
Bachelorarbeit am Institut für Informatik der Freien Universität Berlin
Arbeitsgruppe Corporate Semantic Web

# Entwurf und Implementierung einer Architektur zur asynchronen Verarbeitung von Batch-Jobs in einem Unternehmen

Konstantin Bork
Matrikelnummer: [redacted]
E-Mail: [redacted]

Gutachter: Prof. Dr. Adrian Paschke
Zweitgutachter: Prof. Dr. Lutz Prechelt
Berlin, 21. September 2015

# Abstract

In einem Unternehmen werden oft sogenannte Batch-Jobs durchgeführt, die eine bestimmte Aufgabe ohne Nutzereingaben erledigen. Solche Jobs können nicht ohne weitere Anpassungen parallel ausgeführt werden. Ein bekanntes Konzept ist die Verwendung von Workern, die die Jobs von einem Master erhalten. Worker werden in Form von lokalen Prozessen oder anderen Computern in einem Netzwerk realisiert. Hier liegt allerdings die Schwierigkeit in der Verteilung der Batch-Jobs an die Worker. Eine Lösung ist die Verwendung von Messaging, um Prozesse oder auch Computer unabhängig von Programmiersprachen und Betriebssystemen miteinander kommunizieren zu lassen. Zusammen mit einer Integrationslösung kann diese Lösung in die bestehende Infrastruktur eines Unternehmens eingebunden werden.

# Inhaltsverzeichnis

# I. Abbildungsverzeichnis

# II. Listings



# III. Tabellenverzeichnis

# 1. Einleitung

In der vorliegenden Arbeit wird die Entwicklung und Implementierung eines Programms zur asynchronen Verarbeitung von Batch-Jobs beschrieben. Dieses Programm wird als Plugin entwickelt, welches in die bisher bestehende Architektur des Content-Management-Systems[G] des Unternehmens eingefügt werden soll.
Die Bonial International GmbH bietet Konsumenten einen Service zur Anzeige von Broschüren verschiedener Einzelhändler in über zehn Ländern an. Die von den Händlern geschickten Broschüren werden in ein mit dem Web Application Framework Grails[G] entwickeltes Content-Management- System hochgeladen und für die angebotenen Apps und die Webseite aufbereitet. Diese Applikation läuft auf einem Tomcat[G]-Server und ist großteils sequentiell aufgebaut. Dadurch tritt bei großen Broschüren ein Timeout[G] des Servers auf, wodurch der Benutzer nach etwa zehn Minuten keinen Einfluss mehr auf den aktuellen Upload hat. Trotzdem läuft die Verarbeitung auf dem Server weiter. Als Lösung wurde der Timeout des Servers erhöht, was aber nur einen Workaround[G] darstellt.
Im Rahmen dieser Bachelorarbeit wird eine auf einer Warteschlange basierende Architektur entwickelt und mithilfe des Grails-Frameworks implementiert. Ziel des entwickelten Programms soll es sein, den Timeout des Servers zu umgehen, mehrere gleichzeitige Uploads zu ermöglichen und so insgesamt die Arbeit mit dem Content-Management-Systems effizienter zu gestalten.
Die Arbeit definiert zunächst das Problem genauer und zählt die Anforderungen an das zu entwickelnde Programm auf. Darauf folgen die theoretischen Grundlagen, die dann zur Beschreibung der entwickelten Architektur führen. Es folgt eine Evaluation verschiedener Technologien, die zur Implementierung der Architektur in Betracht gezogen werden. Anschließend wird das Vorgehen bei der Implementierung beschrieben, wobei das Ergebnis mit den Anforderungen verglichen wird, und die Ergebnisse dieser Arbeit werden zusammengefasst.
In der Arbeit werden überwiegend die deutschen Fachbegriffe verwendet. Einige englische Begriffe, die im Deutschen akzeptiert und verwendet werden, werden allerdings nicht übersetzt. Außerdem existiert am Ende der Arbeit ein Glossar, in dem einige in der Arbeit verwendete Fachbegriffe de-

finiert werden. Die im Glossar befindlichen Begriffe werden bei der ersten Verwendung in der Arbeit mit einem [G] markiert.

# 2. Probleme und Anforderungen

## 2.1. Problembeschreibung

Die Bonial International GmbH verwendet zur Verwaltung von Broschüren und anderen für den Service relevanten Daten ein selbstentwickeltes Content-Management-System, welches mit dem Grails-Framework entwickelt wurde und auf einem Apache-Tomcat-Server läuft. Das für die Inhalte des Service zuständige Content Team lädt neue Broschüren, aber auch Informationen zu Filialen verschiedener Einzelhändler in das Content-Management-System hoch. Neben dem reinen Upload werden die Daten vom System für den Benutzer des Service aufbereitet, sodass Prospekte bestimmten Schlagwörtern zugeordnet oder Links zu weiteren Informationen eines im Prospekt gezeigten Produkts hinzugefügt werden.
Bislang muss der Benutzer beim Upload eines Prospekts darauf warten, dass diese vom System fertig verarbeitet wurde, bevor er weitere Einstellungen vornehmen kann. Dabei treten hauptsächlich zwei Probleme auf:

1.) Beim Upload großer Datenmengen, wie sie z.B. bei Broschüren von großen Einzelhändlern mit regionalen Angeboten vorkommen, benötigt die Applikation einige Zeit zur Verarbeitung der einkommenden Daten. Hierdurch kommt es beim Tomcat-Server zu einer Zeitüberschreitung, die normalerweise einen Paketverlust bei der Datenübertragung anzeigen soll. Allerdings laufen der Upload und die Verarbeitung der Daten im Hintergrund auf dem Server weiter. Unerfahrene Benutzer können nun einen neuen Auftrag starten, wodurch die Daten in der Datenbank kompromittiert werden.

2.) Der Benutzer des Content-Management-Systems ist bei der Arbeit blockiert. Er kann keine neuen Broschüren oder Filialdaten hochladen und keine Einstellungen an bereits hochgeladenen und verarbeiteten Daten vornehmen. Hierfür muss er einen Kollegen beauftragen, der eine neue Session im System starten kann und somit nicht vom Dateiupload blockiert wird.

Zum erstgenannten Problem existiert bisher ein Workaround, welcher den Timeout des Servers hochsetzt. Allerdings kann aufgrund des hohen Timeout-Werts, der mittlerweile bei etwa einer Stunde liegt, nicht schnell auf Übertragungsfehler reagiert werden. Somit ist eine Lösung dieses und des zweiten

Problems erforderlich, die das Hochsetzen des Timeouts nicht mehr notwendig macht, die Arbeit mit dem Content-Management-Systems beschleunigt und insgesamt das Nutzererlebnis verbessert.

## 2.2. Anforderungen

Als Lösung soll ein agentenbasiertes System entwickelt werden, welches Nutzeranfragen asynchron verarbeiten soll, die an die bestehende Applikation geschickt werden. Die Agenten sollen anfangs Threads[G] in der Software sein, die auf einkommende Aufgaben warten. Allerdings soll die Möglichkeit eines verteilten Systems offen gehalten werden, bei dem die Anfragen vom Server an physische Instanzen in einem Netzwerk weitergeleitet werden. Nutzeranfragen sollen vor der Verarbeitung durch das System in einer Warteschlange gesammelt werden, sodass man eine große Anzahl an Aufträgen an das System schicken kann, die dann nach Verfügbarkeit der Agenten im Hintergrund abgearbeitet werden. Die Möglichkeit der Priorisierung von Aufträgen soll möglichst offen gehalten werden. Nicht zuletzt soll die Lösung möglichst generisch sein, sodass sie auch mit möglichst kleinen Änderungen in anderen eingesetzten Applikationen verwendet werden kann.

### 2.2.1. Die Agenten

Anfangs sollen Threads die Agenten darstellen, die die einkommenden Aufträge bearbeiten. Die Anzahl der Agenten soll konfigurierbar sein und sie sollen eine Schnittstelle zum Starten, Stoppen und zur Statusabfrage von Aufträgen bieten. Die Aufträge sollen sie aus einer In-Memory-Queue erhalten, die nebenläufige Prozesse blockt und die einkommenden Aufträge nach ihrer Priorität sortiert. Trotz dieses Aufbaus sollen die Agenten und die Warteschlange mit kleinen Änderungen auch Teil eines verteilten Systems werden, wobei dann speziell für die Auftragsverarbeitung bereitgestellte Computer die Aufträge von der lokalen Warteschlange entgegennehmen und die Kommunikation über Nachrichtenkanäle erfolgt.

### 2.2.2. Die Aufträge

Die Aufträge sollen die zu verarbeitenden Daten als Parameter speichern und eine Schnittstelle bereitstellen, über die sie gestartet und gestoppt wer-

den können und über die man ihren aktuellen Status abfragen kann. Zur besseren Identifizierung sollen die Aufträge eine ID speichern, damit Aufträge nicht doppelt erledigt werden. Außerdem sollen sie möglichst generisch aufgebaut sein, damit man verschiedene Dateitypen verarbeiten kann.

### 2.2.3. Nichtfunktionale Anforderungen

Neben allen genannten funktionalen Anforderungen soll die zu entwickelnde Software wiederverwendbar und durch einen modularen Aufbau einfach erweiterbar sein. Der modulare Aufbau soll die Lösung auch wartungsfreundlich machen, was zusätzlich durch eine gute durchgängige Dokumentation begünstigt werden soll. Um die Lösung sicher nutzen zu können, sollen Fehler, die durch falsche oder unvollständige Benutzereingaben hervorgerufen werden, abgefangen werden. Abschließend soll sich die Lösung ohne große Änderungen der bisherigen Architektur in die bestehende Applikation einbauen lassen.

# 3. Theoretische Grundlagen der Software

## 3.1. Architekturmuster

Die Entwicklung der Architektur soll genau wie die Lösung von konkreten Teilproblemen nach bekannten Mustern erfolgen. Hierfür existieren analog zu Entwurfsmustern[G] auch Architekturmuster, die die grundlegende Organisation und Interaktion zwischen den Anwendungskomponenten bestimmen [28].

### 3.1.1. Model-View-Controller

Das *Model-View-Controller*-Muster dient in erster Linie zur Entkopplung von Aussehen und Logik eines Programms. Die Kommunikation zwischen diesen beiden Komponenten findet über ein *Controller*-Objekt statt, das die Benutzereingaben von den *View*-Objekten an das *Model*-Objekt leitet und die Ergebnisse des *Model*-Objektes an die *View*-Objekte schickt. Durch diesen Aufbau können mehrere *View*-Objekte an ein *Model*-Objekt angebunden werden, um verschiedene Datenrepräsentationen anbieten zu können, und es können jederzeit neue *View*-Objekte erzeugt werden, ohne die Programmlogik verändern zu müssen [9, S. 30-32].

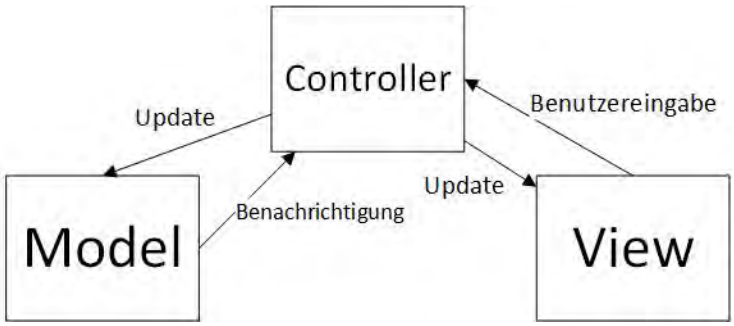


Abbildung 1: Schematische Darstellung des Model-View-Controller-Muster

### 3.1.2. Master-Worker

Das *Master-Worker*-Muster wird bei der parallelen Datenverarbeitung benutzt. Die Aufgaben werden von einem *Master* an mehrere *Worker* verteilt, die die eigentliche Aufgabenbearbeitung durchführen. Die *Worker* können hierbei lokale Prozesse auf dem selben Computer wie der *Master* oder physische Instanzen innerhalb eines verteilten Systems darstellen [10].

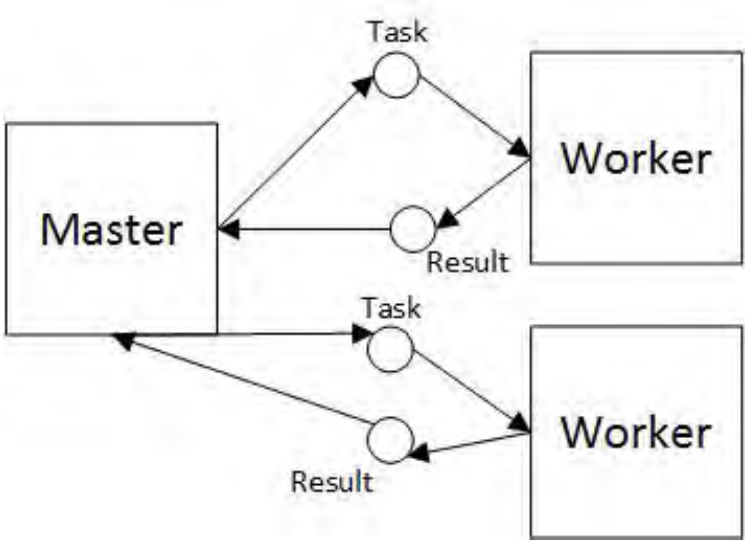


Abbildung 2: Schematische Darstellung des Master-Worker-Muster

### 3.1.3. Producer-Consumer

Die Hauptbestandteile dieses Musters sind ein *Produzent* und ein *Konsument*, die normalerweise über eine Warteschlange miteinander verbunden sind. Der *Produzent* schickt Objekte zur späteren Bearbeitung an die Warteschlange und muss sie nicht direkt bearbeiten. Der *Konsument* kann dann jederzeit das Objekt aus der Warteschlange nehmen und bearbeiten. Durch diesen Aufbau können beliebig viele *Produzenten* und *Konsumenten* unabhängig voneinander innerhalb eines Systems existieren [16].

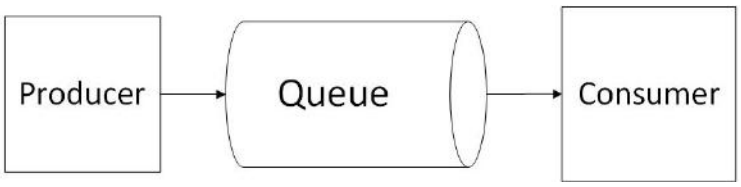


Abbildung 3: Schematische Darstellung des Producer-Consumer-Musters

## 3.2. Batch-Processing

Batch-Processing ist eine Betriebsart eines Computers, bei der Jobs eines Benutzers als Ganzes abgearbeitet werden ohne dem Benutzer eine Eingriffsmöglichkeit zur Bearbeitung der Jobs zu bieten [14]. Batch-Prozesse haben laut [3] folgende Eigenschaften:

- Es werden große Mengen an Eingabedaten verarbeitet und gespeichert, was auch zu einer hohen Anzahl an Datenzugriffen und großen Mengen an Ausgabedaten führt.
- Batch-Prozesse müssen normalerweise nicht sofort auf Anfragen antworten, allerdings sollten sie innerhalb einer bestimmten Zeit, dem sogenannten *Batch Window* [G], abgeschlossen werden.
- Informationen werden aus großen Datenmengen erzeugt.
- Ein geplanter Batch-Prozess kann aus der Ausführung von mehreren Hundert bis mehreren Tausend Jobs bestehen, die in einer festgelegten Reihenfolge abgearbeitet werden.

Beispiele für solche Batch-Prozesse sind im Einzelhandel Abbuchungen von den Bankkonten der Kunden, die mit einer Kredit- oder Girokarte bezahlt haben, oder das Konvertieren von Dateien in ein anderes Dateiformat. Realisiert werden können solche Batch-Prozesse z.B. mittels Unix-Shellskripten oder Windows-Batchdateien.

## 3.3. Middleware

Als Middleware wird eine zusätzliche Schicht in einer komplexen Softwareumgebung bezeichnet. Aufgabe dieser zusätzlichen Schicht ist es, einfache Zugriffsmöglichkeiten auf untere Softwareschichten bereitzustellen und ihre Infrastruktur zu verbergen. Dadurch wird einerseits die Anwendungssoftware entlastet, andererseits steigt die Produktivität bei der Softwareentwicklung, da der Programmierer sich nicht um die Bereitstellung von Diensten aus den unteren Softwareschichten kümmern muss. Es wird zwischen zwei Formen von Middleware unterschieden [17]:

- **Kommunikationsorientiert:** Die Middleware definiert die Kommunikationsinfrastruktur verteilter Anwendungen in einem verteilten System. Es können sowohl synchrone als auch asynchrone Kommunikationsformen unterstützt werden. Synchrone Kommunikation wird u.a. mit *Remote Procedure Calls*[G] realisiert, die einen entfernten Prozeduraufruf realisieren. Bei asynchroner Kommunikation kommt meist *Message Oriented Middleware* zum Einsatz. Hier kommt in der Regel eine Warteschlange zum Einsatz, die alle einkommenden Nachrichten sammelt und es dem Empfänger erlaubt, die Nachricht zu ei-

nem beliebigen Zeitpunkt zu empfangen. Es wird hier weiterhin zwischen zwei Modellen unterschieden: Das *Request-Reply-Modell*, bei dem eine synchrone Kommunikation simuliert wird, und das *Publish-Subscribe-Modell*, bei dem Empfänger ein Thema abonnieren und nur Nachrichten mit diesem Thema erhalten [18].

- **Anwendungsorientiert:** Kommunikationsorientierte Middleware wird um verschiedene Komponenten erweitert. Hierzu gehören eine Laufzeitfunktionalität, die Verbindungen zwischen den Komponenten eines Programms verwaltet und Programmierschnittstellen bereitstellt, und Dienste für die Anwendungsschicht wie einer Sitzungsverwaltung. Oft wird sie auch um ein Komponentenmodell, das Anwendungsobjekte als Komponenten definiert, ergänzt.

Im Rahmen dieser Arbeit liegt der Fokus auf Message Oriented Middleware, da mit der zu entwickelnden Software eine asynchrone Bearbeitung von Batch-Jobs erfolgen soll.

## 3.4. Messaging

Messaging ist eine Technologie, die schnelle asynchrone Kommunikation mit verlässlicher Zustellung zwischen Programmen erlaubt. Dabei sind die Programme über Kanäle miteinander verbunden, über die sie Nachrichten in Form von Datenpaketen schicken. Nachrichten bestehen aus zwei Teilen, dem Header und dem Body. Der Header enthält Meta-Informationen wie Informationen zum Sender und zum Empfänger und wird vom Messaging-System ausgelesen. Der Body enthält die Applikationsdaten und wird vom Empfänger gelesen, während das Messaging-System diese Daten ignoriert [15, S. XXX f.].
Das Messaging koordiniert die Nachrichtenübermittlung zwischen den Programmen. Nachrichten werden dabei in fünf Schritten übertragen [15, S. XXXII]:

1.) Erstellung: Der Nachrichtensender erstellt die Nachricht mit allen erforderlichen Daten.

2.) Senden: Die Nachricht wird an einen Nachrichtenkanal gesendet.

3.) Zustellung: Das Messaging-System bewegt die Nachricht vom Sender zum Empfänger.

4.) Erhalt: Der Empfänger liest die Nachricht aus dem Nachrichtenkanal.

5.) Bearbeitung: Der Empfänger extrahiert die Daten aus der Nachricht.

Gegenüber anderen Methoden zur Datenübertragung wie Datentransfers oder geteilten Datenbanken hat Messaging diverse Vorteile. Erstens können separate Programme miteinander kommunizieren und Daten übertragen. Zweitens ermöglicht Messaging asynchrone Kommunikation, wodurch ein Sender nach Versand einer Nachricht weitere Aufgaben erledigen kann und sich nicht um die Nachrichtenkoordination kümmern muss. Außerdem erlaubt Messaging verlässliche Kommunikation, was besonders gegenüber *Remote Procedure Calls* einen großen Vorteil bedeutet. Nachrichten sind atomare, unabhängige Einheiten, die bis zum erfolgreichen Erhalt beim Empfänger vom Messaging-System wiederholt gesendet werden [15, S. XXXIV f.].

# 4. Architektur

## 4.1. Erster Entwurf

Wie bereits in den Anforderungen beschrieben soll ein agentenbasiertes System entwickelt werden, das Nutzeranfragen zunächst in einer Warteschlange sammelt und dann zur Bearbeitung an die Agenten weiterleitet. Weiterhin soll dies In-Memory geschehen, wobei die Option eines verteilten Systems offengehalten werden soll. Außerdem soll eine Statusabfrage der Aufträge jederzeit möglich sein. Diese funktionalen Anforderungen machen eine modulare Softwarearchitektur fast unumgänglich, die Anforderung einer wartungsfreundlichen und leicht erweiterbaren Software begünstigt die Entwicklung solch einer Architektur.
Drei Komponenten sind aufgrund der gegebenen Anforderungen schon bekannt: Eine Warteschlange, ein Statusmonitor und die Agenten. Um die Warteschlange einfach austauschbar bzw. erweiterbar zu gestalten, wird die Architektur um einen Produzenten und einen Konsumenten erweitert. Die Aufgabe des Produzenten ist es, mithilfe der vom Benutzer bereitgestellten Daten die Aufträge zu erstellen, die in der Warteschlange gespeichert werden sollen, während der Konsument die Aufträge aus der Warteschlange liest und an die Agenten verteilt.
Um die Programmlogik von der Benutzeroberfläche zu trennen, wird zusätzlich ein Controller verwendet. Der Controller dient als Schnittstelle zum Produzenten, um neue Aufträge zu erstellen, und zum Statusmonitor, um den Status der Aufträge abzufragen. Zusätzlich sollen die Agenten von einem Agenten-Handler verwaltet werden, damit Agenten sich dynamisch am System an- und abmelden können und so die Erweiterbarkeit in Richtung eines verteilten Systems gegeben ist. Die Architektur wird im UML-Klassendiagramm in Abbildung 4 noch einmal gezeigt.

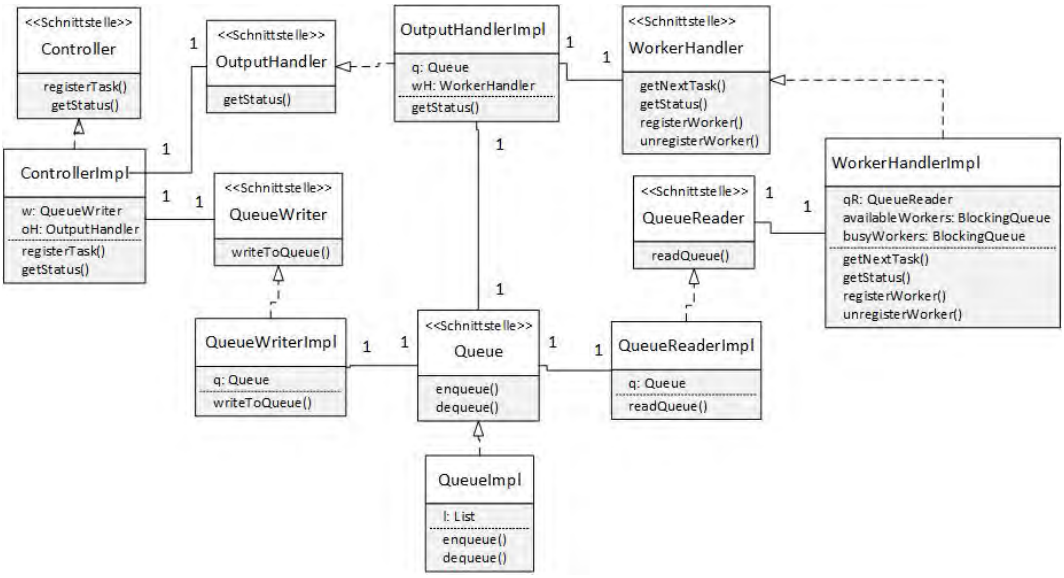


Abbildung 4: UML-Klassendiagramm des ersten Architekturentwurfs

## 4.2. Überarbeitung des ersten Entwurfs und Entwicklung eines Prototypen

Nach näherer Betrachtung ist der erste Entwurf bezüglich des Handlings der Agenten überladen. Die Funktionalität des Agenten-Handlers wird nun in den Konsumenten verlagert, dessen Schnittstelle entsprechend überarbeitet wird. Der nun aktuelle Architekturentwurf wird in einen Prototypen überführt, um die Architektur auf ihre Funktionalität hin zu testen. Das UML-Klassendiagramm in Abbildung 5 verdeutlicht den überarbeiteten Aufbau.

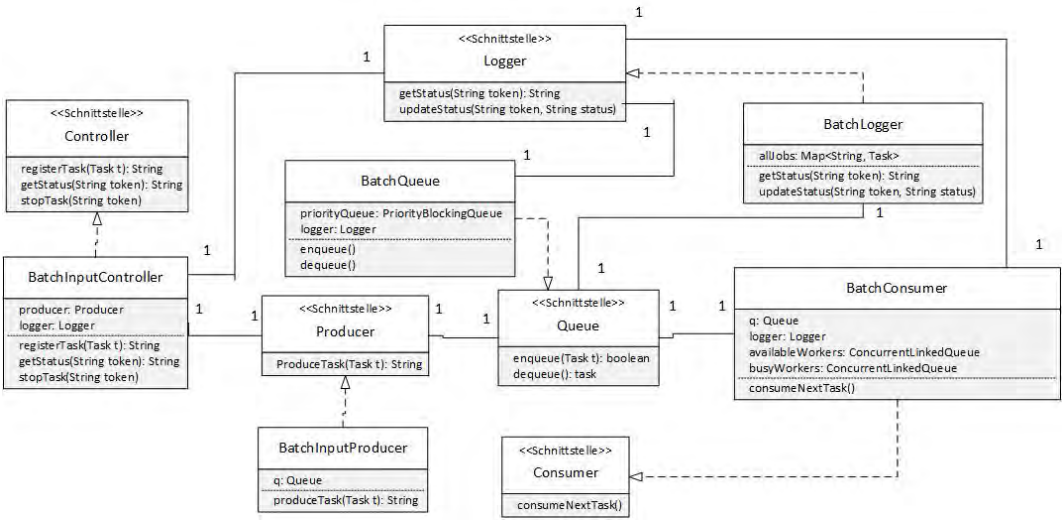


Abbildung 5: UML-Klassendiagramm des überarbeiteten Architekturentwurfs

Der Prototyp wird in Java 8 geschrieben und es werden keine Frameworks verwendet. Aufgrund fehlender Frameworks werden die Aufgaben und Ergebnisse selbst definiert, wobei sich die Komplexität auf die Ausgabe einer Textnachricht in der Standardausgabe des Nutzers beschränkt. Der Fokus dieses Schritts liegt allein auf der Evaluation der Architektur. Es soll überprüft werden, mit welcher Geschwindigkeit sich die Architektur implementieren lässt und ob die Architektur wie geplant funktioniert.
Der Prototyp ist innerhalb eines Tages entstanden, wodurch die Architektur als schnell implementierbar eingestuft wird. Nach verschiedenen positiv verlaufenen Tests ist die Architektur als funktionsfähig genug für die zu erfüllende Aufgabe eingestuft worden, allerdings gehen bei dem Prototypen bei großen Nachrichtenmengen viele Nachrichten verloren. Deshalb ist die Suche nach einer gut skalierbaren Lösung für die Warteschlange essentiell.

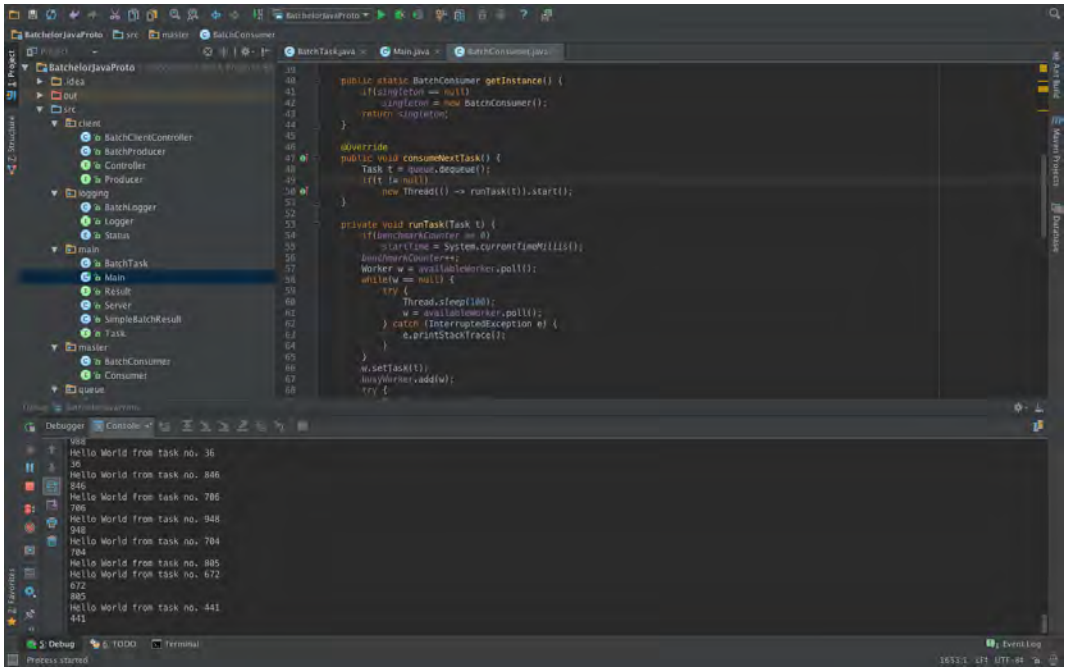


Abbildung 6: Ausführung des Prototypen in IntelliJ IDEA 14

# 5. Evaluation von Programmbibliotheken

Nach Entwicklung der Architektur sollen nun Lösungen für die Komponenten der Applikation evaluiert werden. Am wichtigsten ist hier die Evaluierung von Messaging-Systemen, aber es sollen auch Lösungen zur Enterprise Application Integration und zur Definition von Batch-Jobs betrachtet und evaluiert werden.

## 5.1. Messaging-Systeme

Für die Entwicklung des Programms werden insgesamt sechs verschiedene Messaging-Systeme miteinander theoretisch verglichen. Die wichtigsten Kriterien sind die unterstützten Programmiersprachen, die Unterstützung von Nachrichtenpriorisierungen und eines Queue-Modus. Die vollständige Vergleichstabelle befindet sich unter Tabelle 1 [5,7,8,19,20,23], nachfolgend werden die beiden aussichtsreichsten Lösungen vorgestellt und anschließend miteinander auf ihre Eignung als Messaging-System für die Software hin verglichen.

### 5.1.1. Apache ActiveMQ

ActiveMQ ist nach eigener Aussage das populärste und leistungsstärkste Open-Source-Messaging-System auf dem Markt. Es unterstützt verschiedene Messaging-Protokolle, darunter den Java-exklusiven Java Message Service[G], aber auch Programmiersprachen-unabhängige Protokolle wie OpenWire[G] und AMQP[G]. Intern arbeitet ActiveMQ allerdings nur mit OpenWire. Außerdem kann man Applikationen mit ActiveMQ einfach in bestehende Unternehmenssysteme einbinden, hierzu ist Apache Camel (siehe 5.2.1) nötig. Das Framework besitzt außerdem eine REST API für webbasierte Anwendungen und Ajax[G]- Unterstützung über DHTML, damit auch Webbrowser Teil des Messaging-Systems werden können. ActiveMQ besitzt nicht zuletzt die Möglichkeit, als In-Memory-Message-Provider genutzt zu werden [5].

### 5.1.2. RabbitMQ

RabbitMQ ist im Gegensatz zu ActiveMQ ein vollständiger Messaging-Server, der als eigenständiges Programm auf einem System läuft und Nachrich-

ten von Clients empfängt. Es unterstützt wie ActiveMQ diverse Messaging-Protokolle, u.a. MQTT[G] und AMQP. Verschiedene RabbitMQ-Server können zu einem gemeinsamen Cluster zusammengefasst werden. Außerdem können Warteschlangen auf mehreren Systemen gespiegelt werden, wodurch bei Ausfall eines Systems Warteschlangen und damit die Nachrichten immer noch verfügbar sind. Es existieren Clients für eine Vielzahl an Programmierplattformen, darunter auch nativ per Plugin für Grails. RabbitMQ selbst kann auch mit Plugins um neue Funktionalität erweitert werden [24].

### 5.1.3. Vergleich zwischen den Messaging-Systemen

Auf den ersten Blick eignen sich beide Lösungen für die Software. Sie unterstützen die wichtigsten Messaging-Protokolle, darunter AMQP und JMS, sind Programmiersprachen- unabhängig und unterstützen einen Queue- Modus. Beide Lösungen bieten, zumindest nach eigener Aussage, verlässliche und sichere Nachrichtenübertragungen und unterstützen Grails, was eine wichtige Anforderung ist.
Allerdings ist nach der Evaluation ActiveMQ das Messaging-System der Wahl. Es bietet im Gegensatz zu RabbitMQ die Speicherung von Nachrichten innerhalb des für die Applikation reservierten Speichers an. Die REST API und die Ajax- Unterstützung bieten weitere Möglichkeiten zur Erweiterung der Software und des Unternehmenssystems. Am wichtigsten ist jedoch, dass kein weiterer Server auf einem System laufen muss, was weitere Administrationsarbeit durch den Entwickler erfordert.
Doch nach Absprache mit den anderen Entwicklern ist die Wahl von ActiveMQ nicht optimal. Im Unternehmen wurden bisher schlechte Erfahrungen mit ActiveMQ gemacht, weshalb man langfristig auf RabbitMQ oder eine andere Messaging-Lösung umsteigen will. Dieser Umstand muss bei der Entwicklung berücksichtigt werden, weshalb eine reine In-Memory-Lösung fürs Messaging gesucht werden soll. Außerdem soll der Fokus nun auf der Implementierung einer gut integrierbaren Lösung liegen.

| **Name** | Apache ActiveMQ | HornetQ | Apache Kafka | Apache Qpid | RabbitMQ | Spread Toolkit |
|---|---|---|---|---|---|---|
| **Entwickler** | Apache Software Foundation | JBoss | Apache Software Foundation | Apache Software Foundation | Pivotal Software | Spread Concepts LLC |
| **Lizenz** | Apache 2.0 | Apache 2.0 | Apache 2.0 | Apache 2.0 | Mozilla Public License 1.0 | Spread Open-Source License 1.0 |
| **Sprache** | Java | Java | Scala | Java | Erlang | C |
| **Unterstützte Sprachen** | Java, C, C++, ... | Java | Java | Java, C, C++, Python | Java, Scala, Groovy, Erlang, ... | C, C++, Java, Python |
| **Priorisierung** | ja | ja | nein | ja | nein | nein |
| **Grails-Unterstützung** | ja | ja | ja | ja | ja | nein |
| **Persistent** | ja | ja | ja | ja | ja | ja |
| **Queue-Modus** | ja | ja | nein | ja | ja | nein |
| **Dead-Letter-Queue** | ja | ja | ? | ja | ja | nein |
| **Logs** | ja | ja | ja | ja | ja | nein |

Tabelle 1: Vollständige Liste aller betrachteten Messaging-Systeme

## 5.2. Enterprise-Integration-Frameworks

Nachdem zunächst keine Messaging-Lösung verwendet, diese Möglichkeit aber offen gelassen werden soll, liegt der Fokus nun auf der Wahl einer guten Enterprise-Integration-Lösung.

### 5.2.1. Apache Camel

Die erste mögliche Lösung ist Apache Camel. Mit Camel werden Routing-Regeln definiert, nach denen Nachrichten übermittelt werden. Es benutzt URIs, um verschiedene Transport- und Messaging- Protokolle und -lösungen zu unterstützen. Hierzu gehören u.a. JMS und ActiveMQ, aber auch HTTP. Hierfür implementiert es alle in [15] beschriebenen Muster. Es verwendet für alle Nachrichtenübermittlungen dieselbe API, weshalb Entwickler einen geringen Lernaufwand leisten müssen. Desweiteren kann Apache Camel in Spring-Applikationen verwendet werden und wird nahtlos integriert [6].

### 5.2.2. Spring Integration

Spring Integration ist ein Enterprise-Integration-Framework für Spring- basierte Applikationen. Es erlaubt performantes Messaging und unterstützt die Integration von Spring-Applikationen in bestehenden Unternehmenssysteme. Java-Objekte werden per Messaging miteinander verbunden, wodurch starke Abhängigkeiten unter ihnen vermieden werden.
Spring Integration implementiert genau wie Apache Camel alle in [15] vorgestellten Enterprise-Integration-Muster, darunter Endpunkte, Nachrichtenkanäle und Filter. Messaging kann dabei über verschiedene Protokolle wie HTTP und FTP und mit unterschiedlichen Technologien erfolgen. Zu letzterem gehört auch die Integration von JMS- und RabbitMQ-Clients [26].

### 5.2.3. Auswahl der Integrationslösung

Trotz des schwierigen Umgangs mit Kanälen mithilfe von Spring XML-Dateien und Problemen beim Testen von geschriebenem Code [22], wird für die Applikation das Spring-Integration- Framework verwendet. Beide Lösungen bieten einen ähnlichen Funktionsumfang, was der Tatsache geschuldet ist, dass sie bekannte und anerkannte Muster implementieren. Beide Lösungen unterstützen weiterhin eine Vielzahl an Transport- und Messaging-Lösungen, weshalb sie sich momentan als auch in Zukunft als Integrationslösung im

Unternehmen eignen.
Entscheidend für die Wahl von Spring Integration ist die Erfahrung mit Spring innerhalb des Unternehmens. Der Großteil der selbstentwickelten Programme verwendet das Grails-Framework, welches wiederum das Spring- Framework verwendet. Bei der Beschreibung von Kanälen lässt sich somit auf bestehende Erfahrung zurückgreifen. Außerdem kann mit Hilfe der erfahrenen Entwickler die Applikation über das Spring-Framework getestet werden.

## 5.3. Batch Processing

### 5.3.1. Spring Batch

An letzter Stelle steht eine Lösung zur Definition von Batch-Jobs. Da sowohl Grails als auch Spring Integration verwendet werden sollen, liegt die Entscheidung schnell beim Spring-Batch-Framework. Spring Batch stellt wiederverwendbare Funktionen zur Verarbeitung großer Datenmengen bereit. Hierzu gehören Logging, ein Transaktionsmanagement, das Bereitstellen von Statistiken und die Möglichkeit, definierte Jobs zu pausieren, zu überspringen oder auch ganz zu stoppen [25].
Ein Batch-Job ist in Spring Batch ein Container für *Steps* und wird anhand seines Namens, seiner *Steps* und seiner Möglichkeit eines Neustarts definiert. Von jedem Job werden *JobInstance*-Objekte erzeugt, die sich durch die übergebenen Parameter unterscheiden. Von jedem *JobInstance*-Objekt existiert dann mindestens ein *JobExecution*-Objekt, dass die eigentliche Ausführung des Jobs beschreibt. Sollte eine Ausführung fehlschlagen, so wird zu einem späteren Zeitpunkt ein neues *JobExecution*-Objekt erzeugt, solange der Job einen Neustart zulässt. Wie bereits erwähnt werden Jobs durch *Steps* beschrieben. Dabei beschreibt ein *Step* eine unabhängige sequentielle Phase des Jobs und kann so einfach oder komplex wie gewünscht sein. Von jedem *Step* existieren auch Objekte, die die Ausführung beschrieben. Der ganze Mechanismus wird durch ein *JobLauncher*-Objekt gestartet. Alle genannten Komponenten sind in einem *JobRepository*-Objekt gespeichert [27].

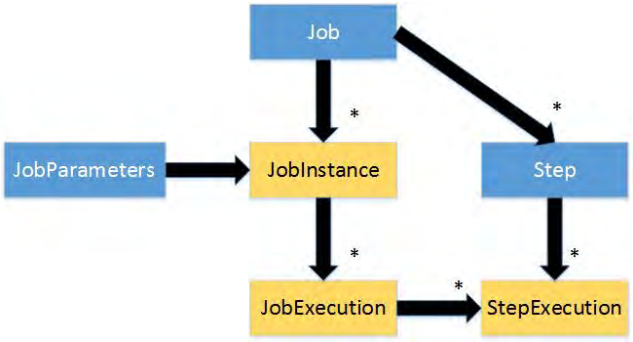


Abbildung 7: Aufbau von Jobs in Spring Batch

### 5.3.2. Zum Vergleich: Easy Batch

Easy Batch ist ein weiteres Framework, welches Batch-Processing mit Java vereinfachen soll. Der Unterschied zu Spring Batch liegt in der Datenverarbeitung: Während Spring Batch Daten in Blöcken verarbeitet, werden Daten in Easy Batch Eintrag für Eintrag verarbeitet. Es soll eine leichtgewichtige Alternative darstellen, die zwar einfach zu lernen ist, aber nicht den gesamten Funktionsumfang von Spring Batch bietet [13]. Easy Batch verfolgt nicht das Job-Konzept von Spring Batch, stattdessen wird eine Verarbeitungskette verwendet. Zunächst werden Daten gelesen und gefiltert. Danach werden die gelesenen Daten auf Java Objekte abgebildet, validiert und abschließend verarbeitet und gespeichert. Das Ergebnis der Verarbeitung wird weiterhin in einem Bericht gespeichert [12]. Easy Batch bietet allerdings nativ keine Möglichkeit, fehlgeschlagene Jobs neuzustarten, Daten können nicht partitioniert werden und es bietet keine Fehlertoleranz [11, Folie 46].

# 6. Implementierung der Lösung

In diesem Abschnitt wird zunächst der Aufbau der Entwicklungsumgebung mit allen verwendeten Technologien beschrieben. Darauf folgt eine Beschreibung der implementierten Architektur zusammen mit den Lösungen in der Implementierung. Darin enthalten ist ein Vergleich mit der anfänglich geplanten Architektur. Die implementierte Lösung wird anschließend den Anforderungen gegenübergestellt und es wird ein Beispielprojekt vorgestellt, in dem die Software verwendet wird.

## 6.1. Verwendete Technologien und Entwicklungsumgebung

Die Entwicklung erfolgt auf einem Macbook Air aus dem Jahr 2014, welches mit einem Intel Core i5 Prozessor mit 1,4 GHz Taktrate und 4 GB Arbeitsspeicher ausgestattet ist. Als Betriebssystem kommen die zu den jeweiligen Zeitpunkten aktuellen Beta-Versionen von Mac OS X 10.11 El Capitan zum Einsatz.
Als Hauptframework kommt zunächst Grails in der Version 2.3.11, später in der Version 2.2.4 zum Einsatz. Der Gebrauch von Grails ist unumgänglich, da die im Unternehmen verwendeten Applikationen, allen voran das Content-Management-System, dieses Framework verwenden. Somit ist es außerdem möglich, die Implementierung als Plugin bereitzustellen. Mit Grails 2.2.4 wird als Programmiersprache zwangsweise Groovy 2.0 verwendet [1], allerdings kann man den Quellcode beliebig mit Java-Quellcode mischen. Zum Ausführen des Programms kommt Java 7 zum Einsatz, in dessen Bytecode der Programmcode kompiliert wurde. Für die Definition der Batch-Jobs wird Spring Batch 2.1.9 verwendet, dessen Verwendung durch ein Grails-Plugin stark vereinfacht wird [4]. Mithilfe des Plugins ist es möglich, Jobs mit der Spring Domain Specific Language[G] anstatt wie üblich mit XML zu beschreiben. Diese Vorgehensweise erhöht die Übersichtlichkeit des Programmcodes merklich und vereinfacht den Umgang mit den definierten Jobs. Zur Bereitstellung von Nachrichtenkanälen und einer Warteschlange kommt Spring Integration 2.2.6.RELEASE zum Einsatz.
Als Entwicklungsumgebung wird IntelliJ IDEA 14 Ultimate verwendet, das ein Plugin zur einfacheren Entwicklung mit Groovy und Grails bereitstellt.

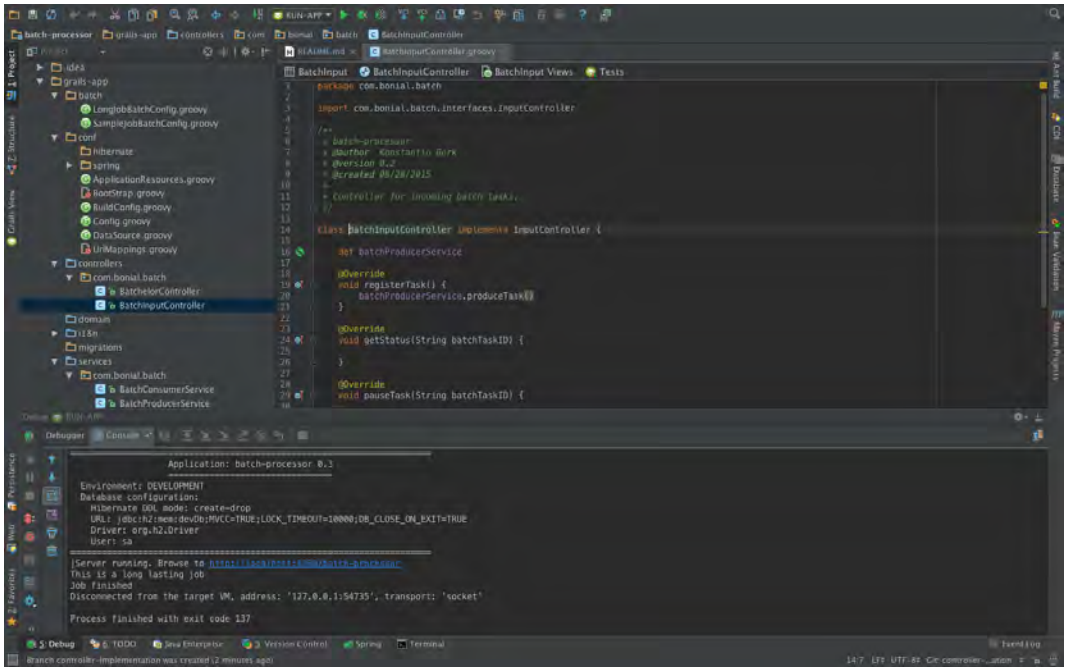

Abbildung 8: IntelliJ IDEA 14 mit dem Quellcode des Controllers

Der Quellcode wird über Git[G] verwaltet und in einem privaten Repository auf Github gehostet.

## 6.2. Die implementierte Architektur

Die implementierte Architektur unterscheidet sich nicht stark von der in Kapitel 4.2 beschriebenen Architektur, es gibt nur kleinere Anpassungen bezüglich der Verwendung der verschiedenen Frameworks. Durch die Verwendung von Spring Batch ist es nicht mehr nötig, eine eigene Job-Klasse zu definieren. Die mit Spring Batch definierten Jobs bieten eine Schnittstelle, um sie zu starten, zu stoppen und nach ihrem aktuellen Status zu fragen. Letzteres führt zunächst dazu, den Statusmonitor aus der Architektur zu entfernen und die von dem Spring-Batch-Plugin bereitgestellten Daten direkt für den Benutzer zugänglich zu machen. Ohne viel Programmieraufwand ist hier allerdings nicht die Einführung einzigartiger Token für die Aufträge möglich. Außerdem ist durch die Einführung eines eigenen Statusmonitors die Quellcodebasis bekannt und man kann die Software schnell bezüglich Ausgaben in Form von Logdateien erweitern.
Insgesamt besteht das Programm, welches die Anforderungen erfüllen soll, aus acht implementierten Klassen und fünf Schnittstellen. Die Schnittstellen spezifizieren den Produzenten, den Konsumenten, die Warteschlange, die

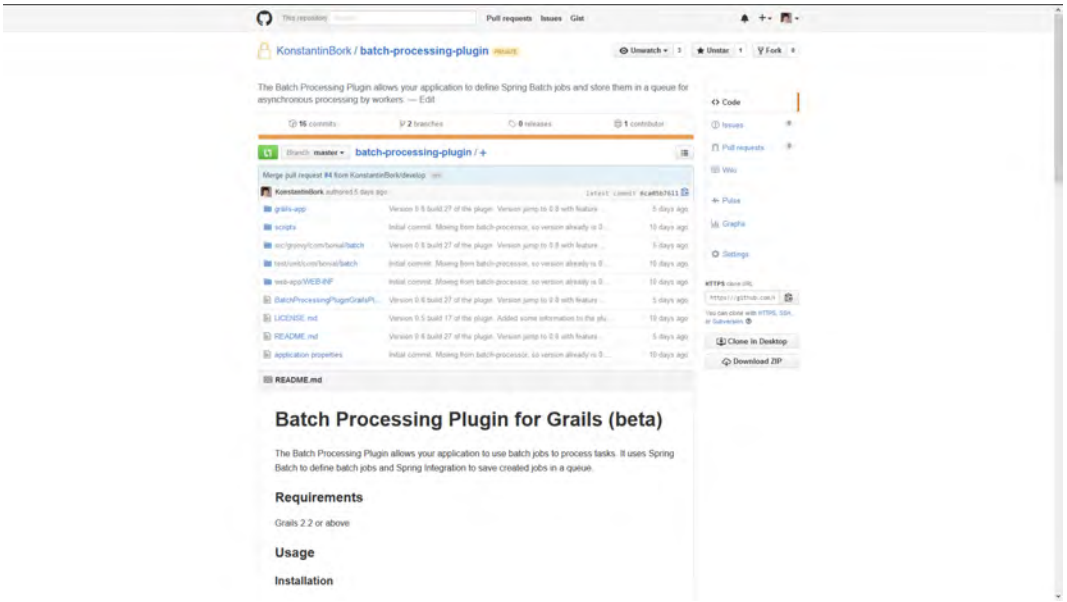


Abbildung 9: Repository des Plugins auf Github

Agenten und den Controller. Zu allen Schnittstellen existiert eine Klasse, für den Produzenten und die Warteschlange existieren zwei Implementierungen, die jeweils mit einer einfachen Warteschlange und einer Prioritätswarteschlange arbeiten. Zusätzlich zu den Implementierungen der Schnittstellen existiert noch der schon beschriebene Statusmonitor, hinzu kommen weitere Grails-spezifische Skripte zur Konfiguration der Applikation, die Dateien für die Benutzeroberfläche der Applikation in Form von Groovy Server Pages[G] und Beispiele zur Spezifizierung von Spring-Batch-Jobs. Bei der Implementierung wurde wie nach Spezifikation erforderlich auf einen modularen Aufbau geachtet, um die Komponenten bei Bedarf erweitern oder ersetzen zu können.
Der Controller nimmt die Nutzerdaten an und leitet sie an das Programm weiter. Außerdem bietet er dem Nutzer eine Schnittstelle an, über die Aufträge gestoppt und ihr aktueller Status abgefragt werden kann. Wie vorher definiert hat der Controller zur Erfüllung der Spezifikationen Verbindungen zum Produzenten und zum Statusmonitor. Zusätzlich existiert eine Verbindung zum Spring-Batch-Plugin, um die bereitgestellten Methoden zum Umgang mit Batch-Jobs verwenden zu können. Es existiert außerdem eine Verbindung zum Konsumenten, um dort einen Thread zum Konsum der Aufträge zu starten. Diese Verbindung soll in einer späteren Version entfernt werden.
Der Produzent nimmt die Daten vom Controller entgegen und bereitet sie zur Speicherung in der Warteschlange vor. Hierfür existiert eine Verbindung

zur Warteschlange, darüber hinaus existieren Verbindungen zum Statusmonitor, um den Status eines Auftrags aktualisieren zu können, und zum Spring-Batch-Plugin.
Die Warteschlange speichert die einkommenden Nachrichten vom Produzenten in einem von dem Spring-Integration-Framework bereitgestellten Nachrichtenkanal, der eine BlockingQueue verwendet [21]. Es existiert nur eine Verbindung zum Statusmonitor, um den Status eines Auftrags aktualisieren zu können.
Der Konsument nimmt die Nachrichten von der Warteschlange entgegen, um sie an die Agenten weiterzuleiten. Hierfür besitzt er zwei *ConcurrentLinkedQueue*-Objekte, die die verfügbaren und beschäftigten *Worker*-Objekte speichern. Um Nachrichten aus der Warteschlange konsumieren zu können, besitzt der Konsument eine Verbindung zu ihr.
Alle bisher genannten Klassen außer dem Controller sind als *Grails Services* implementiert. Dadurch haben sie bestimmte Eigenschaften. Sie sind für Transaktionen mit Datenbanken konfiguriert, wodurch bei Laufzeitfehlern automatisch ein Rollback von Daten durchgeführt wird. Außerdem wird standardmäßig nur ein Objekt von jedem Service erzeugt und mittels *Dependency Injection*[G] können Services allein durch ihren Namen von anderen Klassen gefunden werden [2].
Zuletzt existieren die Agenten, die die Nachrichten vom Konsumenten erhalten und aus ihnen den Job und die Parameter zur Bearbeitung erhalten und den Job dann durchführen. Ihre Anzahl ist im Konsumenten konfigurierbar.
Die Beziehungen der Komponenten untereinander werden im UML- Klassendiagramm in Abbildung 10 dargestellt.

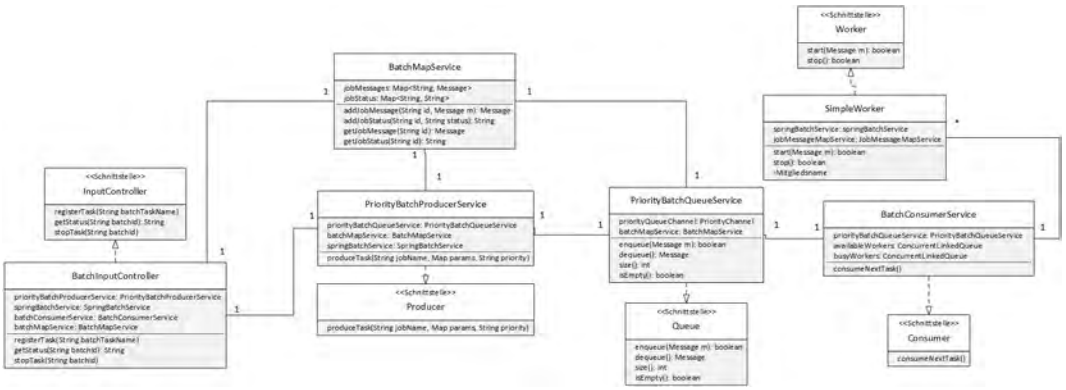


Abbildung 10: Finale Architektur der Applikation

## 6.3. Vorgehensweise bei der Implementierung

Nach Evaluierung der benötigten Softwarekomponenten wird zuerst das Projekt lokal erzeugt und in ein privates Repository auf Github geladen. Da aufgrund von Problemen bei der Auflösung von Plugin-Abhängigkeiten das Einbinden von erforderlichen Grails-Plugins nicht möglich ist, wird das Projekt zunächst als vollständige Grails-Applikation und mit Grails 2.3.11 entwickelt. Hier offenbart sich eine Schwäche von Grails: Das Framework nimmt Programmierern viel Arbeit bei der Projektverwaltung ab, dafür ist dem Programmierer oft nicht der Grund eines Problems offensichtlich, z.B. wenn Abhängigkeiten nicht aufgelöst werden können. Aufgrund des beschriebenen Umstands ist es geplant, die Applikation erst zu einem späteren Zeitpunkt in ein Grails-Plugin zu portieren, das unter Grails 2.2.4 läuft.

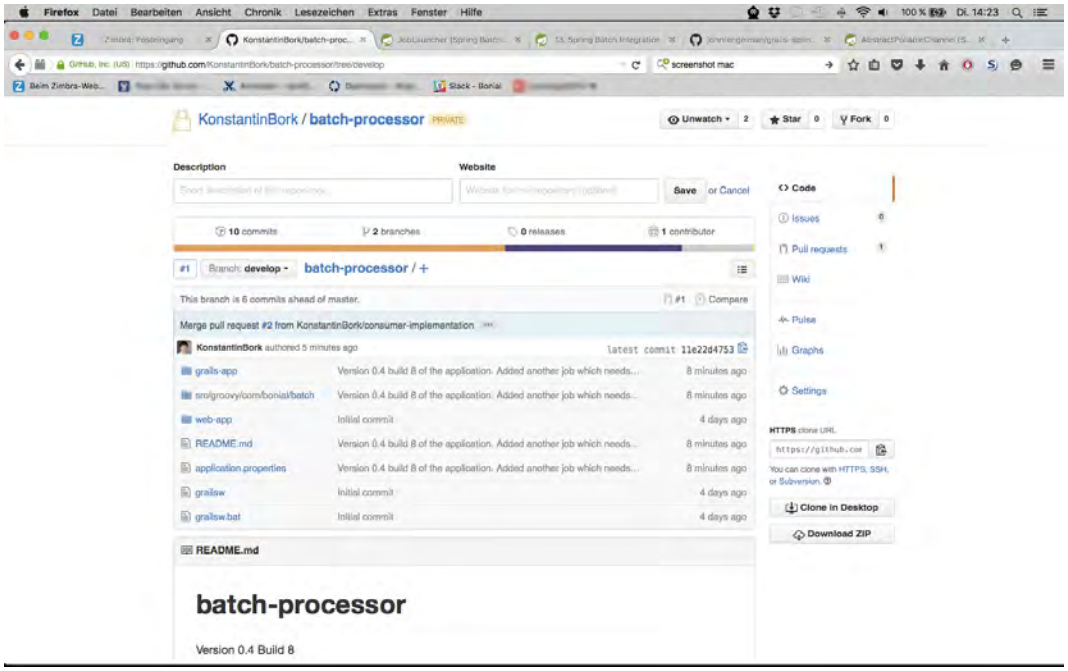


Abbildung 11: Erstes Projekt-Repository auf Github

Als nächstes wird die Struktur des Repositorys definiert. Anfangs existiert im Git-Repository nur ein Entwicklungszweig namens “master”, auf dem man normalerweise nicht entwickeln sollte. Die übliche Vorgehensweise ist es, von diesem Zweig aus einen neuen Entwicklungszweig “develop” zu erstellen, von dem man wiederum für jede neue zu entwickelnde Funktionen einen Zweig erstellt. Alle von “develop” erzeugten Zweige werden dann erst nach erfolgreicher Begutachtung des Codes durch einen anderen Entwickler und erfolgreichen Tests wieder mit “develop” zusammengeführt. Ziel die-

ser Vorgehensweise ist es, dass die beiden Entwicklungszweige “master” und “develop” immer eine lauffähige Version der Applikation haben.
Entsprechend gestaltet sich die Implementierung der einzelnen Funktionen. Zunächst wird die Projektstruktur eingerichtet, was in diesem Fall die Definition der Schnittstellen bedeutet. Anschließend wird mit Spring Batch ein Beispieljob implementiert, der eine “Hello World!”-Nachricht in der Standardausgabe druckt. Hierfür sind eine BatchConfig-Datei, die den Job definiert, und mindestens eine Klasse, die den Job implementiert, notwendig. Listing 1 definiert den Job *sampleJob*, der aus einem Schritt besteht, in dem ein Tasklet ausgeführt wird. Die zweite Möglichkeit für einen Schritt kann das Lesen, Bearbeiten und Schreiben von Daten sein, wie Listing 3 zeigt. Die Implementierung des Jobs findet dann in einer eigenen Datei statt, im Falle des *sampleJob* wird dies in Listing 2 getan.

```
1  import com.bonial.batch.jobs.SampleJobTasklet
2
3  /**
4   * batch-processor
5   * @author Konstantin Bork
6   * @version 0.1
7   * @created 08/28/2015
8   *
9   * An example for the look of Spring Batch job
        configurations.
10  */
11
12 beans = {
13
14     xmlns batch:"http://www.springframework.org/schema/batch
           "
15
16     /**
17         A Spring Batch job consists of at least one step.
               Each step is performed one after the other.
18         Each job and each step have an ID to identify them
               so prevent having jobs and steps with the same
               name.
19         Each step can then look in one of two ways:
```

```
20              1) Data are read from a source, processed and
                   written persistently into a file, database
                   etc.
21              2) One tasklet executes a task.
22          The tasklet is referenced in the step.
23       */
24      batch.job(id: 'helloWorldJob') {
25          batch.step(id: 'printStep') {
26              batch.tasklet(ref: 'printHelloWorld')
27          }
28      }
29
30      /**
31          Reference of the tasklet. It gets the Tasklet
               implementation as a parameter and sets a bean.
32       */
33      printHelloWorld(SampleJobTasklet) { bean ->
34          bean.autowire = "byName"
35      }
36
37  }
```

Listing 1: Erster Entwurf von SampleJobBatchConfig.groovy

```
1  package com.bonial.batch.jobs
2
3  import org.springframework.batch.core.StepContribution
4  import org.springframework.batch.core.scope.context.
       ChunkContext
5  import org.springframework.batch.core.step.tasklet.Tasklet
6  import org.springframework.batch.repeat.RepeatStatus
7
8  /**
9   * batch-processor
10  * @author Konstantin Bork
11  * @version 0.1
12  * @created 08/28/2015
13  *
14  * Example for the look of a tasklet.
15  */
16
```

```
17 class SampleJobTasklet implements Tasklet {
18
19     @Override
20     RepeatStatus execute(StepContribution stepContribution,
           ChunkContext chunkContext) throws Exception {
21         println("Hello World!")
22         return RepeatStatus.FINISHED
23     }
24
25 }
```

Listing 2: Erster Entwurf von SampleJobTasklet.groovy

```
 1 import com.bonial.batch.jobs.FileLineMapper
 2 import com.bonial.batch.jobs.NameItemProcessor
 3 import com.bonial.batch.jobs.NameItemWriter
 4 import org.springframework.batch.item.file.
       FlatFileItemReader
 5
 6 beans {
 7
 8     xmlns batch:"http://www.springframework.org/schema/batch
           "
 9
10     batch.job(id: 'paramInputJob') {
11         batch.step(id: 'paramInput') {
12             batch.tasklet {
13                 batch.chunk (
14                         reader: 'fileReader',
15                         processor: 'lineProcessor',
16                         writer: 'fileWriter',
17                         'commit-interval': 5
18                 )
19             }
20         }
21     }
22
23     fileReader(FlatFileItemReader) { bean ->
24         bean.scope = 'step'
25         resource = "#{jobParameters['file']}"
26         lineMapper = ref('fileLineMapper')
```

```
27      }
28
29      fileLineMapper(FileLineMapper)
30
31      lineProcessor(NameItemProcessor) { bean ->
32          bean.autowire = "byName"
33      }
34
35      fileWriter(NameItemWriter) { bean ->
36          bean.autowire = "byName"
37      }
38
39  }
```

Listing 3: ParamInputJobBatchConfig.groovy als Beispiel für Jobs mit Parameter

Für jede neue Funktion wird nun ein neuer Entwicklungszweig erzeugt und nach Vollendung der Implementierung mit dem “develop”-Zweig zusammengeführt. So werden zunächst eine einfache Warteschlange und der Produzent implementiert. Danach folgen der Konsument zusammen mit den Agenten und zuletzt der Controller. Desweiteren werden weitere Jobs definiert und implementiert, um die Funktionalität der Implementierung testen zu können.
Nachdem diese Hauptfunktionalität implementiert ist, beginnt die Portierung des Programms als Plugin auf Grails 2.2.4. Die anfangs erwähnten Probleme tauchen nun nicht mehr auf, sodass die letzten Anforderungen implementiert werden können. So wird nun eine Prioritätswarteschlange implementiert, damit Jobs nach ihrer Wichtigkeit gespeichert und entsprechend abgearbeitet werden können. Außerdem wird der Umgang mit den Agenten überarbeitet, sodass sie auch mit Sicherheit durch einen Job blockiert werden. In der ersten Fassung ist dies nicht gegeben, denn nach Start eines Jobs sind sie sofort wieder bereit zur Bearbeitung eines anderen Jobs. Außerdem wird nun ein Statusmonitor für die Jobs implementiert, um ihren Status von der Erstellung bis zum Abschluss der Bearbeitung verfolgen zu können. Anschließend wird mittels Integrationstests die Funktionalität der entwickelten Software getestet. Das Besondere der Integrationstests gegenüber den üblich genutzten Unit-Tests ist, dass bei der Durchführung die gesamte Applikation geladen wird und somit beispielsweise per *Dependen-*

*cy Injection* Objekte anhand ihres Namens in die Tests eingefügt werden können. Ein Beispiel für Integrationstests findet sich in Listing 4.

```
1  package com.bonial.batch
2
3  import org.springframework.integration.Message
4  import org.springframework.integration.message.
       GenericMessage
5
6  import org.junit.*
7
8  class BatchConsumerTests {
9
10     PriorityBatchQueueService priorityBatchQueueService
11
12     @Before
13     void setUp() {
14         // Setup logic here
15     }
16
17     @After
18     void tearDown() {
19         // Tear down logic here
20     }
21
22     @Test
23     void testConsumeNextTask() {
24         assert priorityBatchQueueService.isEmpty()
25         Message m = new GenericMessage<>("Hello World!")
26         priorityBatchQueueService.enqueue(m)
27         assert priorityBatchQueueService.size() == 1
28         Message n = priorityBatchQueueService.dequeue()
29         assert m == n
30     }
31
32 }
```

Listing 4: BatchConsumerTests.groovy als Beispiel für Integrationstests

Anschließend wird das entwickelte Plugin in ein Beispielprojekt eingebunden und die Funktionalität getestet.

## 6.4. Beispielprojekt mit dem entwickelten Plugin und Tests

Um das Plugin testen zu können, wird ein Beispielprogramm erstellt und das Plugin darin eingebunden. Dies geschieht mit der Zeile `grails.plugin.location."batch-processing-plugin" = "../batch-processing-plugin"` in der *BuildConfig.groovy*-Datei im Projekt. Um alleine die Funktionalität des entwickelten Plugins zu testen, wird nur eine Controller-Klasse implementiert, die das Plugin anspricht. Gesteuert wird der Controller über ein Webinterface (siehe Abbildung 12).

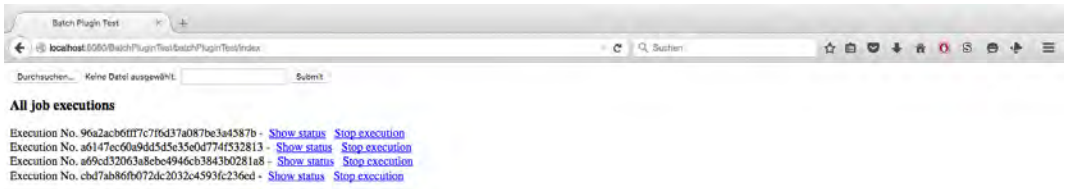


Abbildung 12: Webinterface der Beispielanwendung

Um die Leistungsfähigkeit des Plugins zu testen, wird der in Listing 3 definierte und in Listing 5 implementierte Job ausgeführt. Der Job erhält als Parameter eine Testdatei, die 100000 Datensätze mit Vor- und Nachnamen in umgekehrter Reihenfolge enthält. Der Job soll nun die Reihenfolge der Nennung ändern, sodass zuerst der Vor- und dann der Nachname genannt wird. Das Ergebnis wird in der Standardausgabe ausgegeben.

```
1  import org.springframework.batch.item.ItemProcessor
2  import org.springframework.batch.item.ItemWriter
3  import org.springframework.batch.item.file.LineMapper
4
5  class FileLineMapper implements LineMapper<NameItem> {
6
7      @Override
8      NameItem mapLine(String s, int i) throws Exception {
9          NameItem item = new NameItem()
10         String[] tokens = s.split(/,/)
11         if(tokens.length == 1)
12             item.lastName = tokens[0]
13         else if(tokens.length == 2) {
14             item.lastName = tokens[0]
15             item.firstName = tokens[1]
16         }
17         return item
18     }
19
20 }
21
22 class NameItemProcessor implements ItemProcessor<NameItem,
       Name> {
23
24     @Override
25     Name process(NameItem nameItem) throws Exception {
26         Name name = new Name()
27         name.with {
28             firstName = nameItem.firstName
29             lastName = nameItem.lastName
30             completeName = nameItem.toString()
31         }
32         return name
33     }
34
35 }
36
37 class NameItemWriter implements ItemWriter<Name> {
38
```

```
39      @Override
40      void write(List<? extends Name> list) throws Exception {
41          list.each {
42              println(it.completeName)
43          }
44      }
45
46  }
```

Listing 5: ParamInputJob.groovy

Erwartungsgemäß dauert die Bearbeitung der Datei einige Zeit, doch so kann die parallele Ausführung mehrerer Jobs getestet werden. Dabei tritt das Problem auf, dass ab sieben parallelen Ausführungen das Webinterface nicht reagiert, bis mindestens ein Job vollständig durchgeführt wurde. Dieses Verhalten tritt auch auf einem deutlich leistungsstärkeren PC mit einem Intel Xeon-Prozessor der Haswell-Generation und 8 GB Arbeitsspeicher auf. Der Grund für das Verhalten ist unklar, da die verwendete In-Memory-Queue nicht leistungsfähig genug sein kann oder manche Abhängigkeiten unter den Komponenten noch zu stark sind. Im Rahmen dieser Arbeit wird die Maximalanzahl an parallel ausgeführten Jobs auf fünf beschränkt, für den späteren Produktivbetrieb soll der Grund des beschriebenen Verhaltens jedoch untersucht werden.
Abgesehen von diesem Problem ist es wie gewünscht möglich, dass man den Status eines Jobs abfragen kann und seine Ausführung jederzeit stoppen kann. Beide Funktionen sind mithilfe der vom Spring-Batch-Plugin mitgelieferten Controller überprüfbar, die Informationen zu allen im System gestarteten Jobs bereitstellen. Somit ist zumindest ein Testbetrieb mit dem Plugin möglich.

# 7. Zusammenfassung

## 7.1. Vergleich des fertigen Programms mit den Anforderungen

Das entwickelte Programm ist wie gefordert ein agentenbasiertes System, welches Nutzeranfragen in Form von Nachrichten asynchron verarbeitet. Es ist als Plugin entwickelt worden, ist aufgrund der Verwendung von Nachrichten zur Kommunkation generisch und somit in viele bestehende Applikation integrierbar. Hilfreich ist hierbei die Verwendung des Spring- Integration-Frameworks. Die Verwendung dieses Frameworks erlaubt außerdem die spätere Verwendung eines ausgewachsenen Messaging-Systems wie RabbitMQ, obwohl zurzeit eine In-Memory-Queue verwendet wird. Die Agenten selbst bieten eine Schnittstelle zum Starten und Stoppen von Jobs, außerdem kann man den Status des Jobs abfragen, an dem sie zurzeit arbeiten. Weiterhin ist die Anzahl wie gefordert konfigurierbar, auch wenn wie in Kapitel 6.4 beschrieben derzeit die Nutzeroberfläche ab einer bestimmten Anzahl gleichzeitig durchgeführter Jobs träge reagiert. Die Aufträge bieten eine Schnittstelle zum Starten, Stoppen und zur Statusabfrage, sie sind über eindeutige Token identifizierbar und durch die Verwendung von Nachrichten generisch für verschiedene Datei- und Datentypen aufgebaut. Allerdings müssen neben der Lösung der Oberflächenproblematik noch Abhängigkeiten einzelner Komponenten abgebaut werden, damit einzelne Komponenten einfacher gewartet oder ausgetauscht werden können.

## 7.2. Ausblick

Auch nach dieser Arbeit soll das entwickelte Programm weiterentwickelt werden. Die im letzten Abschnitt erwähnten Probleme sollen möglichst zeitnah gelöst werden. Sobald diese Aufgaben erledigt und Entwicklerressourcen im Unternehmen frei sind, soll das Plugin in die bestehenden Applikationen integriert werden. Außerdem sollen Implementierungen der Warteschlange zur Verwendung eines Messaging-Systems geschrieben werden. Wunsch des Autors ist es weiterhin, das entwickelte Plugin als Open-Source-Projekt zu veröffentlichen, wobei zunächst ein Fork für die Weiterentwicklung innerhalb des Unternehmens erstellt werden soll. Deshalb wird der Code des Projekts auf Github gespeichert. Sobald die Freigabe durch

das Unternehmen erfolgt, findet man das Projekt unter
https://github.com/konstantinbork.

# Literatur

[1] Graeme Rocher & Peter Ledbrook & Marc Palmer & Jeff Brown & Luke Daley & Burt Beckwith. Introduction - Reference Documentation. http://grails.github.io/grails-doc/2.2.4/guide/introduction.html, abgerufen am 12.09.2015.

[2] Graeme Rocher & Peter Ledbrook & Marc Palmer & Jeff Brown & Luke Daley & Burt Beckwith. The Service Layer - Reference Documentation. http://grails.github.io/grails-doc/2.2.4/guide/services.html, abgerufen am 12.09.2015.

[3] IBM Corporation. Mainframes working after hours: Batch processing. http://www-01.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos .zmainframe/zconc_batchproc.htm, abgerufen am 12.09.2015.

[4] John Engelman et al. Spring Batch Grails Plugin. https://github.com/johnrengelman/grails-spring-batch, abgerufen am 12.09.2015.

[5] Apache Software Foundation. Apache ActiveMQ. http://activemq.apache.org/, abgerufen am 15.09.2015.

[6] Apache Software Foundation. Apache Camel. http://camel.apache.org/, abgerufen am 15.09.2015.

[7] Apache Software Foundation. Apache Kafka. http://kafka.apache.org/, abgerufen am 15.09.2015.

[8] Apache Software Foundation. Apache Qpid. https://qpid.apache.org/, abgerufen am 15.09.2015.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. mitp Professional. mitp/bhv, 2015.

[10] Shay Hassidim. Master-worker pattern. http://docs.gigaspaces.com/sbp/master-worker-pattern.html, 2009, abgerufen am 12.09.2015.

[11] Mahmoud Ben Hassine. Easy Batch. https://speakerdeck.com/benas/easy-batch, 2014, abgerufen am 16.09.2015.

[12] Mahmoud Ben Hassine. Architecture. http://www.easybatch.org/documentation/architecture.html, abgerufen am 16.09.2015.

[13] Mahmoud Ben Hassine. Frequently Asked Questions. http://www.easybatch.org/documentation/faq.html, abgerufen am 16.09.2015.

[14] Springer Gabler Verlag (Herausgeber). Gabler Wirtschaftslexikon, Stichwort: Stapelbetrieb, online im Internet. http://wirtschaftslexikon.gabler.de/Archiv/56393/stapelbetrieb-v11.html, abgerufen am 12.09.2015.

[15] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2012.

[16] Roger Hughes. The producer consumer pattern. https://dzone.com/articles/producer-consumer-pattern, 2013, abgerufen am 12.09.2015.

[17] ITWissen.info. Middleware. http://www.itwissen.info/definition/lexikon/Middleware-middleware.html, abgerufen am 12.09.2015.

[18] ITWissen.info. Mom (message oriented middleware). http://www.itwissen.info/definition/lexikon/message-oriented-middleware-MOM.html, abgerufen am 12.09.2015.

[19] JBoss. HornetQ. http://hornetq.jboss.org/, abgerufen am 15.09.2015.

[20] Spread Concepts LLC. Spread Toolkit. http://www.spread.org/, abgerufen am 15.09.2015.

[21] Pivotal. Class QueueChannel. http://docs.spring.io/spring-integration/docs/2.2.6.RELEASE/api/org/springframework/ integration/channel/QueueChannel.html, abgerufen am 12.09.2015.

[22] Christian Posta. Top posts of 2013: Apache camel vs. spring integration. https://dzone.com/articles/light-weight-open-source, 2013, abgerufen am 15.09.2015.

[23] Pivotal Software. RabbitMQ. https://www.rabbitmq.com/, abgerufen am 15.09.2015.

[24] Pivotal Software. RabbitMQ Features. https://www.rabbitmq.com/features.html, abgerufen am 15.09.2015.

[25] Pivotal Software. Spring Batch. http://projects.spring.io/spring-batch/, abgerufen am 15.09.2015.

[26] Pivotal Software. Spring Integration. http://projects.spring.io/spring-integration/, abgerufen am 15.09.2015.

[27] Pivotal Software. The domain language of batch. http://docs.spring.io/spring-batch/2.1.x/reference/html/domain.html, abgerufen am 16.09.2015.

[28] Wikipedia. Architekturmuster. https://de.wikipedia.org/wiki/Architekturmuster, abgerufen am 15.09.2015.

# Glossar

**Advanced Message Queuing Protocol (AMQP):** offenes Netzwerkprotokoll für eine Message Oriented Middleware, das unabhängig von der Programmiersprache ist

**Asynchronous JavaScript and XML (Ajax):** Konzept zur asynchronen Datenübertragung zwischen einem Webbrowser und einem Server

**Apache Tomcat:** Open-Source-Webserver und Webcontainer, der in Java geschriebene Webanwendungen ausführen kann

**Batch Window:** Zeitfenster, in dem wenige Nutzerinteraktionen mit dem System stattfinden, und das deshalb zur Ausführung von Batch-Jobs geeignet ist

**Content-Management-System:** Software zur Erstellung, Bearbeitung und Organisation von Inhalten

**Dependency Injection:** ein Entwurfsmuster, das die Abhängigkeiten eines Objekts zur Laufzeit reglementiert

**Domain Specific Language (DSL):** formale Sprache zur Interaktion zwischen Mensch und Computer für ein bestimmtes Problemfeld

**Git:** Software zur verteilten Versionsverwaltung von Dateien

**Grails:** Webapplikationsframework für die Programmiersprache Groovy

**Groovy Server Pages:** View-Objekte in Grails-Applikationen

**Java Message Service (JMS):** Programmierschnittstelle für Message Oriented Middleware in der Programmiersprache Java

**Message Queue Telemetry Transport (MQTT):** offenes Nachrichtenprotokoll für Machine-to-Machine-Kommunikation zur Übertragung von Telemetrie-

Daten

**OpenWire:** offenes Netzwerkprotokoll für eine Message Oriented Middleware, das unabhängig von der Programmiersprache ist

**Remote Procedure Calls:** Technik zur Realisierung von Interprozesskommunikation, um den Aufruf von Funktionen auf einem anderen Computer zu ermöglichen

**Thread:** eigenständige Aktivität innerhalb eines Prozesses, die einen von anderen Prozessteilen unabhängigen Verarbeitungsstrang darstellt

**Timeout:** Zeitspanne, die ein Vorgang vor Abbruch mit einem Fehler in Anspruch nehmen darf bzw. der Fehler selbst

**Workaround:** Hilfsverfahren zur Umgehung eines Problems, um die Symptome von Fehlverhalten eines technischen Systems zu vermeiden ohne das Problem selbst zu lösen

Freie Universität Berlin

# Selbstständigkeitserklärung

| | |
|---|---|
| Name: Bork | (Nur Block- oder Maschinenschrift verwenden.) |
| Vorname: Konstantin | |
| geb.am: [redacted] | |
| Matr.Nr.: [redacted] | |

Ich erkläre gegenüber der Freien Universität Berlin, dass ich die vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Universität als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: _____ Unterschrift: _____

(_____)

# A. Anhang

## A.1. Inhalt der CD

Dieser Arbeit ist eine CD beigelegt, die folgende Dateien enthält:

- Diese Arbeit in elektronischer Fassung als PDF
- Der Quelltext der zuerst entwickelten Applikation
- Der Quelltext des entwickelten Plugins
- Der Lizenztext zu beiden Programmen (Apache License, Version 2.0)