infrastruktur zu sehen, die bei *Hetzner* untergebracht ist.

[28] Stand: Oktober 2012, Quelle: *runtastic*
[29] http://www.hetzner.de/
[30] http://rubyonrails.org/
[31] http://tomcat.apache.org/
[32] http://haproxy.1wt.eu/
[33] http://www.mysql.com/
[34] http://glassfish.java.net/

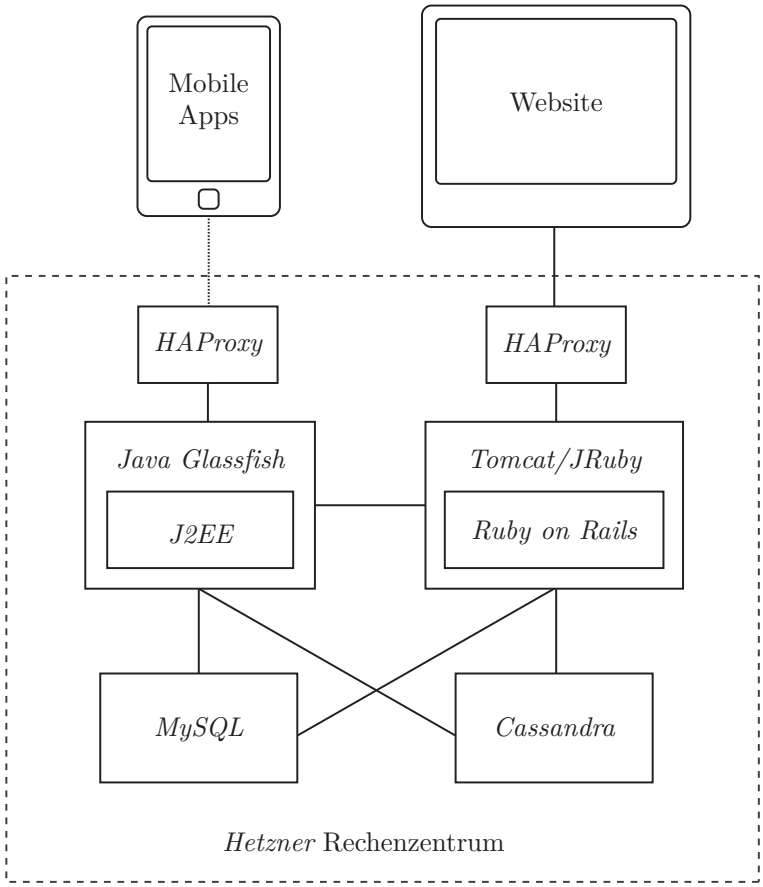


Abbildung 2.4: Schematische Darstellung der *runtastic* Infrastruktur bei *Hetzner* im Rechenzentrum in Deutschland.

Kapitel 3

Verwandte Systeme und Anforderungen

Die Aufgabe eines *Gamification Services* ist, Berechnungen und Entscheidungen auf Basis von eingehenden Aktionen und Daten durchzuführen, die Resultate zu persistieren und verfügbar zu machen, sowie auf Basis der Resultate weitere Aktionen anzustoßen. In diesem Kapitel werden sowohl die Anforderungen an ein *Gamification Service* allgemeiner Natur als auch aus der Sicht von *runtastic* beschrieben. Außerdem wird gezeigt, welche Systeme bereits für den Einsatz als *Gamification Service* erhältlich sind bzw. für diesen Einsatzzweck als Ausgangsbasis verwendet werden könnten.

3.1 Verwandte Systeme

In diesem Abschnitt werden *Gamifcation Services* vorgestellt, die bereits erhältlich sind. Außerdem werden Systeme angeführt, deren Ansätze zur Aufgabenabarbeitung dem in dieser Arbeit vorgestellten *Gamification Service* ähnlich sind.

3.1.1 Gamification Systeme

Gamification Systeme können als Services von kommerziellen Anbietern gemietet oder als Open Source Lösung eigens integriert werden. Allerdings bietet sich im Open Source Bereich derzeit nur ein System zur direkten Verwendung als *Gamification Service* an.

Kommerzielle Lösungen

Mit zunehmender Popularität des Themas Gamification entstanden auch immer mehr Unternehmen, die Dienstleistungen in diesem Bereich anbieten. Diese Unternehmen bieten Plattformen als Service an, mit denen Gamification Elemente in eigene Web- und Mobilapplikationen integriert werden

können, ohne die dafür benötigten Technologien und Webservices selbst entwickeln und hosten zu müssen [9].

Die Integration erfolgt üblicherweise mit vordefinierten Widgets, die an die gewünschte Funktionalität und das gewünschte Design angepasst und in eine Webseite oder App eingebettet werden können. *APIs* der Anbieter ermöglichen es, die eigene Anwendung an die jeweilige Gamification Plattform anzubinden, um z. B. bestimmte durch Nutzer ausgelöste Aktionen zu melden und aktuelle Informationen für einem User auszulesen. Mithilfe einer Weboberfläche können je nach Plattformanbieter u. a. Abläufe, Ziele, Bestenlisten und Belohnungen festgelegt und angepasst, sowie über Statistiken die Auswirkungen durch die verwendeten Gamification Elemente festgestellt werden.

Folgende Unternehmen bieten u. a. *Gamification-Plattformen* als Service an:

- *badgeville.com*[1]
- *BigDoor*[2]
- *Bunchball*[3]
- *Gigya*[4]

Wie diese Plattformen bei den genannten Unternehmen im Detail technisch aufgebaut sind konnte nicht festgestellt werden. Diese Informationen sind öffentlich nicht zugänglich, somit konnten nur die allgemeinen Produktunterlagen als Grundlage für das Verständnis der Funktionalität herangezogen werden.

Open Source Lösungen

Obwohl das Thema Gamification ein starkes Interesse nach kommerziellen Lösungen erfahren hat, konnte trotz intensiver Recherche nur eine Open Source Lösung für diesen Bereich gefunden und analysiert werden.

userinfuser: *userinfuser* ist ein als Open Source Software zur Verfügung gestelltes *Gamification Service* [69]. Dieses Service kann sowohl selbst gehostet als auch bei der unterstützenden Entwicklungsfirma *cloudcaptive*[5] als Cloud-Lösung verwendet werden. Mithilfe von Clientbibliotheken, die in *PHP*, *Python*, *Java* und *Ruby* ebenfalls als Open-Source verfügbar sind, kann das System in eigene Anwendungen eingebunden werden. Dazu unterstützt es ähnlich wie die kommerziellen Lösungen die Verwendung von Widgets, die

[1] http://www.badgeville.com
[2] http://www.bigdoor.com
[3] http://www.bunchball.com
[4] http://www.gigya.com
[5] http://www.cloudcaptive.com

einfach in eine Webseite eingebunden werden können [68]. Mehr dazu findet sich in [69].

3.1.2 Verteilte Taskverarbeitungssysteme

Apache Hadoop

Apache Hadoop ist ein Framework zur Bearbeitung von großen Datenmengen auf verteilten Systemen und implementiert dabei das *MapReduce* Verfahren. Es ist für die Berechnung von langlebigen Bearbeitungsjobs ausgelegt und nicht sinnvoll für Echtzeitberechnungen einsetzbar. Für die Organisation der Tasks verwendet *Apache Hadoop* das *Apache ZooKeeper* Service. Zur Datenhaltung kann u. a. auch *Apache Cassandra* herangezogen werden. Mehr dazu findet sich in [21].

Storm

Storm ist ein von Nathan Marz entwickeltes System zur Echtzeitberechnung von Datenströmen, welches von Twitter[6] als Open-Source Software verfügbar ist [64]. Die Basis von *Storm* ist in Java entwickelt. Berechnungstasks können allerdings in jeder beliebigen Programmsprache geschrieben werden. Storm verwendet eine Topologie bestehend aus so genannten *Spouts* und *Bolts* um den Pfad einer Berechnung zu definieren. Als *Spouts* werden Knoten in der Topologie bezeichnet, die Daten in den Ablauf einleiten. Als *Bolts* werden hingegen die einzelnen Berechnungstasks bezeichnet.

Sobald eine Topologie definiert ist, kann diese auf einen *Storm* Cluster hochgeladen werden. Dieser Cluster besteht aus dem *Nimbus* Server, der sich um das Hochladen der Topologie, die Verteilung der Tasks und um das Monitoring kümmert, einem *ZooKeeper* Cluster, das zum Management der Tasks verwendet wird und mehreren *Supervisor* Servern, die sich um die Ausführung von einzelnen Tasks kümmern [65]. Die Architektur von Storm ist oberflächlich vergleichbar mit Hadoop, weil es in beiden Architekturen einen Masterknoten gibt. Während jedoch Hadoop einzelne Jobs berechnet und diese danach beendet, läuft bei Storm der Datenstrom kontinuierlich [16].

Resque

Resque ist eine Ruby Bibliothek, die es ermöglicht, jegliche Aufgaben asynchron abzuarbeiten. Die auszuführenden Tasks werden in einer Klasse implementiert und während der Laufzeit einer Applikation mithilfe eines Befehls asynchron in die *Task Queue* aufgenommen. Zur Abarbeitung der Tasks gibt es die so genannten *Resque Worker*. Diese können als eigener Prozess auf beliebig vielen Servern gestartet werden und arbeiten dann die für sie

[6] http://www.twitter.com

definierten Warteschlangen ab. Bei Resque wird für die Abarbeitung jedes Jobs ein eigener Kindprozess für die Ausführung erzeugt, damit Tasks sich gegenseitig nicht beeinflussen können [56]. Dies hat allerdings den Nachteil, dass für jeden Prozess ein eigener Speicherbereich angelegt wird und die Speicherauslastung mit der Anzahl der Worker ansteigt.

Resque verwendet zur Haltung seiner Warteschlange die *Queue* Möglichkeiten der Datenbank *Redis*[7]. Die *Queues* werden innerhalb von *Redis* im RAM-Speicher des Servers gehalten und periodisch, abhängig von der Konfiguration, mehr oder weniger oft persistiert [54]. Wie bei anderen Datenbanksystemen kann bei Serverausfällen und Datenverlusten mit einer Datenreplikation auf mehreren Servern abgeholfen werden [55].

In der Verwendung eignet sich *Resque* vor allem für Ansätze, bei denen nach Start eines asynchronen *Tasks* keine Antwort von diesem benötigt wird und die Aufgaben keine Abhängigkeiten zu anderen *Tasks* besitzen.

Sidekiq

Sidekiq ist ein mit *Celluloid* implementiertes System zur Taskbearbeitung, das mit *Resque* kompatibel ist, weil es dasselbe Nachrichtenformat und dieselbe Datenbank (*Redis*) verwendet. Im Gegensatz zu *Resque* verwendet *Sidekiq* keine eigenen Prozesse für jeden Task, sondern packt diese in Threads, weshalb die Speicherauslastung für dieselbe Anzahl an Worker weitaus geringer ausfällt. Um diese Vorteile ausspielen zu können, ist allerdings die Nutzung eines Ruby Interpreters unabdingbar, welcher kein Global Interpreter Lock aufweist, wie z. B. *JRuby*. Mehr dazu findet sich in [62].

3.2 Allgemeine Anforderungen

Damit ein *Gamification Service* in eine bestehende Infrastruktur integrieren werden kann, sollten die in diesem Abschnitt beschriebenen Anforderungen erfüllt sein:

3.2.1 Skalierbarkeit

Damit ein *Gamification Service* sinnvoll für eine beliebige Anzahl von gleichzeitigen Anfragen einsetzbar ist, muss es flexibel für die jeweils zu bewältigenden Lasten skaliert werden können. Dabei ist von Bedeutung, dass sich keiner der eingesetzten Dienste als Flaschenhals für die Erweiterung der Kapazitäten erweisen darf. Dazu soll sich das *Gamification Service* selbst, sowie sämtliche verwendete Systeme, auf mehreren Servern beliebig einsetzen lassen, um durch Erweiterungen der Serverkapazitäten einen höheren Durchsatz erzielen zu können, ohne Architekturänderungen an der Software vollziehen zu müssen.

[7] http://www.redis.io

3.2.2 Unabhängigkeit

Das *Gamification Service* soll ein eigenes, in sich abgeschlossenes System darstellen, welches von außen nur durch eine *REST-Schnittstelle* angesprochen werden kann. Sämtliche zur Berechnung benötigten Informationen müssen über die Schnittstelle übermittelt werden, damit keine Abhängigkeiten zu anderen Systemen entstehen und das Service unabhängig implementiert und genutzt werden kann.

3.2.3 Antwortzeit

Die Antwortzeit des *Gamification Services* spielt eine entscheidende Rolle, damit das System für einen User sinnvoll nutzbar ist. Lange Wartezeiten wirken sich negativ auf die Akzeptanz einer Applikation bei den Benutzern aus [10, Kap. 2]. Deshalb sollten die Antwortzeiten des Systems den Erwartungen der User der jeweiligen Plattform entsprechen, für die das *Service* eingesetzt wird.

3.2.4 Datenkonsistenz

Die Daten, die in das *Gamification Service* fließen und dort berechnet werden, müssen immer in einem konsistenten Zustand verfügbar sein, damit sich keine Fehler durch inkorrekte Ergebnisse einschleichen können. Insbesondere bei offensichtlichen Berechnungsfehlern würden betroffene User schnell bemerken, dass etwas nicht stimmt. Deshalb sind geeignete Sicherheitsmaßnahmen zu treffen, um inkonsistente Zustände soweit wie möglich ausschließen zu können.

3.2.5 Hochverfügbarkeit

Damit ein *Gamification Service* auch bei einem Serverausfall ohne Verfügbarkeitsunterbrechung verwendet werden kann, ist es notwendig, alle dafür verwendeten Systeme ausfallsicher auszulegen. Außerdem soll die Verfügbarkeit auch dann gewährleistet sein, wenn aus unerwarteten Gründen eine höhere als übliche Auslastung des Services besteht.

3.2.6 Erweiterbarkeit

Das *Gamification Service* soll auf Basis eines Frameworks erweitert werden können, um Anforderungen für Abläufe, Inhalte, Regeln und Aktionen ohne Eingriffe in das Grundsystem abbilden und implementieren zu können. Die Implementierung neuer Funktionalität soll dabei soweit wie möglich abstrahiert werden, damit nur Kenntnisse über relevante Schnittstellen benötigt werden. Außerdem soll das *Gamification Service* ermöglichen, Anpassungen und Erweiterungen am Grundsystem einfach zu implementieren, ohne gröbere Änderungen an der Architektur durchführen zu müssen.

Abstraktion von Inhalten

Aus der Rolle eines Game-Designers soll es nicht nötig sein, über die eigentliche Abarbeitung von Anfragen an ein *Gamification Service* im Detail Bescheid zu wissen. Dieser sollte sich nur mit der Zusammenstellung von Regelsätzen beschäftigen müssen, die das Service dann dazu nützt, z. B. *Badges* und *Records* für den Benutzer der Applikation zu berechnen.

Abstraktion von Erweiterungen

Aus der Sicht eines Erweiterungsprogrammierers soll es ebenfalls nicht nötig sein, die Details der Abarbeitung innerhalb eines *Gamification Services* zu kennen. Dieser braucht nur die Schnittstellen zu kennen, die für seine Erweiterung des Systems relevant sind.

3.3 Anforderungen von runtastic

Zusätzlich zu den allgemeinen Anforderungen hat *runtastic* für seinen Einsatzzweck weitere Anforderungen, damit ein *Gamification Service* sinnvoll in die bestehende Infrastruktur integriert werden kann. Das Szenario für die hauptsächliche Verwendung des *Services*, das auch in Abb. 3.1 dargestellt wird, sieht folgendermaßen aus:

1. Ein Sportler zeichnet seinen Lauf mit der *runtastic* App auf und übermittelt die *Sport Session* an das *runtastic* Webservice für mobile Endgeräte.
2. Dieser Webservice Server leitet die Session unter anderem an das *Gamification Service* weiter.
3. Das *Gamification Service* persistiert die zur *Session* gehörenden Daten und berechnet auf Basis der *Session* alle davon abhängigen Erfolge, wie z. B. *Abzeichen* und *Rekorde*.
4. Nach Abschluss aller notwendigen Berechnungen antwortet das *Gamification Service* mit dem aktuellen Zustand der Erfolge an den *runtastic Webservice* Server, welcher die Ergebnisse zurück an das Mobiltelefon sendet, damit die *runtastic* App dem User seine neuen Erfolge anzeigen kann.
5. Außerdem sollte das *Gamification Service* die Sendung von E-Mails oder *Push-Notifications*[8] zum User einleiten, sofern diese aufgrund der Berechnungsergebnisse als weiterführende Aktionen ermittelt wurden.

[8] http://en.wikipedia.org/wiki/Push_technology

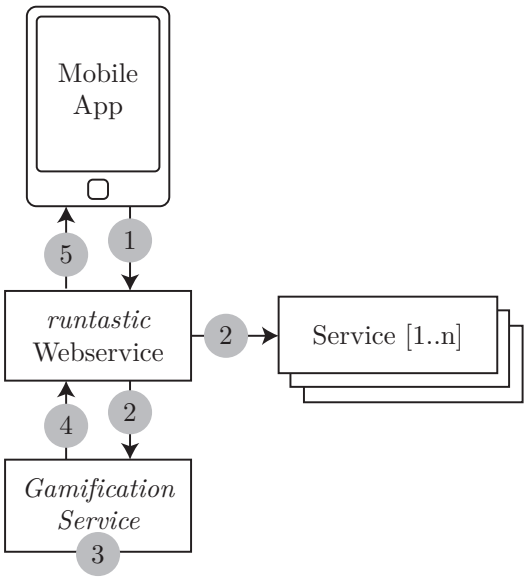


Abbildung 3.1: Szenario für die hauptsächliche Verwendung des *Gamification Services*.

3.3.1 Maximale Antwortzeit

Die maximale Antwortzeit des *Services* spielt eine entscheidende Rolle, damit es sinnvoll in einer verteilten Infrastruktur wie bei *runtastic* eingesetzt werden kann. Dabei ist zu beachten, dass das *Gamification Service* nur ein Teil der Systeme ist, die z. B. bei einem Upload von *Sport Sessions* abgefragt werden. Eine ungewöhnlich lange Antwortzeit des *Gamification Services* würde auch das Gesamtsystem verlangsamen.

Unter Einberechnung der Transferdauer zwischen dem Mobiltelefon des Users und den Webservern, sowie den zusätzlichen für die Antwortaufbereitung verwendeten Systemen, sollte eine Anfrage an das *Gamification Service* bei *runtastic* in der Regel maximal 500 ms dauern [10, Kap. 2]. Insbesondere bei Anfragen, die das Ergebnis der Berechnungen erwarten, sollten die dafür benötigten Operationen innerhalb des *Services* so schnell und effizient wie möglich ausgeführt werden.

3.3.2 Vermeidung von Logik am Client

Die vom Service zur Verfügung gestellte Schnittstelle soll sowohl von mobilen Endgeräten auf verschiedenen Betriebssystemen als auch von einem Webserver zur Darstellung der Daten auf einer Webseite angesprochen werden können. Dabei ist darauf zu achten, dass die jeweilige Client-Applikation

sowenig Logik wie möglich zu implementieren hat und sich hauptsächlich auf die Darstellung und Aktualisierung der Inhalte beschränken soll.

3.3.3 Vielseitigkeit

Das *Gamification Service* soll grundsätzlich auf den Einsatz für Gamification-spezifische Berechnungen optimiert sein. Allerdings wäre es sinnvoll, wenn das System offen für Erweiterungen ist, die ähnliche Abläufe für andere Einsatzzwecke, wie z. B. Onlinespiele oder statistische Berechnungen in Echtzeit, nutzbar machen könnten.

Kapitel 4

Konzept

In diesem Kapitel wird das Konzept für ein *Gamification Service* erläutert. Mit dem Konzept wird versucht, die Anforderungen aus Kapitel 3 zu erfüllen und gleichzeitig einen Leitfaden für eine Implementierung zu geben.

4.1 Prinzipien und Funktionalitäten

Damit die Umsetzung des *Gamification Services* in einem festgelegten Rahmen vonstatten gehen konnte, wurde eine Sammlung von Prinzipien und Funktionalitäten definiert, die bei der Implementierung angewendet wurden.

4.1.1 Trennung von Service und Framework

Das *Service* stellt die Schnittstellen nach außen zur Verfügung und kümmert sich um die Abarbeitung von Anfragen, während das *Framework* für die einzelnen Aufgaben, Inhalte und Berechnungen zuständig ist. Abgesehen von Bereichen, bei denen Funktionalitäten von *Service* und *Framework* voneinander abhängig sind, kann mit dieser Trennung an beiden Komponenten unabhängig weiterentwickelt werden. Zudem ist eine gemeinsame *Core* Komponente sinnvoll, um darin geteilte Funktionalitäten an einem Ort unterzubringen. Diese Vorgehensweise der Teilung in mehrere Komponenten, wie sie in Abb. 4.1 dargestellt ist, wird bei größeren *Ruby* Projekten wie z. B. *Rails*[1] oder *Travis CI*[2] ebenfalls verwendet.

4.1.2 Parallelisierung

Damit die Anforderungen an die Performance und Antwortzeiten des *Gamification Services* eingehalten werden können, wurde darauf geachtet, dass Vorgänge innerhalb des Systems soweit wie möglich parallelisiert werden können. Dazu werden alle für eine Anfrage benötigten Operationen soweit wie

[1] http://www.github.com/rails/rails
[2] https://github.com/travis-ci

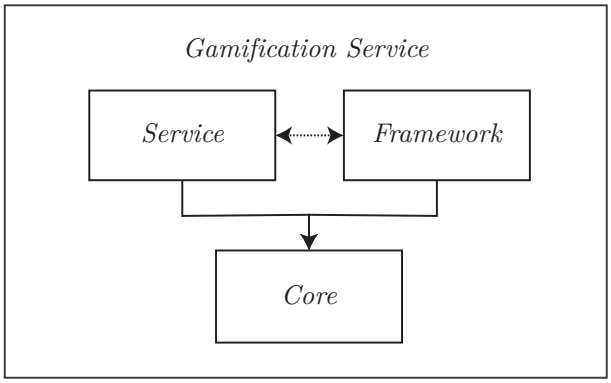


Abbildung 4.1: Unterteilung der Komponenten im *Gamification Service*.

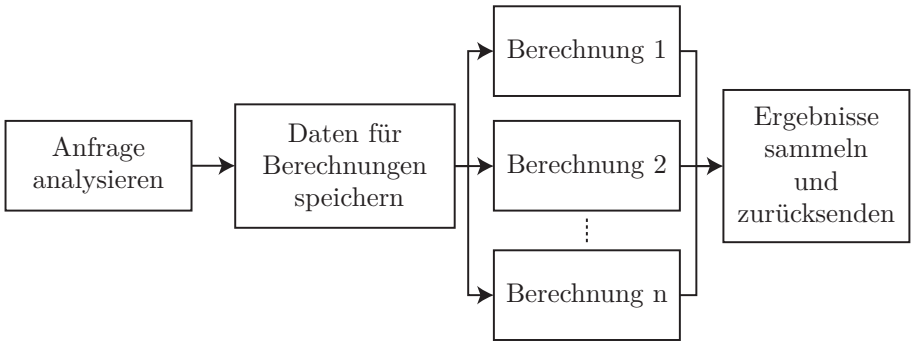


Abbildung 4.2: Parallelisierung im *Gamification Service*.

möglich in unabhängige Teile, so genannte *Tasks*, zerlegt, um diese parallel ausgeführen zu können. Mithilfe einer *Message Queue* können *Tasks* auch auf mehrere Server verteilt werden, um eine Parallelisierung über Servergrenzen hinaus zu erreichen. Falls Abhängigkeiten zwischen den einzelnen Aufgaben bestehen, werden diese aufgelöst, indem die Aufgaben statt parallel, in ihrer Abhängigkeitsreihenfolge ausgeführt werden. Dadurch wird sichergestellt, dass jede Aufgabe immer die benötigten Daten verfügbar hat, wie in Abb. 4.2 dargestellt wird.

Datenhaltung und -konsistenz

Ein kritischer Bereich bei der Parallelisierung der Datenhaltung ist die Datenkonsistenz. Damit diese soweit wie möglich gewährleistet werden kann benötigt es wirksame Sicherheitsmechanismen, die Datenkorruption durch gleichzeitige Bearbeitung von Inhalten verhindern können:

Locking: Mithilfe von *Locks*, die auf allen Servern des *Gamification Services* gültig sind, kann gewährleistet werden, dass Aufgaben und Änderungen in der Datenbank zum selben Zeitpunkt nur genau einmal ausgeführt werden.

Versionierung: Innerhalb des *Gamification Services* werden Daten nie überschrieben, sondern immer in neuen Versionen abgespeichert. Damit wird vermieden, dass bei Berechnungen immer eine komplette Historie der Berechnungsschritte gespeichert werden muss, da immer eine alte Version der Daten vorhanden ist, die beim Berechnen von Zustandsänderungen zum Wiederherstellen des Ursprungszustandes verwendet werden kann.

Threadsicherheit: Damit Komponenten innerhalb des *Gamification Services* von mehreren Threads benutzt werden können, wurde darauf geachtet, dass diese threadsicher implementiert sind. Gerade bei externen Komponenten muss dies nicht immer der Fall sein, weshalb dies als wichtiges Kriterium bei der Auswahl der richtigen Bibliotheken gespielt hat.

4.1.3 Datentypen als Ressourcen

Innerhalb des *Gamification Frameworks* werden alle extern ansprechbaren Datentypen Ressourcen genannt. Ressourcen können als *Framework Erweiterungen* implementiert werden und umfassen derzeit die Typen *User*, *Run Session*, *Static Quest*, *Badge* und *Record*. Von einem Ressourcentyp gibt es wiederum mehrere verschiedene einzelne Ressourcen, die sich anhand einer numerischen Ressourcenidentifikationsnummer, kurz Ressourcen ID, unterscheiden. Der Ressourcentyp ergibt gemeinsam mit der Ressource ID eine eindeutige Ressource im System.

Beziehungen und Abhängigkeiten

Ressourcen können in Bezug zueinander stehen, indem entweder eine als Entität verwendete Ressource oder eine normale Ressource auf eine andere Ressource verweist. Im Fall einer Beziehung zu der normalen Ressource handelt es sich üblicherweise um eine Abhängigkeit, bei der eine Ressource an der Erstellung, Änderung oder Löschung der Inhalte einer anderen Ressource interessiert ist. Dieses Interesse an der Veränderung einer anderen Ressource wird durch die interessierte Ressource bei der Initialisierung des *Gamification Services* in einer Abhängigkeitsliste deponiert, damit bei der Ausführung einer Anfrage, die eine Ressourcenveränderung zufolge hat, die abhängige Ressource darüber informiert wird. Diese Abhängigkeiten können sowohl auf alle Ausprägungen eines Ressourcentyps als auch auf nur eine bestimmte Ressource (Paar bestehend aus Ressourcentyp und Ressourcen ID) festgelegt werden. Falls sich die Abhängigkeit auf den gesamten Ressourcen-

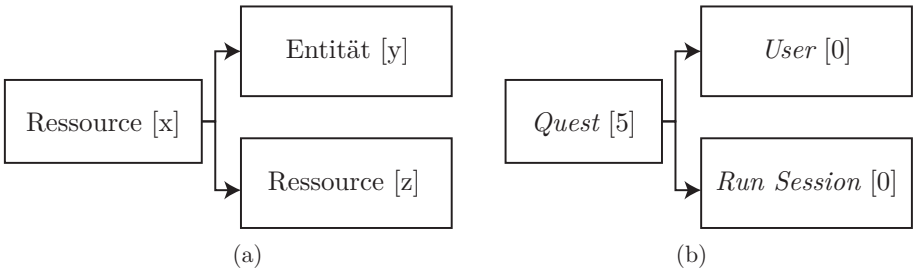


Abbildung 4.3: Abhängigkeiten innerhalb des *Gamification Services*. Allgemeine Darstellung der Abhängigkeit einer Ressource von einer Entität und einer weiteren Ressource (a). Beispiel, bei dem ein *Quest* zu jeder übermittelten *Run Session* für jeden *User* berechnet wird (b).

typ bezieht, wird für die Ressourcen ID der Wert 0 verwendet. Damit eine abhängige Ressource in die Abarbeitung einer Anfrage einbezogen wird, muss sowohl die Entität als auch die Ressource der Abhängigkeit übereinstimmen, wie in Abb. 4.3 (a) allgemein und anhand eines Beispiels in Abb. 4.3 (b) dargestellt wird.

Ressourcen und Entitäten

Entitäten sind als Typen intern ebenfalls Ressourcen, die dazu verwendet werden, um eine Ressource mit einer anderen Ressource in Bezug zu stellen. Derzeit wird für das *Gamification Service* üblicherweise nur die User Ressource als Entität verwendet, um die dazugehörigen anderen Ressourcen wie *Run Sessions*, *Badges* und *Records* zu verändern oder abzufragen. In Zukunft könnten weitere Entitäten, wie Gruppen oder andere übergeordnete Instanzen, Verwendung finden. Deswegen wurden Beziehungen nicht alleine auf den Typ *User* konzipiert und das System damit offen für Anwendungsgebiete mit ähnlichen Anforderungen gehalten.

Verwendung für Berechnungen

Jede Ressource kann ebenfalls als Datenquelle für Berechnungen fungieren. Dazu wird sie als Datenursprung in eine Regelkette aufgenommen und damit zu einer Abhängigkeit für den Container, der die Regelkette enthält. Die Verwendung von Ressourcen für Berechnungen wird in Abb. 4.4 exemplarisch dargestellt.

Verwendete Ressourcentypen

Um die Berechnung von *Quests* auf Basis von übermittelten *Run Session* Daten und die Abfrage von *Badges* und *Records* für einen *User* zu ermöglichen,

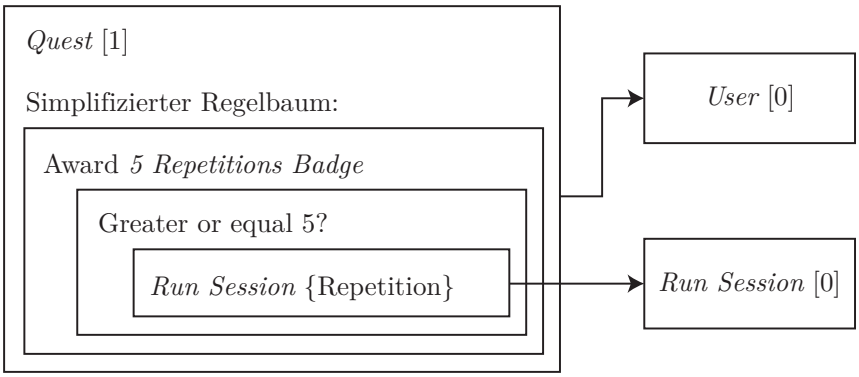


Abbildung 4.4: Der *Quest 1* enthält einen vereinfacht dargestellten Regelbaum, der die *Run Session* Ressource als Datenquelle für die Vergabe eines *Badges* enthält. Weil die Ressource in dem Regelbaum enthalten ist, wird sie automatisch zu einer Ressourcenabhängigkeit des *Quests*. Ein *Quest* ist automatisch von allen *Usern* abhängig, weshalb diese Entität nicht im Regelbaum vorkommen muss.

wurden diese als Ressourcentypen konzipiert.

User: Die *User* Ressource wird als Entität und als Platzhalter für zukünftige userbezogene Daten verwendet, die für Berechnungen herangezogen werden können. Ein *User* muss deswegen derzeit auch nicht im *Service* angelegt werden, um für Abfragen verwendet werden zu können. Alle anderen Ressourcen stehen bei deren Speicherung in Bezug zu einem *User*.

Run Session: Die Run Session Ressource wird zum Speichern und Auslesen der vom Client für einen bestimmten *User* übermittelten Sportdaten verwendet. Diese Daten werden von abhängigen Ressourcen, wie z. B. *Static Quests*, zu Berechnungen herangezogen.

Static Quest: Die *Static Quest* Ressource fungiert als Container für sämtliche Berechnungen von Missionen, die mit dem *Gamification Service* abgebildet werden. Weil diese Aufgaben in einer statischen Definitionsdatei mithilfe einer *DSL* beschrieben werden, sind diese *Quests* statisch. In Zukunft soll es auch möglich sein, durch User definierte Missionen und Wettkämpfe zu erstellen, die dann dynamisch aus der Datenbank geladen werden.

Mithilfe eines *Templates* wird definiert, welche Erweiterungstypen in welchen Bereichen vorkommen dürfen (s. Abb. 4.5), und wie die Abläufe innerhalb des *Static Quests* bei einer Berechnung ausgeführt werden. Dieses *Template* wir dann als Grundlage für die einzelnen *Static Quest* Definiti-

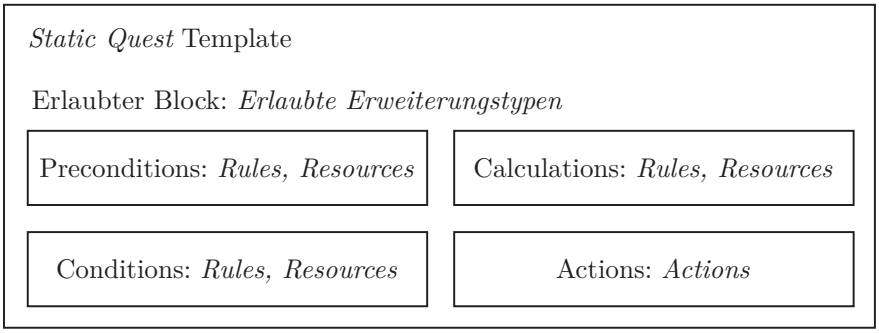


Abbildung 4.5: Stellt dar, welche Blöcke innerhalb von *Static Quest* Templates verwendet werden können und welche Erweiterungen diese aufnehmen können.

onsdateien herangezogen, in denen die Regeln für die einzelnen Missionen definiert werden. Kommt ein Ressourcentyp als Datenquelle in einem Regelbaum vor, so wird dies automatisch als Abhängigkeit erkannt und für den definierten *Quest* beim Abhängigkeitsmanager registriert (s. Abb. 4.4). Deshalb wird dieser bei einer Anfrage, welche die besagte Abhängigkeit enthält, den *Static Quest* darüber informieren, dass sich die Ressource geändert hat und die Mission somit neuberechnet werden muss.

Badge und Record: Die *Badge* und *Record* Ressourcen werden dazu verwendet, die *Badges* und *Records* eines *Users* auszugeben. Beide Ressourcen kümmern sich um die Verknüpfung statischer Daten mit dem für den jeweiligen User berechneten Zustand. Die statischen Daten können z. B. Metainformationen zur Darstellung, Pfade zu Bildern oder Übersetzungstexte enthalten. Die eigentliche Vergabe von *Badges* und *Records* wird in einem *Static Quest* vorgenommen. Dieser enthält einen Regelbaum, in den die jeweilige *Badge-* bzw. *Record Action* eingebunden wird. Sobald die zur Vergabe benötigten Regeln erfüllt sind, werden die *Badges* und *Records* verteilt.

4.1.4 Domänenspezifische Sprache

Inhalte wie *Quests*, *Badges* und *Records* werden im *Gamification Service* von einem *Game Designer* mithilfe einer dafür geschaffenen *Domain Specific Language* erstellt. Diese Sprache enthält genau die Bausteine, welche für die Erstellung der Inhalte benötigt werden. Sämtliche Inhalte werden in jeweils einzelnen Dateien abgelegt, in denen sich die Definitionsbeschreibungen befinden. Für *Badges* und *Records* können sämtliche Metadaten für die clientseitige Darstellung und die Pfade zu Bildern mithilfe der Definitionsdatei beschrieben werden. Außerdem wird die Lokalisierung von Texten, die

an den Client gesendet werden, mithilfe einer gemeinsamen Sprachdatei für alle statischen Ressourcen ermöglicht. Darin können für jede Ressource beliebige Texte definiert werden, die automatisch bei einer Anfrage ausgeliefert werden.

DSL für Definitionsdateien

Mithilfe der *domänenspezifischen Sprache für Definitionsdateien* ist es im *Gamification Service* möglich, das Regelwerk beliebiger Berechnungsabläufe mit *DSL*-Befehlen auszugestalten. Die *DSL* wertet Regelbäume von innen nach außen aus und übermittelt die Ergebnisse auf diesem Weg weiter. Außerdem ermöglicht die Sprache die Verknüpfung von Blöcken, damit Ergebnisse von einem Block in andere Blöcke einfließen können. Durch dieses Prinzip kann das mehrfache Schreiben von gleichen Regelsätzen vermieden werden. Folgende Arten von Erweiterungen wurden für das *Gamification Service* definiert:

Rules: Dabei handelt es sich um Regeln, die Überprüfungen und Berechnungen innerhalb eines Regelbaums durchführen können.

Resources: Ressourcen werden dazu verwendet, Werte aus den übermittelten Daten für die Berechnungen zu ermitteln. Dabei handelt es sich um dieselben Ressourcen, die auch für die Datentypen in Abschn. 4.1.3 vorgestellt wurden, weil diese automatisch für die *DSL* übernommen werden.

Actions: Aktionen werden dazu verwendet, um *Badges* oder *Records* zu vergeben.

DSL für Erweiterungen

Mithilfe einer *domänenspezifischen Sprache für Erweiterungen* ist es im *Gamification Service* möglich, Erweiterungen für die *DSL für Definitionsdateien* zu implementieren. Diese *DSL für Erweiterungen* erlaubt es ohne Kenntnis des Einsatzzwecks Regeln, Berechnungs- und Datenbeschaffungsmethoden umzusetzen. Nur Kenntnisse über die eingehenden und ausgehenden Daten sowie die möglichen Zustände sind nötig, damit eine Erweiterung programmiert werden kann.

4.2 Systemaufbau

Die Komponenten des *Gamification Services* setzen sich aus ausgewählten Systemen zusammen, die in gegenseitigem Einklang stehen, um die in Kap. 3 gewünschten Anforderungen wie z. B. Skalierbarkeit, Hochverfügbarkeit und Datenkonsistenz erfüllen zu können. In diesem Abschnitt wird begründet, weshalb die jeweiligen Systeme zur Implementierung des *Gamification Ser-*

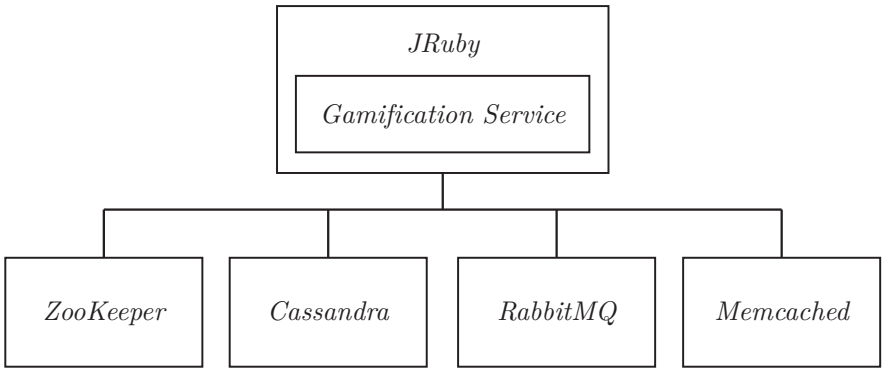


Abbildung 4.6: Systemaufbau des *Gamification Services*.

vices ausgewählt wurden. Die Abb. 4.6 bietet zusätzlich einen Überblick zu den verwendeten Komponenten.

4.2.1 Ruby

Als Programmiersprache wird *Ruby* verwendet, weil bei *runtastic* zum Zeitpunkt der Konzeption und Implementierung diese Sprache bevorzugt verwendet wurde und sie sich für die Umsetzung von Webapplikationen und *Domain Specific Languages* gut eignet (s. Abschn. 2.7.3). Durch die Verwendung von *JRuby* kann zudem die Problematik des *Global Interpreter Locks* umgangen werden, weshalb eine Nutzung aller verfügbaren Ressourcen eines Servers möglich ist.

4.2.2 Apache Cassandra

Cassandra wurde zum Zeitpunkt der Konzeption und Implementierung bei *runtastic* ebenfalls eingesetzt. Die bereits vorhandene Infrastruktur und Erfahrungen mit *Cassandra* waren ein Hauptgrund für die Verwendung, weil damit für den kritischen Bereich der Datenhaltung kein Neuland betreten werden musste. Die eingeschränkten Abfragemöglichkeiten im Vergleich zur *MySQL* Datenbank, die bei *runtastic* ebenfalls eingesetzt wird, wurden in Kauf genommen, weil diese für die vorgesehenen Anwendungsfälle als ausreichend empfunden wurden. Dagegen ist die Skalierbarkeit von *Cassandra* ein wichtiger Eckpfeiler, damit das System auch mit einer anwachsenden Anzahl an Daten und Abfragen mitwachsen kann, ohne dass an der Architektur der Applikation etwas geändert werden muss.

4.2.3 Apache ZooKeeper

Damit das *Gamification Service* auf mehreren Servern gleichzeitig ausgeführt werden kann, ist eine Synchronisation der beteiligten Server unentbehrlich. Außerdem ist die Sicherstellung der Datenkonsistenz ein weiterer wichtiger Faktor, weil *Cassandra* kein *Locking* und auch keine *Transaktionen* auf Datenbankebene bietet. *ZooKeeper* ist bestens geeignet, weil damit alle genannten Szenarien abgedeckt werden können. Mithilfe der *ZooKeeper* Primitiven ist es u. a. möglich, *Locks* abzubilden. Diese *Sperren* werden im *Gamification Service* verwendet, um zu verhindern, dass während eines Zugriffes auf *Cassandra* dieselben Daten von jemandem anderen bearbeitet werden. Außerdem wird mit einem weiteren *Lock* unterbunden, dass zur selben Zeit derselbe *Task* auf einem anderen Server ausgeführt wird. In *ZooKeeper* werden für das *Gamifiaction Service* zudem sämtliche *Taskinformationen* hinterlegt, die Details über die jeweils auszuführenden Aufgaben und deren aktuellen Zustand enthalten.

4.2.4 RabbitMQ

Damit Aufgaben auf unterschiedlichen Servern bearbeitet werden können, wird eine *Message Queue* benötigt, die als Kommunikationskanal zwischen den Servern fungiert. Obwohl *Redis* bereits bei *runtastic* im Produktiveinsatz war und *Resque* bzw. *Sidekiq* ebenfalls darauf setzen, wurde *RabbitMQ* mit dem *AMQP-Protokoll* ausgewählt. Die Flexibilität von *AMQP* war Hauptgrund für diese Entscheidung. Mit dem Protokoll werden viele Szenarien ermöglicht, die nicht nur im aktuellen Konzept sondern auch für zukünftige Erweiterungen von Vorteil sind.

Mithilfe der Möglichkeiten von *AMQP* wird bei einer Nachricht an einen anderen Server des *Gamification Services* in den Metadaten mitgeliefert, über welche *Antwort Queue* eine Rückmeldung erfolgen soll. Außerdem bleibt eine Nachricht bei aktiviertem *Message Acknowledgement* so lange in der *Message Queue*, bis die Abarbeitung einer Aufgabe mit einer Bestätigung fertiggestellt wurde. Falls der verarbeitende Server abstürzt oder die Nachricht abgelehnt wurde, versucht die *Message Queue* eine erneute Zustellung zu einem späteren Zeitpunkt. Damit kann gewährleistet werden, dass die Nachricht auch wirklich erfolgreich bearbeitet wird.

4.2.5 Memcached

Damit die Datenbank für Leseanfragen nicht zu oft abgefragt werden muss, ist ein Caching System sehr vorteilhaft, welches die am meisten verwendeten Daten im Zwischenspeicher des Servers vorhält. Dies wird mithilfe des *Memcached* Services erreicht, welches ebenfalls bei *runtastic* bereits im Produktiveinsatz verwendet wurde. Insbesondere in Bezug auf die Datenkonsistenz ist zu beachten, dass keine Daten im Cache bleiben, die bereits durch neuere

ersetzt wurden. Außerdem muss beachtet werden, dass bei einem fehlenden Cacheeintrag nicht ohne weiteres die Abfrage aus der Datenbank wieder in den Cache geschrieben werden kann, weil in der Zwischenzeit bereits die Daten in der Datenbank verändert worden sein könnten. Aus diesem Grund ist gerade beim Caching von flexiblen Daten Vorsicht geboten, damit im Betrieb keine unerwarteten Nebenwirkungen, wie Datenverluste oder Fehlberechnungen, auftreten.

4.2.6 Einbindung in die runtastic Infrastruktur

Damit das *Gamification Service* von mobilen Endgeräten verwendet werden kann, wird es in die bestehenden Abläufe der *REST-Schnittstelle* eingebunden, die bereits für die mobilen Anwendungen genutzt wird. Das *Gamification Service* muss aus diesem Grund nicht von außerhalb des *runtastic* Servernetzwerkes erreichbar sein, wodurch der Aufwand für Sicherheitsmaßnahmen verringert werden kann, wie in Abb. 4.7 dargestellt wurde.

Daten, die für das *Gamification Service* relevant sind, werden vom Server der Schnittstelle direkt an das *Gamification Service* weitergeleitet. Erwartet die App bei einer Anfrage eine Antwort mit den aktuellsten *Gamification* Neuigkeiten, so werden diese vom *Gamification Service* an das *Webservice* geliefert, das es dann gemeinsam mit anderen Informationen an das Mobiltelefon zurücksendet.

Für die Datenhaltung in *Cassandra* kann auf das bereits vorhandene Cluster bestehend aus 3 Servern zurückgegriffen werden, welches sich im Produktiveinsatz bereits erfolgreich bewährt hat. *ZooKeeper*, *RabbitMQ*, *Memcached* und das *Gamification Service* werden dagegen auf eigenen Servern ausgeführt, wodurch keine unnötigen gegenseitigen Einflüsse entstehen können.

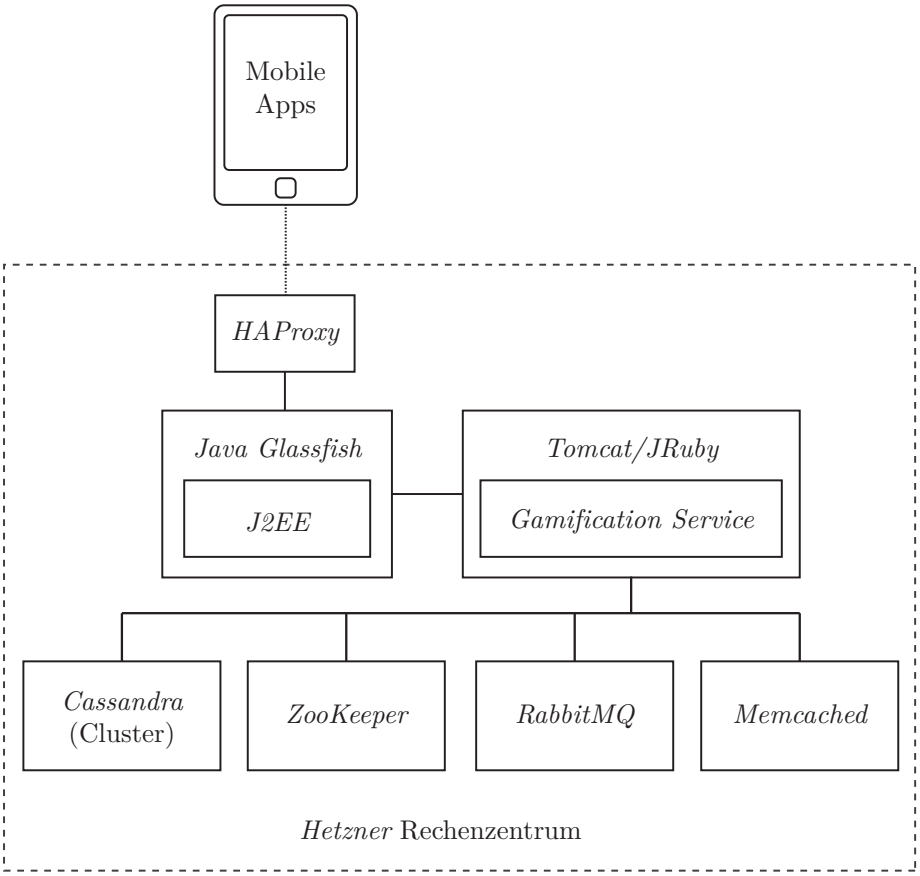


Abbildung 4.7: Darstellung der Integration des *Gamification Services* in die bestehende *runtastic* Infrastruktur.

Kapitel 5

Implementierung

In diesem Kapitel wird auf die konkreten Implementierungsdetails zu den einzelnen Komponenten im *Gamification Service* eingegangen, die auf Basis des Konzeptes umgesetzt wurden. Am Ende werden außerdem die zwei Abläufe beschrieben, die üblicherweise im *Gamification Service* durchgeführt werden.

5.1 Core

Der *Core* des *Gamification Service* enthält alle Komponenten, die sowohl vom *Service* als auch vom *Framework* genutzt werden. Dabei handelt es sich hauptsächlich um Methoden zur Herstellung von Verbindungen mit Systemen wie *Cassanda*, *ZooKeeper*, *RabbitMQ* und *Memcached*, sowie zur Behandlung von Fehlern. Außerdem ist im *Core* das Abhängigkeitsmanagement enthalten, welches zur Abbildung der Abhängigkeiten zwischen Anfragen und den dazugehörigen *Tasks* verwendet wird. Zusätzlich sind einige Helfermethoden (*Helpers*) zentral untergebracht, die an vielen Stellen im gesamten *Gamification Service* Verwendung finden.

5.1.1 Verbindungen zu externen Systemen

Der *Core* enthält alle Klassen, welche die Herstellung der Verbindungen mit den im *Gamification Service* verwendeten Systemen ermöglichen. Sämtliche Verbindungen zu *Cassandra*, *ZooKeeper* und *Memcached* werden in einen *Connection Pool* gelegt. Die Anzahl der Verbindungen entspricht denen der Threads, die im *Service* verwendet werden. Damit wird sichergestellt, dass für jeden im *Service* verwendeten Thread eine eigene Verbindung zur Verfügung steht und deshalb die Verbindungen nur einmal in der Initialisierungsphase und nach einem etwaigen Verbindungsabbruch hergestellt werden müssen. Eine Ausnahme bildet die Verbindung zu *RabbitMQ*. Diese wird direkt im *Broker* des *Services* erstellt, da sich dort die komplette Abwicklung für die

Nachrichtenübertragung befindet und dafür nur eine einzige Verbindung benötigt wird.

5.1.2 Logging und Fehlerbehandlung

Der *Core* enthält Methoden, mit denen z. B. das Loggen, Abfangen und Melden von Informationen, Warnungen und Ausnahmefehlern (*Exceptions*) innerhalb des *Services* einfach implementiert werden können. Diese Funktionalität ist der des *Sidekiq* Projektes nachempfunden, da *Sidekiq* ebenfalls die *Celluloid* Bibliothek als Basis verwendet und dabei bereits auf die Eigenheiten von Multithreading in Hinsicht auf Logging und Fehlerbehandlungen Rücksicht genommen wurde. Im Gegensatz zu *Sidekiq* enthält das Error Handling zusätzlich eine *Exception Mailer* Komponente, die jeden Ausnahmefehler an eine konfigurierte E-Mail Adresse sendet, damit keine Fehler während der Laufzeit des *Gamification Services* unbeachtet bleiben. Außerdem kann auf einfache Weise an jeder Stelle des Quellcodes eine Überprüfung auf fehlerhaftes Verhalten eingebaut werden, wobei ebenfalls ein E-Mail verschickt wird, sobald der Fehler auftritt.

5.1.3 Dependency Graph

Der *Dependency Graph* ermöglicht es, Abhängigkeiten von Elementen topologisch zu sortieren, damit Elemente, die von anderen Elementen abhängig sind, erst nach diesen Abhängigkeiten in einer Liste eingeordnet werden. Zur topologischen Sortierung wird das *Ruby*-eigene *TSort* Mixin verwendet, das den Algorithmus von *Tarjan* zur Bestimmung starker Zusammenhangskomponenten implementiert. Bei der Verwendung gilt es zu beachten, dass durch diesen Algorithmus keine zyklischen Abhängigkeiten aufgelöst werden können [12].

5.1.4 Dependency Connector

Der *Dependency Connector* speichert mithilfe des *Dependency Connector Models* Verbindungen zwischen verwandten Ressourcen in *Cassandra* und wird hauptsächlich im *Framework* verwendet. Dort wird er dazu benötigt, um festzustellen, welche Versionen von Ressourcen bereits in die Berechnung von abhängigen Ressourcen eingeflossen sind, damit bei einer Veränderung der Ausgangsressource zuerst die vorherige verwendete Version aus einer Berechnung herausgenommen und danach die aktuellste Version hineingenommen werden kann. Außerdem ermöglicht der *Dependency Connector* in Zukunft Versionen von Ressourcen aus der Datenbank zu entfernen, wenn davon keine andere Ressource mehr abhängig ist.

5.1.5 Request Dependency Manager

Der *Request Dependency Manager* kümmert sich um die Verwaltung aller Abhängigkeiten zwischen einzelnen Entitäten und Ressourcen. Die Abhängigkeitspaare werden bei dem Hochfahren des *Gamification Services* durch die Initialisierung der einzelnen Ressourcen des *Frameworks* an den *Request Dependency Manager* übermittelt und in einer Liste gespeichert.

Bei der Abarbeitung einer Anfrage an die *API* liefert der *Request Coordinator* die aktuelle Entität und die aktuelle Ressource an den *Request Dependency Manager*, um feststellen zu lassen, welche Abhängigkeiten für den aktuellen *Request* zutreffen. Der *Request Dependency Manager* holt mithilfe seines *Models* sämtliche Abhängigkeiten, die direkt mit der Entität oder der Ressource verbunden sind. Danach werden diese Elemente gefiltert, wobei nur Ressourcen übrig bleiben, welche sowohl von der übermittelten Entität als auch von der übermittelten Ressource abhängig sind. Für jede dieser gefilterten Ressourcen werden mithilfe des Models wiederum sämtliche direkt und indirekt davon abhängigen Ressourcen gesammelt.

Bei den zur ursprünglichen Entität gefundenen Ressourcen wird in weiterer Folge die Abhängigkeit von der Entität zur Ressource geändert, damit sich alle gefundenen Abhängigkeiten nur mehr darauf beziehen. Außerdem wird danach die Ressourcen ID für alle Abhängigkeiten auf die bei der Abfrage definierte Ressourcen ID umgesetzt, die 0 als Ressourcen ID angegeben hatten, um für alle Ressourcen eines Typs gültig zu sein. Mit diesen Maßnahmen wird erreicht, dass der Abhängigkeitenbaum exakt eine Ursprungsressource hat. Nachdem alle Abhängigkeiten definiert wurden, wird außerdem noch die Ursprungsabhängigkeit in die Liste aufgenommen, die den obersten Knoten im Abhängigkeitsbaum darstellt. Danach wird einem optional an den *Request Dependency Manager* übergebenen Block die Möglichkeit zur letzten Bearbeitung der Abhängigkeitenliste gewährt. Diese Liste wird dann an den *Dependency Graph* übergeben und von diesem auf Basis der Abhängigkeiten sortiert und vom *Request Dependency Manager* als Antwort zurückgeliefert.

5.2 Service

Das *Service* Modul enthält alle Komponenten, die zur Abarbeitung einer Anfrage an das *Gamification Service* benötigt werden. Das Aufgabenfeld erstreckt sich von Parsing, Validierung und Analyse der *Requests*, über die Ermittlung der dafür abzuarbeitenden Aufgaben und die Ausführung und Überwachung der Tasks, bis hin zur Rücksendung von Antworten an den Client. Außerdem beinhaltet das *Service* den Endpunkt für die *REST-API*, auf die von außen zugegriffen werden kann. Die eigentliche Logik und Ausführung eines *Tasks* wird wiederum durch das *Framework* übernommen.

5.2.1 API

Die *API* wurde mit dem *Grape* Framework implementiert. Dieses kümmert sich neben der Erstellung der benötigten Routen auch im das Parsen und Validieren der übermittelten Parameter eines *Requests*. Für die Schnittstelle wird eine von *Grape* zur Verfügung gestellte Versionierung verwendet, um für alte Clients bei API-Veränderungen die gewohnte Schnittstelle weiterhin anbieten zu können. In Tabelle 5.1 wird beschrieben, welche Pfadparameter in welcher Reihenfolge an die *API* übergeben werden können. Die *URL* für eine Beispielanfrage mit den definierten Parametern sieht folgendermaßen aus:

```
1 http://vegas/1/api/gamification/user/1234/resources/badge/10
```

GET

Um Daten aus dem *Gamification Service* abzufragen, kann die oben genannte Abfrage mit dem *GET* Befehl getätigt werden. Abgesehen davon gibt es im *Gamification Service* mit `reply_with` noch einen speziellen *Parameter*, der zusätzlich übergeben werden kann, um in einer Abfrage Daten über einen Ressourcentyp hinaus erhalten zu können. Dieser Parameter besteht aus den in Tabelle 5.2 genannten Feldern und kann mehrfach an eine Abfrage angehängt werden. Außerdem lässt sich die Sprache für die in der Antwort des *Gamification Services* enthaltenen Übersetzungstext mithilfe des `locale` Parameters festlegen. Kombiniert sieht eine Beispielanfrage folgendermaßen aus:

```
1 http://vegas/1/api/gamification/user/1234/resources/badge/10?reply_with[
      badge][all]=1,2,5&reply_with[record][state]=0&locale=en
```

Für die *Badges* 1, 2 und 5 werden alle Daten als Antwort zurückgeliefert, für die *Records* nur die Zustandsdaten. Die bei den *Badges* enthaltenen Lokalisierungen sind in englischer Sprache in der Antwort enthalten.

POST und PUT

Mithilfe der *POST* und *PUT* Befehle können beim *Gamification Service* Daten angelegt und verändert werden. Intern wird automatisch festgestellt, ob eine Ressource bereits vorhanden ist und danach entschieden, ob sie neu erstellt oder geändert werden muss. Damit muss das zuführende System nicht darüber Buch führen, ob Daten bereits vom *Gamification Service* verarbeitet wurden oder nicht. Abgesehen von der *URL*, die der des *GET-Requests* gleicht, werden im *Body* des *Requests* die in Tabelle 5.3 definierten Inhalte im *JSON-Format* erwartet.

Tabelle 5.1: Reihenfolge und Beschreibung der Pfadparameter für die *API* des *Gamification Services*.

Nr.	*Feld*	*Beispielwert*	*Beschreibung*
1	Version	`1`	Version der *API*, die genutzt werden soll. Derzeit ist die Version 1 verfügbar.
2	API	`api`	Fixer Platzhalter, der einen *API-Zugriff* signalisiert.
3	Namespace	`gamification`	Grenzt das Einsatzgebiet für das *Service* ab. Damit könnten u. a. mehrere verschiedene Anwendungen vom selben Server aus bedient werden.
4	Entitätentyp	`user`	Gibt an, für welchen Entitätentyp die Abfrage gemacht werden soll. Derzeit wird die *User* Ressource üblicherweise als Entität verwendet.
5	Entitäten ID	`1234`	Gibt an, für welche Entitäten ID die Abfrage gemacht werden soll.
6	Abfragetyp	`resources`	Platzhalter, um später andere Varianten als den Zugriff auf Ressourcen zu ermöglichen.
7	Ressourcentyp	`badge`	Gibt an, für welchen Entitätentyp die Abfrage gemacht werden soll. Derzeit werden üblicherweise die Ressourcen *Badge* und *Record* darüber angesprochen.
8	Ressourcen ID	`10`	Gibt an, welche Ressource genau verwendet werden soll. Bei einem *GET-Request* kann auch `0` als Wert übergeben werden, dadurch enthält die Antwort alle Ressourcen mit dem angegebenen Ressourcentyp.

DELETE

Der *DELETE* Befehl verwendet dieselben Parameter wie *GET* und veranlasst die Löschung einer Ressource im *Gamification Service*.

5.2.2 Broker

Der *Broker* ist als *Celluloid Actor* implementiert und kümmert sich um die Abstraktion der Kommunikation über die *RabbitMQ Message Queue*. Dazu wird auf das *AMQP Gem* zurückgegriffen, welches einen einfachen Zugriff

Tabelle 5.2: Beschreibung der Felder für den `reply_with` Parameter.

Nr.	*Feld*	*Beispielwert*	*Beschreibung*
1	Ressourcentyp	`badge`	Ressourcentyp, der abgefragt werden soll.
2	Inhalte	`all`	Ermöglicht die Selektion, welche Art von Daten geliefert werden sollen. `all` steht für alle Informationen, `static` für alle statischen Daten, `state` für alle Zustandsdaten und `header` für den Kopfbereich der Ressourcendaten.
3	Ressourcen IDs	`1,2,5`	Ermöglicht die genaue Selektion der zu erhaltenden Ressourcen. Die IDs werden per Komma separiert.

Tabelle 5.3: Beschreibung der übermittelten Inhalte einer *POST* oder *PUT* Anfrage an die *API* des *Gamification Services*.

Feldname	*Beschreibung*
`app_data`	Dieses Feld ist optional und kann Informationen über die Anwendung enthalten, die den *Request* abgesetzt hat.
`resource_data`	Beinhaltet sämtliche Daten, die für die übermittelte Ressource gespeichert und verarbeitet werden sollen.
`resource_timestamp`	Zeitstempel, der für die Ressource zur internen Sortierung und Weiterverarbeitung verwendet werden soll.
`reply_with`	Dieses Feld ist optional, funktioniert auch als *URL-Parameter* und kann die Parameter aus Tabelle 5.2 in *JSON-Form* enthalten, um die gewünschten Inhalte für die Antwort festzulegen.
`locale`	Dieses Feld ist optional, funktioniert auch als *URL-Parameter* und kann ein Kürzel für die Sprache enthalten, in welcher Lokalisierung die Antwort gewünscht ist.

auf sämtliche Funktionalitäten des von *RabbitMQ* verwendeten *AMQP* Protokolls ermöglicht. Aufgrund der Abhängigkeit des *AMQP Gems* von der *EventMachine* Bibliothek wird vor dem Verbinden mit dem *RabbitMQ* Server unterschieden, ob das *Gamification Service* in einem Webserver läuft, der selbst *EventMachine* verwendet, und deshalb das *AMQP Gem* mit diesem bereits bestehenden Threads mitlaufen kann, oder ob dafür ein eigener

Thread gestartet werden muss.

5.2.3 Manager

Der *Manager* ist als *Celluloid Actor* implementiert und kümmert sich hauptsächlich um den Empfang und die Verteilung der *Message Queue* Nachrichten für die von ihm über den *Broker* abonnierten *Message Queues*. Außerdem ist er für die Rücksendung der asynchron erhaltenen Antworten an den Webserver verantwortlich, sofern dieser das asynchrone *Request Dispatching* unterstützt. Wenn ein *Request* Antworten aus mehreren *Tasks* erwartet, sammelt der *Manager* die einzelnen über die *Message Queue* eintreffenden Responses zusammen und sendet sie in einer Antwort zurück an den Webserver. Falls ein *Request* bis zur Rücksendung einer Antwort länger als ein definiertes Zeitfenster benötigt, sendet der *Manager* eine *Timeout*-Fehlermeldung an den Webserver zurück, damit die Verbindung zum Client, z. B. im Fehlerfall, nicht endlos geöffnet bleibt.

5.2.4 Dispatcher

Der *Dispatcher* enthält die Einstiegsmethoden sowohl für die *Request-* als auch für die *Taskbearbeitung*, wird als *Celluloid Actor* verwendet und jeweils in einem *Thread Pool* zur gewünschten Funktionalität gewidmet und instanziiert. Das Pooling ist ebenfalls mithilfe von *Celluloid* abgebildet, weil die Funktionalität davon nativ zur Verfügung gestellt wird. Da der *Dispatcher* als Einstiegspunkt für sämtliche Anfragen durch die *API* fungiert, wird hier in jeder extern aufrufbaren Methode eine Fehlerbehandlung vorgenommen, die verhindern soll, dass eine *Exception* innerhalb des *Dispatchers* zum Absturz des Threads oder gar zum Ausfall des gesamten *Gamification Services* führen kann. Fehler, die an dieser Stelle auftreten, werden an einen allgemeinen *Exception Handler* gesendet, der dafür sorgt, dass solche Ausnahmefälle nicht unbemerkt bleiben. Außerdem wird in allen Einstiegsmethoden die Messung der Ausführungsdauer aller beteiligter Services und Objekte gestartet und beendet, damit eventuelle Performanceprobleme lokalisiert und bereinigt werden können.

Request Handling

Wird der *Dispatcher* als *Request Dispatcher* verwendet, so ist die Methode `dispatch_request` der Einstiegspunkt für den Aufruf durch die *API*. Darin wird zuerst mithilfe des *Rack Environments* festgestellt, ob der eingesetzte Webserver eine asynchrone Abarbeitung des *Requests* unterstützt. Dies ist bei *Ruby*-Webservern nur in Ausnahmefällen möglich, weil synchrone Webanwendungen ohne die Verwendung von *Callbacks* einfacher zu implementieren sind. Allerdings ergibt eine asynchrone Abarbeitung bei parallelisierten Vorgängen mehr Sinn, da kein aktives Warten betrieben werden muss, sondern

die Antwort bei Fertigstellung aller Aufgaben per *Callback* direkt an den Webserver und damit an den Client zurückgegeben werden kann. Außerdem können *Tasks* auf sämtlichen Servern ausgeführt werden, auf denen das *Gamification Service* läuft.

Sofern der Webserver nur die synchrone Abarbeitung von *HTTP Requests* unterstützt, können *Requests* hingegen derzeit nur auf einem Server ausgeführt werden. Außerdem wartet der aktuelle *Request Dispatcher* Thread aktiv in einer Endlosschleife auf die Resultate der parallel laufenden *Task Dispatcher* Threads. Ansonsten wird dieser nach Vorbereitung aller *Tasks* für die parallelisierte Abarbeitung zur Bearbeitung weiterer *Requests* wieder in den *Thread Pool* zurückgelegt.

Task Handling

Wird der Dispatcher als Task Dispatcher verwendet, so ist die Methode `dispatch_task` der Einstiegspunkt für den Aufruf über die *Message Queue* bzw. den *Request Dispatcher*. Falls ein Auftrag über die *Message Queue* entgegengenommen wird, dann gibt der *Task Dispatcher* am Ende der Abarbeitung eines *Tasks* ein *Acknowledgement* über einen *Metadata Callback* an die *Queue* zurück. Dieses *Ack* signalisiert, dass die Aufgabe erfolgreich abgearbeitet wurde und die *Message Queue* die Nachricht aus der Warteschlange entfernen kann.

Falls diese Bestätigung nicht abgegeben werden konnte, weil während der Ausführung des *Tasks* ein Fehler aufgetreten oder der Server ausgefallen ist, so würde die *Message Queue* versuchen, die Aufgabe ein weiteres Mal an einen der abonnierten *Gamification Service* Server zuzustellen.

5.2.5 Request Coordinator

Der *Request Coordinator* wird vom *Request Dispatcher* für die Abarbeitung eines *Requests* verwendet und implementiert die dafür benötigte Logik. Diese beinhaltet Methoden für die Gültigkeitsprüfung sowie das Parsen der Parameter von jeglichen Anfragen, die über die *REST-Schnittstelle* entgegengenommen werden. Ferner findet der *Request Coordinator* mithilfe des *Request Dependency Managers* heraus, welche Aufgaben für die jeweilige Anfrage zu erledigen sind und er kümmert sich um das Anlegen der *Tasks* in *Zookeeper*.

Request Parsing, Prüfung und Persistierung

Bevor ein *Request* näher behandelt wird, wird dieser auf seine Gültigkeit geprüft. Falls vorhanden und möglich, wird der `reply_with` Parameter in eine standardisierte Form umgewandelt, weil dieser in verschiedenen Varianten übergeben werden kann, je nachdem, ob es sich um einen *GET* oder *POST Request* handelt. Danach wird geprüft, ob die übermittelten Typen der Entitäten und Ressourcen im System vorhanden sind. Zudem wird geprüft, ob

die erforderlichen Parameter je nach Art des *Requests* übermittelt wurden. Ist dies nicht der Fall, wird der *Request* bereits hier mit einer Fehlermeldung zur Art des Problems abgebrochen. Wenn die Anfrage als gültig bewertet wurde, wird zunächst der Request in die Datenbank persistiert, wenn es sich um einen schreibenden *Request* handelt. Danach werden sämtliche Parameter wie z. B. die angeforderte Sprache, der letztgültige Updatezeitstempel vom Client, eine eindeutige *Request ID*, die Art des *Requests* und der für den User aktuell gültige *Request Zeitstempel* in ein *Request Objekt* gepackt.

Generierung des Request Zeitstempels

Damit bei der Übertragung von Antworten zum Client Datenmenge gespart werden kann, wird immer nur das Delta zwischen dem letztgültigen Zustand am Client und den in der Zwischenzeit am Server durchgeführten Änderungen übermittelt. Bei der ersten Anfrage des Clients muss der Updatezeitstempel auf 0 gesetzt sein, damit der Client alle Daten als Antwort erhält. In der Rücksendung befindet sich dann auch der aktuelle *Request Timestamp*, den der Client für seinen nächsten Request verwenden kann, um dadurch nur die Differenzinhalte zu erhalten.

Weil vom selbigen Client in Ausnahmefällen mehrere *Requests* gleichzeitig abgesendet werden können oder ein *Request* im Fehlerfall nicht vollständig ausgeführt werden könnte, wird bei zeitgleich laufenden Anfragen eine spezielle Generierung des *Request Timestamps* unter der Zuhilfenahme von ZooKeeper nötig:

1. Zuerst wird versucht, einen Knoten mit dem aktuellen Zeitstempel für den User des *Requests* anzulegen.
2. Wenn das Anlegen erfolgreich ist, kann der *Request* mit dem aktuellen Zeitstempel weiterarbeiten. Schlägt dies jedoch fehl, ist bereits ein anderer *Request* mit der Durchführung beschäftigt und der vorher hinterlegte Zeitstempel muss verwendet werden.

Der *Request Timestamp* wird nicht nur für die Antwort zum Client sondern auch als Zeitstempel für die Speicherung von Änderungen in der Datenbank verwendet. Damit bei parallelen *Requests* keine Probleme mit inkonsistenten Zuständen am Client entstehen, müssen bei *Requests*, die bereits einen laufenden *Request* vorfinden, die Zeitstempel so modifiziert werden, dass sie für den bereits laufenden *Request* überdeckend sind. Das heißt, dass die Daten für jeden nach der ersten Anfrage parallel gestarteten *Request* mit einem Zeitstempel um eine Millisekunde erhöht abgespeichert werden müssen, damit der Client vom ersten *Request* diese Änderungen bei einer weiteren Abfrage mitgeliefert bekommt.

Gleichzeitig wird der *Request Zeitstempel*, der an den Client zurückgesendet wird, bei allen parallelen *Requests* außer dem ersten, um eine Millisekunde reduziert, damit diese Clients bei einer weiteren Anfrage alle Än-

derungen vom ersten *Request* erhalten können. Sämtliche Clients bekommen durch diese Modifikationen bei einer Folgeanfrage den korrekten Deltazustand übermittelt, weshalb es hier nie zu *False Negatives* kommen kann.

Request Analyse und Taskerstellung

Auf Basis der Anfrageparameter wird im *Request Coordinator* analysiert, welche *Tasks* für die korrekte Verarbeitung des *Requests* abgearbeitet werden müssen. Dafür registrieren sich in der Initialisierungsphase des *Gamification Services* alle im *Framework* verfügbaren Ressourcen bei dem *Request Dependency Manager* für die für sie relevanten Entitäts- und Ressourcentypen. Bei der Ausführung eines *Requests* wird folgendermaßen vorgegangen:

1. Der *Request Coordinator* holt sich vom *Request Dependency Manager* alle vom *Request* abhängigen *Tasks*.
2. Bei einem *Request*, der neue Daten an das *Service* übermittelt hat, wird für jeden *Task* in der ermittelten Liste überprüft, ob er irgendwelche *Forechecking Filterprüfungen* besitzt. Falls ein Filter existiert, wird diese Prüfung auf die übermittelten Ressourcendaten angewendet. Wenn diese Prüfung fehlschlägt, muss der jeweils geprüfte *Task* für die aktuell übermittelten Daten nicht ausgeführt werden und wird deshalb aus der Abhängigkeitsliste für den *Request* gestrichen. Durch dieses Verfahren wird verhindert, dass unnötige *Tasks* ausgeführt werden, wenn bestimmte Daten nicht übermittelt oder bestimmt Werte nicht gesetzt worden sind.
3. Danach sucht er in dieser Liste alle *Tasks* heraus, die keine weiteren *Nachfolgetasks* mehr haben, um damit bei einem *Request*, bei dem der Client eine Antwort erwartet, die *Auslesetasks* als Nachfolgeaufgabe nach allen Änderungen anhängen zu können.
4. Die sich daraus ergebenden *Schlusstasks* werden dann mit der Anzahl aller *Endtasks* annotiert, damit am Ende eines *Requests* festgestellt werden kann, ob bereits alle *Tasks* beendet wurden und deshalb auch der *Request* beendet werden kann.
5. Nach den Anfügungen und Filterungen der *Tasks* ordnet der *Request Dependency Manager* die *Tasks* nach ihrer Ausführungsreihenfolge in eine sortierte Liste.
6. Diese Liste übernimmt der *Request Coordinator* und filtert darin für die Weiterverarbeitung nicht benötigte Metadaten. Danach fügt er bei jedem *Task* für die *Nachfolgetasks* die Anzahl der jeweiligen *Vorgängertasks* ein, damit diese Bescheid wissen, wie viele *Tasks* zuvor für den jeweiligen *Nachfolgetask* fertiggestellt sein müssen, bevor dieser gestartet werden kann.
7. Danach werden alle Tasks in der Liste durchgegangen und die dafür benötigten Knoten in *ZooKeeper* angelegt, sowie die *Starttasks*, die keine

Vorgänger besitzen, in eine Startliste gegeben, die als Rückgabewert vom *Request Coordinator* zurückgegeben werden.

5.2.6 Task Coordinator

Der *Task Coordinator* kümmert sich um die Abarbeitung eines ihm übergegebenen *Tasks* und ruft das *Framework* zur eigentlichen Aufgabenbearbeitung auf. Danach startet er die *Nachfolgetasks*, sofern bereits alle jeweiligen *Vorgängertasks* erfüllt wurden, beendet den *Request*, falls alle *Tasks* erledigt wurden, und kümmert sich um die Rückgabe der Bearbeitungsergebnisse an den Aufrufer.

Ausführung eines Tasks

Der *Task Coordinator* liest mithilfe der übermittelten Informationen aus der *Task Message* die eigentliche Aufgabenbeschreibung aus dem *ZooKeeper Taskknoten*. Wurde der *Task* bereits mit einer gewisse Anzahl an Versuchen ausgeführt und dabei nie fehlerfrei beendet, dann wird er bereits an dieser Stelle abgebrochen. Als erster Schritt bei der Ausführung des *Tasks* wird versucht, ein *exklusives Lock* für den *Task* in *ZooKeeper* anzulegen, damit der jeweilige *Task* nur einmal gleichzeitig auf allen Servern ausgeführt werden kann. Sollte nach einer gewissen Wartezeit das *Lock* nicht gewährt werden, wird die Abarbeitung abgebrochen.

Sobald das *Task Lock* erfolgreich gewährt wurde, wird über den *Resource Coordinator* ermittelt, ob es sich um einen schreibenden *Task* handelt und wenn ja, welches *Resource Lock* deswegen angelegt werden soll. Dieses *Lock* sorgt dafür, dass eine Ressource auf allen Servern nur einmal gleichzeitig bearbeitet werden kann, egal in welchem *Task* diese Bearbeitung stattfinden würde. Wird das *Lock* nicht nach einer gewissen Zeit gewährt, so wird die Ausführung des *Tasks* abgebrochen. Wenn alle benötigten *Locks* gewährt wurden, wird der *Task* an den *Resource Coordinator* des *Frameworks* übergeben, der sich um die eigentliche Bearbeitung der Aufgabe kümmert.

Start der Nachfolgetasks

Nachdem ein *Task* erfolgreich abgearbeitet wurde, prüft der *Task Coordinator*, ob der gerade bearbeitete *Task* Folgeaufgaben besitzt. Ist dies der Fall, so wird für jeden dieser Folgetasks mithilfe von ZooKeeper geprüft, ob bereits alle Vorgängeraufgaben erledigt wurden. Diese Prüfung erfolgt mittels eines Algorithmus, der an das *ZooKeeper Barrier* Rezept angelehnt ist: Zuerst erstellt der *Task* beim *Nachfolgetask* einen Kinderknoten mit seinem eigenen Tasknamen. Dieser Vorgang symbolisiert, dass der aktuelle *Task* fertiggestellt wurde und deshalb aus dessen Sicht der *Nachfolgetask* gestartet werden könnte.

Da für jeden *Nachfolgetask* bekannt ist, wie viele Vorgänger dieser hat, muss danach geprüft werden, ob die Anzahl der Kinderknoten des *Nachfolgetasks* mit der definierten Anzahl der Vorgänger übereinstimmt. Ist dies der Fall, so darf nur einer der *Tasks*, der diesen Zustand sieht, die Ausführung des *Nachfolgetasks* anstoßen. Dazu versucht jeder *Task*, aus dessen Sicht die Anzahl der Kinderknoten mit der Anzahl der benötigten *Vorgängertasks* übereinstimmt, einen weiteren Kinderknoten als Signal zum Start des nächsten *Tasks* für den *Nachfolgetask* anzulegen. War das Erstellen erfolgreich, hat der Ersteller das Recht, den *Nachfolgetask* anzustoßen, indem der *Task* bei synchroner Aufgabenabarbeitung an die Liste der als nächstes auszuführenden Aufgaben angefügt wird, die nach der Fertigstellung des gerade bearbeiteten *Tasks* gestartet werden sollen. Bei asynchroner Abarbeitung werden die nächsten *Tasks* direkt in die *Task Message Queue* zur weiteren Verarbeitung gelegt.

Sind jedoch bei der Prüfung nicht genügend *Vorgängertasks* des *Nachfolgetasks* fertiggestellt worden oder konnte der aktuelle *Task* den Signalknoten nicht anlegen, weil bereits ein anderer *Task* die Ausführung des *Nachfolgetasks* angestoßen hat, dann darf dieser nicht, bzw. kein weiteres Mal, gestartet werden.

Beendigung des Requests

Hat der aktuelle *Task* jedoch keine Folgeaufgaben, so wird mittels *ZooKeeper* geprüft, ob bereits alle anderen Aufgaben erledigt wurden, die ebenfalls keine Nachfolger besitzen. Dazu wird ein ähnlicher Algorithmus wie bei dem Starten eines *Folgetasks* angewendet:

Zuerst wird ein Kinderknoten, der das Ende des aktuellen *Tasks* symbolisiert, unterhalb des Endknotens erstellt, welcher den gleichen Namen trägt wie der *Request*. Weil jeder *Task*, der keinen Nachfolger besitzt, die Anzahl aller nachfolgerlosen *Tasks* kennt, kann danach geprüft werden, ob bereits alle *Schlusstasks* beendet wurden, um die Beendigung des *Requests* einzuleiten. Wenn diese Bedingung erfüllt ist, versucht der *Task Dispatcher* einen weiteren Knoten unterhalb des Endknotens zu erstellen, der symbolisiert, dass die Beendigung gestartet wurde. War das Anlegen erfolgreich, so wird asynchron die Beendigung und damit das gleichzeitige Aufräumen des gesamten Requests gestartet, wobei sowohl im *ZooKeeper* alle Knoten als auch der in der Datenbank erstellte Requesteintrag gelöscht werden.

5.3 Framework

Das *Framework* enthält alle Komponenten, welche die Erstellung und Auswertung von Inhalten mit dem *Gamification Service* ermöglichen. Es stellt die Grundtypen sowie die *Domain Specific Languages* zur Verfügung, mit denen Designer und Entwickler ihre Inhalte und Erweiterungen abbilden

können, ohne das *Framework* oder andere Kernkomponenten verändern zu müssen. Außerdem kümmert es sich beim Start des *Gamification Services* um das Laden und Initialisieren aller Inhalte und Erweiterungen, damit diese für Anfragen zur Verfügung stehen und dabei instanziiert sowie ausgeführt werden können. Zudem ermöglicht das *Framework* Entwicklern Vorlagen für bestimmte Ressourcentypen anzulegen, damit Game Designer darauf aufbauende Inhalte definieren können, ohne über Details des Systems Bescheid zu wissen. Überdies inkludiert das *Framework* ein Asset System, mit dem sowohl Übersetzungen für beliebige Sprachen als auch Verweise auf externe Inhalte wie Bilder integriert werden können.

5.3.1 Resource Coordinator

Der *Resource Coordinator* ist der Vermittler zwischen dem *Task Coordinator* und den eigentlichen *Framework Ressourcen*. Er kümmert sich um die Beschaffung der für die vom *Task Dispatcher* zur Taskbearbeitung der jeweils auszuführenden Ressource benötigten Lock-Informationen. Außerdem kümmert er sich um die eigentliche Instanziierung der vom *Task Dispatcher* gewünschten Ressourcen und versorgt diese mit den übermittelten Details zur Ausführung und Antworterstellung. Nebenbei wird der *Resource Coordinator* vom *Request Coordinator* zur Feststellung der Verfügbarkeit von bestimmten Ressourcentypen innerhalb des *Frameworks* verwendet.

Ermittlung der Lock Strategie für Ressourcen

Die Strategien, mit welchen Feldern ein *Lock* angelegt werden kann, werden über die eigentliche Ressource ermittelt, die ausgeführt werden soll. Die zu verwendenden Felder werden über eine Option innerhalb einer Ressource definiert, die vom *Resource Coordinator* bei der Ermittlung der Strategie ausgelesen wird.

Ausführung einer Ressource

Bei der Ausführung der Ressource durch den *Resource Coordinator* wird beachtet, welche Ausführungsart im übermittelten *Task* definiert ist:

1. Sofern die Ressource direkt eingefügt, bearbeitet, gelöscht oder abgefragt werden soll, wird entweder die `insert`, `update`, `delete` oder `select` Methode aufgerufen. Als Parameter werden sämtliche Informationen über die zu speichernden bzw. zu lesenden Informationen aus dem übermittelten *Task* des *Task Coordinators* mitgegeben. Bei den `insert` und `update` Methoden inkludiert das zusätzlich auch die neuen Daten, die eingefügt werden sollen.
2. Falls eine Ressource aktualisiert wurde, die in Beziehung mit der durch den aktuellen *Task* zu bearbeitenden Ressource steht, wird die Metho-

de `related_resource_updated` aufgerufen. Diese bekommt ebenfalls die Informationen über den aktuellen *Task* mitgeliefert. Zusätzlich wird sie darüber informiert, ob die verknüpfte Ressource neu erstellt, bearbeitet oder gelöscht wurde, damit jeweils auf die richtige Art und Weise auf Veränderungen reagiert werden kann.

5.3.2 Context Store

Der *Context Store* ist die Hauptkomponente für die Speicherung der Ressourcendaten. Sämtliche Ressourcendaten werden mithilfe des *Context Store Models* in *Cassandra* abgespeichert. Allerdings werden Daten bei einem Aktualisierungs- oder Löschvorgang vom *Context Store* nie überschrieben, sondern immer in einer neuen Version angelegt. Dadurch wird ermöglicht, dass Ressourcen, die von Daten anderer Ressourcen abhängig sind, die alte Version einer abhängigen Ressource aus ihrem Zustand entfernen und deren neue Version hinzufügen können.

Damit Ressourcen auch wieder gefunden werden können, werden zusätzlich zum eigentlichen Datensatz manuelle Indizes gepflegt, weil *Cassandra* selbst dazu nicht besonders gut geeignet ist (s. Abschn. 2.2.1). Diese Indizes ermöglichen dem *Context Store*

- ohne Abfrage aller Ressourcen die neueste Versionsnummer einer Ressource zu erhalten,
- die Ressourcen nach dem Zeitstempel der Ressourcen zu sortieren
- und für eine Entität die dazugehörigen Ressourcen, bzw. für eine Ressource alle dazugehörigen Entitäten, herauszufinden.

Weil *Cassandra* keine Transaktionen unterstützt, werden immer zuerst die Indizes in die Datenbank geschrieben und danach der eigentliche Datensatz. Damit kann über einen Index zwar ein vermeintlicher Datensatz gefunden werden der gar nicht existiert, jedoch ist es nicht möglich, dass Daten existieren, die in einem Index nicht vermerkt sind[1]. Sämtliche Methoden im *Context Store*, die über einen der Indizes an die Schlüsselwerte zum eigentlichen Datensatz wollen, sind darauf vorbereitet, dass dieser evtl. gar nicht existiert und er deshalb ignoriert werden muss. Falls dieser Fehler öfters auftritt, könnten diese fehlerhaften Datensätze in Zukunft auch automatisch bei einem Zugriff entfernt werden, wobei dafür eine eigene Logik mit integriertem *Locking* nötig wäre.

5.3.3 Context Linker

Der *Context Linker* kümmert sich um die Verbindung zwischen Ressourcen, die statische Inhalte beinhalten, und Ressourcen, die den eigentlichen berechneten Wert der Ressource beinhalten. Zum Beispiel sind innerhalb der

[1] Sofern keine Datenbank- oder Programmierfehler vorhanden sind.

Badge Ressource nur die statischen Daten hinterlegt, die eigentliche Berechnung des *Badges* erfolgt jedoch in einem *Static Quest*, in dem sich die *Badge Action* befindet. Weil sich die *Badge Action* beim Starten des *Gamification Services* mit ihrem Ressourcentyp und ihrer Ressourcen ID bei dem *Context Linker* registriert, können bei einer Abfrage des *Badges* die Zustandsdaten aus dem verknüpften *Static Quest* ausgelesen werden.

5.3.4 Resource Base

Die *Resource Base* kümmert sich um das Auslesen und Speichern der Zustandsdaten einer Ressource in *Cassandra* mithilfe des *Context Stores* und des *Dependency Connectors*.

5.3.5 Templated Resource Base

Die *Templated Resource Base* ist eine Erweiterung der *Resource Base* und kümmert sich um das Einlesen der *Template Definitionen*, das Laden der dazugehörigen statischen Informationen und das Zusammenfügen der statischen Informationen mit den Zustandsdaten, die durch die *Resource Base* geliefert werden.

Initialisierung der Definitionen

Beim Start des *Gamification Services* werden sämtliche Erweiterungen initialisiert, die eine Initialisierungsmethode zur Verfügung stellen. Die *Templated Resource Base* nutzt diese Möglichkeit, um alle relevanten *Template Definitionen* mithilfe des *Class Loaders* einzulesen.

Aufgrund der Implementierung auf Klassenebene werden beim Laden der Definitionsdatei die darin enthaltenen Blöcke, welche die Berechnungen, Bedingungen und Aktionen für die Inhalte festlegen, durch den *Extension Parser* ausgewertet und als Ausführungsplan in der Klasse für die spätere Abarbeitung hinterlegt. Damit der *Extension Parser* weiß, welche Erweiterungen akzeptiert werden, lädt dieser davor alle Erweiterungen aus den Erweiterungsverzeichnissen, die bei den jeweiligen im *Template* verfügbaren Blöcken definiert wurden.

Nachdem alle Definitionen geladen und initialisiert wurden, extrahiert die *Templated Resource Base* daraus folgende für die Ausführung relevante Informationen:

- Die verwendeten Erweiterungen zugeordnet nach Blöcken, wie sie der *Extension Parser* ausgelesen hat.
- Den Ausführungsplan für die Erweiterungen, die der *Extension Parser* generiert hat.
- Die *Forechecking Filter*, sofern diese definiert wurden.

- Sämtliche definierte Abhängigkeiten, um diese dem *Request Dependency Manager* zu melden.
- Außerdem werden für jede Definition sämtliche statischen Inhalte wie z. B. Lokalisierungen und Bilderpfade ausgelesen. Zu diesen Inhalten werden zusätzlich noch die Zeitstempel der gelesenen Dateien ausgelesen und davon der aktuellste ermittelt, um bei einer Anfrage durch den Client nur die Daten zu liefern, die neuer sind als sein lokaler Datenbestand.

Nachdem die relevanten Informationen gesammelt wurden, werden die darin definierten Entitäten- und Ressourcenabhängigkeiten an den *Request Dependency Manager* übermittelt, damit das *Service* bei einer Abfrage die im *Template* definierte Ressource in seine auszuführenden *Tasks* einbindet. Außerdem werden sämtliche von Erweiterungen definierte Verknüpfungen an den *Context Linker* übermittelt, damit deren Zustände bei einer Abfrage gefunden werden können.

5.3.6 Resource Item Template Base

Die *Resource Item Template Base* bietet die Basis zum Erstellen von Vorlagen für Definitionsdateien. Sie erlaubt es, Methoden zu definieren, die einen Block mit Methodenbäumen akzeptieren und kümmert sich um deren Verwandlung in eine ausführbare Form für das *Service*. Dazu wird der *Extension Parser* herangezogen, um die Blöcke mit den in der Vorlage erlaubten Erweiterungstypen auszuwerten. Die dabei entstandene Ausführungsliste wird inklusive Metadaten in einer eigenen Kategorieliste für die jeweils in der Vorlage definierten Blockkategorien eingeordnet. Diese Listen können dann u. a. durch die *Templated Resource Base* abgefragt und verwendet werden.

Außerdem stellt die *Resource Item Template Base* eine Methode zur Erstellung eines Ausführungsplans zur Verfügung, der die einzelnen Ausführungslisten in die Reihenfolge bringt, damit Abhängigkeiten von Ein- und Ausgabewerten der in den Ausführungslisten definierten Erweiterungen berücksichtigt werden. Diese Methode verwendet für diese Sortierung den *Dependency Graph*.

5.3.7 Extension Parser

Der *Extension Parser* ist Teil der domänenspezifischen Sprache für Erweiterungen und kümmert sich um das Einlesen der *Ruby Blöcke*, die innerhalb von Definitionsdateien (siehe z. B. Prog. 5.1) enthalten sind und wandelt die im Block enthaltene Baumstruktur in eine Ablaufliste zur späteren Ausführung um. Damit er weiß, welche Methoden innerhalb der Blöcke gültig sind, wird vor dem Parsen der Blöcke angegeben, welche Arten von Erweiterungen innerhalb des Blocks gültig sind. Der *Extension Parser* liest dafür bei der ersten Verwendung eines Erweiterungstyps sämtliche *DSL*-Erweiterungen

aus dem vorgesehenen Unterordner im *Extensions* Verzeichnis aus. Innerhalb der Erweiterung ist wiederum definiert, für welche Methodennamen sie zu registrieren ist. Der *Extension Parser* versucht dann, diese Methode in seinem eigenen Objektkontext hinzuzufügen, sofern diese noch nicht existiert, ansonsten wird sie ignoriert und eine Warnung ausgegeben. Das Erzeugen von Methoden während der Laufzeit wird mithilfe der Möglichkeiten zur Metaprogrammierung in *Ruby* implementiert.

Nach dem Anlegen der Methoden innerhalb des *Extension Parsers* wird der zur Verarbeitung an ihn übergebene Block im Kontext des *Extension Parsers* evaluiert. Dies hat zur Folge, dass die vorher definierten Methoden aufgerufen werden, welche selbst nur als Vermittler dienen, und die Metadaten zur zugehörigen Erweiterung sowie eventuell definierte Parameter und Kinderblöcke an die eigentliche Verarbeitungsmethode des *Extension Parser* weiterleiten. Diese Verarbeitungsmethode kümmert sich darum, dass die durch das *Service* für die spätere Abarbeitung benötigten Erweiterungsmetadaten mithilfe der in der Erweiterung und in den Parametern hinterlegten Informationen aufbereitet werden, und diese in die richtige Reihenfolge in einer Ausführungsliste hinterlegt werden. Falls ein Block übergeben wurde, wird dieser rekursiv im Kontext des *Extension Parser* evaluiert. Sobald alle Methoden ausgeführt wurden, gibt der *Extension Parser* die Ausführungsliste inklusive Metadaten als Antwort an den Aufrufenden zurück.

5.3.8 Extension Runner

Der *Extension Runner* fungiert als Gegenstück zum *Extension Parser* und wird dazu verwendet, Ausführungspläne und -listen tatsächlich zur Ausführung zu bringen. Dabei muss zuerst mit einer vom *Extension Runner* zur Verfügung gestellten Methode ein *Runner Context* erstellt werden. Danach wird dieser Context neben den Ausführungslisten und dem Ausführungsplan an die Planausführungsmethode `run_execution_plan` des *Extension Runner* übergeben. Aufgrund des Planes holt sich die Methode die passende Ausführungsliste und führt diese mit der Methode `run_list` aus. Bei der Ausführung werden sämtliche Erweiterungen in der angegebenen Reihenfolge ausgeführt und die benötigten Daten zwischen den Erweiterungen geteilt. Daten, die für die Durchführung weiterer Ausführungslisten benötigt werden, werden danach ebenso wie das Ergebnis der Auswertung zurückgeliefert. Dieser Vorgang wird für sämtliche Ausführungslisten wiederholt, bis alle abgearbeitet wurden.

5.3.9 Extension DSL

Die *Extension DSL* ermöglicht die Erstellung von Erweiterungen, die vom *Extension Parser* innerhalb von Blöcken evaluiert und mithilfe des *Extension Runners* ausgeführt werden können. Erweiterungen leiten ihre Klasse einfach

Programm 5.1: Beispiel für eine Definitionsdatei anhand eines *Static Quests*, der einen *Badge* für die erste Nutzung der *PushUps App* vergibt, wenn mindestens ein Liegestütz gemacht wurde.

```
1  module Definitions
2   module Gamification
3     module StaticQuests
4       class PushupsFirstUseBadgeQuest < StaticQuestTemplate
5         set :resource_id, 10010001
6
7         forechecking_filter :app_data, :app_branch, :pushups
8
9         preconditions do
10          equal :pushups, compare_as: :symbol do
11            run_session :sport_type
12          end
13        end
14
15        conditions :first_use do
16          greater_equal 1 do
17            run_session :extended_data, :fitness, :max_repetitions
18          end
19        end
20
21        actions do
22          badge 10010001, states: [:secret, :awarded], references: :
       conditions_first_use
23        end
24
25      end
26    end
27  end
28 end
```

von der *Extension DSL* ab und können dadurch Blöcke mit ihrer Funktionalität implementieren, wobei die *DSL* im Hintergrund die Zuordnung der Eingangs- und Ausgangswerte und der Zustandsvariablen übernimmt. Außerdem stellt sie Methoden zur Verfügung, mit der durch die eingehenden Werte iteriert werden kann, ohne dafür unnötig viel Quellcode schreiben zu müssen.

5.4 Abläufe

Zum besseren Verständnis wird hier der Zusammenhang zwischen den einzelnen Komponenten des *Gamification Services* anhand der zwei typischen Abläufen innerhalb des *Services* erläutert.

5.4.1 Ablauf der Initialisierung

1. Das Gamification Service wird mit einem für Ruby-Webanwendungen typischen *rackup* Skript ausgeführt. Bei diesem Skript wird über eine Systemvariable das aktuelle Environment mitgegeben, damit das Service weiß, wo es ausgeführt wird.
2. Mithilfe des *Environment* Parameters werden die dafür festgelegten Konfigurationsdateien geladen und für alle anderen Komponenten verfügbar gemacht.
3. Nach dem Laden der Konfiguration wird diese verwendet, um alle benötigen Verbindungen zu *Cassandra*, *ZooKeeper* und *Memcached* aufzubauen. Nach dem Verbindungsaufbau werden alle Verbindungen je Service in einen *Connection Pool* gelegt. Die Anzahl der benötigten Verbindungen setzt sich aus der Anzahl der verwendeten *Request-* und *Task Dispatcher* zusammen, weil diese in eigenen Threads ablaufen.
4. Sobald die Verbindungen zu den externen Systemen aufgebaut wurden, startet die Initialisierung des *Frameworks*. Dabei werden zuerst alle Klassen instanziiert, die zum Laden der *Framework* Erweiterungen benötigt werden.
5. Mithilfe des *Translation Loaders* werden die Übersetzungen aus dem `assets` Ordner geladen und im *Framework* verfügbar gemacht.
6. Der *Class Loader* lädt danach alle Erweiterungen, die sich im `extensions` Ordner befinden, und macht sie ebenfalls im Framework verfügbar.
7. In weiterer Folge werden sämtliche Erweiterungen durch das *Framework* initialisiert, sofern diese eine Initialisierungsmethode zur Verfügung stellen. Diese Bedingung trifft zu, wenn die Erweiterung von der *Templated Resource Base* abgeleitet wurde.
8. Wenn dies der Fall ist, werden alle für die aktuelle *Templated Resource Base* relevanten *Templatedefinitionen* mithilfe des *Class Loaders* eingelesen.
9. Nach dem Einlesen registriert die *Templated Resource Base* sämtliche Abhängigkeiten und Inhaltsverlinkungen für jede Definition beim *Request Dependency Manager* bzw. beim *Context Linker*, damit diese vom *Service* für die Abarbeitung eines *Requests* berücksichtigt werden.
10. Wurden alle Erweiterungen erfolgreich initialisiert, dann setzt das *Framework* mit der Instanziierung der noch benötigten Klassen fort.
11. Nach der erfolgreichen Initialisierung des Frameworks startet die Initialisierung des *Services*. Das *Service* erzeugt jeweils einen Pool mit *Request-* bzw. *Task Dispatchern*, deren Anzahl über die Konfigurationsdatei festgelegt wurde.
12. In weiterer Folge wird der *Manager* initialisiert, welcher wiederum sei-

nerseits den *Broker* initialisiert.

13. Der *Broker* stellt eine Verbindung zum *RabbitMQ* Server her und registriert alle vom Manager gewünschten *Channels*, *Exchanges* und *Queues*.
14. Im letzten Schritt der Initialisierung wird dann die *API Rack* Anwendung als Antwort der Initialisierungsmethode an das *rackup* Skript zurückgeliefert, damit dieses den Webserver starten und die *REST-Schnittstelle* zur Verfügung stellen kann.

5.4.2 Ablauf eines Requests

1. Eine Client-Applikation sendet ihre Anfrage an die *API* des *Gamification Services*.
2. Die *API* holt einen *Request Dispatcher* aus einem *Thread Pool* von *Dispatchern* und übergibt ihm die eingegangen Parameter des *Requests*.
3. Der *Request Dispatcher* prüft zuerst mithilfe des *Request Coordinators*, ob die Parameter der Anfrage überhaupt gültig sind. Falls sie fehlerhaft sind, werden die falschen Parameter gesammelt und mit einer Fehlermeldung als Antwort an die *API* und somit an den Client retourniert.
4. Nachdem die Parameter als korrekt festgestellt wurden, übergibt der *Request Dispatcher* die Parameter erneut an den *Request Coordinator*, damit dieser daraus ein *Request Objekt* erstellt, welches sämtliche Informationen über die Anfrage des Clients enthält, unter anderem, ob der *Request* eine Antwort erfordert oder es sich nur um eine lesende Anfrage handelt.

   Sofern eine schreibende Anfrage vorliegt, erfolgt auch das Persistieren des *Requests* in der Datenbank. Diese Sicherheitsmaßnahme dient dazu, bei einem potenziellen Ausführungsfehler oder einem Problem mit einem der anderen eingesetzten Systeme, jederzeit den *Request* wiederholen zu können, ohne dabei auf das erneute Einspielen der Anfrage von außerhalb des *Gamification Services* angewiesen zu sein.
5. Falls der *Request* keine Antwort erfordert, bekommt die *API* als Antwort vom *Request Dispatcher* eine Mitteilung, dass die Anfrage erfolgreich angenommen wurde und die Antwort an den Client weitergegeben wird. Damit braucht der Anfragensteller nicht auf die Beendigung der Weiterverarbeitung zu warten.

   Wenn die Anfrage jedoch eine Antwort erfordert, kann der *Request* intern sowohl asynchron als auch synchron ausgeführt werden. Sofern der eingesetzte *Rack*-Webserver die asynchrone Verarbeitung ermöglicht, wird diese eingesetzt, ansonsten wird bei sämtlichen parallelen Vorgängen auf aktives Warten zurückgegriffen.
6. Nach der Eintscheidung über die Art der Weiterverarbeitung durch den *Request Dispatcher* übergibt dieser das *Request Objekt* an den *Re-*

quest Coordinator, damit dieser die für die Anfrage benötigten *Tasks* ermitteln und erstellen kann. Zu diesem Zweck übergibt wiederum der *Request Coordinator* die in der Anfrage mitgelieferte Entität und Ressource an den *Request Dependency Manager*.

7. Dieser ermittelt auf Basis der übergebenen Parameter, welche Aufgaben relevant sind und in welcher Reihenfolge sie ausgeführt werden müssen, um allfällige Abhängigkeiten problemlos auflösen zu können. Bei der Implementierung der *Tasks* ist zu beachten, dass zyklische Abhängigkeiten vorweg ausgeschlossen wurden. Nach der Filterung und Sortierung der Abhängigkeiten werden sie an den *Request Coordinator* zurückgegeben.
8. Dort werden zusätzliche *Abfragetasks* als Abhängigkeit an alle für die Antwort relevanten *Tasks* angefügt, sofern eine Antwort für den *Request* vom Client gewünscht wurde.
9. Nachdem alle *Tasks* für den *Request* feststehen, erstellt der *Request Coordinator* zuerst einen neuen *Request-Knoten* im *ZooKeeper* Server und fügt diesem Hauptknoten dann jeden einzelnen auszuführenden *Task* als Kinderknoten hinzu, welche die jeweils zur Aufgabe dazugehörigen Metadaten und den aktuellen Initialzustand der Aufgabe enthalten.
10. Sobald alle *Tasks* in *ZooKeeper* hinterlegt sind, sucht der *Request Coordinator* in der Taskliste alle Aufgaben heraus, die keine Abhängigkeiten von anderen *Tasks*, also keine Vorgänger haben. Diese sind gleichzeitig die ersten Aufgaben, die problemlos parallel ausgeführt werden können, ohne dass es eine Überschneidung geben kann.
11. Der *Request Dispatcher* bekommt die Liste der ersten auszuführenden Aufgaben als Antwort vom *Request Coordinator* zurückgeliefert und füllt diese *Tasks* bei asynchroner Abarbeitung als *Task Message* in die *RabbitMQ Task Queue* ein oder ruft bei synchroner Abarbeitung nach dem Prinzip des aktiven Wartens parallel in einer Schleife die Abarbeitung der *Tasks* durch einen *Thread Pool* von *Task Dispatchern* auf und wartet auf die jeweiligen Antworten.

    Falls die *Tasks* als Nachricht in die *Task Queue* gelegt wurden, werden diese Nachrichten durch *RabbitMQ* nach dem *Round Robin* Prinzip an einen der Server verteilt, auf denen das Gamification Service ausgeführt wird. Innerhalb des *Gamification Services* ruft der *Manager* dann asynchron einen *Task Dispatcher* aus dem *Thread Pool* auf und übermittelt ihm die *Task Message*.
12. Der *Task Dispatcher* übergibt die an ihn übermittelte *Task Message* weiter an den *Task Coordinator*, welcher sich um die eigentliche Ausführung des *Tasks* kümmert.
13. Der *Task Coordinator* liest mithilfe der übermittelten Informationen

aus der *Task Message* die eigentliche Aufgabenbeschreibung aus dem *ZooKeeper* Taskknoten und kümmert sich um das *Task Lock* und mithilfe des Resource Coordinators um das eventuell benötigte *Resource Lock*. Werden die *Locks* nicht nach einer gewissen Zeit gewährt, so wird die Ausführung des *Tasks* abgebrochen.

14. Nachdem die *Locks* aktiviert wurden, übermittelt der *Task Coordinator* die zum *Task* zugehörigen Informationen zur weiteren Ausführung an den *Resource Coordinator*.
15. Dieser erzeugt eine Instanz der zum *Task* zugehörigen *Resource Extension*, die üblicherweise von *Resource Base* abgeleitet ist, und entscheidet nach der Art des *Tasks*, welche Methode mit welchen Parametern dabei aufgerufen werden soll (s. Abschn. 5.3.1).
16. Wenn nur die Ressource selbst eingefügt oder abgefragt werden soll, nutzt diese die übermittelten Parameter dazu, mithilfe des *Context Stores* die Daten in die *Cassandra* Datenbank zu schreiben bzw. von dort wieder zu extrahieren. Sofern Daten ausgelesen wurden, werden diese als Ergebnis an den *Resource Coordinator* retourniert.

    Bei einer veränderten verwandten Ressource muss die aktuelle Ressource zuerst den eigenen Zustand, der in der *Cassandra* Datenbank hinterlegt ist, über den *Context Store* ermitteln. Ebenfalls aus dem *Content Store* lädt die Ressource in einem weiteren Schritt die vorher verwendete und die neue Version der verwandten Ressource. Danach wird die Neuberechnung der aktuellen Ressource angestoßen, um die Veränderungen der verwandten Ressource in den Zustand der aktuellen Ressource einzubeziehen. Dies geschieht mithilfe einer Methode, die in einer von der *Resource Base* abgeleiteten Klasse enthalten ist, wie z. B. in der *Templated Resource Base*.
17. In der *Templated Resource Base* wird die passende *Template Definition* für die Anfrage ermittelt und daraus der Ausführungsplan für die eingesetzten Regel- und Berechnungserweiterungen geladen. Dieser Ausführungsplan wird zusammen mit dem aktuellen Zustand der Ressource und der neuesten Version der verwandten Ressource an den *Extension Runner* übergeben.
18. Der *Extension Runner* verwendet den aktuellen Ressourcenzustand dazu, die einzelnen Zustände, der für das *Template* auszuführenden Regeln, nach einer vorherigen Ausführung wiederherzustellen, um *Request*-übergreifende Zustände zu ermöglichen. Danach führt er eine Erweiterung nach der anderen aus und erlaubt dabei den Zugriff auf die bereits von der *Resource Base* abgefragten, aktuellen Daten der verwandten Ressource, damit die Erweiterungen diese für ihre Arbeit heranziehen können. Außerdem werden sämtliche Ergebnisse von einer Erweiterung als Eingabewert in die nächsten davon abhängigen Erweiterungen im Ausführungsplan eingespeist.

19. Nachdem der *Extension Runner* das Regelwerk evaluiert hat, gibt er die neuen Zustände zurück an die *Templated Resource Base*, welche ihrerseits die Veränderungen an die *Resource Base* weitergibt.
20. In weiterer Folge speichert die *Resource Base* die neuesten Daten mithilfe des *Context Stores* in die *Cassandra* Datenbank und meldet eine erfolgreiche Ausführung an den *Resource Coordinator* zurück.
21. Die Antwort der Ressourceninstanz wird vom *Resource Coordinator* als Return Wert zurück an den *Task Coordinator* geliefert.
22. Der *Task Coordinator* prüft als nächstes, ob der gerade bearbeitete *Task* Folgeaufgaben besitzt und kümmert sich darum, dass diese in die als nächstes auszuführende Liste von *Tasks* aufgenommen werden, sofern alle dafür benötigten *Vorgängertasks* abgeschlossen wurden. Außerdem sorgt er auch dafür, dass der *Request* beendet wird, wenn keine weiteren *Tasks* mehr durchzuführen oder in Ausführung sind.
23. Der *Task Dispatcher* bekommt die etwaigen Resultate des *Tasks*, sowie die Liste der nächsten auszuführenden *Tasks*, als Antwort des *Task Coordinators* zurück. Ist diese Liste leer, so wurde keine Aufgabe zur Fortführung ausgewählt oder ein asynchroner Task durchgeführt. In synchroner Ausführung gibt der *Task Coordinator* das Resultat und die Folgeaufgabenliste zurück an den *Task Dispatcher*, welcher diese Daten wiederum an den *Request Dispatcher* als Ergebnis liefert.
24. Im synchronen Fall hat der *Request Dispatcher* auf die Fertigstellung des *Tasks* aktiv gewartet. Danach sammelt er etwaige Antworten für den Client in einer Antwortliste. Sofern *Folgetasks* auszuführen sind, werden diese unverzüglich gestartet. Sind alle *Tasks* ohne weitere Folgeaufgaben beendet, wird die Liste mit den Antworten an die *API* zurückgeliefert.
25. Die *API* verwandelt die Antwortliste in das *JSON* Format und liefert diese zurück zum Client.

Kapitel 6

Erfahrungen und Resultate

Die Entwicklung des *Gamification Services* dauerte ungefähr ein Jahr ab Start der Konzeption bis zur Markteinführung für *runtastic*. Durch häufige Änderungen der Anforderungen und des Projektumfeldes waren laufend Anpassungen für das Konzept und in späterer Folge auch in der Implementierung nötig, damit das System so funktionieren konnte, wie es beim Zeitpunkt der Einführung dann auch wirklich war.

6.1 Konzeption und Implementierung

Am Anfang war die Auslegung des *Gamification Services* rein für die Berechnung von *Badges* und *Records* gedacht. Erst nach weiterer Ausarbeitung des Konzeptes wurde der Fokus auf einen allgemeineren, breiteren Ansatz gelenkt. Mit dem aktuellen Ansatz ist es abgesehen von naheliegenden Erweiterungen, wie z. B. weitere *Gamification* Elemente umzusetzen, auch möglich, komplett andere Szenarien zu implementieren, sofern die Basiselemente des Systems wiederverwendet werden können – z. B. könnte es für die Berechnung der Zustände in einem Onlinespiel verwendet werden.

Für die Markteinführung des *Gamification Services* wurden 32 *Static Quests* definiert, die insgesamt 68 *Badges* und 4 *Records* vergeben können. Dank der *DSL* war die Implementierung der *Quests*, *Badges* und *Records* einfach umzusetzen, da die Abstraktion zwischen Erstellung von Inhalten, Erweiterungen und dem eigentlichen Service, das sich um die Abarbeitung der Berechnungen kümmert, erwartungsgemäß funktioniert hat. Insbesondere durch die selbsterklärende Art der Darstellung konnten viele Fehler bei der Erstellung der Inhalte vorweg vermieden werden.

6.2 Produktivbetrieb

Das *Gamification Service* wurde in Verbindung mit der Markteinführung der neuen *runtastic Fitness Apps*[1] in den Produktivbetrieb eingeführt (s. Abb. 6.2), damit die Last auf das System kontinuierlich anwachsen und nicht mit dem hohen Datenaufkommen der bereits am Markt etablierten Apps überfordert werden konnte. Diese Vorgehensweise war insofern wichtig, weil aufgrund des Zeitdrucks, der durch den Wunsch entstanden war, die neuen *Fitness Apps* auf dem *Pioneers Festival*[2] am 30. Oktober 2012 in Wien vorzustellen, eine sonst übliche Testphase nicht möglich war [60]. Abgesehen von der *Cassandra* Datenbank, die in einem Cluster bestehend aus 3 Servern ausgeführt wurde, waren alle anderen Systeme vorerst nur auf einem einzigen geteilten Server aufgesetzt, weil zu dem Zeitpunkt keine weitere Hardware zur Verfügung stand und ein möglicher Ausfall des *Services* hingenommen werden konnte.

Im Zeitraum von 10 Tagen nach dem Start des *Gamification Services* wurden bereits 100 000 Aktivitäten durch das System berechnet und dabei eine Datenmenge von ca. 1 GB erzeugt. Im Durchschnitt wurden dabei ca. 100 Anfragen pro Minute verarbeitet. Bis zum 3. Dezember 2012 wurde das *Gamification Service* bereits von über 76 000 Benutzern verwendet, wie in Abb. 6.1 zu sehen ist. Aus diesen Erfahrungen lässt sich zuversichtlich ableiten, dass die Architektur sinnvoll aufgebaut wurde und für das Wachstum der Daten, sowie Erweiterungen der Inhalte, adäquat ausgelegt wurde. Die Bestätigung dafür wird aber erst ein weiterer Ausbau der Serverlandschaft bringen, wenn weitere Server den produktiven Clusterbetrieb des *Gamification Services* selbst ermöglichen. Dieser konnte zumindest im Testsystem bereits erfolgreich getestet werden.

6.2.1 Antwortzeiten

In der Anfangsphase lagen die Antwortzeiten bei Berechnungsanfragen an das *Service* bei über 5 Sekunden, was nicht den Anforderungen (wie in Abschn. 3 definiert) entsprach. Dieses unerwartete Problem war während der Entwicklung und auch im Testsystem nicht aufgetreten, weshalb es aufgrund des Zeitdrucks vor der Markteinführung nicht mehr behoben werden konnte. Nach näherer Recherche stellte sich das *Software RAID*[3] für *ZooKeeper* als Flaschenhals heraus. *ZooKeeper* wartet immer auf das sichere Schreiben der Daten auf die Festplatte des Servers, bevor eine Anfrage beantwortet wird. Da der vom Betriebssystem zur Verfügung gestellte *fsync* Befehl in Verbindung mit *Software RAID* im Vergleich zu *Hardware RAID* um einiges länger für die Bestätigung des sicheren Schreibens benötigt, wird dementsprechend

[1] http://www.runtastic.com/pumpit
[2] http://www.pioneersfestival.com/
[3] http://en.wikipedia.org/wiki/RAID

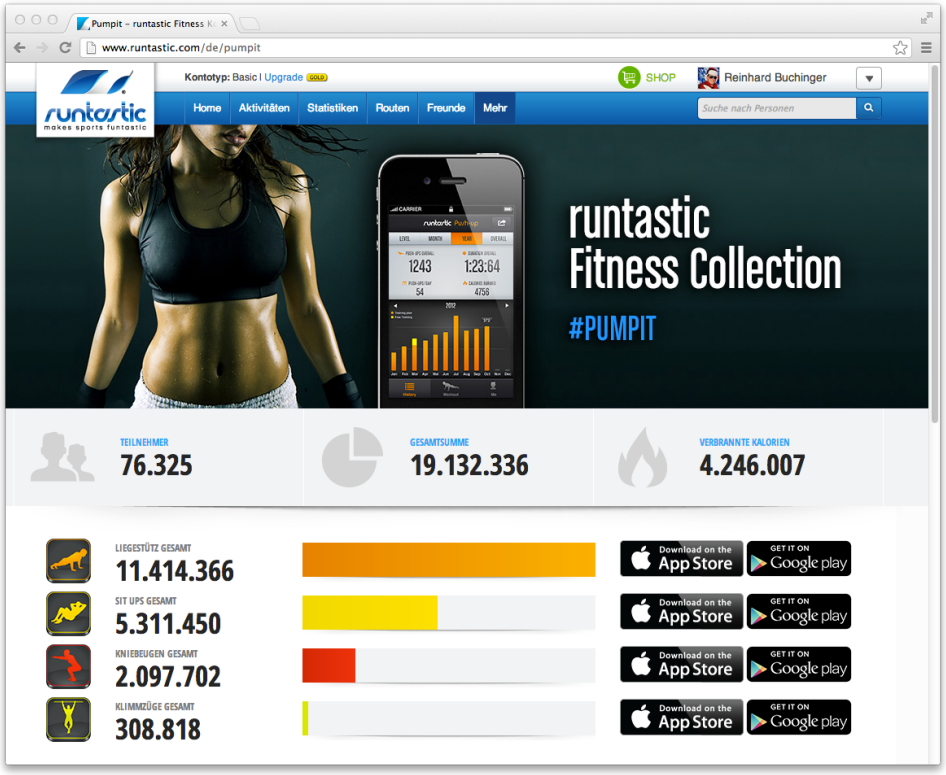


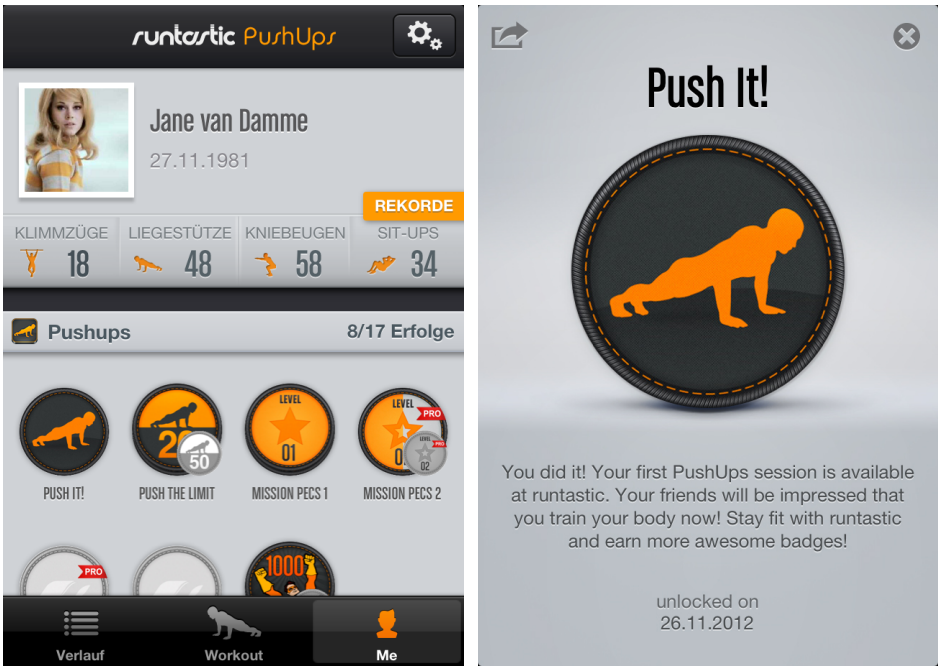
Abbildung 6.1: Aktivitäten der neuen *Fitness Apps* seit der Markteinführung. Stand: 3. Dezember 2012 [59].

das ganze System verlangsamt.

Weil die in *ZooKeeper* hinterlegten Daten im Falle eines Datenverlustes durch einen Serverausfall auf Basis der Daten, die in *Cassandra* gespeichert sind, wiederhergestellt werden könnten, wurde das Abwarten auf das sichere Schreiben der Daten in der *ZooKeeper* Konfiguration deaktiviert. Nach Veränderung dieser Einstellung reduzierte sich die maximale Antwortzeit auf 200 ms, womit die oben angesprochenen Anforderungen in jedem Fall erfüllt wurden.

6.2.2 Race Conditions

Nachdem die Probleme mit der Antwortzeit gelöst waren, trat ein weiteres Problem auf, das vorher nicht in Erscheinung getreten war. Dabei traten *Exceptions* in einem zufällig erscheinenden Muster auf, welche glücklicherweise per *Exception Mailer* übermittelt wurden und dadurch nicht unbemerkt in einem *Logfile* versanken. Nach näherer Analyse konnte festgestellt werden, dass die Ursache am Anfang des Requestablaufs entstanden sein musste, aber



(a) (b)

Abbildung 6.2: Bildschirmfotos entnommen aus der *runtastic PushUps Pro* iPhone App [59]. Darstellung der *Badges* und *Records* im *Me-Tab*. (a). Darstellung einer *Badge*-Detailansicht (b).

beim Auftreten des Fehlers die benötigten Informationen zur Lösung des Problems fehlten. Um die Hintergründe festzustellen, wurden weitere Checks am Anfang des Ablaufs integriert, die im Fehlerfall eine Benachrichtigung mit zusätzlichen Informationen versendeten.

Aufgrund der daraus gewonnenen Daten konnte der Fehler eingegrenzt und schlussendlich gefunden werden. Dieser lag im *Request Dependency Manager*, weil dort fälschlicherweise Instanzvariablen eingesetzt wurden, obwohl dieses Modul nur einmal instanziiert und mit allen *Request Dispatchern* geteilt wird, wodurch eine *Race Condition* bei parallelen Requests entstanden war. Dieser Umstand erklärte auch das zufällige Auftreten der Fehlermeldungen. Mit einer Umwandlung der Instanzvariablen zu Methodenparametern konnte das Problem behoben werden. Weil die Auswirkungen des Problems auf bisherige Berechnungen nicht konkret festgestellt werden konnte, wurden sämtliche Daten aus dem *Gamification Service* zur Sicherheit gelöscht und neu eingefügt.

Kapitel 7

Schlussbemerkungen

7.1 Fazit

In dieser Arbeit wurde das allgemeine Umfeld von *Gamification Services* betrachtet und dabei festgestellt, dass aktuelle Systeme entweder keine Einblicke in ihre Funktionsweise erlauben oder für die in dieser Arbeit behandelten Anforderungen nicht ausreichend gerüstete sind. Danach wurde ein Konzept vorgestellt, auf dessen Basis ein *Gamification Service* und *-Framework* implementiert werden kann.

Die im Konzept vorgeschlagenen Prinzipien und Strukturen wurden in weiterer Folge in einer Implementierung für *runtastic* umgesetzt. Der Umsetzung folgte die erfolgreiche Markteinführung des *Services*, das als Infrastruktur für die Abbildung der *Gamification* Elemente in den *runtastic Fitness Apps* verwendet wurde. Abgesehen von kleinen technischen Problemen im Produktivbetrieb, die aufgrund einer zu kurzen Testphase nicht vermeidbar waren, konnten alle Anforderungen erfüllt und die Funktionsweise des Systems eindrucksvoll bestätigt werden.

7.2 Ausblick

Aufgrund der offenen Auslegung des *Gamification Services* können damit in Zukunft weitere Szenarien abgebildet werden, an die in erster Folge gar nicht gedacht wurde. Dazu zählen u. a. das Sammeln von Statistiken über die Nutzung des *Gamification Services*, die Implementierung einer Spielelogik für ein Onlinespiels und die Umsetzung eines Benachrichtigungssystems, das Entscheidungen auf Basis von Regeldefinitionen trifft.

Abgesehen davon gibt es einige Verbesserungsmöglichkeiten am *Gamification Servce* selbst, die kurz- oder längerfristig implementiert werden sollten:

- Die aktuelle Implementierung der Datenhaltung mit *Cassandra* enthält noch einiges an Potenzial, um die Performance mit ausgefeilteren

Caching Mechanismen weiter zu verbessern und dadurch die Effizienz in der Abarbeitung von *Requests* zu erhöhen. Damit könnten die Antwortzeiten weiter reduziert und das Verhältnis der benötigten Server zu der Anzahl der eingehenden Anfragen verringert werden.

- Derzeit werden *Tasks*, die aufgrund eines Fehlers gestoppt wurden, nicht automatisch neugestartet. Dazu wäre ein System sinnvoll, das die Vorgänge im *System* überwacht und bei Bedarf automatisch eingreift, einen Überblick über den Systemzustand liefert und eine Mitteilung über vorhandene Probleme an den Systemadministrator sendet, falls diese nicht selbst vom System behoben werden können.
- Um bei der Entwicklung von Erweiterungen und Definitionen bereits bekannte Gefahren ausräumen zu können, wären vorgefertigte Testläufe sinnvoll, die in jeder Phase der Inhalte-Implementierung einsetzbar sind.
- Damit in Zukunft *Quests* auch direkt von Benutzern definiert werden können, wird eine Ausweitung der Abhängigkeitenverarbeitung auf Datenbank-basierte Abhängigkeiten benötigt. Entsprechend der Möglichkeiten, die ein Benutzer zur Erstellung eigener *Quests* erhalten soll, wäre die Implementierung einer einfach zu bedienenden Eingabemaske sinnvoll.
- Die Möglichkeit, optionale Abhängigkeiten für *Tasks* definieren zu können wäre nützlich, um nach der Berechnung aller *Quests* eine E-Mail mit den gesammelten Informationen an den Benutzer zu senden, falls mehrere Aktionen innerhalb der *Quest-Regelbäume* eine Sendung von E-Mails auslösen würden.

Anhang A

Inhalt der CD-ROM

A.1 PDF-Dateien

Pfad: /

Buchinger_Reinhard_2012.pdf Diese Masterarbeit in digitaler Form.

A.2 Sourcecode-Dateien

Pfad: /

vegas/ . . . . . Enthält sämtliche Dateien des Gamification Services.

A.3 Onlinequellen-Dateien

Pfad: /

quellen/ . . . . . Enthält sämtliche Onlinequellen nach URL geordnet.

Quellenverzeichnis

Literatur

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[2] Mark Bates. *Distributed Programming with Ruby*. Boston, TX, USA: Addison-Wesley Professional, 2009.

[3] Sebastian Deterding u. a. „From game design elements to gamefulness: defining 'gamification'“. In: *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments*. MindTrek '11. Tampere, Finland: ACM, 2011, S. 9–15.

[4] Arie van Deursen, Paul Klint und Joost Visser. „Domain-specific languages: an annotated bibliography“. In: *SIGPLAN Not.* 35.6 (Juni 2000), S. 26–36.

[5] Brad Fitzpatrick. „Distributed caching with memcached“. In: *Linux Journal* 2004.124 (Aug. 2004), S. 5–.

[6] Patrick Hunt u. a. „ZooKeeper: wait-free coordination for internet-scale systems“. In: *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, S. 11.

[7] William Kent. „A simple guide to five normal forms in relational database theory“. In: *Communications of the ACM* 26.2 (Feb. 1983), S. 120–125.

[8] A. Marczewski. *Gamification: A Simple Introduction*. New Haw, Surrey, UK: Andrzej Marczewski, 2012.

[9] Peter Mell und Tim Grance. „The NIST Definition of Cloud Computing“. In: *National Institute of Standards and Technology* 53.6 (2009), S. 50.

[10] Jakob Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[11] Paolo Perrotta. *Metaprogramming Ruby*. Raleigh, NC, USA: Pragmatic Bookshelf, 2010.

[12] R. Tarjan. „Depth-first search and linear graph algorithms“. In: *SIAM journal on computing* 1.2 (1972), S. 146–160.

[13] Dave Thomas, Chad Fowler und Andy Hunt. *Programming Ruby 1.9: The Pragmatic Programmers' Guide*. 3. Aufl. Raleigh, NC, USA: Pragmatic Bookshelf, 2009.

[14] G. Zichermann und J. Linder. *Game-Based Marketing: Inspire Customer Loyalty Through Rewards, Challenges, and Contests*. Hoboken, NJ, USA: Wiley, 2010.

[15] Gabe Zichermann und Christopher Cunningham. *Gamification by Design: Implementing Game Mechanics in Web and Mobile Apps*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2011.

Online-Quellen

[16] *A Storm is coming: more details and plans for release*. URL: http://engineering.twitter.com/2011/08/storm-is-coming-more-details-and-plans.html (besucht am 03.12.2012).

[17] *About Replication in Cassandra*. URL: http://www.datastax.com/docs/1.1/cluster_architecture/replication (besucht am 03.12.2012).

[18] *About Ruby*. URL: http://www.ruby-lang.org/en/about (besucht am 03.12.2012).

[19] *Aiming To Be A Full-Service Fitness Platform, Runtastic Launches New Indoor App Suite; Hits 14M Downloads*. URL: http://techcrunch.com/2012/10/30/aiming-to-be-a-full-service-fitness-platform-runtastic-launches-new-indoor-app-suite-hits-14m-downloads (besucht am 03.12.2012).

[20] *AMQP Gem*. URL: https://github.com/ruby-amqp/amqp (besucht am 03.12.2012).

[21] *Apache Hadoop*. URL: http://hadoop.apache.org/ (besucht am 03.12.2012).

[22] *Apache ZooKeeper*. URL: http://zookeeper.apache.org/ (besucht am 03.12.2012).

[23] *Apache ZooKeeper Overview*. URL: http://zookeeper.apache.org/doc/r3.4.5/zookeeperOver.html (besucht am 03.12.2012).

[24] *Async-Rack Gem*. URL: https://github.com/rkh/async-rack (besucht am 03.12.2012).

[25] *Cassandra – A structured storage system on a P2P Network*. URL: https://www.facebook.com/note.php?note_id=24413138919&id=9445547199&index=9 (besucht am 03.12.2012).

[26] *Celluloid Gem*. URL: https://github.com/celluloid/celluloid (besucht am 03.12.2012).

[27] *Celluloid Gem – Futures*. URL: https://github.com/celluloid/celluloid/wiki/Futures (besucht am 03.12.2012).

[28] *Connection Pool Gem*. URL: https://github.com/mperham/connection_pool (besucht am 03.12.2012).

[29] *Dalli Gem*. URL: https://github.com/mperham/dalli (besucht am 03.12.2012).

[30] *Download Ruby*. URL: http://www.ruby-lang.org/en/downloads (besucht am 03.12.2012).

[31] *EventMachine Gem*. URL: http://eventmachine.rubyforge.org/ (besucht am 03.12.2012).

[32] *Facebook Releases Cassandra as Open Source*. URL: http://perspectives.mvdirona.com/2008/07/12/FacebookReleasesCassandraAsOpenSource.aspx (besucht am 03.12.2012).

[33] *Foursquare für iPhone*. URL: https://itunes.apple.com/at/app/id306934924/id306934924?mt=8 (besucht am 03.12.2012).

[34] *Game Feature: Leaderboards*. URL: http://gamification.org/wiki/Game_Features/Leaderboards (besucht am 03.12.2012).

[35] *Game Mechanic: Achievements*. URL: http://gamification.org/wiki/Game_Mechanics/Achievements (besucht am 03.12.2012).

[36] *Gamification Facts And Figures*. URL: http://enterprise-gamification.com/index.php/de/fakten (besucht am 03.12.2012).

[37] *Gembundler – Gemfile*. URL: http://gembundler.com/gemfile.html (besucht am 03.12.2012).

[38] *Grape Gem*. URL: https://github.com/intridea/grape (besucht am 03.12.2012).

[39] *Introducing Rack*. URL: http://chneukirchen.org/blog/archive/2007/02/introducing-rack.html (besucht am 03.12.2012).

[40] *Introduction to Composite Columns*. URL: http://www.datastax.com/dev/blog/introduction-to-composite-columns-part-1 (besucht am 03.12.2012).

[41] *JRuby*. URL: http://jruby.org/ (besucht am 03.12.2012).

[42] *JRuby 1.7.0. Released*. URL: http://jruby.org/2012/10/22/jruby-1-7-0.html (besucht am 03.12.2012).

[43] *Memcached*. URL: http://www.memcached.org/ (besucht am 03.12.2012).

[44] *Mizuno Gem*. URL: https://github.com/matadon/mizuno (besucht am 03.12.2012).

[45] *Parallelism is a Myth in Ruby*. URL: http://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby (besucht am 03.12.2012).

[46] *RabbitMQ – Highly Available Queues*. URL: http://www.rabbitmq.com/ha.html (besucht am 03.12.2012).

[47] *RabbitMQ – Publish/Subscribe Tutorial*. URL: http://www.rabbitmq.com/tutorials/tutorial-three-python.html (besucht am 03.12.2012).

[48] *RabbitMQ*. URL: http://www.rabbitmq.com/ (besucht am 03.12.2012).

[49] *RabbitMQ – Routing Tutorial*. URL: http://www.rabbitmq.com/tutorials/tutorial-four-python.html (besucht am 03.12.2012).

[50] *RabbitMQ – RPC Tutorial*. URL: http://www.rabbitmq.com/tutorials/tutorial-six-python.html (besucht am 03.12.2012).

[51] *RabbitMQ – Topics Tutorial*. URL: http://www.rabbitmq.com/tutorials/tutorial-five-python.html (besucht am 03.12.2012).

[52] *RabbitMQ – Work Queues Tutorial*. URL: http://www.rabbitmq.com/tutorials/tutorial-two-python.html (besucht am 03.12.2012).

[53] *Rack Gem*. URL: https://github.com/rack/rack (besucht am 03.12.2012).

[54] *Redis Persistence*. URL: http://redis.io/topics/persistence (besucht am 03.12.2012).

[55] *Redis Replication*. URL: http://redis.io/topics/replication (besucht am 03.12.2012).

[56] *Resque Gem*. URL: https://github.com/defunkt/resque (besucht am 03.12.2012).

[57] *Ruby 1.9 Speed vs. JRuby speed*. URL: http://shootout.alioth.debian.org/u32/benchmark.php?test=all&lang=yarv&lang2=jruby (besucht am 03.12.2012).

[58] *Ruby, Concurrency, and You*. URL: https://www.engineyard.com/blog/2011/ruby-concurrency-and-you (besucht am 03.12.2012).

[59] *runtastic Fitness Collection Übersicht*. URL: http://www.runtastic.com/de/pumpit (besucht am 03.12.2012).

[60] *Runtastic sprints past 14m downloads, and launches four new fitness apps for homebodies*. URL: http://thenextweb.com/apps/2012/10/30/runtastic-sprints-past-14m-downloads-and-launches-four-new-fitness-apps-for-homebodies/ (besucht am 03.12.2012).

[61] *Schema in Cassandra 1.1*. URL: http://www.datastax.com/dev/blog/schema-in-cassandra-1-1 (besucht am 03.12.2012).

[62] *Sidekiq Gem*. URL: https://github.com/mperham/sidekiq (besucht am 03.12.2012).

[63] *Sinatra Gem*. URL: http://www.sinatrarb.com (besucht am 03.12.2012).

[64] *Storm*. URL: https://github.com/nathanmarz/storm (besucht am 03.12. 2012).

[65] *Storm – Tutorial*. URL: https://github.com/nathanmarz/storm/wiki/Tutorial (besucht am 03.12.2012).

[66] *The Cassandra Data Model*. URL: http://www.datastax.com/docs/1.1/ddl/about-data-model (besucht am 03.12.2012).

[67] *The Rise Of Gamification*. URL: http://www.adotas.com/2011/07/the-rise-of-gamification (besucht am 03.12.2012).

[68] *userinfuser – API Documentation*. URL: http://code.google.com/p/userinfuser/wiki/API_Documentation (besucht am 03.12.2012).

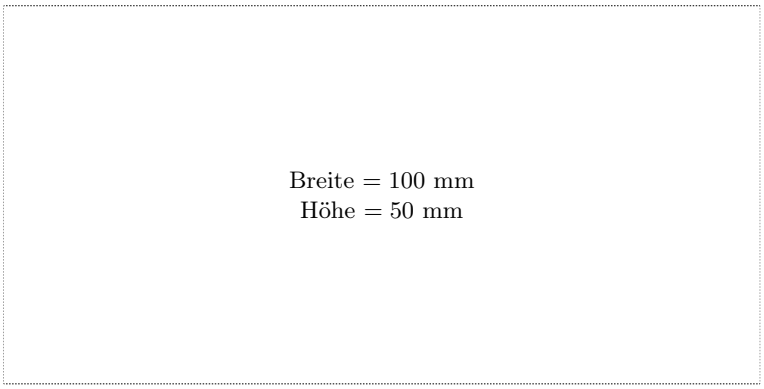
[69] *userinfuser*. URL: http://code.google.com/p/userinfuser (besucht am 03.12.2012).

[70] *Wikipedia – Reactor Pattern*. URL: http://en.wikipedia.org/wiki/Reactor_pattern (besucht am 03.12.2012).

[71] *ZK Gem*. URL: https://github.com/slyphon/zk (besucht am 03.12. 2012).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —