Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Mandantenfähige Cloud Architektur für die Synchronisation von Projektmanagement Softwareanwendungen

Sebastian Hesse

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer/in:	Dr. rer. nat. Matthias Wieland, Dipl.-Inf. Tobias Anstett
Beginn am:	30. Mai 2017
Beendet am:	30. November 2017
CR-Nummer:	D.2.11

Kurzfassung

Projektmanagement-Software wird schon seit einigen Jahren in Unternehmen eingesetzt und spielt heutzutage eine wichtige Rolle, um Projekte erfolgreich umzusetzen. Unternehmen arbeiten häufig auch über Unternehmensgrenzen hinweg gemeinsam mit anderen Unternehmen an einem Projekt und benötigen dafür eine gemeinsame Datenbasis. Allerdings verbieten Datenschutz- und Unternehmensrichtlinien in der Regel den direkten Zugriff auf die eigenen Daten durch andere Unternehmen oder Partner. Da dies jedoch ein häufiger Anwendungsfall ist, haben Software-Hersteller Integrations-Lösungen entwickelt, um Unternehmen in dieser Hinsicht zu unterstützen. Mithilfe einer Integrations-Software werden nur die Daten zwischen den Unternehmen ausgetauscht, welche keine Richtlinien oder Gesetze verletzen. Dieser Anwendungsfall im On-Premise-Kontext deckt jedoch nur eine *1:n* Beziehung ab, denn Unternehmen synchronisieren ihre Daten nur von ihren Systemen in die ihrer Partner. Im On-Demand-Kontext ist die Situation anders. Eine Integrations-Software, welche eine solche Datensynchronisation anbietet, muss *m:n* Beziehungen abbilden, weil die Software von mehreren Mandanten gleichzeitig genutzt wird. Jedoch kann die Migration einer existieren Integrations-Software von On-Premise zu On-Demand schwierig sein. Ein Grund dafür ist häufig, dass eine andere Architektur im On-Demand-Kontext notwendig ist. Die Architektur wird üblicherweise auf die unterstützten Projektmanagement-Systeme ausgerichtet. Moderne Projektmanagement-Systeme stellen mittlerweile Webhooks bereit, um über Änderungen innerhalb des Systems bzw. der Projektdaten zu informieren. Diese Webhooks können durch eine Integrations-Software empfangen werden und auch eine hohe Last erzeugen, wenn viele Änderungen zur gleichen Zeit erfolgen. In einer typischen Cloud-Architektur müssten für dieses Szenario sehr viele Server vorab provisioniert werden. Deshalb wird in dieser Arbeit eine Cloud-Architektur zur Synchronisation von Projektmanagement-Systemen auf Basis eines neuen Trends im Cloud Computing Kontext vorgestellt: dem „Serverless Computing“. Bei der Nutzung des Serverless Computings wird ein System durch einen Cloud Provider erst provisioniert, wenn es tatsächlich benötigt wird. Dadurch werden vor dem Cloud Consumer alle operativen Tätigkeiten verborgen. Es wird daher eine prototypische Implementierung basierend auf einer Serverless Cloud-Architektur bereitgestellt. Zusätzlich wird diese durch Tests validiert. Die Ergebnisse dieser Arbeit zeigen einerseits, dass Serverless Computing sehr gut in Kombination mit Webhooks in einem Integrations-Kontext verwendet werden kann. Andererseits löst es jedoch nicht alle Probleme und weist in gewissen Bereichen noch Nachteile auf, welche durch Alternativen gelöst werden müssen.

Inhaltsverzeichnis

1 Einleitung **13**
1.1 Motivation . . . 14
1.2 Ziele der Arbeit . . . 15
1.3 Gliederung . . . 16

2 Verwandte Arbeiten **17**
2.1 Serverless Cloud Architektur . . . 17
2.2 Mandantenfähigkeit . . . 18
2.3 Synchronsisation von Softwareanwendungen . . . 19

3 Grundlagen **21**
3.1 Projektmanagement-Software . . . 21
3.2 Cloud Computing . . . 28

4 Konzept einer Serverless Cloud Architektur **37**
4.1 Anforderungsanalyse . . . 37
4.2 Cloud-Architektur . . . 40
4.3 Automatisiertes Deployment . . . 51

5 Umsetzung einer prototypischen Implementierung **55**
5.1 Implementierungsarchitektur . . . 55
5.2 Synchronisationsprozess . . . 60

6 Evaluation **71**
6.1 Bewertung . . . 71
6.2 Validierung der Architektur . . . 72
6.3 Weitere Kritik . . . 82
6.4 Fazit . . . 83

7 Zusammenfassung und Ausblick **85**

8 Anhang **87**

Literaturverzeichnis **95**

Abbildungsverzeichnis

3.1 Atlassian On-Premise Add-on Architektur 25
3.2 Atlassian On-Demand Add-on Architektur 26
3.3 Service Layers eines Cloud Providers 29
3.4 Übersicht Serverless Computing 30
3.5 Lastverteilung für AWS Lambda Funktionen 32

4.1 Anwendungsfalldiagramm für das System 39
4.2 System-Architektur 41
4.3 Aufbau einer Synchronisationsbeziehung 43
4.4 Synchronisationsprozess basierend auf Lambda-Funktionen 44
4.5 Benutzer-Autorisierung 48
4.6 OAuth-Prozess 49
4.7 Prozess für eine Anfrage zur Synchronisation 50
4.8 Entwicklungs- und Deploymentprozess 51

5.1 Initiale Trello Board Synchronisierung 60
5.2 Synchronisationsbeziehung 61
5.3 Sequenzdiagramm zur Visualisierung der nicht erkannten Aktualisierungen . 64
5.4 Sequenzdiagramm zum Item Locking 65
5.5 Sequenzdiagramm zum Zurücksetzen eines Item Locks 66
5.6 Aktualisierte System-Architektur 67

6.1 Vergleich der Antwortzeiten bei zehn gleichzeitigen Kaltstarts 74
6.2 Vergleich der Antwortzeiten bei 50 gleichzeitigen Kaltstarts 75
6.3 Vergleich der Antwortzeiten bei 100 gleichzeitigen Kaltstarts 75
6.4 Vergleich der Antwortzeiten einer NodeJS Lambda-Funktion 77
6.5 Vergleich der Antwortzeiten einer Java Lambda-Funktion 77
6.6 Vergleich der Antwortzeiten einer einfachen NodeJS Lambda-Funktion 78
6.7 Vergleich der Antwortzeiten einer einfachen Java Lambda-Funktion 78
6.8 Vergleich zur Gesamtdauer einer Synchronisation 80

8.1 API Client vor der Umstellung 88
8.2 API Client nach der Umstellung 89
8.3 Ablauf eines Sync Requests 90
8.4 Synchronisationsprozess Outgoing Processor 91
8.5 Synchronisationsprozess Incoming Processor 92

Tabellenverzeichnis

4.1 REST API des Power-Ups 42

Verzeichnis der Listings

3.1 Programm-Code einer Lambda-Funktion 31

5.1 Allgemeine Struktur eines CloudFormation Templates 56
5.2 Beispiel einer Lambda-Funktion im CloudFormation Template 57
5.3 Beispiel einer Konfiguration für eine Integration 59

8.1 Beispielhafte „manifest.json“-Datei für ein Trello Power-Up 87
8.2 Programm-Code einer Test-Lambda-Funktion 93

1 Einleitung

Projektmanagement-Software hat sich seit vielen Jahren in Unternehmen etabliert und spielt eine immer wichtigere Rolle, um Projekte erfolgreich zu gestalten. Sie erleichtern die Übersicht, Planung und Organisation eines Projekts, damit diese effizienter umgesetzt werden können. Die ersten Systeme waren darauf ausgelegt, dass sie entweder auf einem lokalen Computer eines Anwenders oder auf einem eigenen Server (On-Premise) installiert werden mussten. Dadurch entsteht der Vorteil, dass Unternehmen die Daten ihrer Systeme und deren Verwendung selbst kontrollieren können. Allerdings führt dieser Ansatz auch dazu, dass die Kosten zum Betrieb dieser Systeme schnell steigen können.

Durch die immer weiter fortschreitende Digitalisierung haben sich deshalb in den letzten Jahren viele neue Anbieter gebildet, die ihre Projektmanagement-Software im Internet zur Verfügung stellen. So müssen Unternehmen die Software nicht mehr auf einem eigenen Server betreiben. Jedem Anwender wird dazu ein Zugang zum System bereitgestellt, sodass er es in seinem Kontext und mit seinen Daten verwenden kann. Häufig sind die angebotenen Systeme jedoch proprietär. Den Anwendern ist es daher nicht so einfach möglich, sie nach ihren Bedürfnissen zu erweitern. Mit der im Jahr 2002 veröffentlichten Projektmanagement-Software JIRA[1] ist eine Erweiterung durch sogenannte *Add-ons* möglich, welche in das bestehende System installiert werden können. JIRA ist speziell an Unternehmen gerichtet und war anfangs ebenfalls nur für On-Premise-Systeme verfügbar. Seit einigen Jahren wird es auch als On-Demand-System in der Cloud betrieben. Der Hersteller dieser Software ist Atlassian, ein australisches Software-Unternehmen, welches ebenfalls die Projektmanagement-Software Trello[2] betreibt. Trello ist ein On-Demand-System, welches anfangs schwerpunktmäßig auf Privatanwender ausgerichtet war, dessen Fokus mittlerweile jedoch auch auf Unternehmen gerichtet wird. Es kann zwar nicht auf eigenen Servern installiert, jedoch auch durch Erweiterungen, sogenannte *Power-Ups*, erweitert werden. Beide Anwendungen können über einen Browser bedient werden und benötigen deshalb keine eigene Software auf dem Computer eines Anwenders.

[1]https://www.atlassian.com/software/jira
[2]https://trello.com/

1.1 Motivation

Atlassian hat ca. 90.000 Organisationen als Kunden und Trello allein besitzt mehr als 19 Millionen Benutzer [Atl17d] [Tre17b]. Die Zahlen zeigen, dass Projektmanagement-Systeme in vielen Unternehmen weit verbreitet sind. Des Weiteren setzen Unternehmen die Produkte nicht nur intern ein, sondern möchten auch über Unternehmensgrenzen hinweg mit anderen Unternehmen bzw. Partnern zusammenarbeiten. Dafür kann es viele Gründe geben: Zum Beispiel müssen zwei IT-Unternehmen an einem Consulting-Projekt für einen gemeinsamen Kunden zusammenarbeiten und sich dabei auf gemeinsame Aufgaben und Abläufe einigen. Oder ein großes Automobil-Unternehmen möchte einzelne Projekte mit seinen Zulieferern abstimmen, damit entsprechende Teilprojekte pünktlich abgeschlossen werden können. In beiden Fällen müssen externe Personen oder Teams Zugriff auf ein System erhalten, welches sie nicht selbst verwalten. Bestenfalls geschieht diese Zusammenarbeit im selben System und auf derselben Daten- und Nutzerbasis. Dies ist jedoch nicht immer möglich oder gewünscht, weil Unternehmen die Datenhoheit behalten möchten. Zusätzlich können Datenschutzrichtlinien den direkten Zugriff auf Daten verbieten oder sonstige Bedenken bestehen.

Aus diesem Grund wurde das Produkt „Backbone Issue Sync“ von der Firma K15t Software GmbH entwickelt[3]. Es bietet die Möglichkeit, dass verschiedene JIRA-Projekte auch auf unterschiedlichen JIRA On-Premise-Systemen untereinander synchronisiert werden. Am Beispiel eines Automobil-Unternehmens und seinem Zulieferer können so zwei JIRA Systeme an unterschiedlichen Standorten betrieben werden, wobei jeder Partner lediglich sein eigenes System nutzt. Auf beiden Systemen wird Backbone Issue Sync als Erweiterung installiert, sodass daraufhin Projektadministratoren eine Synchronisation für ihre Projekte einrichten können. Die Kommunikation der Verbindung kann über verschiedene Kanäle erfolgen, zum Beispiel kann eine E-Mail-Verbindung als Medium genutzt werden. Projektadministratoren können ebenfalls einstellen, welche Daten zwischen beiden Projekten ausgetauscht werden sollen. So können Vorgänge und ihre Felder, Kommentare, Anhänge und Workflow-Status synchronisiert, aber auch nach individuellen Kriterien gefiltert werden.

Die gleichen Möglichkeiten zur Synchronisation gibt es prinzipiell auch bei den On-Demand-Systemen für Trello und JIRA. Allerdings ist Backbone Issue Sync bisher aufgrund von technischen Grenzen nur für die Synchronsation von On-Premise zu On-Premise oder On-Premise zu On-Demand nutzbar, jedoch nicht von On-Demand zu On-Demand. Mittlerweile nutzen ca. 70 % der Atlassian-Kunden On-Demand-Produkte [Atl17d]. Deshalb sehen sich auch immer mehr Hersteller von Erweiterungen dazu veranlasst, ihre Produkte für On-Demand-Systeme zur Verfügung zu stellen. Für bereits existierende Produkte wie Backbone Issue Sync ist eine solche Migration nicht ohne weiteres möglich. Häufig unterscheiden sich die Architekturen für Erweiterungen zwischen On-Premise- und On-Demand-Systemen stark. Im Kontext von On-Demand-Systemen ist es notwendig, dass Add-ons bzw. Power-Ups auf einem eigenen System in der Cloud provisioniert werden. Eine Registrierung mit dem JIRA bzw. Trello System erfolgt

[3] https://www.k15t.com/software/backbone-issue-sync

lediglich per HTTP-Anfrage. Die Provisionierung des Systems kann mithilfe eines Cloud Providers wie Amazon Web Services (AWS) umgesetzt werden [Ama17d]. Bei einem Cloud Provider wie AWS können jedoch nicht nur einzelne Server für verschiedenste Anwendungszwecke verwendet werden, sondern es lassen sich mittlerweile auch „Serverless Computing“-Angebote nutzen [Bar14]. Durch das Serverless Computing fällt die Verwaltung und Wartung einzelner Server weg, weil ein Software Entwickler lediglich seinen Programm-Code bereitstellen muss. Die Infrastruktur hingegen wird nur noch durch den Cloud Provider betrieben und verwaltet. Unternehmen ist sehr daran gelegen, dass sie sich auf die Umsetzung ihres Kerngeschäfts fokussieren können. Deshalb kann die Benutzung des Serverless Computing Ansatzes eine passende Alternative zu der klassischen Server-Provisionierung in der Cloud sein.

1.2 Ziele der Arbeit

Aus diesem Grund wird im Verlauf dieser Arbeit eine Cloud-Architektur auf Basis des Serverless Computing Ansatzes entworfen. Die Umstellung zu einer Erweiterung im On-Demand-Kontext wird mithilfe einer prototypischen Implementierung umgesetzt. Sie wird auch Aufschluss darüber geben, ob Serverless Computing für die Synchronisation im On-Demand-Kontext geeignet ist. Weil Trello in Zukunft für Unternehmen immer wichtiger wird, wird die prototypische Implementierung der Synchronisation auf Basis von Trello umgesetzt. Es wird dabei berücksichtigt, dass andere Projektmanagement-Systeme (wie z.B. JIRA) ebenfalls angebunden und synchronisiert werden können. Das Konzept wird außerdem die Mandantenfähigkeit im On-Demand-Kontext berücksichtigen. Mandantenfähigkeit bedeutet, dass mehrere Mandanten (Nutzer) das System gleichzeitig benutzen können. Deshalb müssen die Behandlung von Daten sowie interne Prozesse darauf ausgerichtet sein.

Weiterhin wird ein Konzept für einen Entwicklungsprozess vorgestellt, welcher das Deployment des Programm-Codes in der Cloud ermöglicht. Dieser Prozess wird weitestgehend automatisiert durchgeführt, sodass keine manuellen Eingriffe notwendig sind.

1.2.1 Abgrenzung

Die Arbeit umfasst mehrere Themengebiete und versucht neue Ansätze mit bestehenden zu verknüpfen. Das führt zu einer Vielzahl an Aspekten, welche im weiteren Verlauf beachtet werden müssen. So gibt es zum Beispiel zur Synchronisation zweier Systeme bereits einige Arbeiten, die sich intensiv mit verschiedenen Ansätzen befasst haben. Deshalb soll hier auch keine neue Form einer Synchronisation vorgestellt werden (vgl. Kapitel 2). Vielmehr liegt der Fokus auf der Erstellung einer Cloud-Architektur auf Basis des Serverless Computings und den damit verbundenen technischen Möglichkeiten zur Umsetzung einer Synchronisation. Im Bereich des Serverless Computings gibt es mittlerweile mehrere große Anbieter. Die Implementierung wird aufgrund des Umfangs nur mithilfe eines Anbieters umgesetzt. Allerdings lassen sich die Ergebnisse aufgrund der Ähnlichkeit der Angebote leicht vergleichen.

1.3 Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Verwandte Arbeiten: Dieses Kapitel nimmt Stellung zu verwandten Arbeiten und nimmt ihre Kernaussagen auf.

Kapitel 3 – Grundlagen: Das Grundlagen-Kapitel erläutert den aktuellen Stand von Projektmanagement-Software anhand von JIRA und Trello. Danach werden die Grundlagen des Cloud Computings beschrieben.

Kapitel 4 – Konzept einer Serverless Cloud Architektur: Dieses Kapitel beschreibt Anforderungen an die zu erstellende Cloud-Architektur sowie das Konzept dieser Architektur und den Einsatz des Serverless Computings für die Synchronisation zwischen zwei Trello Projekten. Des Weiteren wird ein Entwicklungsprozess vorgestellt, welche das Deployment des Programm-Codes in der Cloud ermöglicht.

Kapitel 5 – Umsetzung einer prototypischen Implementierung: Aufbauend auf dem Konzept der Cloud-Architektur wird die prototypische Implementierung sowie aufgetretene Probleme und deren Lösungswege beschrieben.

Kapitel 6 – Evaluation: Anhand von Tests soll evaluiert werden, ob die vorgestellte Cloud-Architektur ein geeigneter Ansatz für eine Synchronisation zweier Projektmanagement-Systeme ist.

Kapitel 7 – Zusammenfassung und Ausblick: Das letzte Kapitel fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte für mögliche Weiterentwicklungen vor.

Kapitel 8 – Anhang: Im Anhang befinden sich weitere Abbildungen, auf die in der Arbeit verwiesen wird.

2 Verwandte Arbeiten

Die Arbeit umfasst mehrere Themengebiete, welche sich in die Bereiche Serverless Cloud-Architektur, Mandantenfähigkeit in der Cloud und Synchronisation von Softwareanwendungen gliedern. Soweit bekannt, gibt es keine wissenschaftliche Arbeit, die sich bereits mit dem Thema einer Serverless Cloud-Architektur zur Synchronisation von Projektmanagement-Softwareanwendungen befasst hat. Aus diesem Grund sollen hier kurz verschiedene Arbeiten und ihre Ansätze für die drei Teilbereiche dieser Arbeit vorgestellt werden. Die Angaben geben einen groben Überblick über die wichtigsten Arbeiten des jeweiligen Themenfelds.

2.1 Serverless Cloud Architektur

Zhang, Cheng und Boutaba [ZCB10] geben in ihrer Arbeit einen Überblick über den Stand der Entwicklungen im Cloud Computing. Es ist grundsätzlich dadurch geprägt, dass Ressourcen wie Server oder virtuelle Maschinen bei Bedarf gestartet werden können. Des Weiteren beschreiben sie den Aufbau der verschiedenen Service-Schichten (IaaS, PaaS, SaaS) und Formen der Cloud (Public, Hybrid, Private) (vgl. Abschnitt 3.2). In den potentiellen Forschungsgebieten nennen die Autoren unter anderem das automatische Provisionieren von Ressourcen als eine besonders wichtige Herausforderung, die aufgrund unerwarteter oder auch erwarteter Lastspitzen noch genauer untersucht werden muss. Vor allem das automatische Provisionieren als Reaktion auf unerwartete Lastspitzen bezeichnet sehr gut eine der Eigenschaften des Serverless Computings. Zum Zeitpunkt der Veröffentlichung existierte bei Cloud Providern wie AWS oder Google jedoch noch kein Service auf Basis des Serverless Computings.

Baldini et al. [BCC+17] geben einen umfangreichen Überblick über den Status des Serverless Computings im Jahr 2017. Neben den proprietären Anbietern wie AWS [Ama17d] wird auch OpenLambda [HSH+16] als offene Alternative genannt. Des Weiteren werden einige beispielhafte Anwendungsfälle zur Verwendung von Serverless Computing und offene Forschungsfragen genannt. Im Vergleich zum klassischen Cloud Computing bietet Serverless Computing zwar den großen Vorteil, dass keine Instanzen dauerhaft provisioniert werden müssen. Dadurch entstehen jedoch auch diverse neue Herausforderungen, wie zum Beispiel der Kaltstart (engl. cold start) einer Serverless-Funktion. Dieser wird auch von Hendrickson et al. [HSH+16] beim OpenLambda-Projekt als Herausforderung bezeichnet.

Als ein Beispiel für die Verwendung des Serverless Computings wird von Baldini et al. [BCC+17] eine Art der Synchronisation zwischen einer Datenbank und einem Projektmanagement-System wie JIRA genannt. Diese Synchronisation soll jedoch nur zur besseren Behandlung von Lastspitzen dienen und entspricht damit nicht dem in dieser Arbeit behandelten Integrationsaspekt. Außerdem stellen sie sich die Frage, ob Legacy-Code ebenfalls auf eine Serverless-Plattform migriert werden kann. Mit dieser Frage haben sich auch andere Arbeiten beschäftigt, indem existierende Applikationen von einer klassischen Server-Architektur zu einer Serverless Architektur migriert wurden [AC17] [MSE+16]. Die Arbeiten zeigen, dass dies zwar mit Anpassungen möglich, jedoch nicht immer sinnvoll ist. Beide heben trotzdem hervor, dass das Serverless Computing sehr gut in einer event-basierten Umgebung verwendet werden kann. Das zeigen auch Yan et al. [YCCI16], die einen Chatbot auf Basis von IBM Cloud Functions [IBM17] präsentieren. Sobald ein Nutzer mithilfe seiner Stimme einen Befehl gibt, wird dieser Befehl in Text umgewandelt und an eine Serverless-Funktion weitergeleitet. Die Funktion verarbeitet daraufhin die Eingabe und führt weitere Aktionen aus.

2.2 Mandantenfähigkeit

SaaS-Produkte werden typischerweise von mehreren Benutzern gleichzeitig verwendet, was auch als Mandantenfähigkeit (engl. multi-tenancy) bezeichnet wird [KMK12]. Mit der zunehmenden Verbreitung von SaaS-Produkten gingen Hersteller dieser Angebote häufiger den Weg, ihre Produkte nicht dediziert pro Kunde zu provisionieren. Der entscheidende Vorteil ist hier, dass sich so Kosten reduzieren lassen [MK11]. Generell wird dabei zwischen den Ausprägungen „Shared Infrastructure", „Shared Middleware" und „Shared Application" unterschieden [MK11]. Eine Shared Infrastructure teilt lediglich die Infrastruktur, jedoch nicht eine Middleware oder Applikation. Diese werden pro Kunde separat provisioniert. Eine Shared Middleware hingegen teilt die Infrastruktur und eine Middleware, sodass nur die Applikation pro Kunde provisioniert wird. Eine Shared Application ermöglicht die Teilung der Infrastruktur, Middleware und Applikation durch alle Kunden. Dadurch lassen sich Kosten einsparen und es wird eine bessere Skalierbarkeit erreicht [GSH+07].

Allerdings müssen bei der Umsetzung eines SaaS-Angebots verschiedenste Aspekte im Hinblick auf die Software-Architektur beachtet werden. Unter anderem muss der korrekte Zugriff auf die Daten eines Benutzers gewährleistet werden [GSH+07]. Bezemer et al. [BZP+10] präsentieren dafür einen Ansatz, wie eine Anwendung, welche nur für einen Mandanten ausgelegt ist, zu einer mandantenfähigen Anwendung migriert werden kann. Diese Anpassungen enthalten die Erweiterung von Datenbankanfragen durch einen Filter. Der Filter wird mithilfe einer Tenant-ID umgesetzt, sodass jeder Benutzer nur auf seine eigenen Daten zugreifen kann. In dieser Arbeit wird die Mandantenfähigkeit mit einem ähnlichen Ansatz sichergestellt.

2.3 Synchronsisation von Softwareanwendungen

Der Bedarf zur Synchronisation von Daten zwischen (verschiedenen) Softwareanwendungen ist im Kontext eines Unternehmens ein Teilbereich der „Enterprise Application Integration (EAI)“. Es lässt sich wie folgt definieren: „EAI is the unrestricted sharing of data and business processes among any connected applications and data sources in the enterprise“ [Lin00]. Mithilfe von EAI lassen sich unter anderem Daten von verschiedenen System, wie zum Beispiel Projektmanagement-Systemen oder Personalplanungs-Software, miteinander verknüpfen (integrieren), um eine gemeinsame Ansicht auf die Daten zu erhalten. Eine solche Lösung ist häufig in großen Unternehmen notwendig, die nicht nur zwei Systeme besitzen, sondern eine sehr große Menge an unterschiedlichsten Systemen mit unterschiedlichen Daten und Prozessen. Puschmann und Alt [PA01] zeigen anhand der Robert Bosch Gruppe, dass eine Integration sehr komplex werden kann und eine entsprechende Verwaltung der Integrationen notwendig ist. Weil viele Integrationen häufig gemeinsame Probleme lösen, gibt es auch eine sehr große Menge an Mustern, wie bestimmte Probleme innerhalb einer Integration gelöst werden können [HW04]. Beispielsweise können Nachrichten, welche die Daten enthalten, über verschiedene Kanäle transportiert, durch bestimmte Komponenten gefiltert oder in ein anderes Format übersetzt werden.

Aus diesen Gemeinsamkeiten sind einige Integrations-Lösungen entstanden, wie zum Beispiel Apache Camel[1]. Damit können individuelle Integrations-Prozesse definiert werden, um zum Beispiel Daten per REST API von einem System auszulesen und daraufhin die Daten über einen weiteren Kanal (z.B. per CSV-Export) in ein anderes System zu übertragen. Mithilfe eines ähnlichen Prozesses wird Apache Camel ebenfalls in Backbone Issue Sync eingesetzt, um die Synchronisation zwischen zwei JIRA Systemen zu ermöglichen.

[1]http://camel.apache.org/

3 Grundlagen

Das Erstellen von Software erfordert heutzutage eine große Anzahl an weiterer Software, welche die Arbeit eines Softwareentwicklers unterstützt. Projektmanagement-Software nimmt hier einen bedeutenden Teil ein, um die Arbeit an einem Projekt zu strukturieren und zu koordinieren. Da verschiedene Firmen jedoch auch unterschiedliche Bedürfnisse und Prozesse für ihr Projektmanagement haben, gibt es bei einigen Systemen die Möglichkeit, dass diese über Schnittstellen individuell erweitert werden können. Daraus ergeben sich auch weitere potentielle Märkte für andere Firmen, bestimmte Erweiterungen als Software-Produkt anzubieten. Nutzer der Projektmanagement-Systeme können so auf eine Eigenentwicklung verzichten und stattdessen auf fertige Erweiterungen zurückgreifen. Genau dies ist das Geschäftsfeld der Firma K15t Software GmbH, die für diese Masterarbeit eine Kooperation eingegangen ist. Im Folgenden werden daher zunächst der aktuelle Stand zur Entwicklung von Erweiterungen für Projektmanagement-Software am Beispiel von Atlassian JIRA und Trello dargestellt. Daraufhin werden die Grundlagen des Cloud Computings und insbesondere Serverless Computing erläutert, weil der zu entwickelnde Prototyp auf dieser relativ neuen Architektur aufbaut.

3.1 Projektmanagement-Software

Es existiert ein sehr großes Angebot an Projektmanagement-Software, wobei sich die Produkte sowohl in ihrem Funktionsumfang als auch in ihren technischen Details sehr stark unterscheiden. Die Angebote können in technischer Hinsicht zwischen klassischen Desktop-Applikationen und webbasierten Systemen differenziert werden. Abhängig vom System können Benutzer die Planungen, Auswertungen und Dokumentation gemeinsam innerhalb des Systems erledigen. Bekannte Produkte sind unter anderem Microsoft Project[1], IBM Rational Team Concert[2], Basecamp[3], Trello[4] und JIRA[5], wobei die beiden letzteren seit 2017 vom gleichen Anbieter Atlassian vermarktet werden [Atl17a].

[1]https://products.office.com/en/project/
[2]https://www.ibm.com/us-en/marketplace/change-and-configuration-management
[3]https://basecamp.com/
[4]https://trello.com/
[5]https://www.atlassian.com/software/jira

Die Firma Atlassian mit Sitz in Sydney (Australien) und San Francisco (USA) ist ein 2002 gegründetes Software-Unternehmen, welches Software für andere Softwareentwickler und -unternehmen anbietet. Mittlerweile werden viele Produkte auch von Nicht-Softwareentwicklern verwendet. Zu den angebotenen Produkten gehören beispielsweise Confluence[6] als browserbasierte Dokumentationssoftware sowie JIRA und Trello zur Unterstützung des Projektmanagements. Die Zielgruppe der Produkte ist überwiegend auf Unternehmen ausgerichtet, die ihre Projekte und Prozesse mit digitalen Werkzeugen nutzen möchten. Eine Ausnahme dazu bildet Trello, welches sich ebenfalls an Privatpersonen richtet, die verschiedene Arten von Projekten verwalten möchten. Es ist jedoch davon auszugehen, dass der Fokus aufgrund der Übernahme von Trello durch Atlassian stärker auf Unternehmenskunden gelenkt wird [Atl17a]. In den folgenden Abschnitten wird der Fokus insbesondere auf JIRA und Trello liegen, da diese beiden Systeme als Grundlage der Arbeit dienen.

3.1.1 Unterschied JIRA und Trello

JIRA und Trello sind zwar beides Projektmanagement-Systeme, jedoch unterscheiden sie sich stark in ihrer Zielgruppe: JIRA ist primär auf agile Software-Teams ausgerichtet, welche hauptsächlich die Planung zur Erstellung ihrer Software verwalten möchten. Trello hingegen ist auf keine bestimmte Art von Team oder Benutzer ausgerichtet. Deshalb werden nun die Unterschiede der beiden Systeme kurz erläutert.

JIRA

In JIRA existieren grundsätzlich zwei verschiedene Datenmodelle: Projekte (engl. projects) und Vorgänge (engl. issues). Innerhalb eines JIRA Systems können mehrere Projekte angelegt und verwaltet werden, wobei ein Projekt zum Beispiel die Entwicklung eines Software-Produkts umfassen kann. Projekte enthalten mehrere Vorgänge und ein Vorgang gehört immer zu einem Projekt. Vorgänge haben eine Zusammenfassung, eine Beschreibung und sind von einem bestimmten Typ, welcher beispielsweise eine Aufgabe (engl. task), ein Problem (engl. bug) oder eine Benutzer-Story (engl. user story) sein kann. Weiterhin gibt es die Möglichkeit, bestimmte Eigenschaften eines Vorgangs zu verändern oder Kommentare und Dateianhänge hinzuzufügen. Zur Planung der Vorgänge und Projekte unterstützt JIRA die agilen Methoden Scrum und Kanban. Beide definieren und visualisieren auf ihre eigene Weise, welche Vorgänge als nächstes bearbeitet werden sollen. Wurden nach einer bestimmten Zeit eine gewisse Menge von Vorgängen abgeschlossen, kann eine neue Version der Software mithilfe von JIRA veröffentlicht werden. JIRA bietet neben diesen Grundfunktionen sehr viele weitere Funktionen, die es vor allem für Software-Teams sehr interessant macht. Zum Beispiel können mithilfe einer sehr mächtigen Workflow-Funktion eigene Prozesse definiert werden,

[6] https://www.atlassian.com/software/confluence

die den Verlauf vom Anlegen einer Aufgabe über die Implementierung bis hin zur Freigabe durch bestimmte Personen genauestens festlegen. In Kombination mit einem ausgereiften Benutzerrechte-System können so sehr komplexe Prozesse abgebildet werden, die z.B. für Software-Produkte in hochregulierten Branchen oder Firmen notwendig sind. Atlassian hat durch die Veröffentlichung von JIRA Core[7] im Jahre 2015 die Ausrichtung etwas verändert und den Fokus auch auf Nicht-Software-Teams verlagert. Damit können Teams ohne Bezug zu agilen Prozessen Aufgaben anlegen, eigene Workflows definieren und Planungen erarbeiten. Dies ähnelt dem Konzept von Trello schon mehr, ist jedoch immer noch sehr stark auf die Planung eines Projekts ausgerichtet.

Trello

Trello erlaubt es Teams sehr frei und individuell die Software zu nutzen. Hier gibt es lediglich drei Datenmodelle: Boards, Listen und Karten. Ein Benutzer (oder Team) kann mehrere Boards besitzen, was vergleichbar mit Projekten in JIRA ist. Jedes Board kann mehrere Listen enthalten, welche wiederum mehrere Karten enthalten können. Karten sind vergleichbar mit Vorgängen in JIRA und bilden den eigentlichen Kern eines Boards. Sie können einen Titel, eine Beschreibung, Kommentare oder auch Dateianhänge enthalten. Listen dienen lediglich dazu, Karten mit ähnlichen Eigenschaften zu gruppieren. Eine solche Gruppierung kann sehr individuell sein. Ein Anwendungsbeispiel ist folgendes: Ein Team möchte zukünftige Blog-Beiträge verwalten und erstellt dafür vier Listen: „Ideen-Liste“, „Nächste Beiträge“, „In Bearbeitung“ und „Veröffentlicht“. Jede Liste enthält wiederum Karten, die einen einzelnen Blog-Beitrag symbolisieren. Eine Karte kann in diesem Fall den Titel des Beitrags, eine Kurzbeschreibung und ein Titelbild enthalten. Anstatt die Listen als eine Art Prozesskette zu verwenden, können die Namen der Listen einfach umbenannt werden, z.B. in „Technik“, „Service“ und „Firmenkultur“, sodass nur die Karten anders sortiert und gruppiert werden. Dies zeigt vereinfacht, wie flexibel das Trello-System ist und dass es nicht nur auf bestimmte Prozesse oder Anwendergruppen festgelegt ist.

3.1.2 Erweiterungen der Software

Die Unterschiede zwischen JIRA und Trello zeigen, dass Anwender verschiedene Bedürfnisse an ihre Projektmanagement-Systeme haben. Aus diesem Grund bieten beide Plattformen die Möglichkeit zur Erweiterung ihrer Software an. Das heißt, andere Softwareentwickler können die bestehenden Systeme individuell verändern und dadurch häufig neue Funktionen anbieten. Diese Erweiterungen werden im Atlassian Umfeld auch „Add-ons“[8] (im JIRA-Kontext) und

[7] https://www.atlassian.com/software/jira/core

[8] Im Laufe dieser Arbeit hat Atlassian in ihrer Kommunikation den Begriff „Add-on“ durch „App“ ersetzt. Weil noch nicht alle Dokumentationen und Ressourcen auf den neuen Begriff umgestellt wurden, wird in dieser Arbeit der Begriff „Add-on“ weiterhin einheitlich verwendet. Des Weiteren ist „Add-on“ nur das englische Äquivalent für „Erweiterung“ und wird in dieser Arbeit deshalb synonym verwendet.

„Power-Ups“ (im Trello-Kontext) genannt. Je nach Plattform fallen die Rahmenbedingungen sehr unterschiedlich aus, weshalb nun näher darauf eingegangen wird.

Erweiterung von On-Premise-Systemen

Atlassian bietet bei vielen seiner Produkte zwei verschiedene Installationstypen an: On-Premise und On-Demand. On-Premise-Systeme sind Systeme, die im eigenen Netzwerk einer Organisation betrieben werden. Im Kontext von JIRA bedeutet dies, dass ein Kunde die JIRA Software herunterladen kann und einen eigenen Server benötigt, auf dem die Software installiert werden kann. Wenn ein Kunde nun Erweiterungen installieren möchte, kann er diese entweder per Browser oder Dateisystem auf die JIRA Instanz hochladen[9]. Im Folgenden werden verschiedene (programmatische) Eigenschaften einer Erweiterung erläutert:

- Intern nutzt JIRA einen OSGi-Container, um dynamisch Add-ons zu verwalten.
- Das Artefakt einer Erweiterung ist eine **.obr* bzw. **.jar* Datei. Im Kern ist dies eine Java-Anwendung.
- Mithilfe von Maven[10] und dem Atlassian SDK für JIRA können Abhängigkeiten zu den Maven-Artefakten von JIRA angegeben werden. So können interne JIRA Klassen und Interfaces verwendet werden.
- Eine Erweiterung enthält eine Konfigurationsdatei namens *atlassian-plugin.xml*, die sich im Klassenpfad der Add-on Anwendung befinden muss. Sie definiert Schnittstellen der Erweiterung und kann Komponenten von JIRA benutzen. Beispielsweise kann das interne Workflow-System erweitert werden, sodass zusätzliche Funktionen nach einer Workflow-Transition ausgeführt werden. Dies geht sogar soweit, dass direkt auf die Datenbank des JIRA Systems zugegriffen werden kann.
- Die Erweiterung wird durch das Hochladen auf eine JIRA Instanz in dem OSGi-Container registriert und automatisch gestartet. Der Startvorgang beinhaltet die Analyse des hochgeladenen Artefakts, indem die Konfigurationsdatei eingelesen wird und die dort enthaltenen Definitionen evaluiert werden. Daraufhin werden entsprechende Komponenten registriert und im JIRA System zur Verfügung gestellt.

Falls die Erweiterung erfolgreich in einer Instanz installiert und gestartet werden konnte, ist sie aktiv und wird entsprechend ihrer Konfiguration eingesetzt. Abbildung 3.1 beschreibt ein Beispielszenario, indem eine Erweiterung auf einer Instanz installiert wurde. Greift ein Benutzer nun auf ein JIRA Issue zu, so antwortet das JIRA System mit der entsprechenden Website. Das

[9]Dieser Prozess ist vergleichbar mit dem Installieren einer App auf einem Smartphone. Eine App erweitert das Smartphone-System ebenfalls und es können nahezu beliebige Erweiterungen dynamisch hinzugefügt werden.

[10]http://maven.apache.org/

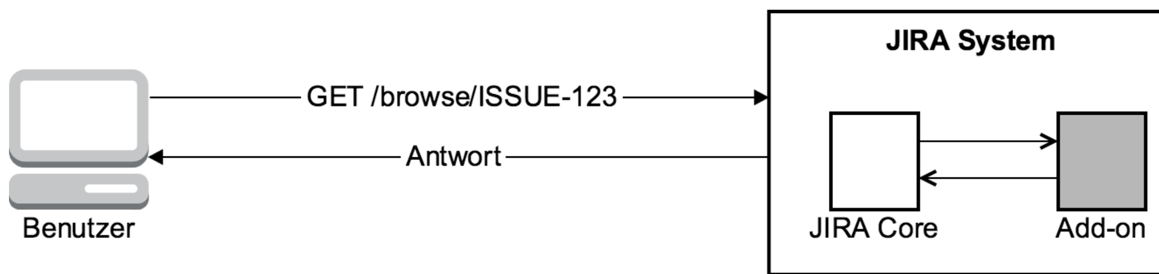


Abbildung 3.1: Die Architektur eines JIRA On-Premise-Systems mit einem Add-on.

Add-on hat sich hier mithilfe der *atlassian-plugin.xml*-Datei im Workflow-System registriert und wird durch JIRA bei einer Änderung des Workflow-Status des Issues benachrichtigt, sodass es weitere Aktionen durchführen kann.

Dieser Prozess funktioniert so oder in einer ähnlichen Art und Weise bei anderen Atlassian Produkten ebenfalls. Als Beispiel sei hier Confluence genannt, welches auf der gleichen Basis die Erweiterung seines Systems ermöglicht. Es bleibt zu erwähnen, dass dies sowohl auf JIRA On-Demand-Systemen als auch auf Trello nicht zutrifft, da diese eine etwas andere Architektur für Erweiterungen verwenden, was im nächsten Abschnitt beschrieben wird.

Erweiterung von On-Demand-Systemen

Im Gegensatz zu On-Premise-Systemen werden On-Demand-Systeme nicht direkt von einem Kunden betrieben, sondern von einem Cloud Provider (vgl. Abschnitt 3.2). Für JIRA On-Demand-Systeme ist Atlassian dafür verantwortlich, dass die entsprechenden Systeme auf Servern installiert sowie im Internet erreichbar sind und der Kunde die Software über den Browser verwenden kann. Das hat den Vorteil, dass für Kunden die Kosten und Arbeit zur Instandhaltung der Instanzen wegfallen. Neben diesem Unterschied gibt es jedoch noch etwas weitaus Wichtigeres für die Add-on Hersteller: die Möglichkeit zum Installieren von individuellen Erweiterungen auf einer Instanz (= Hochladen einer **.jar* Datei) ist nicht mehr gegeben. Add-on Hersteller müssen ihre Erweiterungen nun auf einem eigenen Webserver im Internet bereitstellen und eine Erweiterung mithilfe einer sogenannten „Descriptor“-Datei namens „atlassian-connect.json“ beschreiben. Diese Datei ist von ihrem Verwendungszweck her vergleichbar mit der Konfigurationsdatei *atlassian-plugin.xml* aus dem vorherigen Abschnitt, bietet jedoch aufgrund des anderen Aufbaus der Kommunikation mit dem JIRA System auch andere Einstellungsmöglichkeiten.

In der Descriptor-Datei werden zum einen Meta-Daten wie Add-on Name oder Beschreibung angegeben. Zum anderen werden dort Schnittstellen eines Add-ons beschrieben und welche Zugriffsrechte es zur korrekten Ausführung benötigt. Es kann auch angegeben werden, in welchem Bereich die Oberfläche des JIRA Systems erweitert werden soll. Ein Add-on kann so

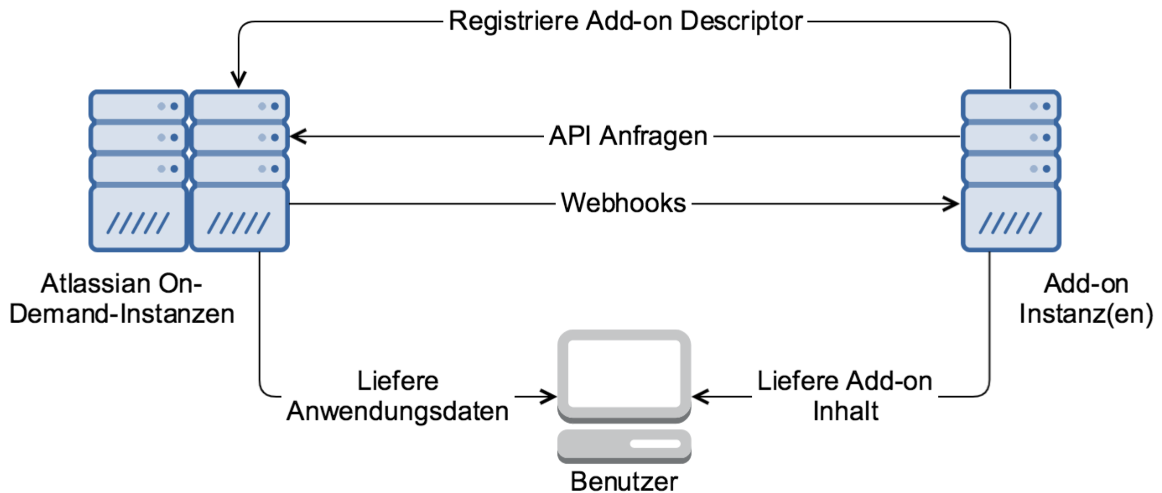


Abbildung 3.2: Die Architektur eines JIRA Systems im Zusammenspiel mit einem On-Demand Add-on [Atl17b].

eigene Menüpunkte oder Dialoge (auch: Popups) definieren. Diese Oberflächen-Elemente werden vom jeweiligen JIRA System in einem HTML IFrame geladen und so direkt in die Webseite integriert. Über eine Javascript-Schnittstelle können Erweiterungen auch weitere Aktionen ausführen. Zum Beispiel können Anfragen an die REST API des Systems gesendet, weitere Dialoge geöffnet oder zusätzliche Informationen dargestellt werden. Durch diesen Aufbau ist allerdings gewährleistet, dass Add-ons die Weboberfläche nicht nach eigenem Belieben verändern können (was bei On-Premise-Systemen durchaus möglich ist). Auch System-Updates sind leichter durchzuführen, da Auswirkungen nur auf Veränderungen der eigens bereitgestellten Schnittstellen überprüft werden müssen. Des Weiteren können Add-on Hersteller nicht mehr auf interne Java-Klassen oder ähnliches zugreifen, sondern können ausschließlich die zur Verfügung gestellte REST API verwenden, um mit dem JIRA System zu kommunizieren. Für den Fall, dass ein Kunde ein Add-on auf seiner On-Demand-Instanz verwenden möchte, muss er lediglich die URL zu der Descriptor-Datei angeben und die JIRA-Instanz stellt daraufhin eine Verbindung zum Add-on her[11]. Abbildung 3.2 verdeutlicht dieses Zusammenspiel zwischen einer On-Demand-Instanz, einem Add-on und einem Benutzer.

Aufgrund der noch recht jungen Akquisition durch Atlassian ist Trello noch nicht vollständig in das beschriebene On-Demand-Ökosystem eingebunden. Es bietet jedoch mit den im Januar 2016 vorgestellten Power-Ups eine sehr ähnliche Möglichkeit zur Integration von Erweiterungen

[11]Diese Angabe ist nicht ganz vollständig: Zusätzlich zu dieser Möglichkeit kann ein Kunde ein Add-on auch über den Atlassian Marketplace installieren. Der Atlassian Marketplace ist ein Marktplatz für Add-ons, ähnlich eines App Stores für Smartphone Apps. Wenn ein Kunde über den Marktplatz ein Add-on kauft, werden im Hintergrund die Descriptor-URL zwischen Marktplatz und jeweiliger JIRA On-Demand-Instanz ausgetauscht. Der Installationsprozess beginnt daraufhin automatisch.

in das Trello-System [Tre16]. Lediglich Details unterscheiden sich, beispielsweise die Art der Benutzung der REST API oder die möglichen Angaben in der Descriptor-Datei, welche bei Trello „manifest.json“ genannt wird[12]. JIRA verwendet für die Authentifizierung der REST API Json Web Tokens (JWT), wobei Trello auf OAuth 1.0a zurückgreift [Atl17c] [Tre17a]. Die beiden Verfahren unterscheiden sich zwischen den Systemen wie folgt:

JIRA: Eine Erweiterung für On-Demand-Systeme erhält vom jeweiligen JIRA System einen eigenen Zugang zu dessen REST API. Dieser beinhaltet einen „Client Key“ sowie ein „Shared Secret“, also eine Art Passwort. Mit diesem Zugang kann ein „Token“ generiert werden, mit dem alle Anfragen an die REST API versehen werden[13].

Trello: Eine Erweiterung für Trello handelt im Namen eines Trello-Benutzers und nicht als einzelne Erweiterung. Das bedeutet, dass ein Trello-Benutzer einer Erweiterung die Erlaubnis erteilt, die REST API in seinem Namen anzufragen. Als Authentifizierung wird dabei ein Schlüssel und ein Passwort verwendet, welche dem Trello-Benutzer zugeordnet werden können. Alle Anfragen an die REST API müssen mit den OAuth-Informationen versehen werden.

Ein weiteres Detail ist - wie bereits erwähnt - die unterschiedliche Ausprägung der Descriptor-Datei. Trellos Descriptor-Datei ist in der Regel weitaus kürzer, da noch nicht so viele Einstellungsmöglichkeiten vorhanden sind. Diese Unterschiede werden hier jedoch nicht weiter vertieft. Ein weitaus wichtigerer Punkt, den JIRA und Trello in ihren Descriptor-Dateien gleichermaßen unterstützen, ist die Verwendung von Webhooks.

Webhooks

Für Webhooks existiert in der Literatur keine klare Definition. Lindsay [Lin07] bezeichnet Webhooks als Callbacks auf Basis von HTTP. Sie funktionieren folgendermaßen: Eine Software registriert sich bei einem System (z.B. Trello) für bestimmte Events mit einer Callback-URL, welche die Erweiterung kontrolliert und unter der sie HTTP-Anfragen empfangen kann. Im Falle des Eintritts eines Events (z.B. eine Trello Karte wird aktualisiert), sendet Trello eine HTTP POST Anfrage an die bereitgestellte Callback-URL der Erweiterung. Die Anfrage enthält im Body-Teil weitere Informationen, nämlich Daten zu dem aufgetretenen Event. Diese Daten können dann in irgendeiner Art und Weise weiterverarbeitet werden. Da sich für solche Events häufig mehr als nur ein Teilnehmer anmelden kann, erinnert die Funktionsweise sehr an ein Publish-Subscribe-System wie MQTT[14]. Im Vergleich dazu sind Webhooks auf HTTP und im Internet erreichbare Teilnehmer beschränkt. Der Vorteil von Webhooks besteht außerdem darin, dass regelmäßiges Abfragen (auch „Polling“ genannt) von etwaigen REST APIs vermieden

[12] Ein Beispiel einer solchen Datei befindet sich im Anhang in Listing 8.1.
[13] Allgemein kann für JIRA auch OAuth 1.0a und OAuth 2 verwendet werden, was jedoch für diesen Anwendungsfall nicht notwendig ist.
[14] http://mqtt.org/

wird. Es wird nur eine Nachricht verschickt, sobald sich etwas geändert hat und dies durch ein Event ausgedrückt werden kann. Ein solcher Ansatz schont Ressourcen sowohl im Netzwerk als auch auf einem Server und dient deshalb auch zur Reduzierung von Kosten.

Auch JIRA und Trello bieten die Möglichkeit, dass sich ein Add-on bzw. Power-Up für Webhooks registriert. So können zum Beispiel Webhooks bei JIRA für Events wie „issue_created", „comment_updated" oder „project_updated" registriert werden. Dies kann durch einen Filter noch weiter verfeinert werden, sodass nicht für alle Vorgänge, Kommentare oder Projekte ein Webhook empfangen wird, sondern nur für Ausgewählte. Trello bietet hingegen die Möglichkeit, sich für Events einzelner Objekte zu registrieren. Beispielsweise können Events für ein gesamtes Board oder auch nur für einzelne Listen oder Karten empfangen werden, wie etwa „createCard", „updateList" oder „commentCard". Hier ist der Nachteil, dass kein Filter vorab angegeben werden kann, sondern die ankommenden Webhooks selbst gefiltert werden müssen.

Webhooks unterstützen den Einsatz einer event-basierten Architektur, indem nur auf die relevanten Änderungen reagiert wird. Eine solche event-basierte Architektur wiederum begünstigt den Einsatz neuester Technologien im Bereich des Cloud Computings, was im folgenden Kapitel detaillierter beschrieben wird.

3.2 Cloud Computing

Seit einigen Jahren ist ein Trend zu beobachten, dass Software-Anbieter immer öfter ihre Produkte für die Cloud entwickeln. Entweder nutzen sie dabei ihre eigenen Server, die sie an die Cloud anbinden, oder greifen auf bewährte Anbieter zurück, die häufig weltweit große Rechenzentren betreiben und diese Ressourcen an Kunden vermieten. Zurzeit sind die vier größten Cloud-Betreiber Amazon Web Services (AWS), Google, Microsoft und IBM mit einem gemeinsamen Marktanteil von 55 %, wobei AWS alleine 34 % besitzt [Syn17]. Mell, Grance et al. [MG+11] definieren *Cloud Computing* als eine Methode zum geeigneten Zugriff auf gemeinsam genutzte Rechenressourcen, welche mit minimalem Aufwand sehr schnell bereitgestellt werden können. Im Allgemeinen sind dabei zwei Parteien involviert: ein Cloud Provider, der etwaige Ressourcen wie (virtuelle) Server zur Verfügung stellt, und ein Cloud Consumer, der diese Ressourcen nach Bedarf benutzt [MG+11]. Ein typisches Vorgehen ist hierbei, dass ein Cloud Provider die Ressourcen einem Kunden nicht exklusiv zur Verfügung stellt, sondern dieser sich die Ressourcen mit anderen Kunden teilt[15]. Die Aufteilung erfolgt durch Virtualisierung der Ressourcen mithilfe von Hypervisoren und virtuellen Maschinen, was zur besseren Auslastung der Hardware führt [ZCB10].

[15]Es ist auch möglich, dass ein Kunde die Ressourcen exklusiv nutzen kann. Eine gemeinsame Nutzung ist jedoch vorteilhafter, da Kosten eingespart werden können.

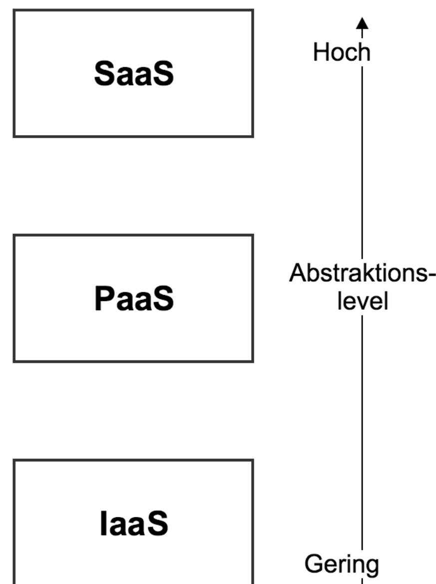


Abbildung 3.3: Die unterschiedlichen Level von bereitgestellten Services der Cloud Provider nach [MG+11] und [ZCB10]. Je höher das Level, desto beschränkter ist der Zugriff auf die darunterliegende Infrastruktur.

Auf dieser Basis können Services angeboten werden, die einem Kunden unterschiedliche Kontrollmöglichkeiten über die verwendete Infrastruktur geben. Eine Einteilung der Service-Stufen ist in Abbildung 3.3 dargestellt. Je höher die Stufe, desto eingeschränkter ist ein Kunde beim Zugriff auf die darunterliegende Infrastruktur. Der Ansatz hat einen großen Vorteil: Ein Cloud Provider kann mehr Ressourcen standardisieren und dadurch günstiger anbieten, was dem Kunden häufig entgegenkommt. Auf dem untersten Level befindet sich *Infrastructure-as-a-Service* (IaaS), welches für das Bereitstellen von Servern, Speicher, Netzwerkkomponenten und ähnlichem steht. Der Cloud Consumer hat auf dieser Stufe die Möglichkeit, selbst zu bestimmen, welche Ressourcen benötigt werden. Beispielsweise kann das Betriebssystem oder die Anzahl der CPU-Kerne ausgewählt werden. Obwohl der Kunde bei dieser Variante sehr viel Freiheit besitzt, kann er die zugrundeliegende Hardware nicht direkt verwalten, sondern muss sich an die Vorgaben des Cloud Providers anpassen. Typische Anbieter der IaaS-Stufe sind AWS [Ama17d], Google [Goo17b] und Microsoft [Mic17a]. Die nächste Stufe *Platform-as-a-Service* (PaaS) abstrahiert einige dieser Einstellungsmöglichkeiten und ermöglicht lediglich das Deployment einer Applikation auf einer vom Cloud Provider unterstützten Plattform [MG+11]. Der Vorteil ist, dass operative Arbeiten zur Verwaltung der zugrundeliegenden Server entfallen, da diese der Cloud Provider übernimmt. Ein Beispiel für einen klassischen PaaS-Anbieter ist Heroku [Her17], der beispielsweise Java-, Scala- oder NodeJS-Applikationen unterstützt. An oberste Stufe der Abstraktion steht *Software-as-a-Service* (SaaS), was alle Applikationen umfasst, die auf einer Cloud Infrastruktur laufen und über einen entsprechenden Client (z.B. ein Browser) oder eine Programmierschnittstelle erreichbar sind. SaaS-Produkte ermöglichen einem Kunden

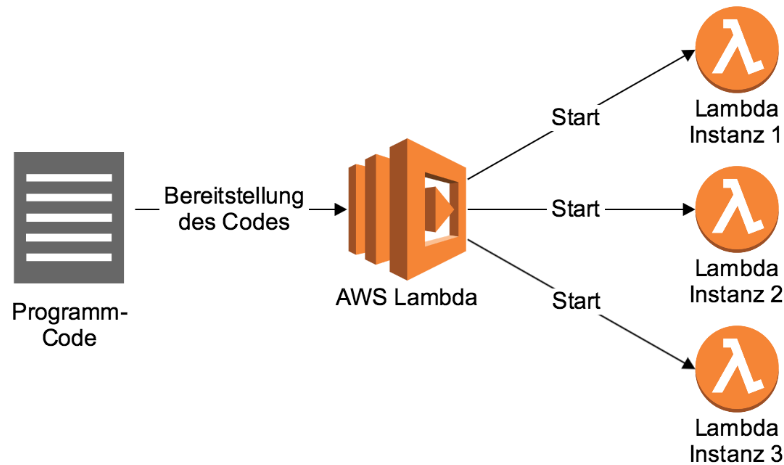


Abbildung 3.4: Übersicht zu Serverless Computing. Der Programm-Code wird an AWS Lambda übergeben, was bei Bedarf einzelne Funktions-Instanzen startet.

die geringsten Konfigurationsmöglichkeiten, da sie die zugrundeliegende Infrastruktur häufig vor dem Kunden verbergen [MG+11]. Beispiele zu SaaS-Anbietern sind Salesforce.com [sal17], Microsoft Office 365 [Mic17c] und auch Trello selbst.

Es ist nicht immer einfach, Cloud Provider einer bestimmten Kategorie zuzuteilen, da ein Service entweder Eigenschaften von zwei Stufen besitzt oder der Cloud-Anbieter mehrere Stufen durch mehrere Services abbildet. Amazon ist solch ein Anbieter, der neben klassischen IaaS- und PaaS-Produkten auch Services anbietet, die nicht konkret einer bestimmten Stufe zugeordnet werden können, z.B. den EC2 Container Service [Ama17c].

3.2.1 Serverless Computing

Ende 2014 hat AWS einen weiteren Service vorgestellt: „AWS Lambda“ [Bar14]. Mit diesem Ansatz muss ein Kunde nur noch den Programm-Code einer einzelnen Funktion schreiben und das Code-Artefakt zu AWS Lambda hochladen. AWS führt daraufhin ein automatisches Deployment auf entsprechende Server-Instanzen und deren Skalierung durch. Das eigentliche Starten der Server-Instanzen und deren Skalierung werden jedoch erst durchgeführt, wenn eine Funktion tatsächlich benötigt wird. Der Auslöser dafür sind Events, die von einer Funktion verarbeitet werden sollen. Die Event-Quellen werden vom Cloud Provider vorgegeben und können in der Regel sowohl HTTP-Anfragen als auch neue Datenbank-Einträge oder Dateien auf einem Dateisystem sein. Amazon hat damit ein neues Abstraktionslevel zwischen SaaS und

Listing 3.1 Beispiel für eine Lambda-Funktion in NodeJS, die per HTTP aufgerufen werden kann und eine Antwort zurückgibt.

```
module.exports.handler = function(event, context, callback) {
  console.log('Handling event: ', event);
  callback(null, {
    statusCode: 200,
    body: 'Success'
  });
};
```

PaaS geschaffen, nämlich „Serverless Computing“[16] [BCC+17]. Es basiert auf dem einfachen Grundgedanken, dass für Kunden keine Notwendigkeit mehr bestehen soll, eigene Server zur Ausführung ihrer Applikationen zu provisionieren, zu starten oder zu stoppen. Daher kommt auch der Name „Serverless“ (deutsch: *serverlos* oder frei übersetzt „Ohne Server“), welcher jedoch nicht ganz korrekt ist: Im Hintergrund werden trotzdem noch Server benötigt auf denen die einzelnen Funktionen ausgeführt werden, allerdings wird dieser Prozess vor dem Kunden versteckt. Mittlerweile haben andere Anbieter wie Google [Goo17a], IBM [IBM17] und Microsoft [Mic17b] ihre eigenen Serverless-Dienste veröffentlicht, aber auch OpenSource-Alternativen wie OpenLambda [HSH+16] haben sich ergeben.

AWS Lambda

Im Folgenden wird der Dienst AWS Lambda genauer vorgestellt, da er für diese Arbeit die wichtigste Grundlage darstellt. Viele Eigenschaften sind jedoch in ähnlicher Form auch bei den Serverless Angeboten anderer Cloud Provider vorhanden.

Bei AWS Lambda hat ein Kunde die Möglichkeit, einzelne Funktionen in einer Programmiersprache wie Python, NodeJS oder Java zu schreiben. Listing 3.1 zeigt ein kleines Beispiel für den grundsätzlichen Aufbau einer Lambda-Funktion in NodeJS[17]. Der *event* Parameter enthält Daten zu dem Event-Objekt, was den Aufruf der Funktion veranlasst hat. Das *context* Objekt enthält u.a. Daten zur Umgebung der Lambda-Funktion und der *callback* Parameter ermöglicht eine Antwort zu liefern, beispielsweise um HTTP-Anfragen zu beantworten. Wie in dem Listing zu sehen, ist das Grundgerüst einer einzelnen Funktion sehr klein. Der Gedanke dahinter ist, dass Funktionen nur für eine bestimmte Aufgabe entwickelt werden. Des Weiteren sollen Funktionen einem statuslosen Programmieransatz folgen, damit die Skalierung automatisch von Amazon durchgeführt werden kann [Ama17i]. Dieser Punkt ist sehr wichtig und unterscheidet sich von den bisherigen Ansätzen, bei denen das Deployment, Starten und

[16] Anzumerken sei hier, dass bisher kein einheitlicher Begriff in der Literatur existiert. Roberts [Rob16] bezeichnet es auch als „Function-as-a-Service“ (FaaS), was den anderen „as-a-Service“-Bezeichnungen wesentlich näher kommt. Der Begriff Function-as-a-Service ist in der Literatur jedoch weitaus weniger verbreitet.

[17] Der Begriff *Lambda-Funktion* wird in dieser Arbeit als Synonym zu einer Funktion verwendet, welche mithilfe des AWS Lambda Services provisioniert wird.

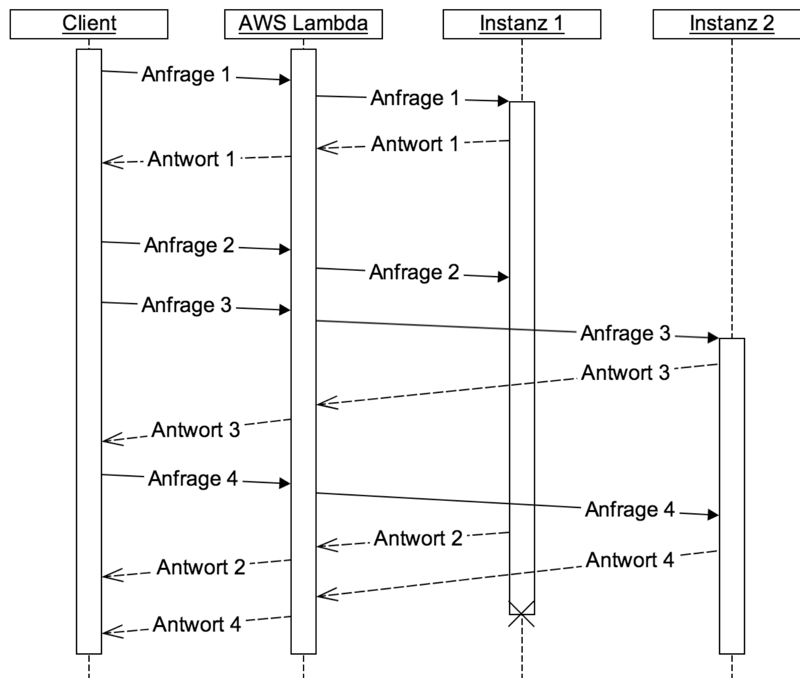


Abbildung 3.5: Die Darstellung zeigt, wann neue Instanzen gestartet werden, wenn beispielsweise HTTP-Anfragen durch einen Benutzer gestellt werden.

Skalieren auch noch durch den Cloud-Kunden durchgeführt werden kann (vgl. auch folgenden Abschnitt zur Abgrenzung zu PaaS).

Abbildung 3.5 verdeutlicht den Prozess zum Starten der Instanzen einer Lambda Funktion, wenn ein Benutzer HTTP-Anfragen an diese sendet. Eine erste Anfrage startet die Instanz 1, welche daraufhin für die Verarbeitung eingeteilt ist. Prinzipiell wird eine Instanz wiederverwendet, wenn sie die Verarbeitung einer vorangegangenen Anfrage abgeschlossen hat und wieder verfügbar ist, was anhand der zweiten Anfrage verdeutlicht wird. Sobald allerdings eine weitere HTTP-Anfrage empfangen wird (in diesem Fall Anfrage 3), muss eine neue Instanz gestartet werden, weil keine freie Instanz verfügbar ist. Anfrage 4 hingegen verwendet Instanz 2 erneut während Instanz 1 noch die zweite Anfrage bearbeitet. Für die Bearbeitung eines Events gibt AWS ein hartes Limit von fünf Minuten vor, welches auch nicht verändert werden kann [Ama17g]. Überschreitet eine Funktion dieses Limit um wenige Millisekunden, wird die Bearbeitung abgebrochen. Nach Ablauf der fünf Minuten kann für eine Instanz jedoch nicht mehr garantiert werden, wie lange sie noch aktiv ist, denn sie kann danach durch AWS einfach gelöscht werden. Dieser Fall ist in Abbildung 3.5 für Instanz 1 beispielhaft durch das *X* dargestellt. Sollte dieser Fall eintreten und daraufhin ein neues Event eintreffen, wird wieder eine neue Instanz gestartet. Das bedeutet, dass es Zeiträume geben kann, in denen keine einzige Instanz einer Funktion existiert.

Wo einerseits ein Nachteil ist, dass eine Instanz erneut gestartet werden muss und dies etwas Zeit benötigt, gibt es andererseits auch einen Vorteil für den Cloud-Nutzer. Aus der geringeren Nutzung folgen weniger Kosten, weil die Abrechnung sich nach dem im Cloud Computing häufig verwendeten „Pay-per-use“-Preismodell richtet [ZCB10]. Im Kontext von AWS Lambda bedeutet es, dass die Anzahl der Aufrufe zusammen mit den Nutzungseinheiten berechnet werden. Eine Nutzungseinheit richtet sich nach der Anzahl an angefangenen 100 ms Intervalle sowie der verwendeten Größe der Lambda-Instanz, welche sich nach der Größe des Arbeitsspeichers richtet [Ama17h]. Nach dem freien Nutzungskontingent kosten eine Millionen Aufrufe für eine Lambda-Funktion mit einer Arbeitsspeichergröße von 128 MB und einer maximalen Laufzeit von 100 ms pro Ausführung lediglich 0,208 $. Dies ist der geringste Preis für eine Millionen Aufrufe und er steigt sobald die gewählte Größe der Lambda-Funktion und ihre Laufzeit erhöht wird. Ein EC2-On-Demand-Server[18] in der kleinstmöglichen Ausführung „t2.nano“ (mit 512 MB Arbeitsspeicher, 1 CPU) kostet im Vergleich dazu 4,176 $ pro Monat (ein Monat = 30 Tage). Daraus ergibt sich, dass eine Lambda-Funktion in der kleinstmöglichen Ausführung mehr als 20 Millionen mal im Monat (etwa sieben mal pro Sekunde) aufgerufen werden muss, damit sie genau so viel kostet wie ein EC2-Server, welcher 24 Stunden am Tag betrieben wird[19].

Abgrenzung zu Platform-as-a-Service

Auch, wenn das Konzept des Serverless Computings sehr ähnlich zu dem des Platform-as-a-Service ist, so unterscheiden sich die beiden in einem wichtigen Punkt: Bei PaaS wird ein Artefakt provisioniert, welches gewissermaßen immer verfügbar ist, um dann über einen längeren Zeitraum verschiedene Aufgaben auszuführen. Das bedeutet auch, dass ein Kunde die Instanz immer bezahlen muss - unabhängig davon, ob die Applikation tatsächlich benutzt wird oder nicht. Im Vergleich dazu ist bei Serverless Computing gewährleistet, dass nur die tatsächlich verbrauchte Zeit abgerechnet wird [BCC+17]. Ein weiterer wesentlicher Unterschied ist auch, dass beim Serverless Computing die Funktionen zwingend statuslos implementiert werden sollten. Das ist aus zwei Gründen wichtig: Erstens ist nicht gewährleistet, dass die Instanz einer Funktion bei einem weiteren Aufruf noch existiert. Zweitens, falls sie doch noch existiert, bedeutet es nicht, dass sie noch einmal verwendet wird. Dies ist insbesondere zu beachten, wenn eine Anwendung auf einen Status angewiesen ist, wie beispielsweise eine Session im Webbereich.

Zusammenfassend lässt sich also durchaus sagen, dass Serverless Computing eine Art Weiterentwicklung des PaaS-Konzepts ist. Ein Kunde hat zwar weniger Möglichkeiten bei der Auswahl

[18] *Elastic Computing Cloud (EC2)* ist der AWS Dienst, welcher die Provisionierung von Servern in der Cloud ermöglicht: https://aws.amazon.com/ec2/

[19] Die Berechnungen stellen nur ungefähre Werte dar, weil sie von verschiedenen Faktoren abhängen. Ein dieser Faktoren ist der verursachte Datentransfer. Sowohl EC2- als auch Lambda-Instanzen benutzen jedoch das gleiche Preismodell für den Datentransfer [Ama17h].

der zugrundeliegenden Infrastruktur, jedoch fallen dadurch auch weniger Verwaltungs- und Wartungsarbeiten an.

3.2.2 Weitere Services

Zusätzlich zu AWS Lambda müssen noch weitere AWS Services betrachtet werden, die im Verlauf dieser Arbeit für das zu entwickelnde System von besonderer Bedeutung sind. Alle folgenden Services sind wie AWS Lambda auch proprietär. Das heißt, niemand außer Amazon kann die Weiterentwicklung und Verbesserung der Services anstreben.

S3

Simple Storage Service (S3) ist ein hochverfügbarer Service, mit dem beliebige Dateien in sogenannten *Buckets* abgelegt werden können, wobei ein Bucket mit einer sehr großen Festplatte in der Cloud verglichen werden kann. Dateien werden innerhalb eines Buckets durch einen eindeutigen Schlüssel referenziert, welcher beliebig gewählt werden kann. Dieser Service kann durch eine Einstellung auch als statischer Webserver verwendet werden, der Dateien öffentlich über das Internet anbietet. Dadurch können zum Beispiel HTML-, JavaScript- oder CSS-Dateien wie über eine normale Website bereitgestellt werden.

DynamoDB

DynamoDB ist ein hochverfügbarer NoSQL-Datenbank-Service, welcher im Hintergrund die Verwaltung einzelner Datenbank-Instanzen sowie die entsprechende Replikation der Daten übernimmt [Ama17b]. Eine Applikation, die diesen Service nutzt, kann dadurch eine oder mehrere Datenbanktabellen erstellen und muss pro Tabelle lediglich ein Schlüssel-Schemata angeben. Ein Schlüssel-Schemata besteht aus der Angabe eines oder zweier Attribute, welche ein Tabelleneintrag besitzen muss, und deren Typ. Zum Beispiel könnten die Attribute einer Tabelle von Onlineshop-Artikeln eine Artikelbezeichnung vom Typ String und eine Artikelfarbe vom Typ String sein. Das Schlüssel-Schemata würde dann die Attribute Artikelbezeichnung als *HASH* und Artikelfarbe als *RANGE* referenzieren[20]. Anhand dieser Angaben können daraufhin Einträge in der Tabelle identifiziert und abgefragt werden, wobei jedoch das Prinzip der *Eventual Consistency* berücksichtigt werden muss, welches bei NoSQL-Datenbanken üblich ist [DHJ+07]. Des Weiteren können noch Indizes erstellt werden, die alternative Zugriffsmöglichkeiten durch andere Schlüssel-Schemata ermöglichen.

[20] *HASH* definiert den *Partition Key*, was im Prinzip ein Primärschlüssel ist, und *RANGE* definiert den *Sort Key*, welcher als Ergänzung dient, falls das Schlüssel-Schemata wie in diesem Beispiel aus zwei Attributen besteht und der Partition Key nicht eindeutig ist.

API Gateway

Mit dem Service API Gateway können beliebige HTTP-Schnittstellen definiert werden, z.B. `/api/users/{id}`. Wenn diese HTTP-Schnittstelle mit einer Lambda-Funktion kombiniert und nach dem Deployment der Funktion durch einen Client aufgerufen wird, leitet das API Gateway die Anfrage an AWS Lambda weiter, sodass eine Lambda-Funktion aufgerufen wird. Das API Gateway bietet daneben noch weitere Punkte an, zum Beispiel der Schutz vor Ausfall der Systeme oder der Autorisierung zur Nutzung der API [Ama17e].

CloudFront

CloudFront ist ein „Content Delivery Network (CDN)“ Service, der auf das Caching und schnelle Ausliefern von Inhalten spezialisiert ist [Ama17a]. An weltweit wichtigen Knotenpunkten des Internets sind CloudFront-Server aufgestellt, die einen kürzeren Weg zwischen dem Sender einer Anfrage und dessen Ziel abdecken, sodass Daten schneller ausgeliefert werden können. Dies kann dazu genutzt werden, um vor allem statische Daten wie HTML- oder JavaScript-Dateien auszuliefern. In Kombination mit „Route53“, Amazons eigenem DNS-Service, sind Dateien auch unter einer gemeinsamen Domain erreichbar.

CloudFormation

Damit die zu verwendende Infrastruktur nicht nur manuell über die AWS Console (siehe nächster Abschnitt) erstellt, sondern auch in einer maschinenlesbaren Form definiert werden kann, existiert der Service CloudFormation [Ama17f]. Dieser Service verfolgt den Ansatz des Infrastructure-as-Code (IaC), welcher es ermöglicht, den Aufbau einer Infrastruktur anhand einer Beschreibungssprache zu definieren [Mor16].

AWS Console

Die AWS Console bietet Zugriff über eine Weboberfläche auf das AWS Konto eines AWS Kunden [Ama17j]. In diesem System können alle eingesetzten Services eingesehen und gegebenenfalls verändert werden. Bei Verwendung eines Infrastructure-as-Code Ansatzes sollten jedoch manuelle Änderungen an Konfigurationen eines Services nicht notwendig sein. Zusätzlich zur Möglichkeit des Zugriffs per Weboberfläche gibt es auch noch für verschiedene Betriebssysteme ein Kommandozeilen-Programm zur Steuerung der Services, nämlich das AWS CLI. Dieses kann auch zur Automatisierung von Abläufen in Skripten verwendet werden.

3.2.3 Zusammenfassung

AWS bietet als derzeit größter Anbieter eine Vielzahl an Cloud-Services an, von denen hier eine kleine Auswahl und insbesondere der Service AWS Lambda vorgestellt wurden. Für den weiteren Verlauf der Arbeit wird eine Architektur und ein zugehöriger Prototyp auf Basis dieser Services erstellt.

4 Konzept einer Serverless Cloud Architektur

In diesem Kapitel wird das Konzept zur Erstellung der Trello-Erweiterung auf Basis einer Cloud-Architektur für On-Demand-Systeme beschrieben. Zuerst werden dafür die Anforderungen ausgearbeitet, die der Prototyp erfüllen soll[1]. Dazu zählen sowohl funktionale als auch nicht-funktionale Anforderungen. Im Anschluss daran wird eine Cloud-Architektur auf Basis der Anforderungen vorgestellt, die als Cloud Provider Amazon Web Services verwendet. Zusätzlich wird ein Entwicklungsprozess vorgestellt, der beschreibt, wie das Deployment des Programm-Codes in der Cloud erfolgt.

4.1 Anforderungsanalyse

4.1.1 Funktionale Anforderungen

FA1 - Synchronisation zweier Trello Boards Der Prototyp soll die Möglichkeit bieten, dass zwei Trello Boards synchronisiert werden können. Zu einem Board zählen Daten wie die Listen eines Boards, die Karten einer Liste sowie Kommentare, Anhänge, Labels und die Beschreibung einer Karte. Individuelle Felder, die etwa durch weitere Power-Ups hinzugefügt werden können, sollen nicht beachtet werden.

FA2 - Field Mappings Die Synchronisation soll ein einfaches 1-zu-1 Mapping der Daten umsetzen. Es kann dabei vom Trello Standardformat ausgegangen werden. Das heißt, dass zum Beispiel Labels lediglich eine Farbe, jedoch keine Beschriftung enthalten. Es müssen keine Daten konvertiert oder vorverarbeitet werden. Allerdings soll eine Möglichkeit bestehen, etwaige Methoden im Nachhinein noch hinzufügen zu können.

FA3 - Änderungen automatisiert synchronisieren Der Prototyp soll Änderungen in einem Trello Board automatisch mit dem verknüpften Board ohne manuelle Eingriffe synchronisieren.

[1]Im weiteren Verlauf wird der Prototyp auch als „Trello Sync“ bezeichnet.

FA4 - Authorisierung Der Prototyp muss sicherstellen, dass eine Synchronisation zwischen zwei Trello Boards erst dann durchgeführt wird, wenn beide Eigentümer der jeweiligen Boards dieser Synchronisation zugestimmt haben. Ein Benutzer muss die Möglichkeit haben, die Synchronisation jederzeit zu beenden.

FA5 - Konfiguration der Synchronisation Der Prototyp muss Benutzern eine Möglichkeit zur Verfügung stellen, dass Einstellungen zur Synchronisation konfiguriert werden können. Einstellungsmöglichkeiten sind zum einen die Wahl des Synchronisations-Partners und zum anderen etwaige Filtermöglichkeiten, zum Beispiel ob alle Listen eines Board synchronisiert werden sollen. Die Einstellungsmöglichkeiten sind hierbei nicht fest vorgeschrieben, sollen jedoch erweiterbar sein.

FA6 - Fehlererkennung Falls während der Synchronisation einer Änderung von einem Board zum anderen ein Fehler auftritt, soll dieser Fehler registriert und dem Benutzer angezeigt werden. Es muss keine automatische Fehlerbehandlung für alle Fehler vorhanden sein.

4.1.2 Nicht-funktionale Anforderungen

NFA1 - Zuverlässigkeit Der Prototyp muss gewährleisten, dass alle Änderungen innerhalb eines Boards erfasst und synchronisiert werden. Falls Änderungen nicht erfasst oder synchronisiert werden, soll der Vorgang entweder wiederholt oder als Fehler registriert werden.

NFA2 - Serverless Architektur Der Prototyp soll auf einer Serverless Architektur aufbauen. Als Cloud Provider soll Amazon Web Services verwendet werden. Die einzelnen Funktionen müssen dabei zwingend statuslos implementiert werden.

NFA3 - Infrastructure as Code Die Architektur soll auf Basis eines Templates in der Cloud provisioniert werden. Dies verringert Fehler während eines Deployments, weil zum einen keine manuellen Schritte notwendig sind und zum anderen die tatsächliche Architektur eindeutig ist. Das heißt, dass bei einem erneuten Deployment immer noch die selben Services provisioniert werden, solange sich an dem Template Code nichts geändert hat.

NFA4 - Erweiterung zu JIRA Sowohl die Architektur als auch die Implementierung soll so aufgebaut sein, dass eine Erweiterung der Synchronisation zu JIRA möglich ist. Dies beinhaltet insbesondere ein Austauschformat, welches flexibel zur Kommunikation eingesetzt werden kann.

NFA5 - Mandantenfähigkeit Der Prototyp muss in der Lage sein, mehrere Mandanten gleichzeitig bedienen zu können, sodass ein individuelles Provisionieren einer Version für einen Kunden nicht notwendig ist. Dafür müssen die Daten pro Mandant gespeichert werden.

NFA6 - Entwicklungsprozess Es soll ein Entwicklungsprozess genutzt werden, der den Source Code des Prototyps automatisiert in der Cloud provisionieren kann.

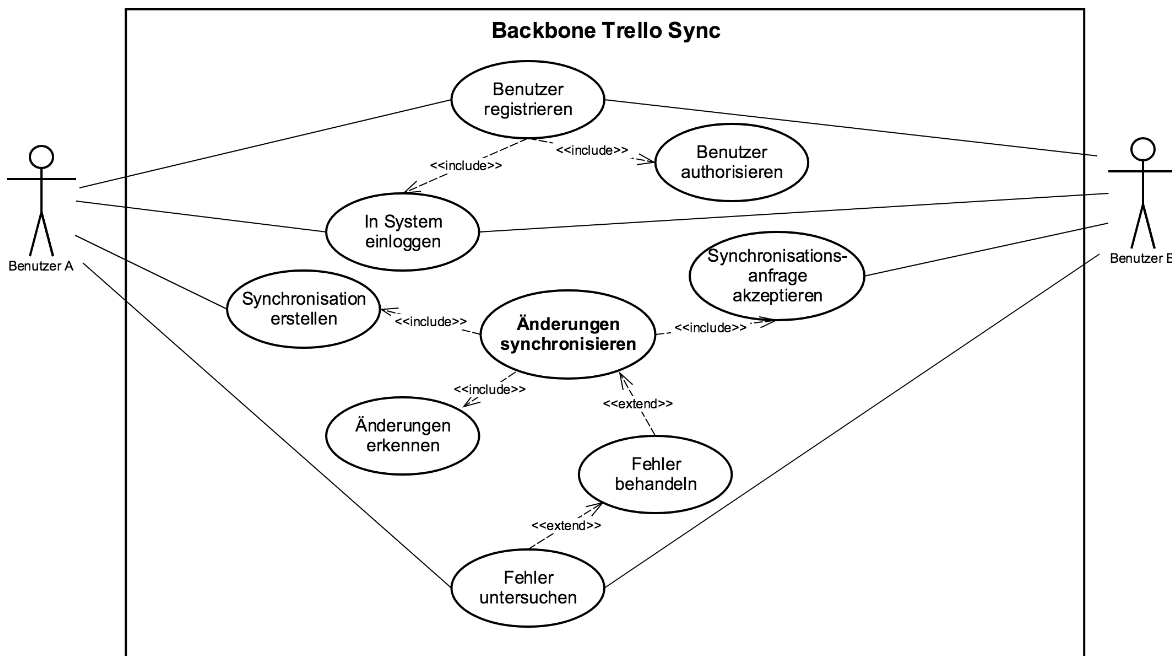


Abbildung 4.1: Anwendungsfälle des Systems.

4.1.3 Anwendungsfälle

Basierend auf den erarbeiteten Anforderungen sind in Abbildung 4.1 die entsprechenden Anwendungsfälle des Systems basierend auf zwei Synchronisations-Partnern dargestellt. Die einzelnen Anwendungsfälle werden im Folgenden genauer beschrieben:

1. **Registrierung** Der Anwendungsfall der Registrierung eines Benutzers tritt lediglich vor der ersten Benutzung des Systems auf. Ein Benutzer muss sich registrieren, damit seine Konfigurationen und sein Trello-Account verknüpft werden können. Damit kann das System mithilfe von OAuth Anfragen die Daten eines Trello-Boards abfragen und weitere Aktionen durchführen, wie in Abschnitt 3.1.2 beschrieben wurde. Wenn dieser Prozess abgeschlossen ist, können nach dem Benutzer-Login Synchronisations-Einstellungen vorgenommen werden. Das Ziel dieses Anwendungsfalles ist die Registrierung zweier Nutzer, die ihre Trello-Boards synchronisieren möchten, sodass die Konfiguration einer Synchronisation möglich ist.

2. **Synchronisation von Änderungen** Die Synchronisation der Änderungen ist der Hauptanwendungsfall des Systems. Dies beinhaltet, dass *Benutzer A* eine Konfiguration zur Synchronisation mit einem anderen *Benutzer B* erstellt, welche wiederum von diesem akzeptiert werden muss. Nach der erfolgreichen Konfiguration beginnt die automatische Synchronisation der Änderungen. Änderungen werden automatisch von dem System erkannt, ohne dass ein Benutzer aktiv werden muss. Eine Fehlerbehandlung erweitert

die Synchronisation und Benutzer sollen die Möglichkeit haben, diese Fehler zu untersuchen. Das Ziel dieses Anwendungsfalles ist die Synchronisation von Listen, Karten, Kommentaren und Anhängen innerhalb eines Boards.

4.2 Cloud-Architektur

Im Folgenden wird eine geeignete Architektur für die erarbeiteten Anwendungsfälle und Anforderungen vorgestellt. Dazu wird zuerst mit der System-Architektur ein Überblick über die Systemkomponenten gegeben. Nachfolgend werden die einzelnen Aspekte zur Synchronisation und der Mandantenfähigkeit innerhalb des Systems näher erläutert. Im Anschluss daran wird das Konzept zum Deployment des Systems in der Cloud vorgestellt.

4.2.1 Bedingungen für die Architektur

Ein Power-Up ist durch die Vorgaben seitens Trello an eine klassische Client-Server-Architektur gebunden. In diesem Szenario stellt das System des Power-Ups den Server dar, welcher die eigenen Benutzer (Clients) bedient. Clients können das Power-Up benutzen, indem sie mithilfe des in die Trello-Oberfläche eingebundenen HTML-IFrame direkt mit dem Power-Up interagieren. Der „Server"-Teil in der Architektur ist in diesem Fall jedoch nur eine Bezeichnung des Gesamtsystems, welches laut Anforderung NFA2 ohne zu verwaltenden Server auskommen soll. Für die Erweiterung bedeutet dies, dass eine Architektur benötigt wird, die nicht nur statische Dateien wie die Descriptor-Datei und weitere notwendige HTML-, JavaScript- und CSS-Dateien ausliefern, sondern auch die entsprechende Synchronisation durchführen kann. Unter Berücksichtigung der nicht-funktionalen Anforderung NFA2, dass AWS als Cloud Provider verwendet werden soll, wird der Service AWS Lambda zur Umsetzung der Serverless Architektur verwendet. Weil AWS Lambda eine eventbasierte Architektur unterstützt und Trello die Möglichkeit zur Webhook-Registrierung bietet, bildet diese Kombination eine sehr gute Grundlage der Architektur. So ist sichergestellt, dass das System nur betrieben wird, wenn es tatsächlich notwendig ist.

4.2.2 System-Architektur

Abbildung 4.2 beschreibt basierend auf den vorherigen Bedingungen eine System-Architektur auf Basis verschiedener AWS Services. Die Architektur verwendet CloudFront, um alle HTTP-Anfragen entgegenzunehmen und diese entsprechend an andere Services weiterzuleiten. Falls sich die Anfrage auf eine statische Datei bezieht, wird die Anfrage entweder an den Simple Storage Service (S3) weitergeleitet oder aus dem Cache beantwortet. S3 wird dazu genutzt, um aus einem Bucket die Descriptor-Datei des Power-Ups und statische Website-Dateien für die Einbindung durch ein HTML-IFrame auszuliefern. Daneben wird ein S3-Bucket genutzt, um

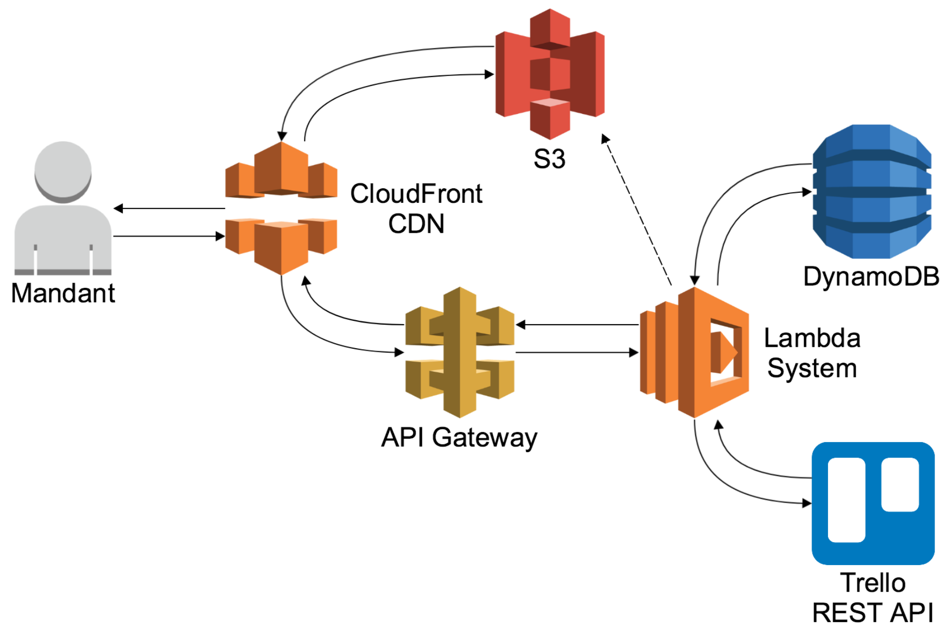


Abbildung 4.2: Die System-Architektur basierend auf den verwendeten Services.

dort Fehlerdateien abzulegen, falls eine Synchronisation fehlschlägt (vgl. Abschnitt 4.2.3). Falls sich die Anfrage jedoch auf die REST API des Systems bezieht, wie z.B. `/api/*`, wird die Anfrage an das API Gateway weitergeleitet (vgl. Abschnitt 4.2.2 - REST API des Systems). Das API Gateway wiederum leitet die Anfrage an den AWS Lambda Service weiter, falls für den URL-Pfad eine Zuordnung zu einer Lambda-Funktion existiert, beispielsweise für `/api/webhooks/update`. AWS Lambda sorgt daraufhin dafür, dass eine Instanz der Lambda-Funktion existiert und mit den entsprechenden Parametern aufgerufen wird. Der zentrale Bestandteil der Architektur ist deshalb das in Abbildung 4.2 bezeichnete „Lambda-System", welches aus mehreren Lambda-Funktionen besteht. Die Funktionen dienen zur Konfiguration einer Synchronisationsbeziehung und führen die Synchronisation zwischen den Trello Boards durch (vgl. Abschnitt 4.2.3).

Weiterhin greifen die Lambda-Funktionen noch auf zwei weitere Services zu: Zum einen auf DynamoDB und zum anderen auf die Trello REST API. Da AWS Lambda nicht als Persistenzschicht geeignet ist, werden DynamoDB-Tabellen genutzt. Sie enthalten Daten für die Tenant-Informationen, Webhooks und Synchronisationen (vgl. Abschnitt 4.2.3 und Abschnitt 4.2.4). In einer DynamoDB-Tabelle können Daten prinzipiell beliebig gespeichert werden, müssen jedoch Attribute für das entsprechende Schlüssel-Schema besitzen. Anhand des Schlüssels können die Daten dann abgefragt oder auch wieder entfernt werden. Zusätzlich müssen bei einer Synchronisation noch Daten von der Trello REST API angefragt werden. Zum Beispiel können anhand der HTTP GET Methode und der URL `https://api.trello.com/1/cards/{id}` Daten zu einer bestimmten Karte abgefragt werden. Für Listen und Boards sind die URLs äquivalent gestaltet, also `https://api.trello.com/1/lists/{id}`

HTTP-Methode	URL
POST	/api/user/register
POST	/api/user/login
POST	/api/oauth/login
POST	/api/oauth/callback
GET, PUT, DELETE	/api/config/{projectId}
POST	/api/config
POST	/api/webhooks/update

Tabelle 4.1: REST API des Power-Ups.

und `https://api.trello.com/1/boards/{id}`. Wird jedoch ein Anhang einer Karte benötigt, muss die URL `https://api.trello.com/1/cards/{id}/attachments/{attachmentId}` abgefragt werden.

REST API des Systems

In Tabelle 4.1 sind die einzelnen HTTP-Schnittstellen des Systems aufgelistet, welche durch das API Gateway bereitgestellt werden. Sie werden im Folgenden noch näher erläutert:

/api/user/* Schnittstellen zur Registrierung und zum Login eines Benutzers des Power-Ups. Dies ist notwendig, damit die OAuth-Daten für den Trello-Benutzer gespeichert und zugeordnet werden können.

/api/oauth/* Schnittstellen, die einen OAuth-Prozess mit Trello einleiten und OAuth-Token und -Secret mit dem Power-Up-Benutzer verknüpfen. Der Registrierungsprozess wird in Abschnitt 4.2.4 detaillierter beschrieben.

/api/config/* Schnittstellen, um die Konfiguration einer Synchronisation zu verwalten: Abrufen (GET), Hinzufügen (POST), Aktualisieren (PUT) und Löschen (DELETE).

/api/webhooks/update Ermöglicht Empfang der Trello Webhooks und startet asynchron eine Synchronisation, indem eine weitere Lambda-Funktionen aufgerufen wird.

Hinter jeder genannten Schnittstelle verbirgt sich eine Lambda-Funktion, welche durch die Weiterleitung der HTTP-Anfrage aufgerufen wird. Die wichtigste Schnittstelle ist die zum Empfang eines Webhooks. Dadurch erkennt das Power-Up, dass eine Änderung innerhalb eines Boards ausgelöst wurde. Wieso das notwendig ist, wird im folgenden Abschnitt genauer dargelegt.

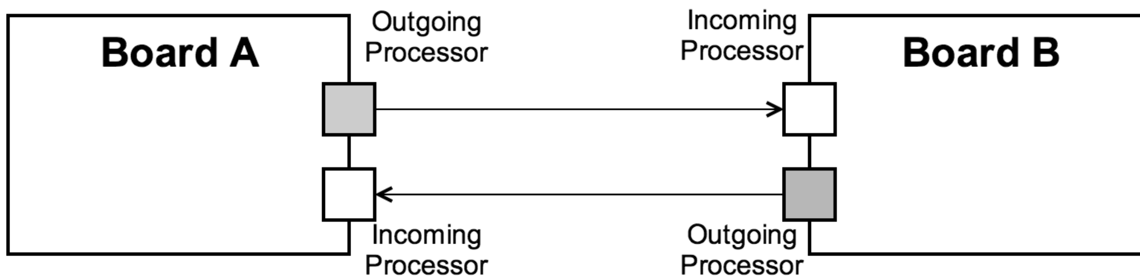


Abbildung 4.3: Aufbau der Synchronisationsbeziehung zwischen zwei Trello Boards mit je einem virtuellen Ausgangs- („Outgoing Processor") und Eingangskanal („Incoming Processor").

4.2.3 Synchronisation

Trello ist ein nach außen hin verschlossenes System, welches über eine REST API verfügt, um Daten abzufragen oder zu verändern. Deshalb ist es nicht möglich, direkt mit der Datenbank zu kommunizieren und mithilfe einer Sprache wie SQL die Trello-Daten zu synchronisieren. Eine einfache und naive Alternative wäre, die REST API zu nutzen, um regelmäßig alle Daten von einem Projekt in ein anderes zu übertragen und dadurch einen konsistenten Zustand zu erreichen. Dieser Ansatz ist jedoch sehr zeit- und kostenintensiv. Eine solche Synchronisation kann unter Berücksichtigung von Netzwerklatenzen, etwaigen Limitierungen von Zugriffen auf eine Schnittstelle und Anzahl an Anfragen viel Zeit in Anspruch nehmen. Eine Optimierung dieses Ansatzes wäre, dass das System in der Lage ist zu erkennen, welche neuen Änderungen vorhanden sind. Dies würde jedoch voraussetzen, dass ein System wie Trello eine Möglichkeit bietet, nach solchen Änderungen direkt zu suchen. Ansonsten müssten sehr viele Daten übertragen und eigenhändig analysiert werden. Nach diesem Prinzip funktioniert das bestehende Backbone Issue Sync System, welches nach Ablauf eines konfigurierbaren Intervalls (standardmäßig fünf Sekunden) nach neuen Änderungen sucht und diese anschließend synchronisiert. Auf einem lokalem System ist dies eine durchaus praktikable Lösung, jedoch führen im Cloud-Kontext Faktoren wie Ausführungszeit, zusätzliche HTTP-Anfragen und verwendete Ressourcen wie Datentransfer oder (Zwischen-)Speicherung der Daten zu erhöhten Kosten. Damit diese Faktoren möglichst gering gehalten werden, ist es sinnvoll, nach Möglichkeit eine eventbasierte Synchronisation zu verwenden [DL12]. Webhooks bieten sich deshalb als geeignete Lösung für dieses Problem an. Bevor näher auf die Anwendung der Webhooks eingegangen wird, soll zuerst der Aufbau einer Synchronisationsbeziehung kurz erläutert werden.

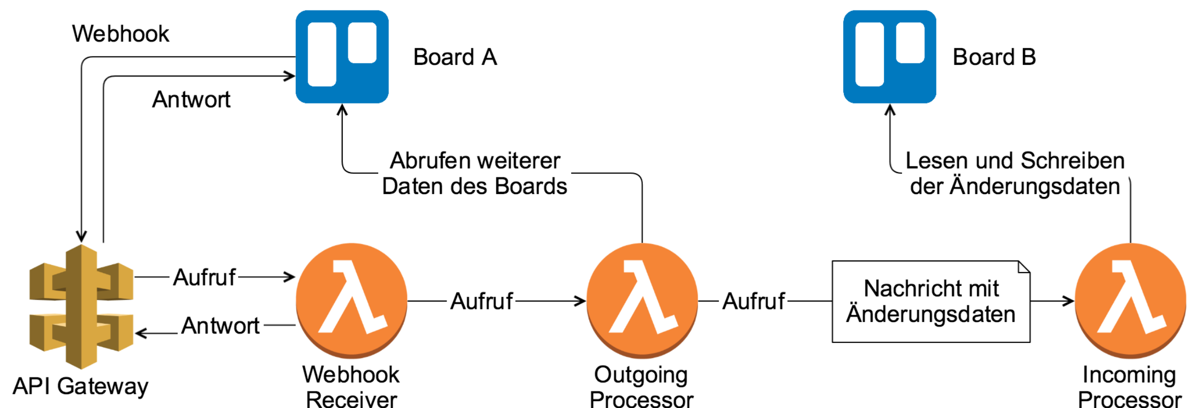


Abbildung 4.4: Ablauf einer Synchronisation zwischen den einzelnen Lambda-Funktionen.

Aufbau einer Synchronisationsbeziehung

Abbildung 4.3 verdeutlicht den prinzipiellen Aufbau einer Synchronisationsbeziehung zwischen zwei Trello Boards[2]. Jedes Board erhält einen virtuellen Ausgangkanal, den „Outgoing Processor“, welcher die Events eines Boards empfängt und zu einer Nachricht aufbereitet. Diese Nachricht wird daraufhin an einen virtuellen Eingangskanal des anderen Boards, den „Incoming Processor“, gesendet, welcher wiederum dafür zuständig ist, dass die Änderungen durch Aufrufen der REST API angewendet werden. Je nach Art der Änderung müssen Daten entweder hinzugefügt, aktualisiert oder gelöscht werden.

Dieser Ansatz ist notwendig, da Trello vielseitig einsetzbar ist und auch die Möglichkeit bietet, einzelne Felder oder Labels zu individualisieren. Durch die Vorverarbeitung der Änderungen im Outgoing Processor können die Daten auf ein Format gebracht werden, das vom Incoming Processor verstanden wird. Für komplexere Verarbeitungen sind ein aufwendigeres Field Mapping sowie entsprechende Konfigurationsmöglichkeiten notwendig, was allerdings nicht Ziel dieser Arbeit ist. Nichtsdestotrotz wäre es möglich, eine solche komplexere Verarbeitung in dieses System zu integrieren, indem die Ein- bzw. Ausgangskanäle erweitert werden. Zusätzlich unterstützt dieser Ansatz die in NFA4 geforderte Erweiterbarkeit zu JIRA, da auch entsprechende Verarbeitungsschritte für ein JIRA-Format integriert werden können.

Synchronisationsprozess

Der Ablauf einer Synchronisation bezogen auf die wichtigsten System-Komponenten sieht nun wie folgt aus (vgl. Abbildung 4.4):

[2]Eine solche Beziehung zwischen zwei Projekten bzw. hier Trello Boards wird auch als „Integration“ bezeichnet.

1. **Auslösen des Webhooks:** Sobald eine Änderung im Trello Board A registriert wurde, wird ein Webhook ausgelöst. Dieser wird durch das API Gateway transportiert und von einer Lambda-Funktion, dem „Webhook Receiver“, entgegengenommen.
2. **Empfang des Webhooks:** Die Webhook Receiver Funktion entnimmt dem Webhook daraufhin die notwendigen Informationen, beispielsweise den Eigentümer des Boards (im Folgenden auch „Tenant“ genannt), die Board ID und die eigentlichen Daten, die im Board verändert wurden. Diese Informationen werden daraufhin an den Outgoing Processor übergeben. Beim Empfang eines Webhooks ist es wichtig, umgehend eine Empfangsbestätigung an Trello zu senden. Ansonsten würde Trello versuchen, den Webhook erneut zu versenden (begrenzt auf max. 2 Versuche).
3. **Nachricht vorbereiten:** Der Outgoing Processor erhält die Daten des Webhooks und prüft, ob der Webhook neue oder bereits bekannte Änderungen enthält. Falls sie bekannt sind, wird der Webhook ignoriert. Falls nicht, bereitet der Outgoing Processor die Nachricht zur Übermittlung der Änderungen auf. Dazu müssen ggf. weitere Daten des Trello Boards abgefragt werden. Im Fall des Prototyps kann der Outgoing Processor den Incoming Processor direkt aufrufen und die Nachricht übergeben. Allerdings wäre es auch möglich, die Nachricht über einen anderen Kanal zu kommunizieren, falls die Erweiterung auf andere Systeme verteilt werden soll.
4. **Nachricht empfangen:** Der Incoming Processor empfängt die Nachricht des Outgoing Processors und führt entsprechende Aktionen an der REST API für das Trello Board B aus. Zum Beispiel werden neue Karten hinzugefügt, die Beschreibung einer Karte aktualisiert oder ein Dateianhang gelöscht.
5. **Auslösen eines weiteren Webhooks:** Durch das Anwenden der Änderungen wird ein neuer Webhook ausgelöst, sodass die Schritte zur Synchronisation erneut durchlaufen werden. In diesem Fall wären dann in Abbildung 4.4 die beiden Trello Boards in ihren Positionen vertauscht, weil das Trello Board B den Webhook ausgelöst hat. Jedoch wird in Schritt 3 der nun ankommende Webhook ignoriert, weil die Änderungen bereits bekannt sind.

Da nicht alle Schritte einer Synchronisation immer einwandfrei durchlaufen werden, müssen auftretende Fehler behandelt werden. Dies kann mit einer Fehlerbehandlung umgesetzt werden, was im folgenden Abschnitt beschrieben ist. Weiterhin ist sicherzustellen, dass alle Änderungen innerhalb eines Trello Boards erkannt werden. Dafür muss zusätzlich zu den Webhooks ein Dienst regelmäßig ausgeführt werden, der die vermissten Änderungen erkennt und deren Synchronisation nachholt. Dies wird im darauffolgenden Abschnitt näher erläutert.

Fehlerbehandlung

Während der Synchronisation können sowohl bei der Aufbereitung einer Nachricht im Outgoing Processor als auch beim Anwenden der Änderungsdaten im Incoming Processor ver-

schiedene Fehler auftreten. Manche können automatisch gelöst werden, indem der Synchronisationsschritt erneut ausgeführt wird. Jedoch ist dies nicht bei allen Fehlertypen möglich. Für den Outgoing Processor kann eine grobe Kategorisierung (OE - Outgoing Error) wie folgt vorgenommen werden:

OE1 - Trello nicht erreichbar Der Outgoing Processor muss häufig weitere Daten des Trello Boards abfragen. Da kann es vorkommen, dass das Trello System nicht verfügbar ist. Ein Ausfall kann sehr kurz, aber auch durchaus länger sein. Für einen solchen Fehlerfall kann eine Wiederholung nach einer kurzen Pause hilfreich sein. Nach 3 fehlgeschlagenen Wiederholungen sollte der Versuch jedoch abgebrochen und der Fehler nicht weiter bearbeitet werden, da die Wahrscheinlichkeit zur Lösung des Problems immer geringer wird.

OE2 - Unbekannter Fehler Ein unbekannter Fehler kann viele Ursachen besitzen: Beispielsweise können Daten, die von der Trello REST API abgefragt wurden, ein unbekanntes Format haben. Dies kann zu einem unerwartetem Verhalten des Programms führen. Solche Fehler können nicht einfach wiederholt werden. Deshalb werden sie lediglich registriert und dem Benutzer angezeigt.

Die Einteilung der Fehlerkategorien auf der Seite des Incoming Processors (IE - Incoming Error) ist etwas differenzierter. In diesem Fall können zusätzlich Fehler auftreten, die ihre Ursache in der Architektur haben.

IE1 - Abhängiges Objekt fehlt Viele Änderungen hängen von anderen Änderungen ab. Zum Beispiel kann eine Karte einem Board nur hinzugefügt werden, wenn sie einer Liste zugeordnet ist. Existiert diese Liste jedoch noch nicht, kommt ein Fehler zustande. Die Ursache kann zum Beispiel sein, dass während der Synchronisation der Liste ein Fehler aufgetreten ist.

IE2 - Ungültige Sequenz Für jedes zu synchronisierende Objekt existiert eine Synchronisationsbeziehung. Die Änderungen pro Beziehung werden sequentiell nummeriert, damit die Änderungshistorie beibehalten werden kann. Unter Umständen kann eine Synchronisationsnachricht nun schneller beim Incoming Processor angekommen sein, was dann zu dem Fehler der ungültigen Sequenz führt. Eine Ursache kann eine lange Verarbeitungszeit der vorherigen Nachricht sein. Ein solcher Fehler kann ebenfalls wiederholt werden, da es durchaus möglich ist, dass die fehlende Nachricht innerhalb kurzer Zeit noch eintrifft.

IE3 - Trello nicht erreichbar Dieser Fehler ist äquivalent zum Fehler des Outgoing Processors.

IE4 - Unbekannter Fehler Dieser Fehler ist äquivalent zum Fehler des Outgoing Processors.

In beiden Lambda-Funktionen werden auftretende Fehler in einer Fehlerdatei auf ein separates S3-Bucket hochgeladen. Eine Fehlerdatei enthält nähere Angaben zum auftretenden Fehler

sowie die Daten, die in der jeweiligen Funktion zu Beginn verfügbar waren. Zur Fehlerbehandlung können die Dateien entsprechend wieder heruntergeladen und auch dem Benutzer im Browser angezeigt werden. Dies ist zwar keine vollständige Fehlerbehandlung, allerdings ausreichend für die Anforderung FA6.

Scheduled Job

Wie bereits in Abschnitt 3.1.2 beschrieben, sind Webhooks ebenfalls Nachrichten, die über das Netzwerk versendet werden und deshalb potentiell verloren gehen können. Dafür gibt es vielfältige Gründe: Falls Trellos Komponente zum Versenden der Webhooks ausfällt, der Cloud Provider Probleme mit der Infrastruktur hat oder weil im Programmcode zum Empfangen der Webhooks sich ein Fehler eingeschlichen hat, können Webhooks nicht empfangen und Änderungen nicht synchronisiert werden. Dies stellt für das zu entwickelnde System ein Problem dar. Deshalb muss eine Möglichkeit geschaffen werden, die auch Änderungen synchronisiert, welche nicht empfangen wurden. Ein Ansatz wäre, bei jedem empfangenen Webhook zu überprüfen, ob seit dem zuletzt empfangenen Webhook weitere Änderungen aufgetreten sind. Falls das zutrifft, werden diese Änderungen auch noch synchronisiert. Mit dieser Methode wäre das Problem relativ elegant gelöst, ohne zusätzliche Komponenten zu entwickeln. Allerdings löst dieser Ansatz das Problem nicht vollständig. Der Ansatz funktioniert nicht, wenn nach einem Webhook keine weiteren Änderungen ausgelöst werden oder Webhooks für eine sehr lange Zeit ausfallen.

Aus diesem Grund muss das System regelmäßig kontrollieren, ob alle letzten Änderungen eines Boards erfasst wurden. Zur Umsetzung eines solchen „Scheduled Job"s kann ebenfalls AWS Lambda verwendet werden, denn AWS bietet die Möglichkeit eine Funktion regelmäßig und geplant aufzurufen. Die Häufigkeit bzw. das Intervall kann mithilfe eines Ausdrucks definiert werden, z.B. täglich oder stündlich. Wenn die Funktion aufgerufen wird, kontrolliert sie für jede Synchronisation zwischen zwei Trello Boards, ob neue Events seit der letzten Synchronisierung aufgetreten sind. Sobald sie erkennt, dass Änderungen nicht synchronisiert wurden, werden die notwendigen Synchronisierungsdaten so aufbereitet, dass sie dem Outgoing Processor übergeben werden können. Daraufhin wird ein regulärer Synchronisationsprozess durchlaufen.

4.2.4 Mandantenfähigkeit

Die Mandantenfähigkeit (engl. multi-tenancy) ist ein zentraler Aspekt vieler Software-Systeme in der Cloud [KMK12]. In diesem Kontext werden Benutzer bzw. eine Gruppe von Benutzern eines Systems auch als „Tenants" (deutsch: Mandanten) bezeichnet [KMK12]. Aufgrund der Anforderung NFA5 soll die Mandantenfähigkeit im zu entwickelnden System umgesetzt werden, da sie in der bisherigen Erweiterung Backbone Issue Sync nur teilweise vorhanden ist. In der existierenden On-Premise-Implementierung ist es nur möglich, *1:n* Beziehungen zu verwalten.

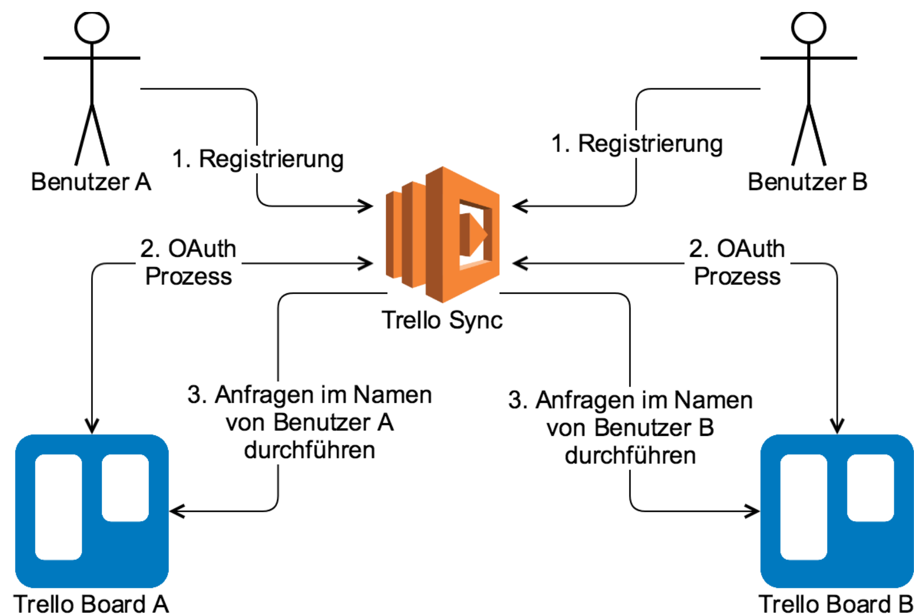


Abbildung 4.5: Der Ablauf der Registrierung bis zur Verwendung der Trello Synchronisation am Beispiel von zwei Benutzern, die ihre Trello Boards synchronisieren.

Das heißt, ein Tenant kann lediglich eine Integration zwischen seinem System und anderen Systemen erstellen. Jedoch können andere Tenants keine Integrationen zu anderen Tenants erstellen, ohne die Erweiterung auf ihrem eigenen JIRA System zu installieren. Der Prototyp muss deshalb im On-Demand-Kontext in der Lage sein, mehrere Tenants mit mehreren Integrationen, also *m:n* Beziehungen, verwalten zu können. Um dieses Ziel zu erreichen, sind zwei Voraussetzungen zu erfüllen: Zum einen müssen sich Trello-Benutzer mit dem Power-Up-Prototyp verknüpfen können. Zum anderen müssen sie die Möglichkeit haben, mehrere Integrationen anzulegen ohne zusätzliche Deployments durchführen zu müssen.

Abbildung 4.5 beschreibt die erste Voraussetzung: Ein Trello-Benutzer registriert sich über eine Weboberfläche mit dem Power-Up und verknüpft beide Logins miteinander. Dabei erteilt er dem Power-Up anhand eines OAuth-Prozesses die Berechtigung, die Trello REST API in seinem Namen zu benutzen. Der genaue Ablauf des OAuth-Prozesses ist in Abbildung 4.6 abgebildet. Dieser Prozess wird durch das Power-Up initiiert. Er ist erfolgreich abgeschlossen, sobald das Power-Up die entsprechenden Authentifizierungsdaten erhalten hat. Diese Daten werden daraufhin in einer Tenant-Tabelle in DynamoDB zusammen mit den anderen Tenant-Informationen gespeichert. Nach erfolgreichem Abschluss des Prozesses ist der Prototyp in der Lage, HTTP-Anfragen an die Trello REST API zu stellen und auch Synchronisationen durchzuführen. Für die Erstellung einer Integration müssen beide beteiligte Tenants diesen Prozess durchlaufen.

Die zweite Voraussetzung umfasst die daraufhin vorhandene Möglichkeit der Tenants, Integrationen zwischen zwei Trello Boards anzulegen. Damit das System weiß, dass eine Aktion

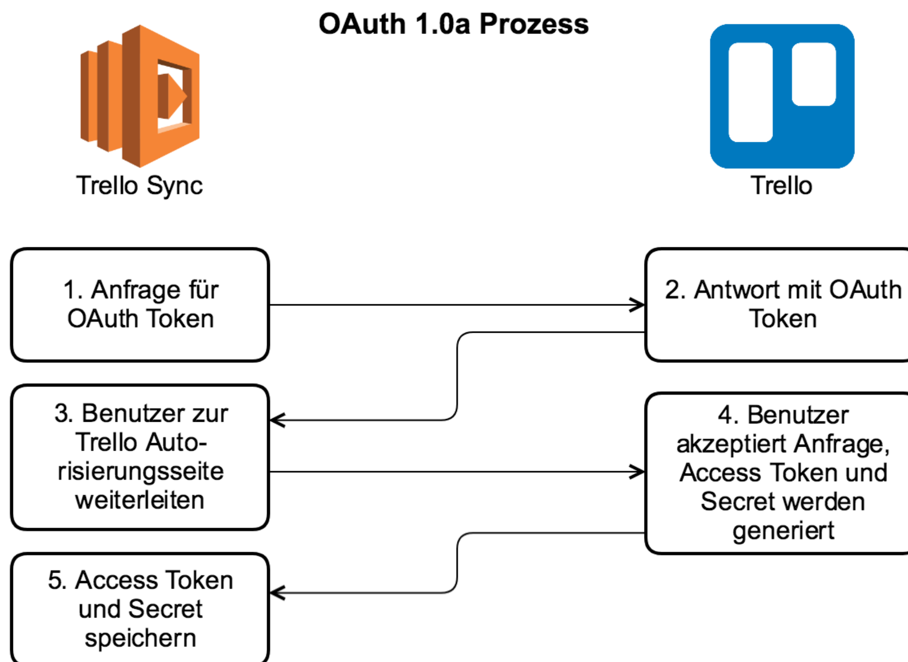


Abbildung 4.6: Der OAuth-Prozess zur Authorisierung des Trello Sync Systems, dass HTTP-Anfragen im Namen eines Benutzers gestellt werden dürfen. Vereinfachte Darstellung nach [Int10].

(z.B. das Anlegen einer Integration) von einem bestimmten Tenant ausgeführt wird, muss eine Zuordnung ermöglicht werden. Diese Zuordnung zwischen Tenant und Integration erfolgt anhand einer Tenant ID, die während der Registrierung generiert wird. Aus diesem Grund ist ein zusätzliches Deployment mit separater Datenbank nicht notwendig und für jeden Tenant können dieselben Datenbank-Tabellen verwendet werden.

4.2.5 Autorisierung einer Integration

Sobald beide Parteien einer Integration die OAuth-Autorisierung erfolgreich abgeschlossen haben, kann eine Integration theoretisch gestartet werden. Allerdings muss gewährleistet sein, dass ein Benutzer keine beliebige Integration mit anderen Benutzern erstellen kann. Dadurch könnten Daten unerlaubt entwendet werden. Dies wäre technisch durchaus möglich, da das System aufgrund der Autorisierung auf alle Daten der Tenants bereits Zugriff hat. Deshalb muss ein Prozess eingeführt werden, welcher die Anforderung FA4 umsetzt und sicherstellt, dass jeder Tenant einer Integration explizit zustimmen muss. Abbildung 4.7 veranschaulicht diesen Prozess anhand eines Aktivitätsdiagramms. Ein Benutzer muss dafür die Integration initial erstellen, wodurch er seine Zustimmung direkt erteilt. So kann ein zweiter Benutzer diese akzeptieren und dadurch seine Zustimmung signalisieren. Das System wird daraufhin

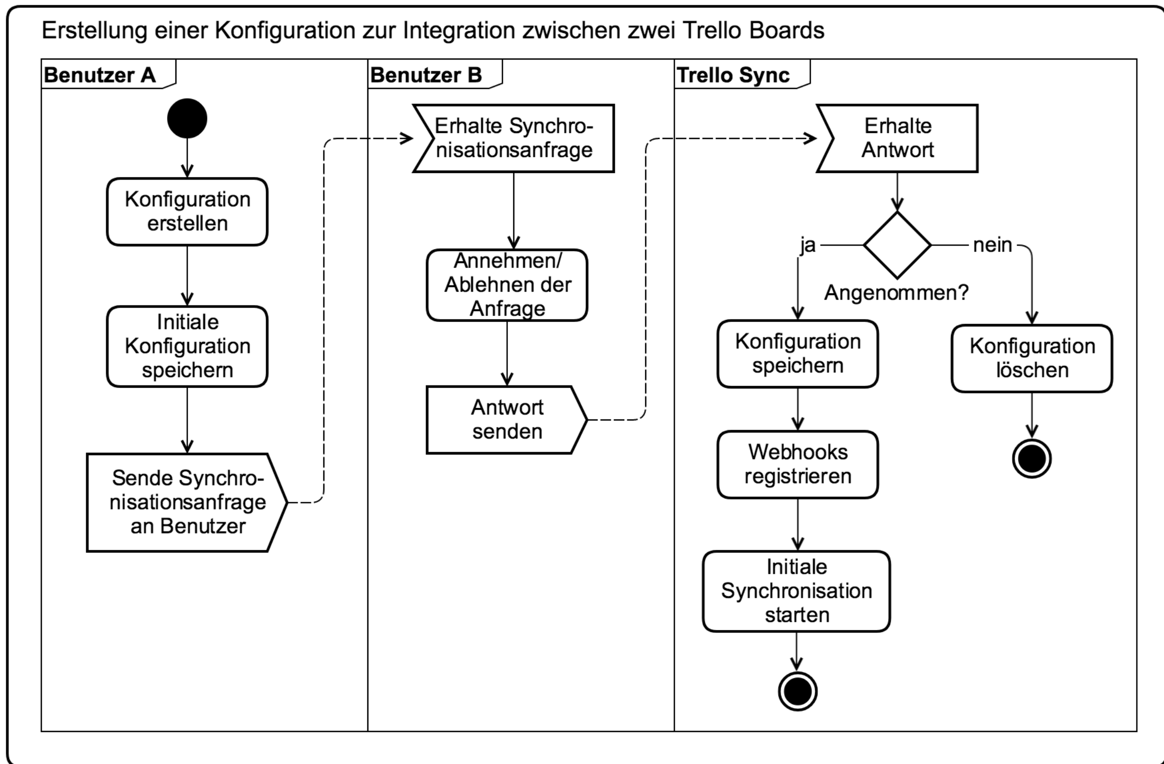


Abbildung 4.7: Aktivitätsdiagramm zum Erstellen einer Integration mit expliziter Zustimmung.

die Webhooks für die entsprechenden Boards bei Trello registrieren[3] und eine initiale Synchronisation starten. Diese ist notwendig, damit ein konsistenter Synchronisationszustand erreicht wird und auf beiden Seiten die Elemente sind, die laut Konfiguration synchronisiert sein sollen. Ein solcher Schritt kann je nach Größe eines Boards etwas Zeit in Anspruch nehmen, weil dafür viele HTTP-Anfragen zwischen den verschiedenen Diensten notwendig sind. Jedoch hat dies wenig Auswirkungen auf die Benutzbarkeit des Systems hat, da die Synchronisation im Hintergrund abläuft. Falls ein Benutzer seine Entscheidung widerrufen möchte, kann er dies tun, indem er die entsprechende Integration löscht. Dadurch werden alle Informationen zur Integration aus den Datenbank-Tabellen entfernt und auch die registrierten Webhooks der Integration gelöscht.

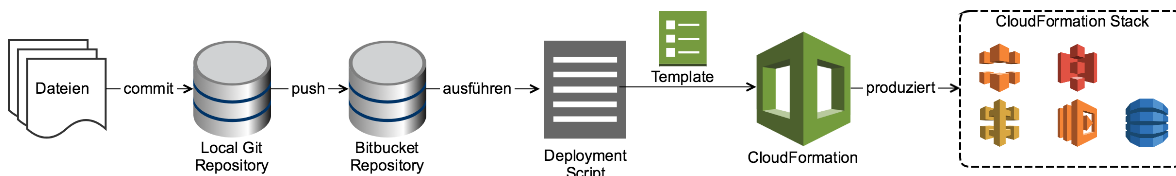


Abbildung 4.8: Entwicklungs- und Deploymentprozess vom Commit bis zur Änderung der Services.

4.3 Automatisiertes Deployment

Ein weiteres Ziel dieser Arbeit ist es, dass ein geeigneter Entwicklungsprozess entwickelt wird. Er soll sicherstellen, dass die Architektur in der Cloud provisioniert wird. Dieser Prozess soll weitestgehend automatisiert sein, damit die Wahrscheinlichkeit manueller Fehler und die Zeit zur Adaption von Änderungen minimiert werden kann. Um dieses Ziel zu erreichen, muss es möglich sein, die Architektur durch eine Beschreibungssprache auszudrücken und manuelle Eingriffe minimal zu halten. Zur Unterstützung eines solchen Ziels bietet sich AWS CloudFormation an. Dabei werden in einer YAML- oder JSON-Datei die einzelnen Datenbanken, S3-Buckets, Lambda-Funktionen und das CloudFront CDN konfiguriert. Diese Datei wird im AWS Kontext auch als „Template“ bezeichnet, was dem eigentlichen Zweck sehr nahe kommt: Es wird eine Vorlage erstellt, aus der automatisch ein Stack mit den entsprechenden Services generiert und provisioniert wird. Das Einlesen des Templates und die Provisionierung wird von AWS CloudFormation ausgeführt. AWS CloudFormation leitet auch entsprechende Rollback-Prozesse ein, falls aufgrund eines Fehlers im Template einzelne Services nicht provisioniert werden können. Das bedeutet, dass ein Stack-Update immer nur ganz oder gar nicht provisioniert wird.

Der gesamte Entwicklungsprozess ist in Abbildung 4.8 dargestellt. Die einzelnen Schritte werden im Folgenden detaillierter beschrieben:

1. **Commit von Änderungen:** Der Quellcode wird mit einem Versionskontrollsystem wie Git[4] verwaltet. Sobald Änderungen des Codes fertiggestellt sind, werden sie in das lokale Repository des Entwicklers eingetragen. Der Quellcode enthält neben den eigentlichen Quelldateien auch ein Deployment Skript und ein CloudFormation Template. Dies hat den Vorteil, dass Änderungen an diesen Dateien (und damit auch an der System-Architektur) ebenfalls versioniert werden.

2. **Push von Änderungen:** Zu einem gewissen Zeitpunkt werden die Änderungen in das Bitbucket Repository hochgeladen. Bitbucket ist ebenfalls ein Service von Atlassian, der für Softwareprojekte eine Versionskontrolle auf Basis von Git anbietet.

[3] Die Webhooks werden pro Integration registriert. So können sie später eindeutig zugeordnet werden.
[4] https://git-scm.com/

3. **Ausführen des Deployment Skripts:** Nachdem die Änderungen in Bitbucket registriert wurden, wird mithilfe des Tools Bitbucket Pipelines automatisch ein Deployment Skript ausgeführt. Das Skript stellt in einzelnen Schritten sicher, dass der Quellcode zu einem Artefakt zusammengefügt wird, automatisierte Tests durchgeführt werden und das Deployment mithilfe von AWS CloudFormation gestartet wird.
4. **CloudFormation Deployment:** Das Deployment Skript ruft den AWS CloudFormation Service auf und übergibt das entsprechende Template. AWS CloudFormation prüft das Template auf formale und inhaltliche Korrektheit (z.B. werden auch Circular Dependencies erkannt), erstellt ein sogenanntes „Change Set“[5] und sorgt durch Anwendung des Change Sets für eine Aktualisierung des Stacks. In diesen Schritt kann nicht manuell eingegriffen, sondern lediglich die Ergebnisse der Einzelschritte in der AWS Console überprüft werden. Erst nachdem alle Aktualisierungen erfolgreich durchgeführt wurden, sind diese auch tatsächlich sichtbar. Ist eine einzelne Aktualisierung nicht erfolgreich, wird ein Rollback aller bisher getätigten Aktualisierungen durchgeführt, sodass der Stack wieder im Originalzustand ist.

Alle aufgeführten Schritte werden automatisch ausgeführt und es ist kein manueller Eingriff notwendig (ausgenommen der Commit und Push in das Repository). Dadurch kann das Ziel erreicht werden, manuelle Fehler zu minimieren. Die Zeit zur Adaption von Änderungen hängt hier von der Dauer der auszuführenden Tests sowie des Deployments ab, worauf nur bedingt Einfluss genommen werden kann.

Anhand des vorgestellten Entwicklungsprozesses ist auch gut zu erkennen, dass lediglich ein einzelnes Gesamtsystem in der AWS Cloud provisioniert wird und deshalb jeder Tenant mit ein und demselben System interagiert. Dies unterscheidet sich von der bisherigen On-Premise-Lösung, da dort für jedes JIRA System ein eigenes Backbone Issue Sync System installiert werden muss.

4.3.1 Verschiedene Deployment-Umgebungen

Basierend auf dem vorgestellten Prozess können mithilfe von Git, Bitbucket und AWS auch verschiedene Stufen (engl. stages) zur Entwicklung verwendet werden. So kann eine Development-, Staging- und Production-Stufe eingerichtet werden, welche für verschiedene Test-Szenarien und Qualitätssicherungsmaßnahmen genutzt werden können. Die Development-Stufe wird zum frühen Testen neuer Features herangezogen und beinhaltet immer den aktuellen Inhalt des *develop*-Branches des Git Repositorys. Hingegen beinhaltet die Staging-Stufe eine bestimmte Menge von neuen Features, welche bereits als Teil eines Releases festgelegt wurden. Da jede Stufe bereits die einzelnen automatisierten Deployment-Schritte aus dem vorherigen

[5] Ein Change Set enthält die Unterschiede zwischen dem aktuellen Stack und dem Stack, welcher durch das neue Template entstehen wird. AWS CloudFormation versucht dieses Change Set anzuwenden. Das heißt, es führt automatisch Anpassungen der Services durch, sodass der im Template angegebene Zustand erreicht wird.

Abschnitt durchlaufen musste, dienen die Stufen lediglich der letzten manuellen Überprüfung. Sofern keine Fehler oder Probleme auftreten, kann die Staging-Version freigegeben und in die Production-Stufe übertragen werden.

Zur Umsetzung dieser Stufen werden im Git-Repository drei entsprechende Branches angelegt: *develop*, *staging* und *production*[6]. Damit Änderungen von einer Stufe in die andere übergeben werden, müssen diese lediglich von einem Branch in den anderen übertragen werden, zum Beispiel durch ein `git merge`. Mithilfe von Bitbucket und Bitbucket Pipelines kann daraufhin eingestellt werden, dass bei jeder Änderung einer dieser Branches ein neues Deployment in die entsprechende Stufe erfolgt.

[6] Alternativ kann der *production*-Branch auch dem *master*-Branch entsprechen.

5 Umsetzung einer prototypischen Implementierung

Nachdem das Konzept im vorherigen Kapitel vorgestellt wurde, wird im Folgenden die Realisierung des Prototyps beschrieben. Zuerst wird die Implementierungsarchitektur im Detail vorgestellt. Daraufhin wird die Konfiguration einer Integration und der Prozess zur Erstellung einer solchen beschrieben. Im Anschluss wird der Synchronisationsprozess sowie dessen Besonderheiten in Bezug auf die Implementierungsarchitektur erläutert. Dieser Abschnitt beschreibt ebenfalls spezielle Herausforderungen, die teilweise eine Anpassung der Architektur erfordern.

5.1 Implementierungsarchitektur

Wie bereits im letzten Kapitel beschrieben wurde, besteht die System-Architektur aus den folgenden Komponenten: Einem CloudFront-Service zur Verteilung und Caching von Anfragen, einem S3-Bucket zur Auslieferung statischer Dateien, einem API Gateway zum Aufrufen der Lambda-Funktionen per HTTP, den einzelnen Lambda-Funktionen, einem weiteren S3-Bucket zur Speicherung von Fehlerdateien und mehreren Datenbanktabellen zur Speicherung von Tenant- und Synchronisationsdaten. Insgesamt existieren vier Datenbanktabellen: Eine Tenant-Tabelle zur Speicherung der Tenant-Informationen (Benutzername, Password, Trello-Benutzername, OAuth-Token und -Secret), eine Integrations-Tabelle zur Speicherung der Integrationsbeziehungen, eine Synchronisations-Tabelle zur Speicherung der Synchronisationsbeziehungen zwischen einzelnen Trello Objekten wie Listen oder Karten und eine Webhook-Tabelle zur Verwaltung der registrierten Webhooks der einzelnen Trello Boards. Im Folgenden wird nun genauer dargelegt, wie diese Architektur im CloudFormation Template abgebildet wird. Danach werden die Speicherung einer Integration und der Synchronisationsprozess beschrieben.

5.1.1 Deployment

In Listing 5.1 ist ein Basis-Template dargestellt, das von AWS CloudFormation verarbeitet werden kann. Es definiert ein Serverless-Format, welches auf die Verwendung von Lambda-Funktionen zusammen mit entsprechenden Eventquellen ausgelegt ist [Ama17k]. Dem Templa-

Listing 5.1 Allgemeine Struktur eines YAML CloudFormation Templates basierend auf dem Serverless Application Model (SAM) [Ama17k].

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Basic structure for a CloudFormation template.

Parameters:
  # define which parameters can be used in this template,
  # e.g. set an environment like dev/stg/prod

Mappings:
  # define static key value maps,
  # e.g. to map an environment to a certain domain

Outputs:
  # define the return data (might be based on the deployed resources),
  # e.g. a URL to access a Lambda function via HTTP

Resources:
  # Lambdas, database tables, S3 buckets, etc.
```

te können Parameter übergeben werden, welche unter *Parameters* definiert werden müssen und zusammen mit *Mappings* in den Ressourcen verwendet werden können. *Outputs* dienen als eine Art Ausgabekanal und können mithilfe der AWS CLI oder per AWS Console abgefragt werden. Lambda-Funktionen sowie alle weiteren AWS Services werden unter dem Punkt *Resources* aufgeführt.

Das Deployment des Templates wird im Deployment Skript mit zwei Aufrufen der AWS CLI gestartet:

```
aws cloudformation package --template-file cfn.yml --output-template-file cfn.packaged.yml
--s3-bucket ${LAMBDA_BUCKET}
```

```
aws cloudformation deploy --template-file cfn.packaged.yml --stack-name ${STACK_NAME}
```

Der erste Aufruf lädt die Code-Artefakte der Applikation auf ein S3-Bucket und erzeugt eine neue Template-Datei, welche Referenzen auf die hochgeladenen Artefakte enthält. Im zweiten Aufruf lädt AWS CloudFormation diese Code-Artefakte wieder herunter, um sie im Stack zusammen mit den Lambda-Funktionen entsprechend zu provisionieren. Hier wäre es möglich, mit dem Argument `--parameter-overrides` die im Template definierten Parameter mit einem Wert zu belegen bzw. zu überschreiben, falls ein Standardwert gesetzt wurde.

Ein Beispiel für die Definition einer Lambda-Funktion zeigt Listing 5.2, welches vier wichtige Bereiche unter der „Properties"-Angabe aufzeigt: der erste Bereich enthält Angaben zum Typ und Artefakt der Lambda-Funktion mit *Handler*, *Runtime*, *Timeout* und *CodeUri*; der zweite Bereich definiert mit *Policies* die Zugriffsrechte der Funktion, welche in diesem Fall auf die Tenant-Tabelle zugreifen und die Lambda-Funktion OutgoingProcessor aufrufen darf; der dritte

Listing 5.2 YAML-Deklaration der Lambda-Funktion zum Empfang von Trello Webhooks.

```
BoardUpdateWebhook:
  Type: AWS::Serverless::Function
  Properties:
    Handler: index.updateWebhook
    Runtime: nodejs6.10
    Timeout: 10
    CodeUri: ../backend/target
    Policies:
      - AWSLambdaBasicExecutionRole
      - Version: '2012-10-17'
        Statement:
         - Effect: "Allow"
           Action: "dynamodb:*"
           Resource:
            - !Sub arn:aws:dynamodb:${region}:${accountId}:table/${TenantTable}
            - !Sub arn:aws:dynamodb:${region}:${accountId}:table/${TenantTable}/index/*
            # further tables...
         - Effect: "Allow"
           Action: "lambda:invokeFunction"
           Resource:
            - !GetAtt [OutgoingProcessor, Arn]
    Environment:
      Variables:
        TENANT_TABLE: !Ref AddonTenantTable
        # further variables...
    Events:
      UpdateWebhookResource:
        Type: Api
          Properties:
            Path: /api/webhooks/update
            Method: post
```

Bereich ermöglicht mit *Environment* das Setzen von Umgebungsvariablen wie z.B. den Namen der Tenant-Tabelle, indem auf sie referenziert wird; der vierte Bereich definiert mit *Events* wie die Lambda-Funktion aufgerufen wird. In diesem Fall wird eine einfache HTTP-Schnittstelle angegeben.

Wie zu sehen ist, lautet die Angabe zur *Runtime* der Lambda-Funktion *nodejs6.10* für NodeJS, allerdings existieren auch Funktionen mit der Angabe *java8* für Java. Insgesamt werden ein NodeJS- und ein Java-Artefakt zur Provisionierung der Lambda-Funktionen verwendet. Beide Artefakte enthalten für verschiedene Lambda-Funktionen den jeweiligen Programm-Code. Eine korrekte Referenzierung auf die entsprechenden Funktionen innerhalb des Artefakts wird durch *Handler* angegeben. Der Grund für die Entscheidung zur Verwendung eines einzigen Code-Artefakts pro Programmiersprache ist die große Menge an gemeinsam genutzten Programm-Code.

Für die Aufteilung in NodeJS und Java gibt es zwei Gründe: Zum einen können NodeJS-Funktionen schneller gestartet werden und eignen sich daher besser für Funktionen, die vom Frontend aufgerufen werden (vgl. dazu auch Kapitel 6). Zum anderen fordert die Anforderung NFA4, dass eine Erweiterung zu JIRA möglich sein soll. Da das bestehende Backbone Issue Sync System in Java implementiert ist, wird die Synchronisierung ebenfalls in Java umgesetzt. Deshalb werden bestimmte Komponenten wiederverwendet und erweitert, welche im Folgenden näher erläutert werden:

REST API Client Das bisherige System besitzt einen API Client, welcher mit der JIRA REST API kommunizieren kann. Dieser wird so angepasst, dass nun sowohl die JIRA als auch Trello REST API angefragt werden können. Abbildung 8.1 und Abbildung 8.2 veranschaulichen die Umstellung des API Clients anhand eines Klassendiagramms. Es existiert eine gemeinsame Oberklasse, die Basis-Methoden wie *get()* oder *post()* bereitstellt. Die jeweiligen Implementierungen müssen lediglich die *executeRequest()*-Methode überschreiben, welche die individuelle Authentifizierung (z.B. JWT oder OAuth) der REST API behandelt. Mit diesem Aufbau können theoretisch auch noch weitere Schnittstellen angebunden werden.

Integration Config Das Datenmodell zur Speicherung einer Integration umfasst Daten der jeweiligen Tenants, die zu synchronisierenden Projekte bzw. Boards sowie Synchronisations-Einstellungen. Die generelle Konfiguration vom Anwendungsfall einer JIRA-zu-JIRA gleicht einer Trello-zu-Trello Synchronisation in vielen Punkten. Deshalb wird die Konfiguration um Trello-spezifische Daten erweitert werden.

Synchronisationsbeziehung Für Trello Objekte muss ebenfalls eine Synchronisationsbeziehung gespeichert werden, um Änderungen an einem Objekt dem korrekten Objekt im anderen Trello Board zuordnen zu können. Die Umsetzung dieser Beziehung wird in einer vereinfachten Form wiederverwendet.

Wie eine solche Konfiguration erstellt wird und aufgebaut ist, wird im folgenden Abschnitt näher erläutert. Danach wird in Abschnitt 5.2 die Umsetzung der Synchronisationsbeziehung beschrieben.

5.1.2 Konfiguration einer Integration

Abbildung 8.3 im Anhang veranschaulicht den Ablauf zur Erstellung einer Integrationsanfrage zwischen zwei Trello Boards. Die Anfrage zur Integration wird hier auch als „Sync Request“ bezeichnet. Ein Benutzer kann in das Formular einen Namen der Integration, den Trello-Benutzernamen des Eigentümers des anderen Boards sowie eine Konfiguration im JSON-Format angeben. Das Format einer solchen Konfiguration ist in Listing 5.3 beispielhaft dargestellt. Dieses enthält Meta-Angaben zur Integration sowie die Einträge *firstConnector* und *secondConnector*, welche jeweils eine Board-Konfiguration der Integration definieren und

Listing 5.3 Beispiel einer Konfiguration für eine Integration im JSON-Format.

```
{
  "integrationKey": "INT-322dc3b8915f730a2a189bb2e7f4ff64ea28a546",
  "summary": "Integration between Board A and Board B.",
  "firstConnector": {
    "connectorKey": "CON-a92edf659d1070a109295a3226768a2744120c87",
    "projectKey": "5994206a4b23b450611850d8",
    "connection": {
      "application": {
        "user": "shesse_test"
      }
    },
    "configuration": {
      "trelloSettings": {
        "syncLists": [
          "Public List"
        ],
        "syncPartnerLists": true
      }
    }
  },
  "secondConnector": {
    /* ... */
  }
}
```

beispielsweise Einstellungen für den Outgoing und Incoming Processor pro Board enthalten können[1]. In diesem Beispiel definiert *firstConnector* die Konfiguration für Board A und *secondConnector* die Konfiguration für Board B. Jeder Benutzer hat jedoch nur die Möglichkeit, seinen eigenen Connector für sein Board zu konfigurieren. Für den Prototyp sind zwei Einstellungsmöglichkeiten pro Connector vorhanden: Zum einen können die Namen der zu synchronisierenden Listen des Boards definiert werden[2]. Zum anderen kann festgelegt werden, ob die Änderungen an Listen des anderen Boards auch wieder zurück synchronisiert werden sollen. Das Beispiel zeigt für den *firstConnector*, dass nur die Liste *Public List* in Board B synchronisiert werden soll und die Änderungen an den Listen, welche aus Board B in das Board A synchronisiert werden, auch wieder zurück synchronisiert werden sollen.

Lehnt der andere Benutzer die Integration ab (vgl. Abbildung 8.3c), so wird sie direkt gelöscht und es findet keine Synchronisation statt. Akzeptiert der andere Benutzer sie hingegen, wird sie umgehend gestartet. Wie bereits im vorherigen Kapitel erwähnt, ist dazu eine initiale Synchronisation notwendig. Der Ablauf einer solchen initialen Synchronisation ist in Ab-

[1]Da laut den Anforderungen keine expliziten Field Mappings zur Verarbeitung von Daten notwendig sind, ist die Konfiguration sehr einfach gehalten. Prinzipiell sind hier jedoch Erweiterungen des Formats möglich.

[2]Dies ist zwar nicht eindeutig, da der Name einer Liste pro Board häufiger vorkommen kann, jedoch soll es auch nur demonstrieren, inwiefern man eine Synchronisation filtern kann.

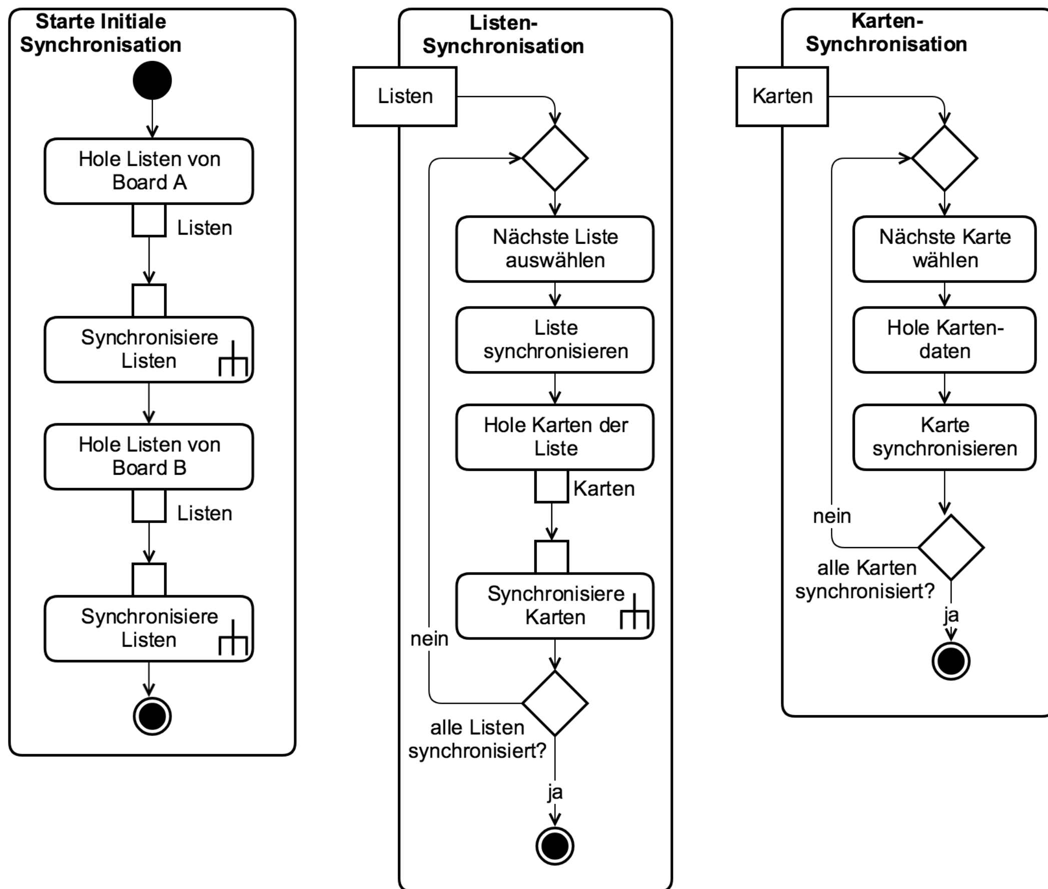


Abbildung 5.1: Aktivitätsdiagramm zur initialen Synchronisation.

bildung 5.1 dargestellt. Es müssen dazu alle Listen eines Boards von der Trello REST API abgefragt werden, sodass diese und ihre jeweiligen Karten synchronisiert werden können. *Invoke list sync* und *Invoke card sync* sind hier die entscheidenden Aktionen, welche die Synchronisierungen der existierenden Objekte asynchron starten. Im folgenden Kapitel wird dieser Synchronisationsprozess deshalb genauer vorgestellt.

5.2 Synchronisationsprozess

Der Synchronisationsprozess umfasst den gesamten Ablauf vom Outgoing bis zum Incoming Processor. Damit ein Objekt eines Trello Boards in ein anderes Board synchronisiert werden kann, muss eine Synchronisationsbeziehung erstellt werden, denn technisch gesehen sind es unterschiedliche Daten in den Boards. Das heißt, dass die Objekte anhand einer eindeutiger Identifikationsnummern miteinander verknüpft werden. Dies ist in Abbildung 5.2 genauer dargestellt ist. Ein „SyncInfo“-Objekt ist ein Repräsentant eines Trello Objektes, z.B.

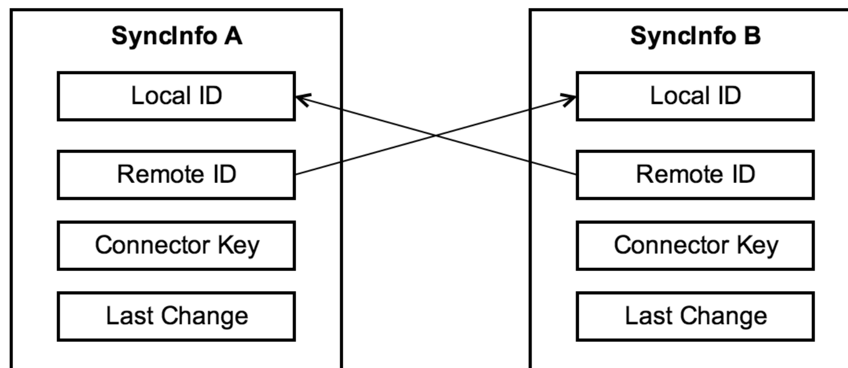


Abbildung 5.2: Die Synchronisationsbeziehung zwischen zwei Objekten aus zwei Trello Boards A und B.

eine Liste oder Karte. Es wird innerhalb des zu entwickelnden Trello Sync Systems in der Synchronisations-Tabelle gespeichert. Die *Local ID* entspricht dabei der ID des Objekts im Trello System. Zur Verknüpfung der Objekte in Board A und B fungiert die *Remote ID* als eine Art Zeiger auf das jeweils andere SyncInfo-Objekt. Des Weiteren besitzt ein SyncInfo-Objekt noch einen Schlüssel, den *Connector Key*, zur Referenzierung des zugehörigen Connectors. Ein Zeitstempel, der *Last Change*, enthält den Zeitpunkt der letzten, verarbeiteten Änderung. Neben den abgebildeten Attributen werden noch Sequenz-Informationen gespeichert, damit der Fehler *IE2 - Ungültige Sequenz* erkannt werden kann.

Der Vorteil eines solchen Aufbaus ist die Erweiterbarkeit des Systems: Eine ID kann theoretisch beliebig gewählt und auch aus mehreren Attributen generiert werden, falls dies notwendig ist. Für die in NFA4 geforderte Erweiterung zu JIRA könnte die ID auch durch den Issue Key (die ID eines Vorgangs) gesetzt werden. Im Falle eines eintreffenden Webhooks im Outgoing Processor des Boards A wird die ID des Trello Objekts dem SyncInfo-Objekt A zugeordnet. Damit der Incoming Processor des Boards B die Änderungen auch dem SyncInfo-Objekt B zuordnen kann, wird dies anhand der *Remote ID* gesucht. Deshalb wird die ID des SyncInfo-Objekts A in die Synchronisationsnachricht eingetragen. Dies ist die in Abschnitt 4.2.3 bereits erwähnte Nachricht, welche zwischen Outgoing und Incoming Processor ausgetauscht wird und die Änderungsdaten einer Synchronisation übermittelt.

Eine Nachricht enthält folgende wichtige Elemente: **feste Attribute**, z.B. den *ConnectorKey* des Connectors, der die Nachricht erstellt und versendet hat; **Header**- bzw. **Meta-Daten**, die z.B. den Typ des Trello Objekts wie Liste oder Karte und die Änderungsart wie *Karte hinzufügen* angeben; **Änderungsdaten**, die z.B. den Namen einer Karte enthalten. Die Angaben der Meta- und Änderungsdaten sind sehr flexibel, weil sie einer Schlüssel-Wert-Datenstruktur folgen, wodurch sie sich auch gut serialisieren lassen. Damit die Nachricht zwischen dem Outgoing und Incoming Processor ausgetauscht werden kann, wird sie als JSON-String serialisiert. Falls in Zukunft weitere Systeme unterstützt werden sollen, muss lediglich die Verarbeitung des Nachrichteninhalts angepasst werden.

Abbildung 8.4 und Abbildung 8.5 im Anhang stellen den Ablauf der Aufbereitung und Verarbeitung der Nachricht im Outgoing bzw. Incoming Processor dar. Wie dort zu sehen ist, werden im Outgoing Processor zuerst bestimmte Daten aus den Datenbanktabellen und Trello zusammengetragen, sodass daraufhin entschieden werden kann, ob der Webhook bzw. dessen Daten synchronisiert werden sollen. Diese Entscheidung kann anhand des SyncInfo-Objekts getroffen werden, indem der Zeitstempel des Attributs *Last Change* mit dem Zeitstempel des Webhooks verglichen wird, welcher dem Zeitpunkt der letzten Änderung entspricht. Weil Trello keine Möglichkeit bereitstellt, dynamische Attribute an einem Objekt zu setzen, ist dies die einzige Möglichkeit, zu erkennen, ob eine Änderung neu ist oder nicht[3]. Dies ist jedoch nur möglich, indem im letzten Schritt des Incoming Processors (vgl. Abbildung 8.5) das entsprechende SyncInfo-Objekt mit dem Zeitstempel der letzten Änderung aktualisiert wird. Der Zeitstempel kann aus der Antwort der Trello REST API ausgelesen werden. Bei einer Erweiterung zu anderen Systemen wäre hier auch eine differenziertere Überprüfung möglich. Zum Beispiel könnten individuelle Angaben im Webhook geprüft werden, an denen erkannt werden kann, dass Änderungen vom Trello Sync System ausgelöst wurden.

Nach der Überprüfung, ob ein Webhook verarbeitet werden soll, folgen die eigentlichen Verarbeitungsschritte. Abhängig vom Typ des Objekts und der Änderung werden unterschiedliche Schritte ausgeführt.

- **Objekt wurde erstellt:** Alle unterstützten und vorhandenen Datenfelder werden der Nachricht hinzugefügt. Am Beispiel einer neuen Karte: Name, ID der Liste, Beschreibung, Position innerhalb der Liste.
- **Objekt wurde aktualisiert:** Nur die geänderten Datenfelder werden der Nachricht hinzugefügt. Am Beispiel einer Karte: Nur der Name, falls dieser sich geändert hat.
- **Objekt wurde gelöscht:** Keine Datenfelder werden der Nachricht hinzugefügt, sondern lediglich ein Meta-Feld, das die Löschung des Objekts im anderen Trello Board signalisiert.

Hingegen werden im Incoming Processor folgende Verarbeitungsschritte ausgeführt:

- **Objekt wurde erstellt:** Eine HTTP POST Anfrage wird mit den vorhandenen Daten aus der Synchronisationsnachricht an die Trello REST API gesendet.
- **Objekt wurde aktualisiert:** Eine HTTP PUT Anfrage wird mit den vorhandenen Daten aus der Synchronisationsnachricht an die Trello REST API gesendet.
- **Objekt wurde gelöscht:** Eine HTTP DELETE Anfrage wird an die Trello REST API gesendet.

[3]Es ist jedoch möglich, dass ein Webhook statische Daten enthält, welche bei der Registrierung übergeben werden müssen.

Sowohl der Outgoing als auch der Incoming Processor sind als statuslose Funktionen implementiert, was dem in Abschnitt 3.2.1 vorgestelltem Konzept entspricht. Deshalb müssen jedes Mal die entsprechenden Daten aus verschiedenen Quellen zusammengetragen werden. Jedoch kann dadurch auch der Vorteil der Skalierung genutzt werden: Mithilfe von AWS Lambda sowie den pro Objekt gespeicherten SyncInfo-Objekten können gleichzeitig mehrere Synchronisationen über unterschiedliche Trello Boards hinweg durchgeführt werden.

Unterstützte Synchronisationsfälle

Die folgende Übersicht fasst die unterstützten Synchronisationsfälle kurz zusammen:

- **Liste**
  - Neue Liste wird hinzugefügt
  - Listenname wird aktualisiert
- **Karte**
  - Neue Karte wird einer Liste hinzugefügt
  - Karte wird aktualisiert (Name, Beschreibung, zugehörige Liste)
  - Karte wird gelöscht
  - Label wird der Karte hinzugefügt
  - Label wird von Karte entfernt
- **Kommentar**
  - Kommentar wird einer Karte hinzugefügt
  - Text des Kommentars wird aktualisiert
  - Kommentar wird von Karte entfernt
- **Anhang**
  - Anhang wird einer Karte hinzugefügt
  - Anhang wird aktualisiert
  - Anhang wird von Karte entfernt

Neben den bereits vorgestellten Fällen, die unterstützt werden, existieren einige Sonderfälle, welche bei der Synchronisation beachtet werden müssen:

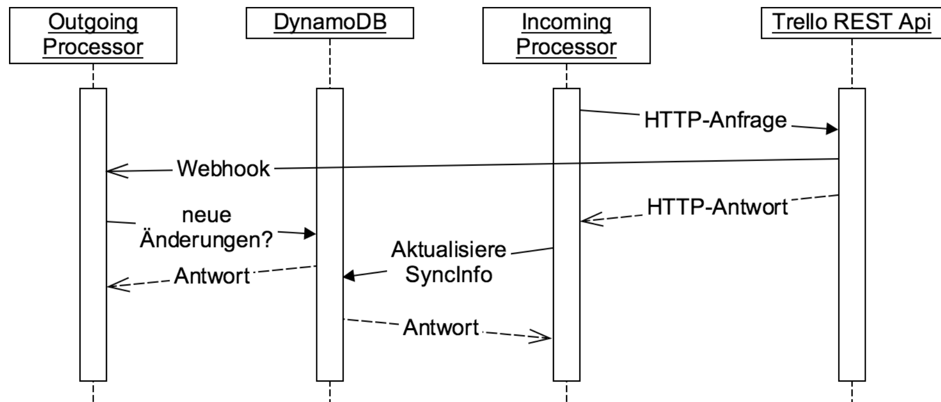


Abbildung 5.3: Sequenzdiagramm zur Visualisierung der nicht erkannten Aktualisierungen.

Karte wird auf nicht-synchronisierte Liste verschoben: Wie in Abschnitt 5.1.2 beschrieben, kann eine Menge an Listen konfiguriert werden, die synchronisiert werden sollen. Falls eine Karte auf eine nicht-synchronisierte Liste verschoben wird, werden alle weiteren Änderungen (sowohl einzelne Felder wie Name und Beschreibung als auch Kommentare und Anhänge, die der Karte hinzugefügt werden) so lange ignoriert, bis sie wieder auf eine Liste verschoben wird, die synchronisiert wird.

Integration wird beendet: Falls ein Benutzer die Integration beendet, werden keine Synchronisationen mehr dafür durchgeführt. Eine spätere Fortführung ist nicht möglich, da die SyncInfo-Objekte mit dem Connector einer Integration verknüpft sind. Es kann jedoch eine neue Integration gestartet werden.

5.2.1 Item Locking

DynamoDB verwendet zur Speicherung der Daten Eventual Consistency (vgl. Abschnitt 3.2.2). Dies kann zu Inkonsistenzen führen, wenn innerhalb des DynamoDB-Services noch nicht alle Knoten die aktualisierten Daten erhalten haben. Abbildung 5.3 veranschaulicht ein daraus resultierendes Problem im Bezug auf Webhooks und den Daten eines SyncInfo-Objekts. So können Daten eines Synchronisationsdurchlaufs noch nicht in der SyncInfo-Tabelle eingetragen sein, obwohl sie für den nächsten eingehenden Webhook abgefragt werden müssen. Dies ist beispielsweise notwendig, um zu entscheiden, ob der Webhook verarbeitet oder ignoriert werden soll. Im Prinzip heißt das, dass nach einer Änderung in Trello der dadurch ausgelöste Webhook schneller versendet wird als eine Änderung in DynamoDB geschrieben werden kann. Auch ein streng konsistenter Lesevorgang kann dieses Problem nicht verhindern, wenn die zu speichernden Daten noch gar nicht an DynamoDB gesendet wurden.

Aus diesem Grund wurde ein „Item Locking“ eingeführt, welches einen SyncInfo-Eintrag in der Datenbank solange sperrt, bis er von einem Prozess wieder freigegeben wird. Dieser Ablauf ist in einem Sequenzdiagramm in Abbildung 5.4 dargestellt. Ein Prozess A (z.B. der

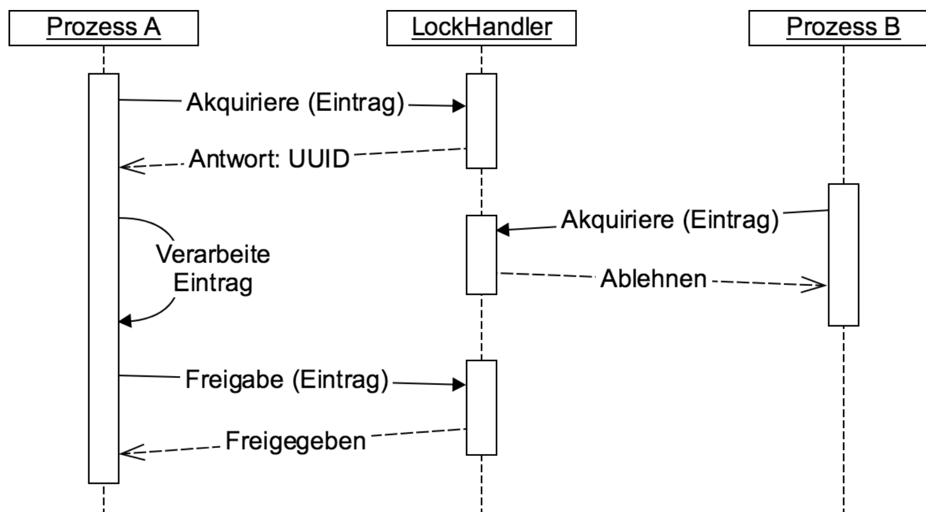


Abbildung 5.4: Sequenzdiagramm zur Visualisierung des Item Lockings.

Incoming Processor) fragt einen LockHandler, ob ein Datenbankeintrag exklusiv gesperrt werden kann. So kann nur dieser Prozess Änderungen am Datenbankeintrag durchführen. Dafür erhält Prozess A vom LockHandler eine *uuid*, die er jeder Datenbankänderung anhängen muss, um sich zu autorisieren. Wenn der Prozess A die Verarbeitung abgeschlossen hat, kann er den Datenbankeintrag wieder freigeben. Falls ein weiterer Prozess B (z.B. der Outgoing Processor) zu der gleichen Zeit den Datenbankeintrag verwenden möchte, teilt der LockHandler dem Prozess mit, dass dies zurzeit nicht möglich ist. Der Prozess B kann dann zu einem späteren Zeitpunkt erneut versuchen (z.B. nach zehn Sekunden), den Datenbankeintrag für seine Zwecke zu sperren. Wendet man diesen Ansatz auf das vorgestellte Problem an, so muss der Outgoing Processor unter Umständen für ein paar (Milli-)Sekunden warten. Erst, wenn der Datenbankeintrag freigegeben wird, kann der Outgoing Processor abfragen, ob der Webhook tatsächlich neue Änderungen enthält. Dies verhindert, dass veraltete Informationen eines SyncInfo-Eintrags gelesen werden.

Der Ansatz stellt sicher, dass ein Datenbankeintrag nicht fälschlicherweise geändert wird oder veraltete Informationen gelesen werden. Allerdings muss auch beachtet werden, dass Prozesse nicht immer fehlerfrei ablaufen. Dies kann verschiedene Gründe haben: Ein Prozess stürzt ab, die Java Virtual Machine (JVM) startet den Garbage Collector und unterbricht die Ausführung des Prozesses oder ein unerwarteter (Netzwerk-)Fehler tritt auf, sei es während der Verarbeitung oder während der Freigabe des Datenbankeintrags. Es ist also durchaus möglich, dass ein Datenbankeintrag von einem Prozess nie wieder freigegeben wird. DynamoDB bietet jedoch keine eigene Möglichkeit, um beispielsweise falsche Daten bzw. Zustände aufzuräumen. Deshalb stellt der Service *ResetLockService* sicher, dass eine Sperre eines Datenbankeintrags nach einer gewissen Zeit wieder gelöscht wird, falls das bis zu dem Zeitpunkt noch nicht geschehen ist. Dieser Ablauf wird in Abbildung 5.5 beschrieben. Der Service *ResetLockService* wurde mithilfe von *AWS Step Functions* umgesetzt. AWS Step Functions bietet die Orches-

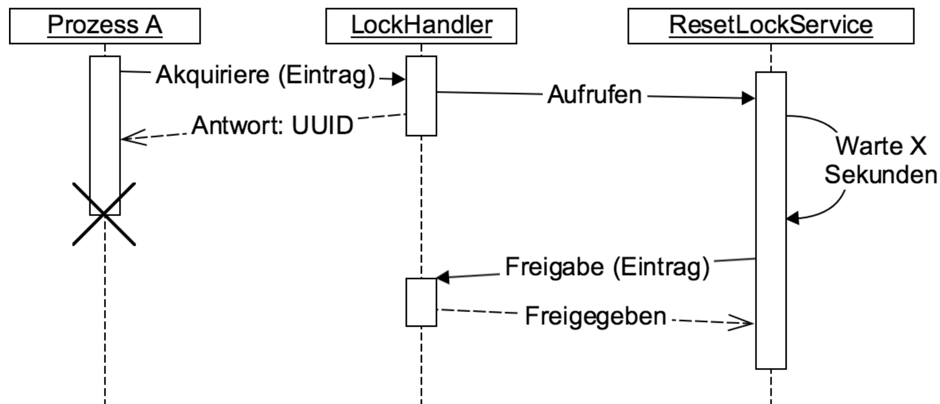


Abbildung 5.5: Sequenzdiagramm zum Zurücksetzen eines Item Locks, falls der anfragende Prozess abbricht.

trierung einzelner Microservices, wie Lambda-Funktionen, in einem Workflow an [Ama17l]. Mithilfe einer JSON-Notation wird der Workflow definiert. Dieser führt nach 30 Sekunden eine Lambda-Funktion aus, welche - falls noch nötig - die Sperre aufhebt[4]. Damit nicht die falsche Sperre aufgehoben wird, erhält der Service als Eingabedaten die *uuid*.

Im Endeffekt bedeutet dies, dass ein Prozess einen exklusiven Lese- und Schreibzugriff zu einem Datenbankeintrag erhält. Durch den Einsatz des ResetLockServices kann auch die Wahrscheinlichkeit für einen Fehler während des Item Lockings als gering angesehen werden, weil dafür mehrere Komponenten vollständig ausfallen müssten. Alternativ präsentieren Patrikalis und Slutsker [PS17] einen ähnlichen Ansatz, um verteiltes Locking mit DynamoDB zu implementieren.

Aktualisierte System-Architektur

Abbildung 5.6 zeigt die aktualisierte System-Architektur basierend auf den Anpassungen durch das Item Locking. Wie zu sehen ist, wird aus den Lambda-Funktionen im Lambda System der Step Functions Service aufgerufen. Dieser ruft gegebenenfalls eine weitere Lambda-Funktion auf, um einen Datenbankeintrag wieder freizugeben. Ansonsten sind keine weiteren Veränderungen an der System-Architektur notwendig.

[4]Die 30 Sekunden Wartezeit sind unter Umständen nicht ausreichend. Als Verbesserung kann eine Wartezeit von 300 Sekunden definiert werden. Diese entspricht der maximalen Laufzeit einer Lambda-Funktion.

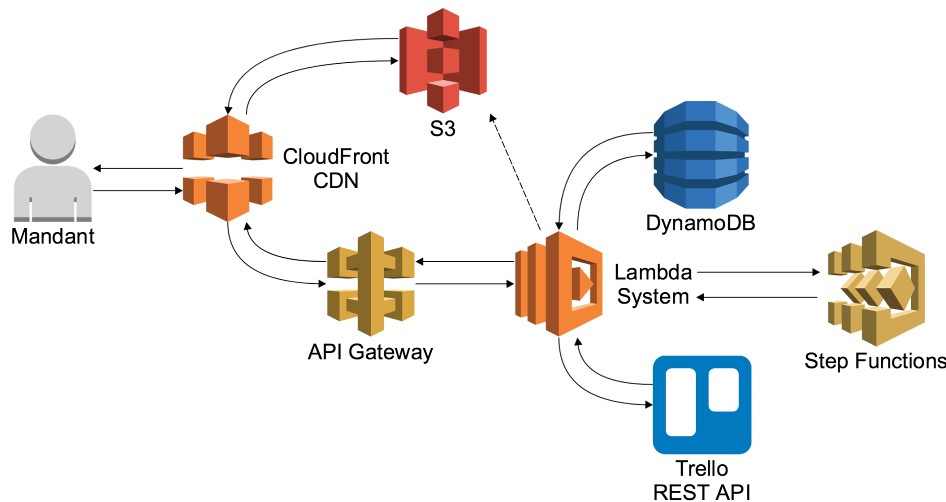


Abbildung 5.6: Die aktualisierte System-Architektur.

5.2.2 Mandantenfähigkeit

Eine weitere zentrale Anforderung ist die Umsetzung der Mandantenfähigkeit aus NFA5. In Abschnitt 4.2.4 wurde bereits erwähnt, dass die Tenant ID eine Zuordnung der Tenants ermöglicht. Deshalb muss sie bei jedem Aufruf einer Lambda-Funktion und jeder Synchronisation vorhanden sein. Dies wird wie folgt umgesetzt: Während der Registrierung eines Benutzers im Trello Sync System wird eine Tenant ID generiert, welche einen Benutzer identifiziert und mit den Login-Daten verknüpft ist. Die Daten werden in der Tenant-Tabelle in DynamoDB gespeichert. Wenn sich ein Benutzer im System anmeldet, wird in seinem Browser ein JSON Web Token (JWT) gespeichert, welches unter anderem auch die Tenant ID enthält. Das JWT wird daraufhin zu jeder HTTP-Anfrage an das Trello Sync System angehängt, sodass die Lambda-Funktionen erkennen können, welchem Benutzer die Anfrage zuzuordnen ist. Falls ein Benutzer eine neue Integration erstellt, wird diese Integration ebenfalls mit der Tenant ID verknüpft und die Daten in der Integrations-Tabelle gespeichert. Des Weiteren wird bei der Registrierung der Webhooks die Tenant ID auch als statisches Attribut angegeben, welches jedem ankommenden Webhook angehängt werden soll. So ist eine Zuordnung des Webhook zu einer Integration bei jeder Synchronisierung möglich. Durch die Verknüpfung der Daten und dem entsprechenden Programm-Code wird sichergestellt, dass eine HTTP-Anfrage nur die Daten als Antwort enthält, welche mit der mitgelieferten Tenant ID verknüpft sind. Zusätzlich werden HTTP-Anfragen validiert, indem entweder die JWT-Daten (für Anfragen aus dem Frontend des Systems) oder die Webhook-Signatur überprüft werden[5].

Neben der vorgestellten Umsetzung zur Speicherung aller Daten in einer Tabelle (z.B. alle Webhook-Daten in einer einzigen Webhook-Tabelle), wäre es alternativ auch möglich, pro

[5]Trello signiert alle Webhooks mit dem API-Key des Systems [Tre17c].

Tenant eine Menge von Tabellen (Webhooks, Integrationen, Synchronisationen) anzulegen. Dies hätte den Vorteil, dass die Daten pro Tenant separat gespeichert und Daten weniger leicht vertauscht werden können. Allerdings erhöht es die Sicherheit nur marginal. Denn trotz dieser Umsetzung muss im Programm-Code sichergestellt sein, dass eben nicht der korrekte Tabellen-Eintrag, sondern die korrekte Tabelle verwendet wird. Das Problem wird daher nur verlagert. Zusätzlich würde dieser Ansatz noch weitere Nachteile mit sich bringen: Durch das dynamische Erzeugen der Tabellen können diese nicht mehr über das CloudFormation-Template verwaltet und aktualisiert werden. Das bedeutet, dass zusätzliche Verwaltungsmöglichkeiten in Anspruch genommen werden müssen, was wiederum Arbeit und Kosten erzeugt.

Im Bezug auf Datensicherheit ist noch zu erwähnen, dass in den Datenbank-Tabellen nur systemrelevante Daten (und - abgesehen von den Login-Daten - keine personenbezogenen Daten) gespeichert werden. Falls es einem Angreifer gelingen sollte, Daten aus diesen Tabellen zu stehlen, so sind diese in der jetzigen Implementierung nicht verschlüsselt. Mit den Daten aus der Tenant-Tabelle könnten sogar weitere Anfragen an die Trello REST API gesendet werden, sodass weitere Daten gestohlen werden. Es gibt seitens AWS auch die Möglichkeit, Daten in DynamoDB zu verschlüsseln, allerdings wurde in dem Prototypen darauf verzichtet, da dies ein potentieller Punkt zur Erweiterung des Systems ist und die eingesetzte Architektur dadurch nicht oder nur minimal verändert werden müsste.

5.2.3 Service Limitierungen

Das entwickelte System besitzt zwei Funktionen, welche unter Umständen sehr viele Daten in sehr kurzer Zeit verwalten müssen: Zum einen der *Scheduled Job*, welche täglich ausgeführt wird, um verpasste Änderungen zu aktualisieren und zum anderen die initiale Synchronisation nach dem Starten einer Integration. Falls die Trello Boards sehr viele Listen und Karten enthalten, sind viele Aufrufe an die Datenbanktabellen und Trello REST API notwendig, damit Daten abgeglichen oder abgefragt werden können. Allerdings kann dies bei 5-10 Aufrufen pro Synchronisationsdurchlauf schnell zum Erreichen von Limitierungen führen, wenn auch noch mehrere Karten und Listen gleichzeitig synchronisiert werden[6]. Die Trello REST API erlaubt beispielsweise lediglich 100 Anfragen pro zehn Sekunden. DynamoDB hingegen verwendet ein etwas komplizierteres Berechnungsmodell: Pro Datenbanktabelle kann im CloudFormation Template je ein Limit für Lese- und Schreibvorgänge festgelegt werden, welche auch als *Read Capacity Units* und *Write Capacity Units* bezeichnet werden [Ama17n]. Die genaue Bedeutung lautet wie folgt:

Read Capacity Unit umfasst einen streng konsistenten (engl. strongly consistent) Lesevorgang oder zwei eventuell konsistente (engl. eventually consistent) Lesevorgänge pro Sekunde für einen Datenbankeintrag mit einer Größe von maximal 4 KB. Falls mehr

[6]Die Anzahl von 5-10 Aufrufen ist eine vorsichtige Schätzung und variiert auch je nachdem was synchronisiert wird.

Zugriffe pro Sekunde benötigt werden, muss die Anzahl der Read Capacity Units erhöht werden. Zum Beispiel müssen bei 40 streng konsistenten Lesevorgängen pro Sekunde für einen Eintrag von 4 KB 40 Read Capacity Units angegeben werden. Bei einem Eintrag von 6 KB Größe müssen hingegen 80 Read Capacity Units angegeben werden (die Größe eines Eintrags wird auf die nächste 4 KB Einheit gerundet, hier also 8 KB).

Write Capacity Unit umfasst einen Schreibvorgang pro Sekunde für einen Datenbankeintrag mit einer Größe von maximal 1 KB. Falls mehr Zugriffe pro Sekunde benötigt werden, muss die Anzahl der Write Capacity Units erhöht werden. Zum Beispiel müssen bei 20 Schreibvorgängen pro Sekunde für einen Eintrag von 1 KB 20 Write Capacity Units angegeben werden. Auch hier wird die benötigte Größe entsprechend aufgerundet.

Des Weiteren erlaubt DynamoDB die kurzfristige Überschreitung eines Limits für einen gewissen Zeitraum, um unerwartete Lastspitzen zu behandeln [Ama17n]. Jede weitere Überschreitung wird allerdings vermieden, was bedeutet, dass ein Fehler zurückgegeben wird. Deshalb werden etwaige Überschreitungen der Limits zurzeit als Fehler erkannt und entsprechend durch die Fehlerbehandlung verarbeitet. Allerdings ist dies keine optimale Lösung, weil lediglich reagiert wird anstatt das Erreichen der Limits aktiv zu vermeiden. Eine Möglichkeit wäre daher die Verwendung der Auto-Skalierung für DynamoDB, sodass bei einer erhöhten Last automatisch weitere Lese- und/oder Schreib-Kapazitäten der Datenbanktabelle hinzugefügt werden [Ama17m]. Jedoch würde dies nur einen Teil des Problems lösen, da die Trello REST API keine solche Skalierungsmöglichkeit anbietet. Allerdings wäre es hier möglich, eine Ausnahme der Limitierung zu beantragen oder die Limitierung zu erhöhen.

5.2.4 Vergleich mit bestehendem System

Nachdem die Cloud-Architektur und ihre Implementierungsdetails nun vorgestellt wurde, soll im Folgenden kurz darauf eingegangen werden, inwiefern sich das neue System vom bestehenden System unterscheidet und vor allem, welche Vorteile sich daraus ergeben. Dies betrifft insbesondere das neue Vorgehen zum Deployment des Systems sowie die auch damit zusammenhängenden Kosten.

Das Deployment des neuen Systems dauert zurzeit ungefähr fünf bis zehn Minuten, abhängig davon wie viele Änderungen an der Infrastruktur gemacht wurden. Damit dauert es deutlich länger als ein Deployment des bestehenden Backbone Systems auf einem JIRA System, welches in der Regel innerhalb von 30 Sekunden aktualisiert ist. Dies liegt daran, dass die neue Cloud-Architektur deutlich mehr Abhängigkeiten zu weiteren Services besitzt. Abhängige Services müssen unter Umständen ebenfalls aktualisiert werden, sodass daraus ein komplexeres System resultiert. Wenn jedoch die gesamte Auslieferungsdauer einer System-Aktualisierung betrachtet wird, so hat sich dies mit der Cloud-Architektur deutlich verbessert. Der Grund dafür ist, dass nach einer Aktualisierung alle Kunden diese unmittelbar erhalten, weil sie über den Browser und das Internet direkt auf das aktualisierte System zugreifen. Im Fall des existierenden Backbone Systems muss jeder Kunde zuerst einmal benachrichtigt werden (häufig per E-Mail),

dass ein Update zur Verfügung steht. Danach kann das Update heruntergeladen und auf dem System installiert werden. Weil jedoch zwischen diesen beiden Schritten je nach Kunde häufig mehrere Stunden, Tage oder auch sogar Wochen liegen, dauert die gesamte Auslieferung einer Aktualisierung deutlich länger und deshalb bringt die Cloud-Architektur einen erheblichen Fortschritt.

Ein weiterer Vorteil des neuen Systems sind die geringeren Wartungs- und anfänglichen Kosten. So muss für eine Synchronisation im On-Premise-Kontext mindestens ein Server (besser zwei zur Erhöhung der Verfügbarkeit und besseren Lastverteilung) zur Verfügung stehen und entsprechend konfiguriert werden. Dazu muss Personal entweder angestellt oder eingekauft werden, um den Server regelmäßig zu warten und das Betriebssystem sowie das JIRA und Backbone System darauf zu aktualisieren. Zusätzlich ist die Installation der Software an hohe Lizenzgebühren gebunden, welche in bestimmten Abständen bei einer neuen Version wiederkehren. Eine Synchronisation im On-Demand-Kontext mithilfe der neuen Cloud-Architektur hingegen kann für einen Kunden ohne eigene Server und eigenes Personal erstellt werden. Stattdessen bezahlt ein Kunde pro Monat lediglich einen fixen Betrag für das Trello Sync System[7]. Insgesamt gesehen werden deshalb vor allem die anfänglichen Kosten sehr viel geringer ausfallen als im On-Premise-Kontext. Auch andere Arbeiten zeigen ähnliche Ergebnisse, welche die geringeren Kosten durch die Nutzung des Serverless Computings feststellen [VGO+17].

[7] Unter Umständen müssen für Trello (oder alternativ JIRA) noch monatliche Kosten betrachtet werden. Dies hängt jedoch bei Trello von den verwendeten Features und bei JIRA von der Anzahl der Benutzer ab.

6 Evaluation

Im Folgenden wird die Umsetzung der Anforderungen überprüft sowie Tests in Bezug auf das entwickelte System durchgeführt. Die Tests geben Aufschluss darüber, ob das neue System wie erwartet automatisch skaliert und wie sich das auf die Geschwindigkeit der Synchronisation auswirkt. Daraufhin wird noch weitere Kritik sowie potentielle Alternativen diskutiert und schließlich ein Fazit gezogen.

6.1 Bewertung

Die folgende Übersicht fasst zusammen, durch welche Maßnahmen die Anforderungen umgesetzt wurden.

Funktionale Anforderungen

FA1 - Synchronisation zweier Trello Boards Die Synchronisation wird mithilfe eines Eingangs- und Ausgangskanal pro Board umgesetzt.

FA2 - Field Mappings Die geänderten Daten von Listen, Karten, Kommentaren und Anhängen werden durch die Lambda-Funktionen Outgoing und Incoming Processor synchronisiert.

FA3 - Änderungen automatisiert synchronisieren Webhooks stellen den automatischen Empfang sicher und stoßen die Synchronisation an.

FA4 - Autorisierung Benutzer müssen ihren Trello-Account verknüpfen und eine Synchronisation muss anhand eines Sync Requests durch beide beteiligten Parteien bestätigt werden.

FA5 - Konfiguration der Synchronisation Die Einstellungen einer Synchronisation werden anhand einer JSON-Konfiguration pro Board angegeben.

FA6 - Fehlererkennung Fehler werden während der Synchronisation erkannt und auf einem S3-Bucket abgelegt, falls sie nicht automatisch behandelt werden können.

Nicht-Funktionale Anforderungen

NFA1 - Zuverlässigkeit Webhooks, eine automatische Fehlererkennung und ein zusätzlicher Scheduled Job sorgen für eine ausgereifte Zuverlässigkeit.

NFA2 - Serverless Architektur Auf Basis von AWS Lambda wurde die Serverless Architektur umgesetzt. Weitere Services wie DynamoDB und S3 unterstützen ebenfalls die Serverless Architektur, weil diese Systeme auch nicht manuell verwaltet werden müssen.

NFA3 - Infrastructure as Code Durch die Verwendung von CloudFormation können Deployments konsistent durchgeführt werden.

NFA4 - Erweiterung zu JIRA Eine Erweiterung zu JIRA ist mit einem überschaubaren Aufwand möglich, da JIRA ebenfalls Webhooks unterstützt und das Austauschformat flexibel genug für andere Daten ist.

NFA5 - Mandantenfähigkeit Mandanten und deren Daten können anhand ihrer Tenant ID jederzeit zugeordnet werden.

NFA6 - Entwicklungsprozess Der vorgestellte Entwicklungsprozess unterstützt eine automatische Provisionierung in der Cloud mithilfe eines CloudFormation Templates und verschiedener Entwicklungsstufen.

Insgesamt konnten alle Anforderungen umgesetzt werden. Die Verwendung von AWS Lambda sorgt dafür, dass keine Server verwaltet oder provisioniert werden müssen. Nichtsdestotrotz schränkt die Nutzung dieses Services die Flexibilität etwas ein, da vor allem ein Wechsel auf einen Service eines anderen Anbieters mit einigen Anpassungen verbunden wäre.

6.2 Validierung der Architektur

Im Folgenden wird die Cloud-Architektur validiert. Dafür wird zuerst der Testaufbau beschrieben. Anschließend werden die durchgeführten Tests und ihre Ergebnisse vorgestellt. Danach werden die Ergebnisse in einem separaten Abschnitt ausgewertet und interpretiert.

In den Tests wird zum einen die Performance eines „Kaltstarts“ einer Lambda-Funktion untersucht, was dem initialen Starten einer Lambda-Funktion entspricht. Im Zusammenhang dazu wird auch die Performance einer Lambda-Funktion untersucht, wenn sie bereits einmal verwendet wurde, also „aufgewärmt“ ist. Zum anderen wird auch die Dauer einer Synchronisation ausgewertet, ebenfalls mit „kalten“ und „aufgewärmten“ Lambda-Funktionen.

Wie bereits in Abschnitt 5.1.1 beschrieben, wurden für die Lambda-Funktionen insgesamt nur zwei Code-Artefakte verwendet: Eins für die NodeJS und eins für die Java Lambda-Funktionen. Die Größe der Artefakte liegt deshalb bei ca. 2,1 MB (NodeJS) bzw. 21,4 MB (Java). AWS rät jedoch dazu, die Artefakt-Größe möglichst gering zu halten. Deshalb soll mit den Tests auch untersucht werden, wie viel Einfluss die Größe auf die Ergebnisse hat.

6.2.1 Testaufbau

Der Testaufbau gestaltet sich daher wie folgt: Die ersten beiden Tests überprüfen die Skalierbarkeit und Antwortzeiten der Lambda-Funktionen, wobei der dritte Test die tatsächliche Dauer einer Synchronisation auswertet. Damit die Ergebnisse der ersten beiden Tests nicht von externen Services abhängen, wurden für die Tests keine der existierenden Lambda-Funktionen verwendet, sondern vier weitere implementiert. Sie ermöglichen die Unterscheidung eines Kaltstarts, da AWS Lambda dafür keine native Möglichkeit bereitstellt. Außerdem ist so eine bessere Messung der minimalen Antwortzeit möglich, da die Funktionen per HTTP-Anfrage aufrufbar sind. Im Anhang in Listing 8.2 befindet sich der Programm-Code für eine der Lambda-Funktionen. Abhängig davon, ob die Funktion zum ersten Mal oder bereits zum wiederholten Mal aufgerufen wurde, wird eine entsprechende HTTP-Antwort zurückgeliefert. Die anderen drei Lambda-Funktionen basieren auf dem gleichen Programm-Code.

Alle vier Lambda-Funktionen unterscheiden sich einerseits in ihrer Programmiersprache (NodeJS und Java) und andererseits, ob sie im bestehenden Code-Artefakt eingebettet sind oder nur mit minimalen Abhängigkeiten in ein extra Artefakt ausgelagert wurden. Mithilfe eines extra Code-Artefakts mit minimalen Abhängigkeiten lässt sich besser vergleichen, inwiefern die Größe des Artefakts die Ergebnisse beeinflusst. Daraus ergeben sich die vier Lambda-Funktionen „**node**“ (implementiert in NodeJS und in bestehendes NodeJS-Artefakt eingebettet, Artefakt-Größe: 2,1 MB), „**java**“ (implementiert in Java und in bestehendes Java-Artefakt eingebettet, Größe: 21,4 MB), „**simple-node**“ (implementiert in NodeJS und in einem separaten Artefakt ohne weitere Abhängigkeiten, Größe: 280 Bytes) und „**simple-java**“ (implementiert in Java und in einem separaten Java-Artefakt, Größe: 11 KB). Für den dritten Test sind die Test-Lambda-Funktionen nicht von Bedeutung, da dieser Test die tatsächlich entwickelten Lambda-Funktionen zur Synchronisation verwendet.

Weiterhin wurden die Test-Lambda-Funktionen im selben CloudFormation-Template wie die übrigen Lambda-Funktionen definiert und mit folgenden Eigenschaften provisioniert: die `MemorySize` wurde auf 312 und das `Timeout` auf 60 Sekunden für alle Test-Lambda-Funktionen gesetzt; als AWS Region des CloudFormation Stacks wurde `us-west-2` (Oregon, USA) ausgewählt. Durch die gleichen Werte kann eine bessere Vergleichbarkeit gewährleistet werden. Zusätzlich wurde für die Durchführung der Tests auf das Caching durch CloudFront verzichtet, um die tatsächlich benötigte Zeit besser vergleichen zu können.

6.2.2 Ziel der Tests

Damit die Ergebnisse bewertet werden können, müssen sie mit Erwartungswerten verglichen werden. Dafür wird angenommen, dass die HTTP-Anfragen an die Test-Lambda-Funktionen echte Anfragen durch einen Benutzer darstellen. Ein solcher Vergleich ist zwar schwierig, da die maximale Antwortzeit vom Kontext und der Wahrnehmung eines Benutzers abhängt. Daher existiert auch kein generell akzeptierter Wert, der die maximale Antwortzeit definiert. Nielsen [Nie93] setzt dafür ein Limit von maximal zehn Sekunden fest bevor der Benutzer eine Website

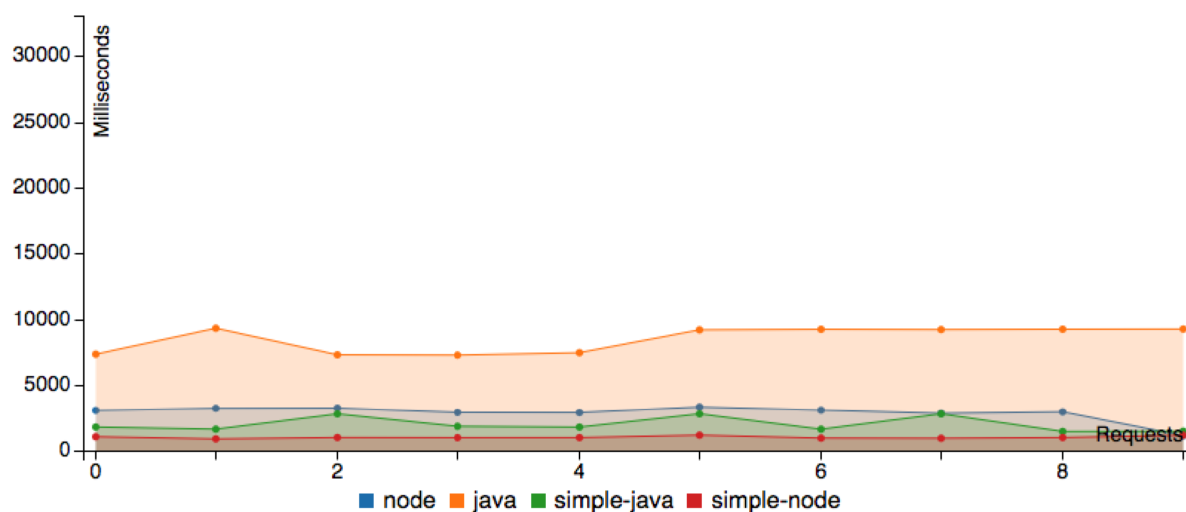


Abbildung 6.1: Antwortzeiten einer NodeJS und Java Lambda-Funktion bei zehn gleichzeitigen Anfragen, sodass für jede Anfrage eine neue Lambda-Instanz gestartet werden muss.

verlässt. Weiterhin wird eine Sekunde als ein Wert angesehen, der eine gute Akzeptanz durch den Benutzer verspricht. Ähnlich beschreibt es Mishunov [Mis15] mit maximal eine Sekunde für eine direkte Antwort. Für eine optimale Benutzererfahrung werden jedoch zwei bis fünf Sekunden angenommen. Deshalb wird für die Antwortzeiten der ersten beiden Tests festgelegt, dass der Durchschnittswert aller gemessenen Werte zwei Sekunden und jeder gemessene Wert fünf Sekunden nicht überschreiten darf. Die Werte des dritten Tests hingegen sollen einen Durchschnittswert von fünf Sekunden nicht überschreiten. Dieser Grenzwert liegt etwas höher, da der Synchronisationsprozess im Hintergrund durchgeführt wird und ein Benutzer keine direkte Antwort erwartet.

6.2.3 Skalierung und Performance der Kaltstarts

Der erste Test untersucht die Skalierung und Performance der Lambda-Funktionen, wenn neue Instanzen einer Funktion gestartet werden müssen. Das Ziel dieses Tests ist die Überprüfung, ob die Funktionen durch AWS Lambda so skaliert werden wie erwartet und inwiefern sich die Masse an Anfragen auf die Antwortzeiten auswirkt. Insbesondere für unerwartete Lastspitzen durch Trellos Webhooks ist diese Erkenntnis sehr relevant. Es ist in diesem Test deshalb notwendig, in einem möglichst kleinen Zeitraum alle HTTP-Anfragen gleichzeitig zu versenden. So wird AWS Lambda gezwungen, für jede Anfrage eine neue Lambda-Instanz zu starten und keine existierende Instanz wiederzuverwenden. Weiterhin soll der Test zeigen, wie effizient diese Skalierung im Hinblick auf die Antwortzeit durchgeführt wird.

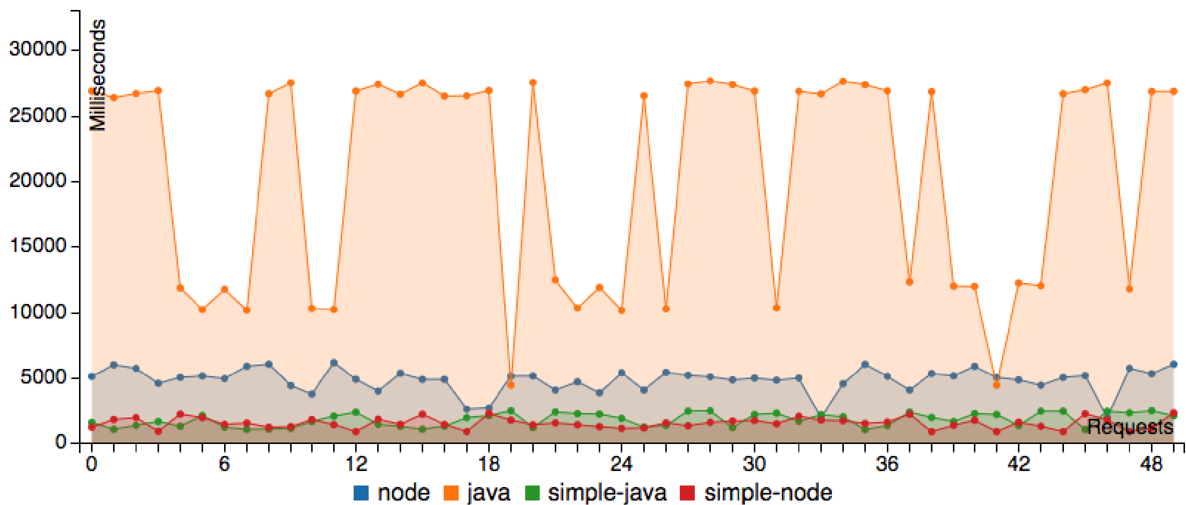


Abbildung 6.2: Antwortzeiten einer NodeJS und Java Lambda-Funktion bei 50 gleichzeitigen Anfragen, sodass für jede Anfrage eine neue Lambda-Instanz gestartet werden muss.

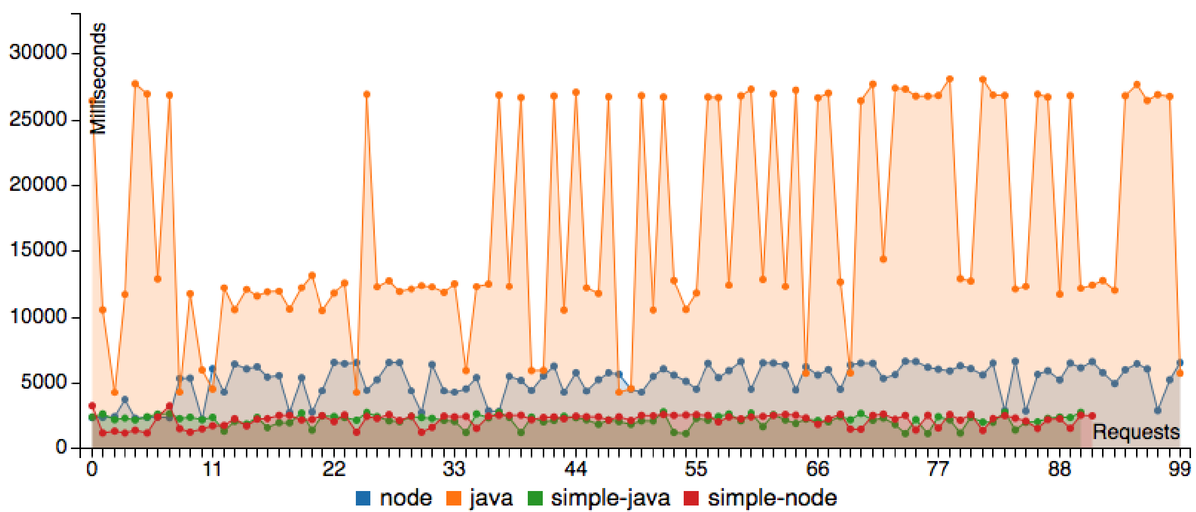


Abbildung 6.3: Antwortzeiten einer NodeJS und Java Lambda-Funktion bei 100 gleichzeitigen Anfragen, sodass für jede Anfrage eine neue Lambda-Instanz gestartet werden muss.

Ein Test-Skript, welches innerhalb von wenigen Hundert Millisekunden eine bestimmte Anzahl an HTTP-Anfragen an die NodeJS und Java Lambda-Funktionen sendet, misst die Zeit bis zum Empfang der Antwort. Abbildung 6.1 veranschaulicht dies für zehn gleichzeitige Anfragen und zeigt, dass die **node** Funktion im Durchschnitt nur 2,9 Sekunden zum Starten und Antworten benötigt, wohingegen die **java** Funktion im Durchschnitt fast das Dreifache (8,5 Sekunden) benötigt. Dieses Ergebnis spiegelt sich auch in Abbildung 6.2 wieder, welche die Ergebnisse mit 50 gleichzeitigen HTTP-Anfragen zeigt. Hier beträgt die durchschnittliche Antwortzeit 4,8 (**node**) bzw. 20,4 (**java**) Sekunden. Abbildung 6.3 mit 100 gleichzeitigen Anfragen zeigt ein ähnliches Ergebnis mit durchschnittlich 5,1 (**node**) bzw. 16,9 Sekunden (**java**). Im Vergleich dazu sind in allen drei genannten Abbildungen auch die Ergebnisse der **simple-node** und **simple-java** Funktionen dargestellt, welche zeigen, dass die Artefaktgröße durchaus wichtig ist und eine große Rolle beim Kaltstart einer Funktion spielt. Bei diesen einfachen Funktionen ist die Wahl der Programmiersprache jedoch weitaus weniger wichtig, was sich an den durchschnittlichen Antwortzeiten erkennen lässt: bei 10 gleichzeitigen Anfragen (siehe Abbildung 6.1) beträgt die durchschnittliche Antwortzeit 1,0 (**simple-node**) bzw. 2,0 (**simple-java**) Sekunden, bei 50 gleichzeitigen Anfragen (siehe Abbildung 6.2) beträgt sie 1,5 (**simple-node**) bzw. 1,8 (**simple-java**) Sekunden und bei 100 gleichzeitigen Anfragen (siehe Abbildung 6.3) beträgt sie 2,1 Sekunden für beide Lambda-Funktionen **simple-node** und **simple-java**. Da die Antwortzeit so gering ist, kommt es hier teilweise zur Wiederverwendung einer Instanz durch AWS Lambda. Daher fehlen für diese beiden Lambda-Funktionen ein paar Messwerte in Abbildung 6.3.

Damit verifiziert werden konnte, dass die Test-Ergebnisse reproduzierbar sind, wurden weitere Tests an anderen Tagen und zu verschiedenen Tageszeiten durchgeführt. Die Tests zeigen immer ein ähnliches Ergebnis: die **java** Lambda-Funktion ist deutlich langsamer beim Kaltstart im Vergleich zur **node** Lambda-Funktion. Ein Grund für die langsamere Startzeit ist sehr wahrscheinlich die Größe des Artefakts, was zehn mal so groß ist wie das der **node** Lambda-Funktion. Dieser Unterschied der Antwortzeiten wird besonders bei 100 gleichzeitigen Anfragen deutlich. Von Test zu Test variierte lediglich die durchschnittliche Antwortzeit, sodass sie beispielsweise bei zehn gleichzeitigen Anfragen teilweise bei 5,8 (**node**) bzw. 10,1 (**java**) Sekunden lag. Das Ergebnis kann hier allerdings leicht durch viele verschiedene Faktoren beeinflusst worden sein, wie beispielsweise die Verfügbarkeit von Instanzen in der jeweiligen AWS Region oder der Geschwindigkeit bzw. Auslastung des eigenen Internetanschlusses. Deshalb ist es wichtig, dass das Verhältnis der Antwortzeiten zwischen **node** und **java** Lambda-Funktionen in etwa reproduzierbar war.

Der Test zeigt, dass die Lambda-Funktionen bei einer Last von bis zu 100 gleichzeitigen Anfragen entsprechend skaliert werden. Allerdings können die gemessenen Werte die vorher festgelegten maximalen Antwortzeiten nicht erreichen. Sie liegen im Durchschnitt häufig bei den doppelten bzw. zehnfachen Werten. Nur die **simple-node** und **simple-java** Funktionen können die Erwartungswerte bei zehn bzw. 50 gleichzeitigen Anfragen unterschreiten. Durch eine entsprechend geringe Größe eines Artefakts lässt sich der Nebeneffekt des Kaltstarts sogar durchaus bemerkbar verringern, wie die Ergebnisse der einfachen Lambda-Funktionen zeigen.

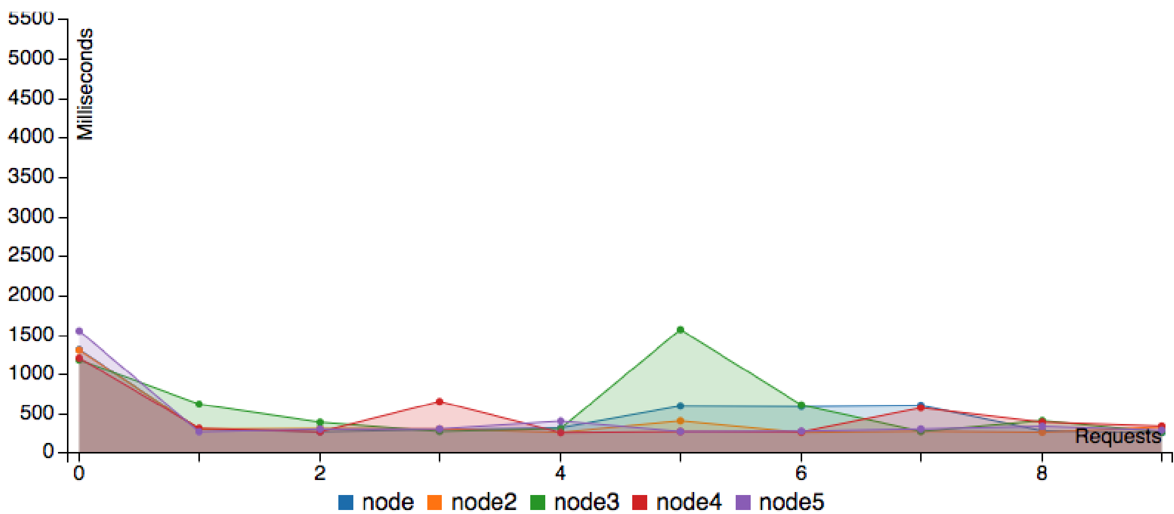


Abbildung 6.4: Vergleich der Antwortzeiten einer NodeJS Lambda-Funktion, die nach einem Kaltstart weitere Anfragen abarbeiten muss.

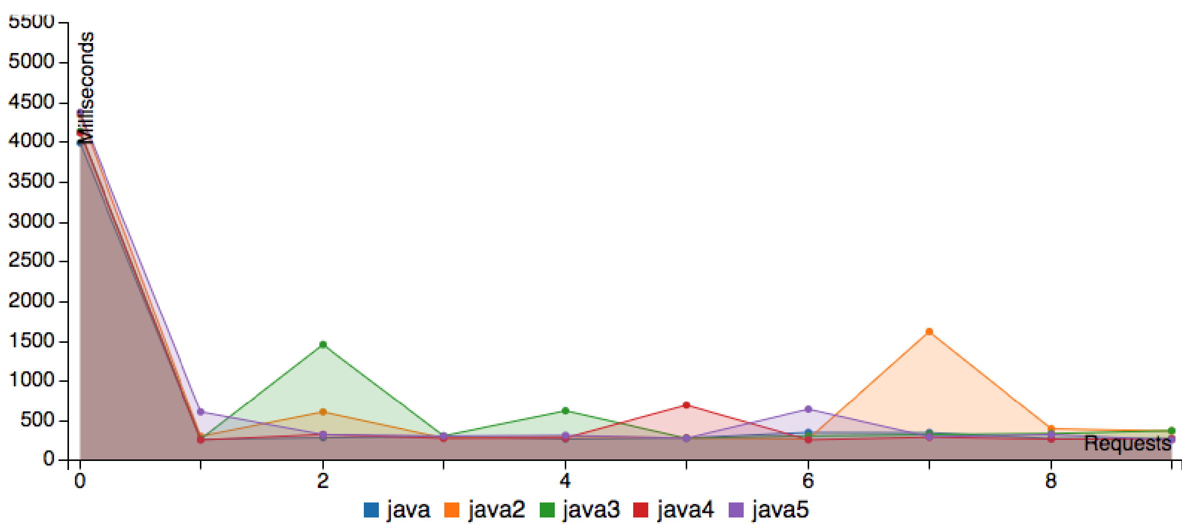


Abbildung 6.5: Vergleich der Antwortzeiten einer Java Lambda-Funktion, die nach einem Kaltstart weitere Anfragen abarbeiten muss.

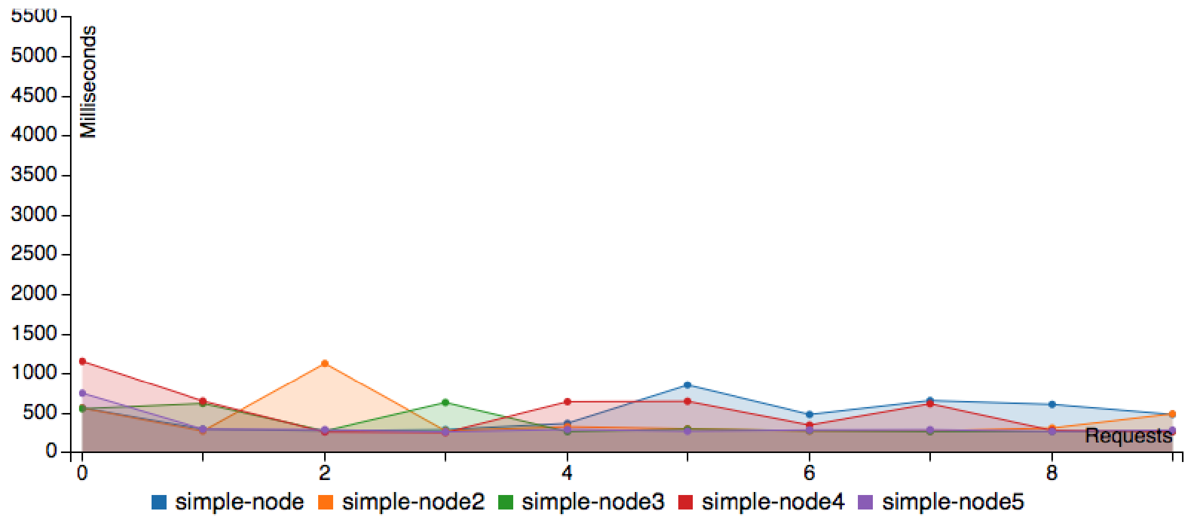


Abbildung 6.6: Vergleich der Antwortzeiten einer einfachen NodeJS Lambda-Funktion, die nach einem Kaltstart weitere Anfragen abarbeiten muss.

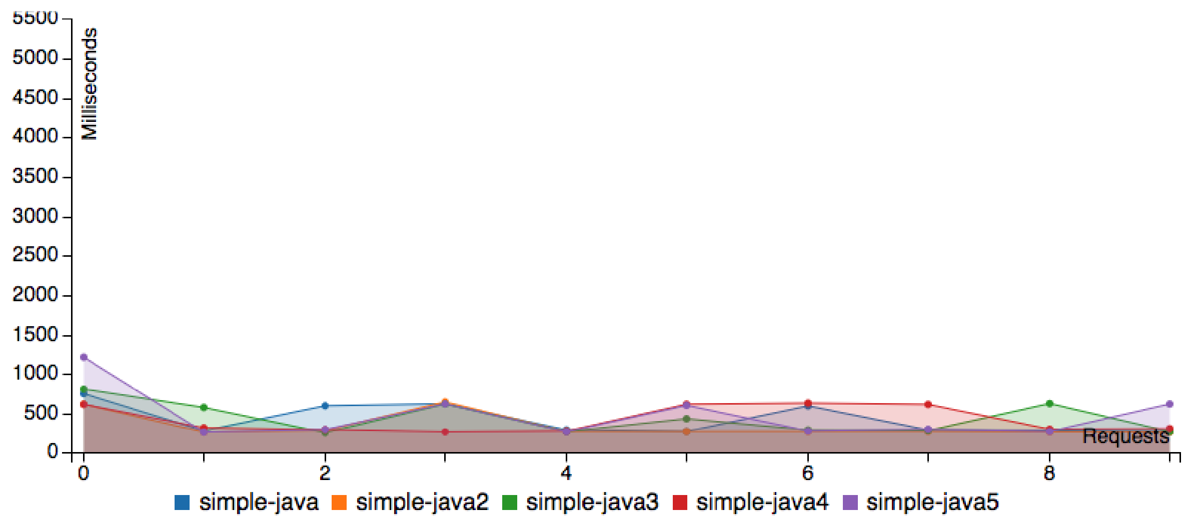


Abbildung 6.7: Vergleich der Antwortzeiten einer einfachen Java Lambda-Funktion, die nach einem Kaltstart weitere Anfragen abarbeiten muss.

6.2.4 Performance der aufgewärmten Lambda-Funktionen

Der zweite Test umfasst die Performance von Lambda-Funktionen anhand einer Sequenz von zehn Anfragen. Er soll zeigen, wie sich die Antwortzeiten verändern, sobald die Lambda-Instanz einmal gestartet wurde und daraufhin wiederverwendet werden kann. Deshalb werden die HTTP-Anfragen nicht gleichzeitig, sondern nacheinander gesendet. So wird sichergestellt, dass immer dieselbe Lambda-Instanz die Anfrage erhält und Anfragen auch im aufgewärmten

Zustand beantwortet[1]. Die Tests werden ebenfalls wieder mithilfe der Lambda-Funktionen **node**, **java**, **simple-node** und **simple-java** durchgeführt. Pro Test sind fünf verschiedene Testdurchläufe aufgeführt, sodass eine Aussage über mehrere Versuche gemacht werden kann.

Abbildung 6.4 und Abbildung 6.5 zeigen die Ergebnisse für **node** und **java**. Es ist bei beiden Abbildungen deutlich zu erkennen, dass die Antwortzeiten beim Kaltstart der Funktionen höher ausfallen (in den Abbildungen entspricht dies dem y-Wert von 0 auf der x-Achse) als die der weiteren Anfragen. Die Antwortzeiten der Kaltstarts entsprechen in etwa denen des vorherigen Tests. Jedoch fällt für die Tests bei den **java** Lambda-Funktionen auf, dass der Kaltstart lediglich zwischen 4 und 4,5 Sekunden beträgt. Im Vergleich dazu lag der Durchschnittswert im vorherigen Kaltstart-Test für zehn gleichzeitige Anfragen bei 8,5 Sekunden. Die Durchschnittswerte der HTTP-Anfragen ohne Kaltstart über alle Testdurchläufe hingegen betragen bei den Tests aus Abbildung 6.4 und Abbildung 6.5 0,378 (**node**) bzw. 0,387 (**java**) Sekunden. Wird der Kaltstart mitberechnet, so fallen die Werte etwas höher aus und belaufen sich auf 0,471 (**node**) bzw. 0,767 (**java**) Sekunden.

Abbildung 6.6 und Abbildung 6.7 zeigen die Ergebnisse der Tests für **simple-node** und **simple-java**. Auffällig ist hier, dass die Antwortzeiten der Kaltstarts sich nur gering von den weiteren Werten im aufgewärmten Zustand unterscheiden. Auch die Durchschnittswerte über alle Testdurchläufe bestätigen diesen Eindruck: 0,389 (**simple-node**) bzw. 0,377 (**simple-java**) Sekunden für die Werte ohne Kaltstart und 0,422 (**simple-node**) bzw. 0,419 (**simple-java**) Sekunden mit Einberechnung des Kaltstarts.

In diesem Test zeigt sich ebenfalls, dass der Kaltstart einer Lambda-Funktion eine wichtige Rolle spielt. Falls eine Lambda-Funktion jedoch bereits aufgewärmt wurde und eine Anfrage erhält, ist der Unterschied der Antwortzeiten kaum messbar. Denn wie die Ergebnisse der Testdurchläufe zeigen, liegen die durchschnittlichen Antwortzeiten bei einer bereits aufgewärmten Funktion zwischen 0,377 und 0,389 Sekunden. Dies sind zwar gute Werte, allerdings muss auch beachtet werden, dass die getesteten Lambda-Funktionen im Prinzip keinerlei Logik enthalten und ihre eigentliche Ausführungszeit einen dementsprechend geringen Anteil an der gesamten Antwortzeit ausmacht. Trotzdem werden damit die vorab definierten Erwartungswerte für die maximale Antwortzeit von allen Testdurchläufen erfüllt.

6.2.5 Dauer einer Synchronisation

Als nächstes wird die Dauer einer Synchronisation getestet. Sie ermittelt, wie schnell eine Änderung von einem Trello Board in das andere Trello Board synchronisiert wird. Dieser Test bedingt die Ausführung von drei Lambda-Funktionen, welche für die Synchronisation

[1] Ein erneuter Kaltstart kann anhand des HTTP-Status Codes festgestellt werden, wie es auch im vorherigen Test angewandt wurde. Dadurch ist ebenfalls sichergestellt, dass keine weitere Lambda-Instanz während eines Testdurchlaufs gestartet wurde.

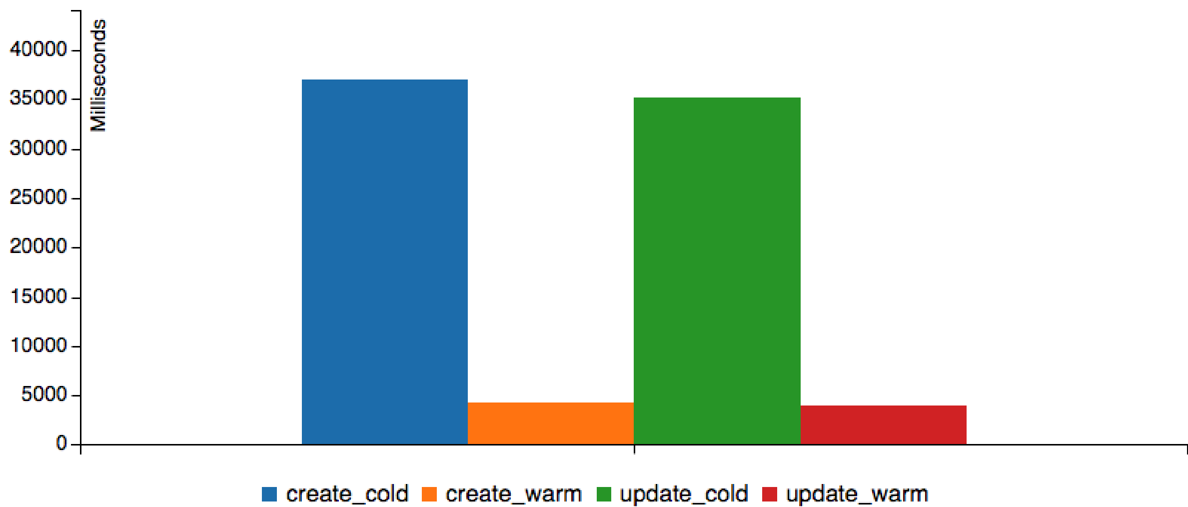


Abbildung 6.8: Gesamtdauer einer Synchronisation basierend auf dem Zustand der Funktionen: (1) Hinzufügen mit Kaltstart, (2) Hinzufügen aufgewärmt, (3) Aktualisierung mit Kaltstart, (4) Aktualisierung aufgewärmt.

verantwortlich sind: *Webhook Receiver*, *Outgoing Processor* und *Incoming Processor* (vgl. Abbildung 4.4 aus Abschnitt 4.2.3). Auch hier wird bei den Ergebnissen unterschieden, ob die Lambda-Funktionen bereits aufgewärmt wurden oder nicht. Für den Test werden daher insgesamt vier Testdurchläufe gestartet: Zwei Durchläufe fügen eine neue Karte einer Liste hinzu (mit/ohne Kaltstart) und zwei weitere verändern eine Karte (mit/ohne Kaltstart). Die Dauer der Synchronisation wird daraufhin ermittelt, indem die Schnittstelle der Trello REST API zur Ermittlung der Historie eines Boards angefragt wird und daraus die Zeitpunkte des Auftretens der Events ausgelesen werden. Im Detail bedeutet dies, dass eine Karte in Board A hinzugefügt bzw. verändert, das Änderungsdatum dieser Aktion in Board A ausgelesen, die Karte in Board B hinzugefügt bzw. geändert und das Änderungsdatum dieser Aktion in Board B ausgelesen wird.

Abbildung 6.8 zeigt die Ergebnisse des Tests: Eine Synchronisation mit Kaltstart dauert in beiden Fällen ca. 35 Sekunden und eine mit aufgewärmter Funktion etwa vier Sekunden. Einerseits erscheinen die Ergebnisse recht hoch, vor allem die Dauer der Synchronisationen mit Kaltstart. Andererseits müssen die Synchronisationen mit Kaltstart nicht nur eine Lambda-Funktion starten, sondern insgesamt drei: *Webhook Receiver* (NodeJS), *Outgoing Processor* und *Incoming Processor* (Java) werden alle nacheinander aufgerufen. Wenn die Durchschnittswerte der vorherigen Tests nun miteinander addiert werden, so ist die Dauer von 35 Sekunden durchaus realistisch. Die einzelnen Lambda-Funktionen enthalten zusätzlich noch Aufrufe zu weiteren Services wie der Trello API und DynamoDB, welche weitere Latenzen mit sich bringen und so die Dauer der Synchronisation erhöhen. Damit hält allerdings lediglich die Synchronisation mit aufgewärmten Funktionen den festgelegten Erwartungswert von maximal fünf Sekunden ein.

6.2.6 Auswertung der Test-Ergebnisse

Die durchgeführten Tests zeigen, dass die gewählte Serverless Architektur sehr gut automatisch skaliert. Sie besitzt jedoch durchaus noch Optimierungsbedarf bei der Performance der Lambda-Funktionen im Falle eines Kaltstarts. Auch andere Arbeiten haben gezeigt, dass Lambdas im Vergleich zu herkömmlichen Systemen, die beispielsweise auf einer Microservice-Architektur basieren, eine schlechtere Antwortzeit besitzen [HSH+16]. Für Lambda-Funktionen, von denen keine direkte Antwort durch einen Benutzer erwartet wird, ist der Kaltstart weitaus weniger bedeutend. In diesem Kontext ist nur wichtig, dass die Funktion automatisch ausgeführt und skaliert wird. Jedoch ist für Lambda-Funktionen, die per HTTP-Anfrage aus dem Frontend aufgerufen werden, eine Antwortzeit von mehr als 2 Sekunden wenig akzeptabel. Nichtsdestotrotz kann bei einer großen Nutzerbasis im Realbetrieb der Nachteil der Kaltstarts auf viele Benutzer verteilt werden, sodass nicht bei jeder HTTP-Anfrage eine neue Instanz gestartet werden muss. Aufgrund der regelmäßigen Nutzung können daher häufig bereits aufgewärmte Funktionen verwendet werden.

Dennoch ergeben sich hier Optimierungspotentiale, mit denen die Kaltstarts reduziert werden können: Zum einen sollte die Größe der Artefakte reduziert werden. Die Tests haben gezeigt, dass diese entscheidend für die Performance ist, insbesondere wenn mehrere Anfragen gleichzeitig gestellt werden. Vor allem für Java-Artefakte sollten die Abhängigkeiten optimiert und die Java Lambda-Funktionen auf jeweils eigene Artefakte aufgeteilt werden. Damit wird sichergestellt, dass jede Lambda-Funktion nur die Abhängigkeiten enthält, die tatsächlich zur Ausführung benötigt werden. Zum anderen wäre eine weitere Optimierung möglich, welche die Logik mehrerer Lambda-Funktionen unter einer Schnittstelle zusammenfasst, wodurch die einzelnen Kaltstarts vermieden werden und nur noch ein Kaltstart notwendig wäre. Am Beispiel von HTTP-Anfragen wäre dies die Zusammenlegung einer Schnittstelle unter einem bestimmten Pfad, welcher durch verschiedene HTTP-Methoden erreichbar ist. Im vorgestellten Prototypen könnten so die Lambda-Funktionen zur Verwaltung der Konfiguration unter `/api/config` zusammengefasst und durch eine einzige Lambda-Funktion verwaltet werden. Die Lambda-Funktion kann daraufhin anhand der HTTP-Methode selbst entscheiden, welche Aktion genau ausgeführt werden soll. Da diese Optimierung jedoch konträr zur vorherigen Optimierungsmöglichkeit ist, sollte dies nur für bestimmte Funktionen durchgeführt werden, welche inhaltlich auch zusammengefasst werden können. In diesem Fall muss eine gute Balance zwischen Artefaktgröße und Anzahl der Lambda-Funktionen gefunden werden.

Zur Verbesserung der Antwortzeiten wäre es ebenfalls möglich, Funktionen warm zu halten, indem jede Minute eine simple Anfrage an alle Funktionen gesendet wird, zum Beispiel durch eine spezielle Lambda-Funktion. Dadurch kann der Kaltstart einer Funktion zwar nicht vermieden werden, jedoch wird die Wahrscheinlichkeit erhöht, dass eine Funktion bereits aufgewärmt ist. Allerdings hat diese teilweise Umgehung des Kaltstarts auch einen Nachteil: Sie kostet Geld, da mehr Ausführungen erfolgen. Weiterhin sollte in dieser Hinsicht auch evaluiert werden, ob die Verwendung eines Servers eine Alternative darstellt. Da die Funktionen dann nicht mehr nach Bedarf gestartet werden, wären sie dauerhaft verfügbar. Jedoch widerspricht

dies in dieser Arbeit der Anforderung, dass das System einer Serverless-Architektur folgen soll. Es sollte auch nur in Betracht gezogen werden, wenn die Antwortzeiten dauerhaft nicht hinnehmbar sind.

6.3 Weitere Kritik

Neben der bereits erwähnten Kritik am Kaltstart existieren auch noch weitere Punkte, welche in die Entscheidung zur Verwendung einer Serverless Architektur einfließen müssen. Ein großer Nachteil der Lambda-Funktionen ist, dass eine Funktion nicht gestoppt werden kann. Sobald eine Funktion durch ein Event gestartet wurde, wird der entsprechende Programm-Code ausgeführt und kann nicht mehr von außen gestoppt werden. Dies kann zu unerwünschten Effekten führen, zum Beispiel einer Endlosschleife. Eine klassische Endlosschleife innerhalb des Programm-Codes würde zwar automatisch nach 5 Minuten Ausführungszeit gestoppt werden, weil die Ausführung nach dieser Zeit durch AWS abgebrochen wird. Jedoch kann eine Ausführung auch das Aufrufen anderer Lambda-Funktionen beinhalten, welche wiederum die initiale Lambda-Funktion aufrufen. Dadurch überschreitet eine einzelne Ausführung das 5 Minuten Limit nicht und die Endlosschleife ist nicht ausschließlich auf eine einzelne Funktion begrenzt. Eine erneute Ausführung der Lambda-Funktion kann lediglich verhindert werden, indem Eigenschaften der Lambda-Funktion verändert werden. Zum Beispiel kann eine Änderung einer Umgebungsvariable oder die Aktualisierung des Code-Artefakts durchgeführt werden, sodass die nächste Ausführung unerwartet abbricht oder gar nicht erst gestartet werden kann.

Weiterhin müssen in On-Demand-Systemen die Zugriffe auf andere Services und Schnittstellen vorsichtig verwendet werden. Zum einen kann ein Zugriff selbst Kosten verursachen. Zum anderen kann eine längere Ausführungszeit der Lambda-Funktion entstehen, was wiederum entsprechend von AWS abgerechnet wird. Für wenige Ausführungen können die Kosten unter Umständen noch nicht ins Gewicht fallen, jedoch wird dies bei steigender Nutzungszahl ein wichtigerer Faktor. Nichtsdestotrotz können durch die Verwendung von AWS Lambda die Kosten im Vergleich zu einer regulären Server-Architektur gesenkt werden, denn dort müssen Komponenten ausfallsicher geplant und für eine hohe Nutzungslast vorbereitet sein [AC17] [VGO+17]. Im Falle von AWS Lambda ist AWS für die Einhaltung der Ausfallsicherheit zuständig. Dies ist gleichzeitig ein weiterer Nachteil der engen Kopplung an ein Cloud-Ökosystem: Cloud Consumer machen sich von Cloud Providern sehr stark abhängig, wenn sie ausschließlich deren Services verwenden. Ansätze wie der des *Serverless Frameworks*[2] versprechen, dass Serverless Functions mithilfe des Frameworks auf jeder der unterstützten Plattformen[3] lauffähig sind. Jedoch können diese Ansätze die Nutzung aller anderen Services

[2] https://serverless.com/

[3] Unter anderem AWS Lambda, Google Cloud Functions oder Microsoft Azure Functions.

eines Cloud Providers nicht vollständig abstrahieren. Deshalb sollte ein Cloud Provider sorgsam ausgewählt werden.

6.4 Fazit

Der Ansatz, eine Serverless Architektur in der Cloud zu verwenden, bringt viele Vorteile mit sich. Zum einen ergibt sich ein günstiger Einstieg in die Technologie, da lediglich die tatsächliche Ausführungszeit in Rechnung gestellt wird. Zum anderen wird die Verwaltung und Skalierung einzelner Server dem Cloud Provider überlassen. Einem Cloud Consumer bieten sich dadurch nicht nur finanzielle Verbesserungen, sondern auch die Möglichkeit, ein Problem mit einer anderen Art von Architektur zu lösen. Durch die Konzentration auf die eigentliche Geschäftslogik können Software-Hersteller deshalb auch schneller auf Veränderungen reagieren. Allerdings muss bei diesem Ansatz neben den bereits erwähnten Nachteilen wie dem Kaltstart auch berücksichtigt werden, dass sich zum einen der Entwicklungsansatz erheblich von bisherigen Modellen unterscheidet und zum anderen eine Serverless Architektur nicht auf jedes Problem anwendbar ist. So zeigt sich auch durch die Tests, dass Lambda-Funktionen nicht optimal für HTTP-Schnittstellen geeignet sind. Zusätzlich sollte der Programm-Code besser auf mehrere Artefakte aufgeteilt werden, um die Artefakt-Größe zu reduzieren. Lambda-Funktionen sind jedoch gut zu verwenden, wenn sie als asynchrone Prozesse gestartet werden und von ihnen keine direkte Antwort erwartet wird. In diesen Fällen ist die Ausführungszeit häufig weniger relevant.

7 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit ist die Ausarbeitung einer Cloud-Architektur zur Synchronisation zweier Projektmanagement-Systeme. Dazu werden in Kapitel 3 die Grundlagen zu Projektmanagement-Systemen anhand von Trello und JIRA sowie deren Architekturen zur Erweiterung der Software beschrieben. Ein zentraler Aspekt beider Architekturen ist die Bereitstellung von Webhooks, welche durch eine Erweiterung empfangen und verarbeitet werden können. Weiterhin wird ein aktueller Überblick zu Cloud Computing und dem relativ neuen Ansatz des Serverless Computings gegeben, welches den Einsatz der Webhooks begünstigt. Daraufhin wird Serverless Computing am Beispiel von AWS Lambda erläutert.

Anschließend werden in Kapitel 4 die Anforderungen an das Konzept der Cloud-Architektur zusammengetragen. Darauf basierend wird das Konzept auf Basis von Trello und diversen AWS Services vorgestellt. Dieses reagiert auf empfangene Webhooks und löst entsprechende Verarbeitungsschritte aus. Innerhalb der Verarbeitung findet die Synchronisation zwischen zwei Trello Boards statt. Ebenfalls werden Fehler während dieser Synchronisation erkannt und teilweise automatisch behandelt. Damit die Architektur auch auf mehrere Benutzer ausgerichtet ist, wird ein Konzept zur Mandantenfähigkeit und Autorisierung vorgestellt. Zusätzlich zeigt ein Entwicklungsprozess, wie der Code automatisiert in der AWS Cloud-Infrastruktur provisioniert werden kann.

Aufbauend auf dem Konzept wird in Kapitel 5 die Umsetzung eines Prototypen beschrieben. Dieser basiert auf AWS CloudFormation und NodeJS- bzw. Java-Funktionen. Neben der Beschreibung des Deployments und der Konfiguration einer Integration wird auch der Synchronisationsprozess im Bezug auf die eingesetzte Datenspeicherung und -kommunikation detailliert erläutert. Dazu zählt auch die Einführung eines Locking-Mechanismus zur Vermeidung der Verwendung falscher Daten. Des Weiteren wird auch auf die Umsetzung der Mandantenfähigkeit und Behandlung von Service Limitierungen eingegangen.

In Kapitel 6 wird die prototypische Realisierung durch Laufzeit- und Performance-Tests evaluiert. Unter anderem werden die Antwortzeiten der Lambda-Funktionen und die Dauer einer Synchronisation untersucht. Die Ergebnisse zeigen, dass vor allem der Kaltstart einer Lambda-Funktion je nach Last sehr viel Zeit in Anspruch nehmen kann. Deshalb ist es von Vorteil, wenn die Zeit und Häufigkeit eines Kaltstarts minimiert werden können. Ein weiterer Vorteil ist die automatische Skalierung der Lambda-Funktionen, welche bei besonders viel genutzten Schnittstellen sehr hilfreich sein kann.

Die Arbeit zeigt, dass das Serverless Computing einen sehr guten Ansatz bietet, um Projektmanagement-Systeme auf Basis von Webhooks zu synchronisieren. Nichtsdestotrotz

sollte es mit Bedacht eingesetzt werden, da es keine Lösung für alle Probleme ist, wie die Testergebnisse zur Antwortzeit auch zeigen. Das heißt zum Beispiel, dass für die Umsetzung einer HTTP-Schnittstelle auch weitere Alternativen, die nicht dem Serverless-Ansatz folgen, evaluiert werden sollten.

Ausblick

In der Arbeit erfolgte eine prototypische Umsetzung des Konzepts. Diese hat gezeigt, dass die Nutzung einer Cloud-Architektur auf Basis des Serverless Computings vielversprechend ist. Allerdings sind dafür noch Weiterentwicklungen und Verbesserungen notwendig.

Der wichtigste Punkt wäre die Optimierung der Benutzeroberfläche, sodass ein Benutzer die Konfiguration nicht anhand eines Textfeldes eingeben muss. Dies würde zur besseren Verwendbarkeit des Systems beitragen, was auch die Zufriedenheit der Kunden erhöht. Damit hängt auch eine verbesserte Fehlerbehandlung zusammen: So könnte die Behandlung der Fehler ausgebaut werden, indem Benutzern ermöglicht wird, fehlgeschlagene Synchronisationen noch einmal durchzuführen. In bestimmten Fällen kann dies hilfreich sein. Beispielsweise wenn ein Fehler auf einer falschen Konfiguration beruht und diese zwischenzeitlich verändert wurde.

Ein weiterer wichtiger Punkt ist die Integration mit anderen Projektmanagement-Systemen, wie zum Beispiel JIRA oder IBM Rational Team Concert[1]. Damit eine Integration gelingt, müssen verschiedene Voraussetzungen gegeben sein. Ein Projektmanagement-System muss in jedem Fall über eine API verfügen, welche die Abfrage von Daten ermöglicht. Von Vorteil wäre außerdem, wenn diese API auch die Suche nach Änderungen über einen Zeitstempel oder ähnliche Merkmale unterstützt. So können regelmäßig neue Änderungen abgefragt und an das eigene System zur Verarbeitung weitergeleitet werden. Im Idealfall unterstützt das Projektmanagement-System Webhooks, so wie es in dieser Arbeit beschrieben wurde. Dadurch wäre eine regelmäßige Abfrage nach Änderungen nicht notwendig. Weiterhin müssen bei einer Anbindung auch entsprechende Erweiterungen zur korrekten Verarbeitung der ein- und ausgehenden Daten implementiert werden.

In Bezug auf die Cloud-Architektur lassen sich ebenfalls noch Verbesserungen vornehmen. Zum einen müssten Metriken und Alarme erstellt werden, um Fehler im System schneller und besser zu erkennen. Beispielsweise kann die Anzahl der fehlerhaften Ausführungen bzw. Aufrufe einer Lambda-Funktion gemessen werden. So könnte ab Überschreitung eines bestimmten Wertes ein Alarm per E-Mail oder SMS versendet werden, um einen Schaden und unangenehme Folgen zu vermeiden. Zum anderen verwendet der Prototyp keine geeigneten Caching-Mechanismen, welche durchaus die Anzahl der Anfragen an Datenbanken und andere Services sowie dadurch entstehende Kosten verringern können.

[1] https://www.ibm.com/us-en/marketplace/change-and-configuration-management

8 Anhang

Listing 8.1 Beispielhafte „manifest.json“-Datei für ein Trello Power-Up. Mit „capabilities“ werden die benötigten Rechte bzw. Erweiterungspunkte innerhalb der Trello Oberfläche angegeben.

```
{
  "name": "Power-Up Name",
  "details": "Describes the Power-Ups capabilities and some steps to use it.",
  "icon": {
    "url": "./logo.png"
  },
  "author": "Sebastian Hesse",
  "capabilities": [
    "authorization-status",
    "board-buttons",
    "callback",
    "show-authorization",
    "show-settings"
  ],
  "connectors": {
    "iframe": {
      "url": "./powerup-index.html"
    }
  }
}
```

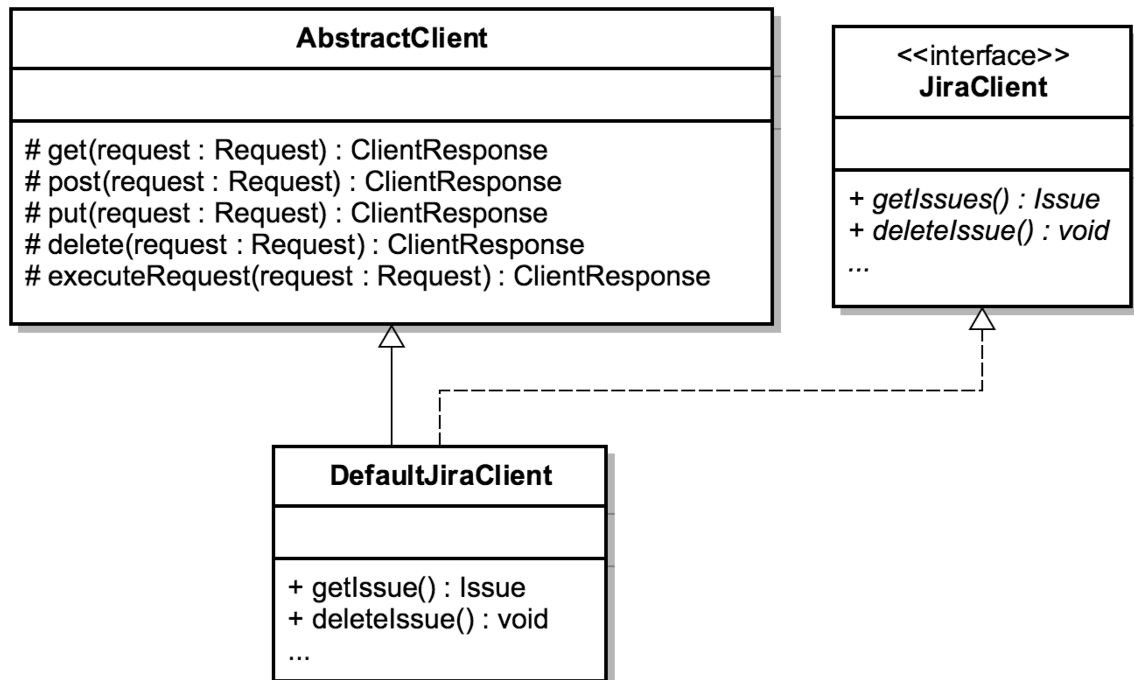


Abbildung 8.1: Klassendiagramm zum Aufbau der API Client Komponente vor der Umstellung. Die einzelnen Methoden der Schnittstelle sind der Übersichtlichkeit wegen nicht vollständig angegeben.

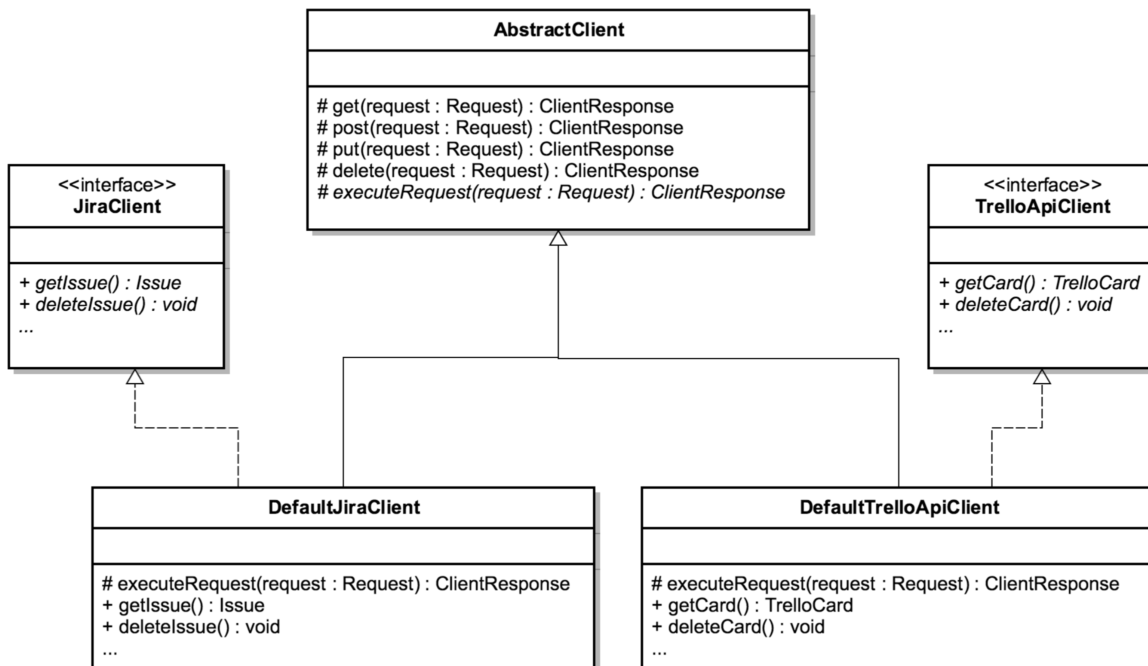
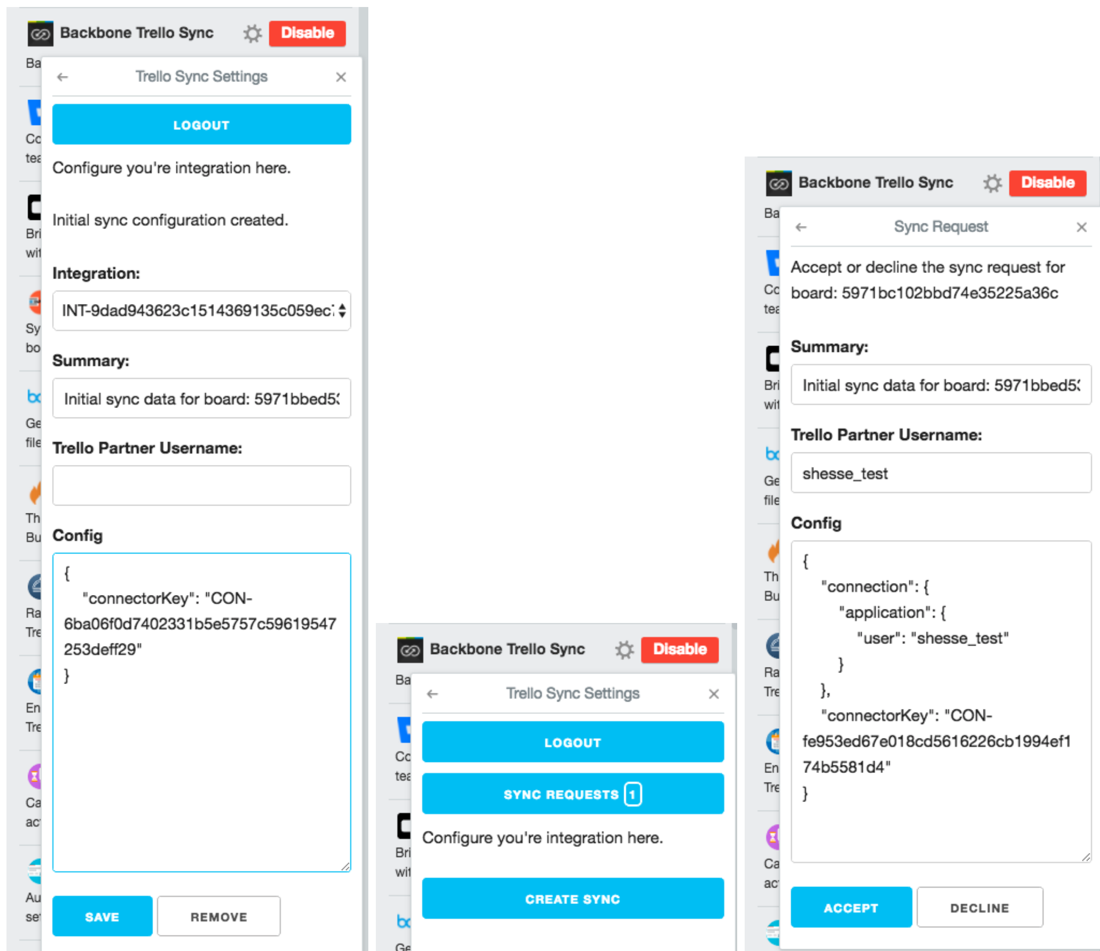


Abbildung 8.2: Klassendiagramm zum Aufbau der API Client Komponente nach der Umstellung. Wie zu sehen, können so zwei Clients für verschiedene REST Apis eine gemeinsame Basis haben, jedoch unterschiedliche Implementierungen für die eigentlichen Schnittstellen. Die einzelnen Methoden der Schnittstellen sind der Übersichtlichkeit wegen nicht vollständig angegeben.



(a) Anlegen eines Sync Requests im Power-Up des Boards A.

(b) Trello-Partner wird in Board B benachrichtigt, dass ein Sync Request von Board A existiert.

(c) Sync Request kann angenommen oder abgelehnt werden.

Abbildung 8.3: Ablauf zur Erstellung einer neuen Integration anhand eines Sync Requests.

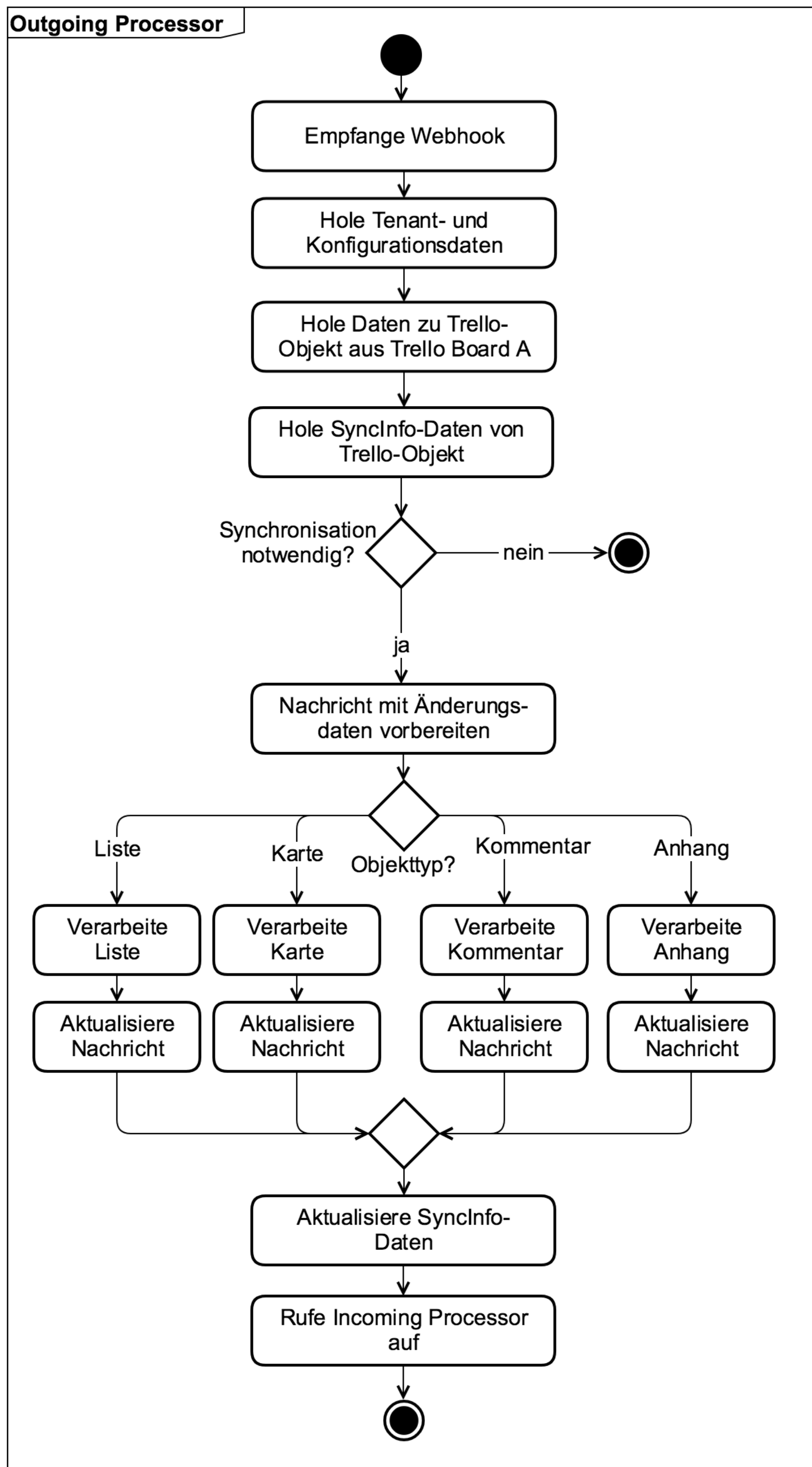


Abbildung 8.4: Ablauf der Synchronisation innerhalb des Outgoing Processors.

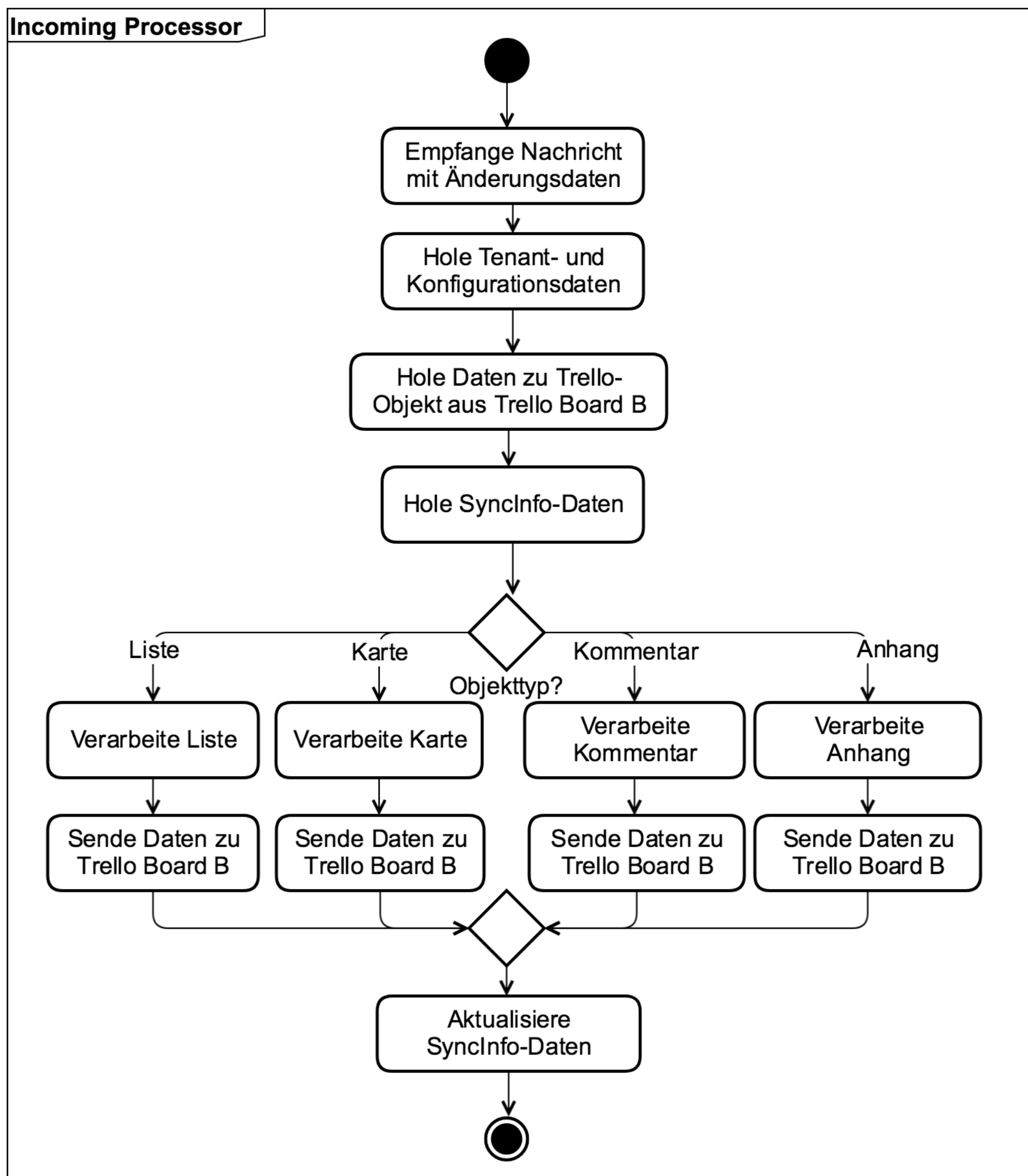


Abbildung 8.5: Ablauf der Synchronisation innerhalb des Incoming Processors.

Listing 8.2 Der Programm-Code für die *simple-node* Test-Lambda-Funktion als Beispiel zur Implementierung.

```
let called = false;

module.exports.simpleLambda = function (event, context, callback) {
   if (!called) {
      called = true;
      callback(null, { statusCode: 200, body: 'Success' });
   } else {
      callback(null, { statusCode: 400, body: 'Already called' });
   }
};
```

Literaturverzeichnis

[AC17] G. Adzic, R. Chatley. „Serverless computing: economic and architectural impact“. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, S. 884–889 (zitiert auf S. 18, 82).

[Ama17a] Amazon Web Services Inc. *Amazon CloudFront*. 2017. URL: https://aws.amazon.com/cloudfront/ (zitiert auf S. 35).

[Ama17b] Amazon Web Services Inc. *Amazon DynamoDB*. 2017. URL: https://aws.amazon.com/dynamodb/ (zitiert auf S. 34).

[Ama17c] Amazon Web Services Inc. *Amazon Elastic Container Service*. 2017. URL: https://aws.amazon.com/ecs/ (zitiert auf S. 30).

[Ama17d] Amazon Web Services Inc. *Amazon Web Services (AWS)*. 2017. URL: https://aws.amazon.com/ (zitiert auf S. 15, 17, 29).

[Ama17e] Amazon Web Services Inc. *AWS API Gateway Product Details*. 2017. URL: https://aws.amazon.com/api-gateway/details/ (zitiert auf S. 35).

[Ama17f] Amazon Web Services Inc. *AWS CloudFormation*. 2017. URL: https://aws.amazon.com/cloudformation/ (zitiert auf S. 35).

[Ama17g] Amazon Web Services Inc. *AWS Lambda Limits*. 2017. URL: http://docs.aws.amazon.com/lambda/latest/dg/limits.html (zitiert auf S. 32).

[Ama17h] Amazon Web Services Inc. *AWS Lambda Pricing*. 2017. URL: https://aws.amazon.com/lambda/pricing/ (zitiert auf S. 33).

[Ama17i] Amazon Web Services Inc. *AWS Lambda - Programming Model*. 2017. URL: http://docs.aws.amazon.com/lambda/latest/dg/programming-model-v2.html (zitiert auf S. 31).

[Ama17j] Amazon Web Services Inc. *AWS Management Console*. 2017. URL: https://aws.amazon.com/console/ (zitiert auf S. 35).

[Ama17k] Amazon Web Services Inc. *AWS Serverless Application Model (SAM)*. 14. Aug. 2017. URL: https://github.com/awslabs/serverless-application-model/blob/master/versions/2016-10-31.md (zitiert auf S. 55, 56).

[Ama17l] Amazon Web Services Inc. *AWS Step Functions*. 2017. URL: https://aws.amazon.com/step-functions/ (zitiert auf S. 66).

[Ama17m] Amazon Web Services Inc. *Managing Throughput Capacity Automatically with DynamoDB Auto Scaling*. 2017. URL: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/AutoScaling.html (zitiert auf S. 69).

[Ama17n] Amazon Web Services Inc. *Throughput Settings for Reads and Writes*. 2017. URL: http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ProvisionedThroughput.html (zitiert auf S. 68, 69).

[Atl17a] Atlassian Inc. *Atlassian + Trello: changing the way teams work*. 9. Jan. 2017. URL: https://www.atlassian.com/blog/announcements/atlassian-plus-trello (zitiert auf S. 21, 22).

[Atl17b] Atlassian Inc. *Integrating with JIRA Cloud*. 2017. URL: https://developer.atlassian.com/cloud/jira/platform/integrating-with-jira-cloud/ (zitiert auf S. 26).

[Atl17c] Atlassian Inc. *JIRA Cloud Platform - Security Overview*. 2017. URL: https://developer.atlassian.com/cloud/jira/platform/security-overview/ (zitiert auf S. 27).

[Atl17d] Atlassian Inc. *Shareholder Letter Q4'17 and Fiscal 2017*. 27. Juli 2017. URL: https://s2.q4cdn.com/141359120/files/doc_financials/2017/Q4/TEAM-Q4-2017-Shareholder-Letter.pdf (zitiert auf S. 14).

[Bar14] J. Barr. *AWS Lambda - Run Code in the Cloud*. 13. Nov. 2014. URL: https://aws.amazon.com/blogs/aws/run-code-cloud/ (zitiert auf S. 15, 30).

[BCC+17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski et al. „Serverless Computing: Current Trends and Open Problems". In: *arXiv preprint arXiv:1706.03178* (2017) (zitiert auf S. 17, 18, 31, 33).

[BZP+10] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, A. Hart. „Enabling multi-tenancy: An industrial experience report". In: *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE. 2010, S. 1–8 (zitiert auf S. 18).

[DHJ+07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. „Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), S. 205–220 (zitiert auf S. 34).

[DL12] O. Demir, J. Lunze. „Event-based synchronisation of multi-agent systems". In: *IFAC Proceedings Volumes* 45.9 (2012), S. 1–6 (zitiert auf S. 43).

[Goo17a] Google Inc. *Cloud Functions - Serverless Environment to Build and Connect Cloud Services*. 2017. URL: https://cloud.google.com/functions/ (zitiert auf S. 31).

[Goo17b] Google Inc. *Google Cloud Computing, Hosting Services & APIs*. 2017. URL: https://cloud.google.com/ (zitiert auf S. 29).

[GSH+07] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, B. Gao. „A framework for native multi-tenancy application development and management". In: *e-commerce Technology and the 4th IEEE International Conference on Enterprise Computing, e-commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on.* IEEE. 2007, S. 551–558 (zitiert auf S. 18).

[Her17] Heroku Inc. *Cloud Application Platform.* 2017. URL: https://www.heroku.com/ (zitiert auf S. 29).

[HSH+16] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau. „Serverless Computation with OpenLambda". In: *Elastic* 60 (2016), S. 80 (zitiert auf S. 17, 31, 81).

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Addison-Wesley Professional, 2004 (zitiert auf S. 19).

[IBM17] IBM. *IBM Cloud Functions.* 2017. URL: https://www.ibm.com/cloud/functions (zitiert auf S. 18, 31).

[Int10] Internet Engineering Task Force (IETF). *The OAuth 1.0 Protocol.* 2010. URL: https://tools.ietf.org/html/rfc5849 (zitiert auf S. 49).

[KMK12] R. Krebs, C. Momm, S. Kounev. „Architectural Concerns in Multi-tenant SaaS Applications." In: *Closer* 12 (2012), S. 426–431 (zitiert auf S. 18, 47).

[Lin00] D. S. Linthicum. *Enterprise application integration.* Addison-Wesley Professional, 2000 (zitiert auf S. 19).

[Lin07] J. Lindsay. *Web hooks to revolutionize the web.* 3. Mai 2007. URL: http://progrium.com/blog/2007/05/03/web-hooks-to-revolutionize-the-web/ (zitiert auf S. 27).

[MG+11] P. Mell, T. Grance et al. „The NIST definition of cloud computing". In: *Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg* (2011) (zitiert auf S. 28–30).

[Mic17a] Microsoft. *Microsoft Azure Cloud Computing Platform & Services.* 2017. URL: https://azure.microsoft.com/ (zitiert auf S. 29).

[Mic17b] Microsoft Inc. *Azure Functions - Serverless Architecture.* 2017. URL: https://azure.microsoft.com/en-us/services/functions/ (zitiert auf S. 31).

[Mic17c] Microsoft Inc. *Microsoft Office 365.* 2017. URL: https://products.office.com/en-us/business/explore-office-365-for-business-a (zitiert auf S. 30).

[Mis15] D. Mishunov. *Why Perceived Performance Matters, Part 1: The Perception Of Time.* 25. Sep. 2015. URL: https://www.smashingmagazine.com/2015/09/why-performance-matters-the-perception-of-time/ (zitiert auf S. 74).

[MK11] C. Momm, R. Krebs. „A Qualitative Discussion of Different Approaches for Implementing Multi-Tenant SaaS Offerings." In: *Software Engineering (Workshops).* Bd. 11. 2011, S. 139–150 (zitiert auf S. 18).

[Mor16] K. Morris. *Infrastructure as code: managing servers in the cloud.* O'Reilly Media, Inc., 2016 (zitiert auf S. 35).

[MSE+16] G. McGrath, J. Short, S. Ennis, B. Judson, P. Brenner. „Cloud Event Programming Paradigms: Applications and Analysis". In: *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on.* IEEE. 2016, S. 400–406 (zitiert auf S. 18).

[Nie93] J. Nielsen. *Response Times: The 3 Important Limits.* 1. Jan. 1993. URL: https://www.nngroup.com/articles/response-times-3-important-limits/ (zitiert auf S. 73).

[PA01] T. Puschmann, R. Alt. „Enterprise application integration-the case of the Robert Bosch Group". In: *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on.* IEEE. 2001, 10–pp (zitiert auf S. 19).

[PS17] A. Patrikalis, S. Slutsker. *Building Distributed Locks with the DynamoDB Lock Client.* 15. Aug. 2017. URL: https://aws.amazon.com/blogs/database/building-distributed-locks-with-the-dynamodb-lock-client/ (zitiert auf S. 66).

[Rob16] M. Roberts. *Serverless Architectures.* 4. Aug. 2016. URL: https://martinfowler.com/articles/serverless.html (zitiert auf S. 31).

[sal17] salesforce.com Inc. *Online CRM Software Systems.* 2017. URL: https://www.salesforce.com/ (zitiert auf S. 30).

[Syn17] Synergy Research Group. *The Leading Cloud Providers Continue to Run Away with the Market.* 27. Juli 2017. URL: https://www.srgresearch.com/articles/leading-cloud-providers-continue-run-away-market (zitiert auf S. 28).

[Tre16] Trello Inc. *Introducing Trello's New Power-Ups Platform.* 18. Jan. 2016. URL: https://blog.trello.com/introducing-the-trello-power-ups-platform (zitiert auf S. 27).

[Tre17a] Trello Inc. *Authorization.* 2017. URL: https://developers.trello.com/page/authorization (zitiert auf S. 27).

[Tre17b] Trello Inc. *Trello is Being Acquired By Atlassian.* 9. Jan. 2017. URL: https://blog.trello.com/trello-atlassian (zitiert auf S. 14).

[Tre17c] Trello Inc. *Webhooks.* 2017. URL: https://developers.trello.com/v1.0/page/webhooks (zitiert auf S. 67).

[VGO+17] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano et al. „Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures". In: *Service Oriented Computing and Applications* 11.2 (2017), S. 233–247 (zitiert auf S. 70, 82).

[YCCI16] M. Yan, P. Castro, P. Cheng, V. Ishakian. „Building a Chatbot with Serverless Computing". In: *Proceedings of the 1st International Workshop on Mashups of Things and APIs.* ACM. 2016, S. 5 (zitiert auf S. 18).

[ZCB10] Q. Zhang, L. Cheng, R. Boutaba. „Cloud computing: state-of-the-art and research challenges". In: *Journal of internet services and applications* 1.1 (2010), S. 7–18 (zitiert auf S. 17, 28, 29, 33).

Alle URLs wurden zuletzt am 26. November 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift