HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

ins institute for networked solutions

Internet Of Things

mit Microsoft Azure

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2014

Autoren:	Freier Dominik
	Marco Leutenegger
Betreuer:	Prof. Hansjörg Huser
Projektpartner:	INS Institute for Networked Solutions

Aufgabenstellung

Internet of Things mit Microsoft Azure

Aufgabenstellung für Dominik Freier und Marco Leutenegger

Einführung
Internet of Things (IoT) bezeichnet ein Szenario, wo Devices (Sensoren, Actors) direkt mit einem eindeutigen Identifier (z.B. IPV6_Adresse) versehen sind und direkt übers Netzwerk, z.B. mit Cloud-basierten Diensten kommunizieren können. Da bis zu mehrere tausend oder sogar Millionen von Devices mit der Cloud kommunizieren können, ist die Skalierbarkeit und die Performance der Cloud-Lösung eine grosse Herausforderung.
Ziel dieses Projektes ist der Aufbau eines IoT-Systems basierend auf der Microsoft Azure Cloud-Plattform. Das IoT- System besteht für Demo-Zwecke aus einem Simulator für die Generierung von Events von vielen Sensoren.
Hauptfokus liegt auf der effizienten Verteilung der Events von den Sensoren sowie für das Kommandieren von Actors.

Aufgabe:
Architektur einer IOT – Anwendung mit Microsoft Azure (Service Bus Event Hubs)
Implementation eines Demo-Systems

- Cloud Services (inkl. persistente Speicherung der Konfigurations- und des Systemzustands)
- Client (Web oder Smart Client)
- Prozess-Simulator zur Lastgenerierung
- Anforderungen SW-Design
- Erweiterbar, konfigurierbar bezüglich Anzahl und Art der Sensoren und Actors

Resultate:

- Lauffähige Software
- Dokumentation gemäss HSR Vorlage

Projektteam

Dominik Freier (dfreier@hsr.ch)
Marco Leutenegger (mleutene@hsr.ch)

Betreuung HSR

Hansjörg Huser, hhuser@hsr.ch, Tel: 055 222 49 12 (HSR Raum 6.010)

Projektabwicklung

Termine:

Siehe auch Terminplan auf https://www.hsr.ch/Termine-Diplom-Bachelor-und.5142.0.html?&L=0

- 15.09.2014: Beginn der Studienarbeit, Ausgabe der Aufgabenstellung.
- 15.12.2014: Die Studierenden senden folgende Dokumente der Arbeit per Email zur Prüfung an ihre Betreuer: - Kurzfassung - A0-Poster Vorlagen sowie eine ausführliche Anleitung betreffend Dokumentation stehen unter den allgemeinen Infos Diplom-, Bachelor- und Studienarbeiten zur Verfügung.
- 19.12.2014, 12h: Die Studierenden senden die vom Betreuer abgenommene und freigegebene Kurzfassung als Word-Dokument an das Studiengangsekretariat (cfurrer(at)hsr.ch).
- 19.12.2014, 17h: Abgabe des Berichtes an den Betreuer.

Arbeitsaufwand

Für die erfolgreich abgeschlossene Arbeit werden 8 ECTS angerechnet. Dies entspricht einer Arbeitsleistung
von mind. 240 Stunden pro Student. Planen Sie eine durchschnittliche Arbeitszeit von
17 Stunden (oder 2 Arbeitstage) pro Woche ein!

Hinweise für die Gliederung und Abwicklung des Projektes:
Gliedern Sie Ihre Arbeit in 4 bis 5 Teilschritte. Schliessen Sie jeden Teilschritt mit einem Meilenstein ab. Definieren Sie für jeden Meilenstein, welche Resultate dann vorliegen müssen!

Folgende Teilschritte bzw. Meilensteine sollten Sie in Ihrer Planung vorsehen:
Schritt 1: Projektauftrag inkl. Projektplan (mit Meilensteinen),

- Meilenstein 1: Review des Projektauftrages abgeschlossen. Projektauftrag von Auftraggeber und Dozent genehmigt
  Letzter Meilenstein: Systemtest abgeschlossen Termin: ca. eine Woche vor Abgabe

- Entwickeln Sie Ihre SW mit einem agilen oder in einem iterativen, inkrementellen Prozess: Planen Sie möglichst früh einen ersten lauffähigen Prototypen mit den wichtigsten und kritischsten Kernfunktionen. In die folgenden Phasen können Sie dieses Kernsystem schrittweise ausbauen und testen.
- Falls Sie in Ihrer Arbeit neue oder Ihnen unbekannte Technologien einsetzen, sollten Sie parallel zum Erarbeiten des Projektauftrages mit dem Technologiestudium beginnen.
- Setzen Sie konsequent Unit-Tests ein! Verwalten Sie ihre Software und Dokumente in einem Versionsverwaltungssystem. Stellen Sie sicher, dass der/die Betreuer jederzeit Zugriff auf das Repository haben und dass das Projekt anhand des Repositories jederzeit wiederhergestellt werden kann.
- Achten Sie auf die Einhaltung guter Programmier- und Designprinzipien
- Halten Sie sich im Übrigen an die Vorgaben aus dem Modul SE-Projekt.

Projektadministration

- Führen Sie ein individuelles Projekttagebuch aus dem ersichtlich wird, welche Arbeiten Sie durchgeführt haben (inkl. Zeitaufwand). Diese Angaben sollten u.a. eine individuelle Beurteilung ermöglichen.
- Dokumentieren Sie Ihre Arbeiten laufend. Legen Sie Ihre Projektdokumentation mit der aktuellen Planung und den Beschreibungen der Arbeitsresultate elektronisch in einem Projektordner ab. Dieser Projektordner sollte jederzeit einsehbar sein (z.B svn-Server oder File-Share).

Inhalt der Dokumentation
Bei der Abgabe muss jede Arbeit folgende Inhalte haben:

- Dokumente gemäss Vorgabe:
  siehe https://www.hsr.ch/Allgemeine-Infos-Diplom-Bach.4418.0.html?&L=0.
- Die Abgabe ist so zu gliedern, dass die obigen Inhalte klar erkenntlich und auffindbar sind.
- Zitate sind zu kennzeichnen, die Quelle ist anzugeben.
- Verwendete Dokumente und Literatur sind in einem Literaturverzeichnis aufzuführen.
- Projekttagebuch, Dokumentation des Projektverlaufes, Planung etc.

Die Dokumentation (inkl. Source) ist vollständig auf CD/DVD in 2 Exemplaren abzugeben. Auf Wunsch ist für den Betreuer eine gedruckte Version zu erstellen.

Fortschrittsbesprechung:
Regelmässig findet zu einem fixen Zeitpunkt eine Fortschrittsbesprechung statt.
Teilnehmer: Betreuer und Studenten, bei Bedarf auch Vertreter der Auftraggeber
Termin: jeweils Dienstag, 17h, Raum 6.010 (Abweichungen werden rechtzeitig kommuniziert)

Alle Besprechungen sind von den Studierenden mit einer Traktandenliste vorzubereiten und die Ergebnisse in einem Protokoll zu dokumentieren, das dem Betreuer und dem Auftraggeber per Email (spätestens innerhalb von 2 Tagen) zugestellt wird.
Standard-Traktanden sind

- Was wurde erreicht, was ist geplant, welche Probleme stehen an
- Review von Code/Dokumentation (Abgabe jeweils einen Tag vor dem Meeting

Falls notwendig, können weitere Besprechungen / Diskussionen einberufen werden.

Beurteilung

Gesichtspunkte	**Gewicht**
Organisation, Durchführung	1/5
Berichte (Abstract, Mgmt Summary, techn. und persönliche Berichte) sowie Gliederung, Darstellung, Sprache)	1/5
Inhalt	3/5

*) Die Unterteilung und Gewichtung von 3. Inhalt wird im Laufe dieser Arbeit festgelegt.
Im Übrigen gelten die Bestimmungen der Abt. Informatik zur Durchführung von Studienarbeiten.

Rapperswil, 15. Sept. 2014

Hansjörg Huser

Abstract

Unsere Arbeit befasst sich mit dem Thema “Internet of Things”. Dazu wurde die Microsoft Azure Cloud eingesetzt. Einerseits soll in dieser Arbeit ein Service entstehen, mit dem Devices (Sensoren und Actors) Events bzw. Commands versenden können. Andererseits soll ein solches Szenario als Demo-System implementiert werden und Multitenancy unterstützen.

Konkret bedeutet das, dass Sensoren über einen Device Controller die gemessenen Daten (Events) in der Cloud auf einen EventHub ablegen.
Die Events auf dem EventHub werden von einer Worker Role konsumiert und verarbeitet. Die Verarbeitung besteht darin, dass die Worker Role die relevanten Daten aus den Events ausliest, ein Entity Objekt erstellt und dieses in einem Table Storage persistiert.

Auf diese Daten kann dann über eine RESTful API zugegriffen werden. Für diesen Zweck stellt eine Web Role Methoden zur Verfügung, um die persistierten Daten nach verschiedenen Kriterien auszulesen.
Die API ist gegen unautorisierten Zugriff geschützt. Darauf zugreifen kann nur, wer im MS Azure Active Directory erfasst wurde. Die Daten sind daher einerseits durch die gesicherte API geschützt, andererseits wird sichergestellt, dass nicht auf die Daten eines anderen Tenants zugegriffen werden kann.

Über das RESTful API können auch Commands gesendet werden. Mit diesen Commands kann beispielsweise die Konfiguration eines Aktors geändert werden.
Der Command wird über die Web API auf eine Queue gelegt, die wiederum von einem Device Controller gelesen wird. Der Device Controller ändert aufgrund des Commands die Konfiguration des Aktors und sendet eine Antwort über die API auf die Antwort-Queue. Wenn die Konfigurationsänderung korrekt abgelaufen ist, wird die Konfiguration im Table Storage aktualisiert.

Zusammengefasst bietet unser System die Möglichkeit, Events von Actors zu sammeln und in einer Cockpit-Anwendung darzustellen. Zudem können die verschiedenen Actors über das Cockpit konfiguriert werden.

Management Summary

Ausgangslage

Diese Studienarbeit befasst sich mit einem Teilgebiet des Konzepts «Internet of Things», nämlich der lokalen Vernetzung von Sensoren und Actors, wie man es beispielsweise in so genannten «Intelligent Home Systems» wiederfindet. Die Arbeit richtet sich in erster Linie an Dienstanbieter derartiger Systeme.

Gemäss der Aufgabenstellung ist diese Arbeit sowohl eine Lösungserstellung, als auch eine Forschungsarbeit. Ziel ist ein System für ein imaginäres IoT-Szenario. Der Forschungsteil befasst sich mit einer abstrakten Architektur und deren Implementierung, sodass viele verwandte Szenarien ein gemeinsames Fundament nutzen können. Im Lösungsteil, dem Demo-System, liegt der Fokus auf der Umsetzung eines konkreten Anwendungsfalls, mit Hilfe dieses Fundaments.

Vorgehen / Technologien

Da wir anfangs der Studienarbeit noch keine Kenntnisse betreffend .NET und MS Azure Cloud hatten, mussten wir uns in einem ersten Schritt mit der Materie auseinandersetzen. Das bedeutete für uns, dass wir uns einerseits in die Theorie einlesen, andererseits verschiedene Tutorials durcharbeiten mussten.
Zeitgleich besuchten wir das Modul «.Net Technologien», mit dem wir uns so einen Wissensstand erarbeiten konnten, um die Arbeit gemäss Anforderungen schrittweise durchzuführen. Durch die wöchentlichen Meetings mit Herrn Huser konnten wir unsere Umsetzung überprüfen und etwaige Fragen klären.

Eingesetzte Technologien:

- Microsoft Azure
- C#
- ASP.NET

Ergebnisse

Unser Service bietet dem Benutzer die Möglichkeit Sensoren und Actors zu registrieren. Die Daten, die der Sensor sammelt, werden in die Cloud gesendet, wo sie verarbeitet und gesichert werden. Diese Daten können dann von einem Client (Webapplikation, Mobile-App etc.) abgerufen werden. Weiter kann der Client dem Actor Commands senden, um dessen Konfiguration zu ändern.

Das Ergebnis beinhaltet ein Demo-System, das (unter anderem) aus einem Feuchtigkeitssensor und einem Sprinkler besteht. Der Sensor sendet die Daten via Cloud zum Client. Über den Client kann dann der Kunde beispielsweise den Sprinkler ein oder ausschalten oder dessen Radius ändern.

Ausblick

Während der Arbeit entstand der Anspruch ein solches System automatisiert aufzusetzen zu können. Dies konnte jedoch in der vorgegebenen Zeit nicht umgesetzt werden.

Inhalt

Technischer Bericht

1 Ausgangslage

Diese Studienarbeit befasst sich mit einem Teilgebiet des «Internet of Things», nämlich der lokalen Vernetzung von Sensoren und Actors, wie man sie beispielsweise in so genannten «Intelligent Home Systems» wiederfindet. Die Arbeit richtet sich in erster Linie an Organisationen, die ein derartiges Intelligent Home System für eine grosse Anzahl an Kunden betreiben möchten.

Gemäss der Aufgabenstellung ist diese Arbeit sowohl eine Lösungserstellung, als auch eine Forschungsarbeit. Gesamtziel ist es, ein System für ein imaginäres IoT-Szenario zu entwickeln. Der Forschungsteil befasst sich damit eine Architektur zu finden, die für viele ähnliche Szenarien wiederverwendet werden kann und die grundlegenden Funktionen eins Intelligent Home Systems abdeckt. Welche Funktionen das sind und wie diese am besten implementiert werden, das muss zuerst erforscht werden. Im Lösungsteil, dem Demo-System, liegt der Fokus auf der Umsetzung eines konkreten Anwendungsfalls, der auf dem Basisszenario aufbaut.

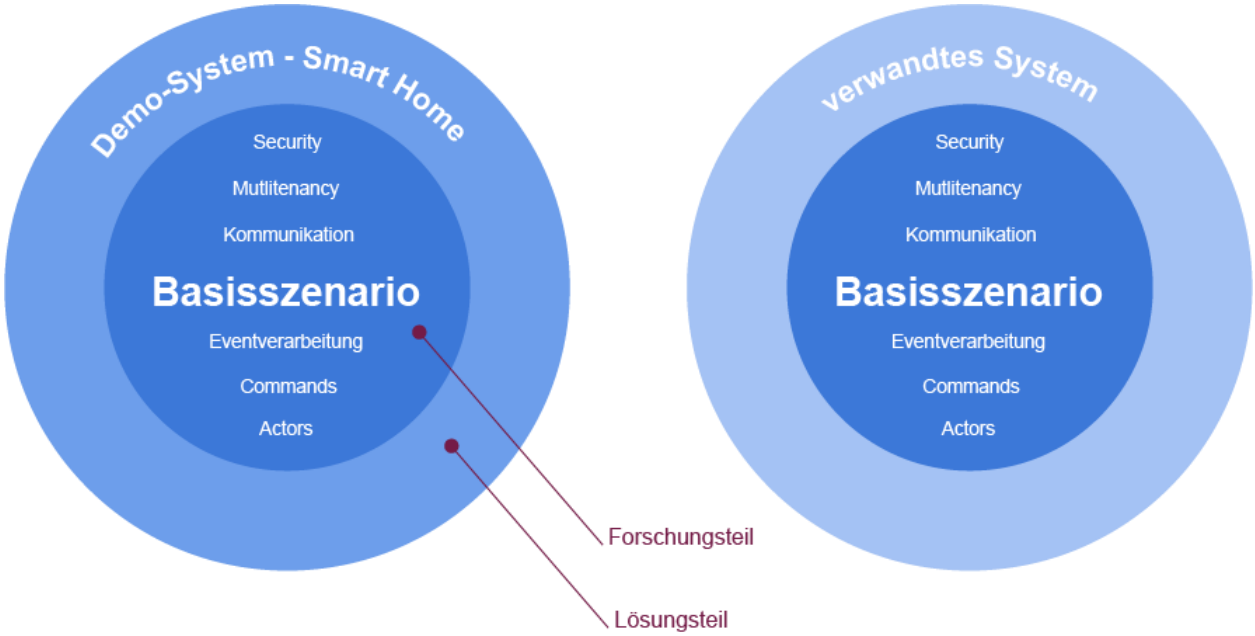

Abbildung 1 – Forschungs- und Lösungsteil

2 Problembeschreibung

2.1 Motivation

Die Arbeit hat ein bestehendes Demo-System zum Vorbild und soll dieses ablösen. Folgende Mängel gaben Anlass dazu:

- Fehlende Multitenancy, wodurch eine komplette Systeminstanz pro Endkunde eingerichtet werden musste
- Skalierbarkeit insbesondere auf Ebene der Eventverarbeitung unzureichend
- Kein wiederverwendbares Basisszenario, spezifische Details des Demo-System ziehen sich durch alle Schichten

2.2 Funktionale Anforderungen

2.2.1 Forschungsteil (Basisszenario)

Während der Analyse für das Basisszenario haben sich die folgenden Entitäten und Beziehungen in der Problemdomäne ergeben:

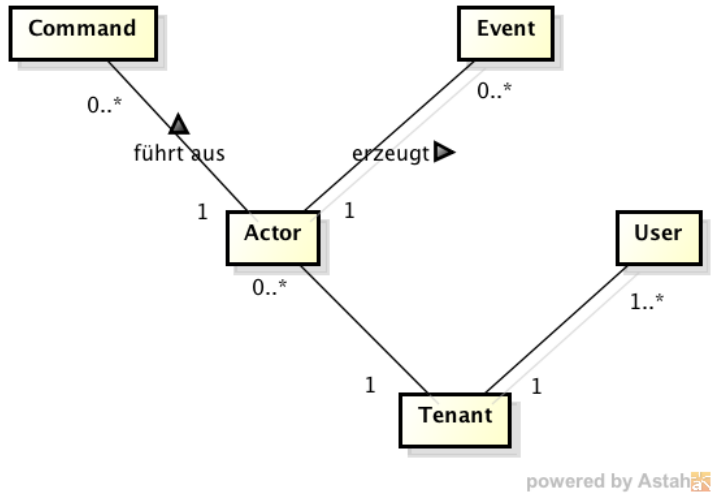

Abbildung 2 – Entitäten der Problemdomäne

Entität	Erklärung
Tenant	Ein Kunde (Organisation, Privatperson, etc) des Systembetreibers. Jeder Tenant hat seine eigene Sicht auf das System.
User	Benutzer, die einem Tenant zugewiesen sind. Der Tenant selbst ist kein User. Alle User handeln im Namen des Tenants.
Actor	Abstraktes Modell eines steuerbaren Subsystems. Ein Actor besitzt eine Schnittstelle, um ihm Befehle zu schicken. Der Actor kann das System über Ereignisse informieren.
Command	Ein Befehl, mit dem ein Actor gesteuert werden kann.
Event	Ein Ereignis beim Actor. Der Actor informiert das System über ein Ereignis.

Tabelle 1 – Legende zu Abbildung 2

F01: Command senden
Um einen Actor zu steuern sendet der User einen Command an das System. Die verfügbaren Commands sind nicht vom System vorgegeben, sondern werden vom Empfänger entsprechend interpretiert. Das System ist dafür verantwortlich den Command zuzustellen.

F02: Command verarbeiten
Actors können Commands über das System empfangen. Sie interpretieren einen Command und reagieren in selbstbestimmter Art und Weise darauf. Für jeden Command wird dem System eine Rückmeldung bekanntgegeben. Die Rückmeldung enthält Statusinformationen, die sich direkt auf den Command beziehen, zum Beispiel zu dessen Gültigkeit oder Vollständigkeit. Das System merkt sich den Zusammenhang zwischen ursprünglichem Command und Rückmeldung.

F03: Event senden
Actors können beliebige Events an das System senden, von dem sie an den richtigen Empfänger zugestellt werden. Events werden typischerweise als Folge von verarbeiteten Commands ausgelöst, wobei der Inhalt von Events nicht spezifiziert wird. Events sind nicht zu verwechseln mit Rückmeldungen auf Commands.

F04: Events abrufen
Der Benutzer kann Events seiner Actors über das System abrufen.

2.2.2 Lösungsteil (Demo-System)

L01: Rasensprinkler konfigurieren
Der Benutzer kann den Actor «Rasensprinkler» mit einem Command konfigurieren. Falls erwünscht kontrolliert der Rasensprinkler in definierten Zeitabständen die Bodenfeuchtigkeit. Falls diese unter einen Schwellwert fällt schaltet sich der Sprinkler so lange ein, bis die minimale Feuchtigkeit wieder erreicht ist. Der Sprinkler benachrichtigt das System über alle ausgeführten Schritte.

2.3 Nichtfunktionale Anforderungen

2.3.1 Multitenancy
Das System soll so ausgelegt sein, dass möglichst viele Ressourcen von mehreren Tenants genutzt werden können, ohne dass sich die Tenants gegenseitig beeinflussen. Das Hinzufügen neuer Tenants darf nicht dazu führen, dass eine zusätzliche Instanz des Systems installiert werden muss. Pro Tenant kann es mehrere User geben, die jedoch ausschliesslich zur Authentisierung dienen und auf denselben Daten operieren.

2.3.2 Sicherheit
Grundsätzlich muss gewährleistet sein, dass alle Daten vor unautorisierten Zugriffen geschützt sind. Sämtliche Kommunikation zwischen Clients und dem System muss verschlüsselt werden.

Vertraulichkeit
Alle Daten eines Tenants dürfen nur für die zugehörigen User sichtbar sein. Dazu gehören:

- Gesendete Commands
- Rückmeldungen auf Commands
- Gesendete Events
- Registrierte Actors

Integrität
Nur die authentisierten User eines Tenants können folgende Daten erzeugen, manipulieren und löschen:

- Commands
- Rückmeldungen auf Commands
- Events
- Actors

2.3.3 Skalierbarkeit

Die Skalierbarkeit stellt bezüglich Cloud ein wichtiges Thema dar. Gemäss Anforderungen soll das System erweiterbar sein. Diese Anforderung zieht mit sich, dass die Lösung gut skalieren muss, da mit weiteren Tenants, Actors und Sensoren immer mehr Last generiert wird.

2.3.4 Technologie

Die Aufgabenstellung dieser Studienarbeit gibt die Microsoft Azure Cloud als Technologie vor. Insbesondere soll ein EventHub für das Zwischenspeichern von Events verwendet werden.

3 Lösungskonzept

Dieses Kapitel beinhaltet die Beschreibungen der Architektur und wichtiger Komponenten. Die eingesetzten Technologien und genauen Implementierungsdetails stehen im Hintergrund und werden im Kapitel 4 aufgegriffen. Neben der Architektur beinhaltet das Lösungskonzept auch eine allgemeine Schnittstellenbeschreibung. Informationen zur technischen Anbindung an die Schnittstellen können dem Abschnitt **Error! Reference source not found.** im Anhang entnommen werden.

3.1 Allgemeine Systemsicht

Anhand der Problembeschreibung wurde ein Plan erarbeitet, der das ganze System verständlich beschreibt. Abbildung 3 stellt die wichtigsten Komponenten und Entitäten aus der Problemdomäne in gegenseitiger Beziehung dar.
Die User eines Tenants können über die Cloud mit ihren Actors gemäss den funktionalen Anforderungen interagieren.

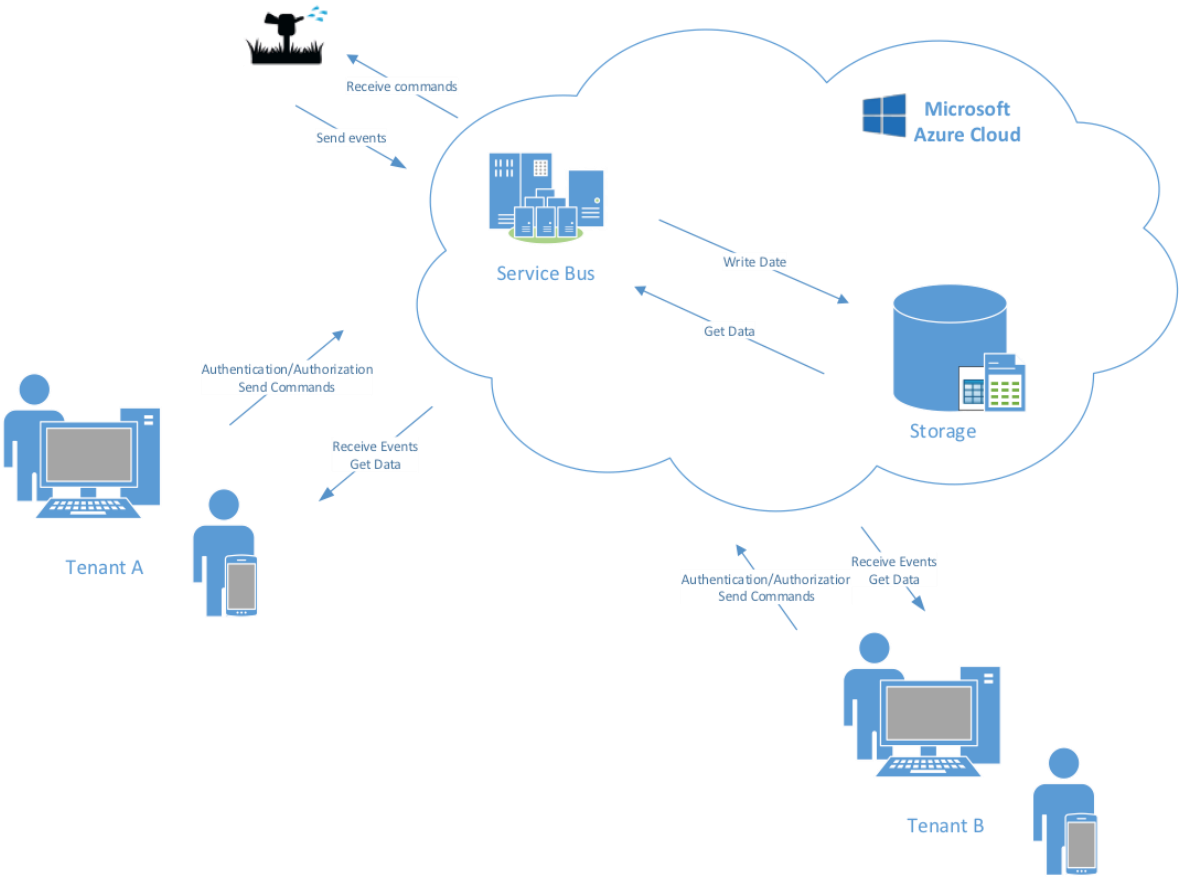

Abbildung 3 – Allgemeine Systemsicht

3.2 Systemabläufe

In Abbildung 4 sind die Systemabläufe anhand eines Beispielszenarios in einem Sequenzdiagramm erklärt. Das Szenario beinhaltet die Use-Cases F01 - F04. Die Cloud wird im Diagramm als Blackbox betrachtet.

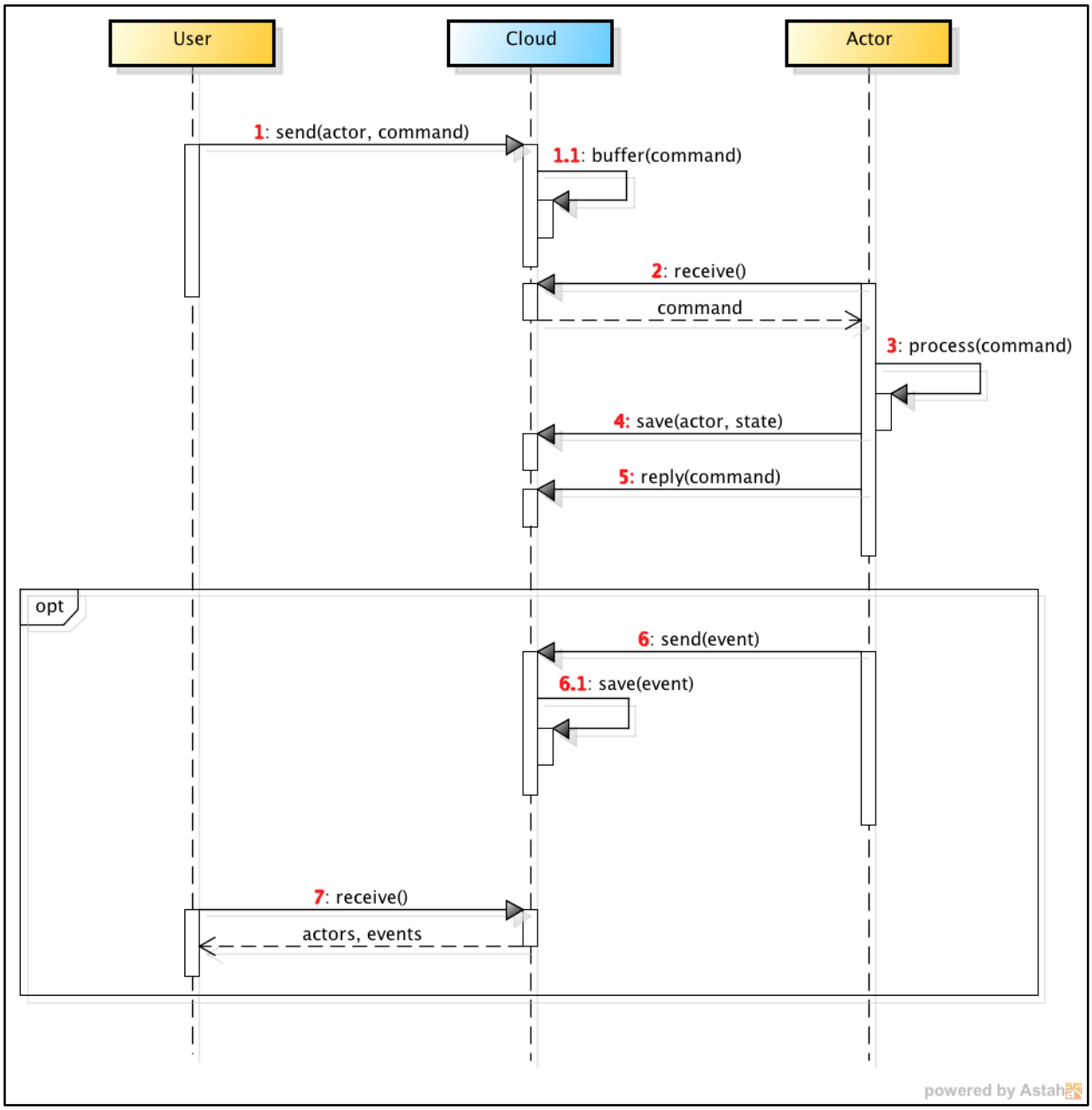

Abbildung 4 – Systemabläufe

Erklärung Sequenzdiagramm

1	User sendet einen Command an den Actor
1.1	Cloud speichert den Command in einem Buffer
2	Actor holt sich den Command aus dem Buffer der Cloud
3	Actor verarbeitet den Command

4	Actor speichert seinen konsistenten Zustand in der Cloud ab
5	Actor sendet eine Rückmeldung auf den Command
6	Actor sendet ein Event and die Cloud
6.1	Cloud speichert das Event ab
7	User fragt die Cloud nach neuen Daten ab

Tabelle 2 – Legende zu Abbildung 4

3.3 Schichten und Tiers

Der grösste Teil an Komponenten wird im Cloud Tier betrieben. Der andere Teil läuft auf einer lokalen Maschine. Abbildung 5 stellt die logischen Schichten und Tiers vereinfacht dar.

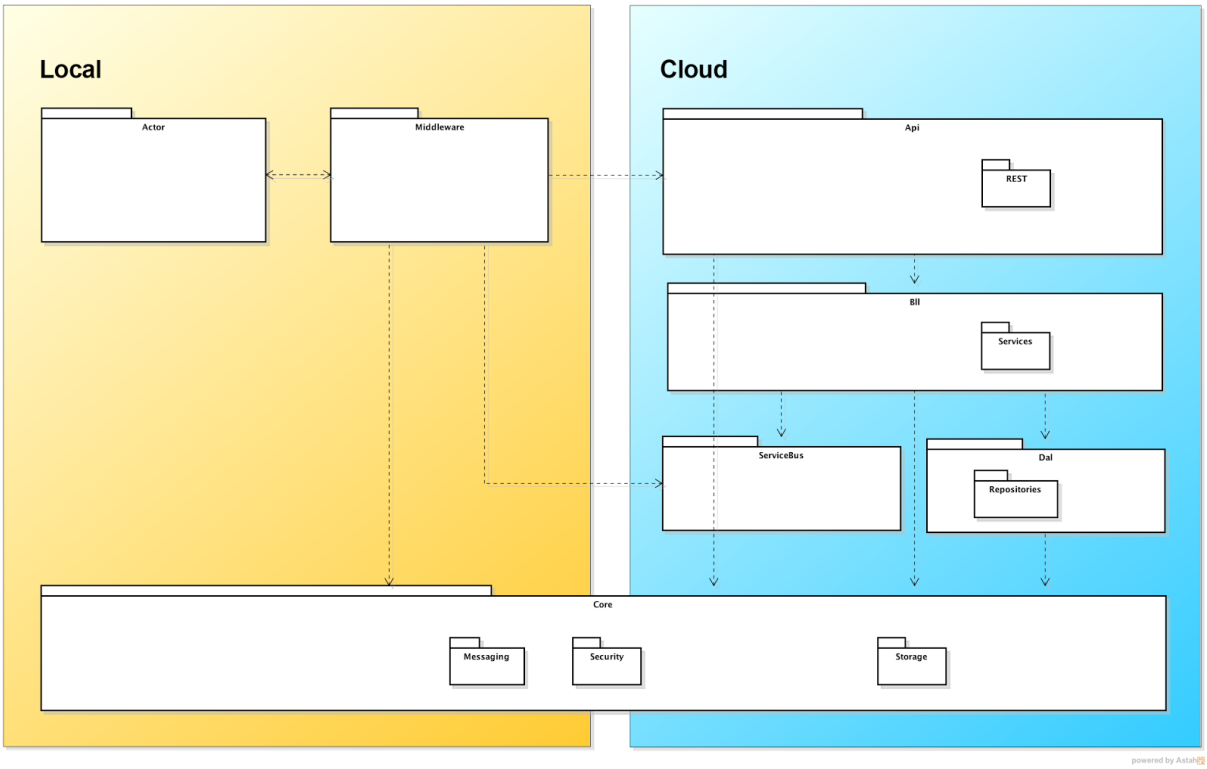

Abbildung 5 – Logische Schichten

Logische Schichten

Package	Beschreibung	Umgebung
Core	Stellt elementare Klassen und Konzepte zur Verfügung, die von allen anderen Packages genutzt werden.	Cloud/Local
Dal	Beschreibt Entitäten, die peristent gespeichert werden und bietet rudimentäre Operationen auf den Entitäten an.	Cloud
Bll	Enthält die Businesslogik, die von den oberen Schichten genutzt wird.	Cloud
Api	Schnittstelle zur Cloud.	Cloud
Actor	Ein Gerät, das über die Cloud gesteuert werden soll.	Local
Middleware	Eine Softwareschicht, die als Vermittler zwischen technologieunabhängigen Actors und der Cloud fungiert.	Local

Tabelle 3 – Legende zu Abbildung 5

3.4 Allgemeine Schnittstellenbeschreibung

Das Cockpit ist eine Anwendung, mit der ein User seine Actors einsehen und konfigurieren kann. Zusätzlich werden im Cockpit die Events der Actors aufgelistet.

Cockpit	
Schnittstelle	**Beschreibung**
GetEvents	Liefert Events aus dem Storage.
GetActors	Liefert eine Liste mit Actors.
GetActorDetail	Liefert die Details eines Actors.
PutConfig	Setzt die Konfiguration eines Actors.

Tabelle 4 – Schnittstellen Cockpit

Die Actors können mit Hilfe der Middleware auf die ClientApi der Cloud zugreifen. Die API bietet Methoden zur Verwaltung und Konfiguration der Actors.

ClientApi	
Schnittstelle	**Beschreibung**
PutActor	Registriert einen neuen Actor.
GetConfig	Liefert die Konfiguration eines Actors.
PutConfig	Setzt die Konfiguration eines Actors.
PutLocation	Setzt die Location eines Actors.

Tabelle 5 – Schnittstellen ClientApi

Queue	**Beschreibung**
Command	Auf dieser Queue werden Commands abgelegt.
Reply	Auf dieser Queue werden die Antworten bezüglich eines Commands abgelegt.

Tabelle 6 – Service Bus Queues

4 Umsetzung

4.1 Technologie und Plattform

Die Aufgabenstellung der Studienarbeit gibt Microsoft Azure als Cloud-Plattform vor. Dazu eine nicht abgeschlossene Liste der verwendeten Microsoft Technologien:

- Service Bus
- Azure Active Directory
- Azure Storage Table
- Web Role
- Worker Role
- Visual Studio

4.1.1 Service Bus

Der Service Bus ist ein Anwendungsdienst von Microsoft Azure. Der Service Bus erlaubt das effiziente und Technologieunabhängige Austauschen von Nachrichten zwischen verschiedensten Geräten, Anwendungen und Diensten. Für ein IoT-Szenario ist der Service Bus deshalb also sehr gut geeignet. Das Senden und Empfangen von Commands, sowie das Senden von Rückmeldungen auf Commands wird mit Service Bus Queues realisiert. Events werden an einen EventHub gesendet. Der EventHub ist äusserst leistungsfähig und kann eine sehr grosse Anzahl an Events pro Sekunde aufnehmen. Der EventHub erfüllt damit eine der wichtigsten Anforderungen des Basisszenarios.

4.1.2 Azure Active Directory

Das Basisszenario sieht eine Lösung zum Authentisieren und Autorisieren von Usern vor. Azure Active Directory ist eine cloudbasierte Identitäts- und Zugriffsverwaltung und damit eine gute Wahl für das System.

4.1.3 Azure Storage Table

Zum Speichern von Daten, die keinem streng relationalen Schema folgen müssen, haben wird die NoSQL Lösung «Azure Table Storage» gewählt. Die Tabellen können effizient abgefragt werden und skalieren nach Bedarf.

4.1.4 Web Role

Eine Azure Web Role ist ein Container für Instanzen von virtuellen Webservern. In einer Web Role kann beispielsweise eine ASP.NET MVC Anwendung betrieben werden. Da die Web Role eine logische Einheit ist, muss sich die Anwendung nicht damit beschäftigen, auf welcher Instanz sie ausgeführt wird. Bei stateful Anwendungen muss die Synchronisierung zwischen Instanzen selbst implementiert werden.

4.1.5 Worker Role

Eine Azure Worker Role ist genau wie die Web Role ein Container. Jedoch nicht für Webserver sondern eher für einen Dienst, der permanent in der Cloud läuft. Instanzen von Worker Roles und Web Roles können über den Service Bus miteinander kommunizieren.

4.1.6 Visual Studio

Als Entwicklungsumgebung wurde Visual Studio 2013 eingesetzt. Die Solution gliedert sich in mehrere Teilprojekte, die im Abschnitt 4.2 erklärt werden.

4.2 Visual Studio Projekte

4.2.1 Saiot.Core

Dieses Library-Projekt setzt die Core-Schicht aus Abbildung 5 um. Alle anderen Projekte hängen vom Projekt Saiot.Core ab.

Namespace	Inhalt
Saiot.Core.Messaging	Hilfsklassen und Abstraktionen zur Kommunikation mit dem Azure ServiceBus.
Saiot.Core.Security	Klassen und Interfaces zur Unterstützung von Authentisierung und Autorisierung.
Saiot.Core.Storage	Komponenten um die Arbeit mit Azure Storage Tables zu erleichtern.

Tabelle 7 – Namespaces Saiot.Core

4.2.2 Saiot.Dal

Data Access Layer gemäss dem Lösungskonzept. Im Library-Projekt Saiot.Dal werden die Abfragen an Azure Storage Tables entkoppelt. Operationen zum Erzeugen, Finden, Aktualisieren oder Löschen von Einträgen sind über einfache Schnittstellen aufrufbar.

Namespace	Inhalt
Saiot.Dal.TableEntities	Definitionen der verwendeten Entitäten.
Saiot.Dal.Repositories	Table Storage Operationen in Repository-Klassen für jede Entität.

Tabelle 8 – Namespaces Saiot.Dal

4.2.3 Saiot.Bll

Der Business Logic Layer ist ein Library-Projekt, das als aufbauenden Layer für das Projekt Saiot.Dal eingesetzt wird. Die Schnittstellen des Business Logic Layer sind nicht mehr so allgemein wie im Data Access Layer, sondern sind sehr nahe an der Problemdomäne. Die Verwender von Saiot.Bll haben keine Abhängigkeit von Saiot.Dal, sodass die Technologie zur Persistenz ausgetauscht werden könnte. Dazu werden die Entitäten aus dem Namespace Saiot.Dal.TableEntities in Data Transfer Objects konvertiert.

Namespace	Inhalt
Saiot.Bll.Services	Klassen mit Methoden für die Geschäftslogik. Nutzt die Repository-Klassen von Saiot.Dal.Repositories.
Saiot.Bll.Dto	Transferobjekte, die von den oberen Schichten genutzt werden.

Tabelle 9 – Namespaces Saiot.Bll

4.2.4 Saiot.WebRole.Common

In diesem Library-Projekt werden Komponenten definiert, die von mehreren ASP.NET MVC Projekten genutzt werden können.

Namespace	Inhalt
Saiot.WebRole.Common.Controllers	ASP.NET Controller-Klassen, die in anderen Projekten genutzt werden können.

Tabelle 10 – Namespaces Saiot.WebRole.Common

4.2.5 Saiot.WebRole.Cockpit

Eine ASP.NET MVC[1] Single-Page Anwendung, die als Web Role in Windows Azure betrieben wird. Die Anwendung dient den Usern als Schnittstelle um mit ihren Actors zu interagieren. Ein Controller[2] verarbeitet den Request und liefert eine HTML-Seite als Response. Alle weiteren Requests erfolgen asynchron an mehrere ApiController[3]. Die Anwendung ist in der Abbildung 1 dem Demo-System zuzuordnen und weniger dem Basisszenario.

Mit Hilfe des User Interface können die registrierten Actors konfiguriert werden. In der Anwendung sind zudem die letzten Events aller Actors eines Tenants aufgelistet.

Die Authentisierung der Benutzer erfolgt über einen Token, der durch den Azure ActiveDirectory Identity Provider ausgestellt wurde. Alle Verbindungen sind TLS geschützt.

Alle ApiController verwenden die Service-Klassen aus Saiot.Bll.Services und stammen von der Klasse Saiot.WebRole.Common.Controllers.ProtectedServiceController ab.

4.2.6 Saiot.WebRole.ClientApi

Im Unterschied zum Cockpit ist die ClientApi ein reines WebApi Projekt und hat kein User Interface. Mit Hilfe der ClientApi Methoden können die Actors mit dem System gemäss der API-Beschreibung im Anhang kommunizieren. Die ClientApi ist Bestandteil des Basisszenario aus Abbildung 1 und ist vom Demo-System weitgehend unabhängig.

Die ClientApi ist nur für authentifizierte Nutzer verfügbar. Die Authentisierung geschieht über einen Token. Da die ClientApi auch Methoden für Zugangsdaten besitzt ist die Verbindung TLS geschützt.

Alle ApiController verwenden die Service-Klassen aus Saiot.Bll.Services und stammen von der Klasse Saiot.WebRole.Common.Controllers.ProtectedServiceController ab.

4.2.7 Saiot.DeviceController

Der DeviceController ist eine Konsolenapplikation und dient als Middleware zwischen Actors und dem System. Der DeviceController empfängt und interpretiert Commands, sodass diese von den Actors ausgeführt werden können. Die Commands werden als Strings an den DeviceController geschickt und folgen einer gewissen Syntax. Weitere Informationen, zu den Commands sind im Anhang «Erläuterungen zum Code» vorhanden.

Die (virtuellen) Actors können den DeviceController über Ereignisse informieren, die der DeviceController dann als Events an den EventHub im ServiceBus der Cloud sendet. Eine weitere Aufgabe des DeviceController ist Anbindung an die ClientApi. Dazu muss er sich vom Azure ActiveDirectory Identity Provider zunächst einen Token ausstellen lassen.

[1] http://www.asp.net/mvc
[2] http://www.asp.net/mvc/overview/getting-started/introduction/adding-a-controller
[3] http://www.asp.net/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api

4.2.8 Saiot.WorkerRole.EventProcessor

Der ServiceBus EventHub kann sehr viele Events unter grosser Last zwischenspeichern. Irgendwann ist der EventHub jedoch voll oder die Events erreichen ihr Ablaufdatum. Die Worker Role «EventProcessor» läuft in der Cloud und liest permanent Events aus dem EventHub aus. Anschliessend werden die Events in Dto's konvertiert und mit Hilfe des Business Logic Layer (Saiot.Bll) persistiert.
Das effiziente Verarbeiten von Events stellt eine wichtige Anforderung an das System dar und wurde in der Aufgabenstellung explizit gefordert. Im Anhang **Error! Reference source not found.** wird genauer erklärt, wie der EventProcessor funktioniert.

4.2.9 Saiot.WorkerRole.Correlation

Wenn der DeviceController einen Command empfangen hat wird er versuchen, den Command von dem richtigen Actor ausführen zu lassen. Dabei können beliebige Fehler auftreten. Deshalb muss dem System eine Rückmeldung zum Ausführungsstatus des Commands geschickt werden. Die Rückmeldung wird in die Reply-Queue geschoben. Die Worker Role «Correlation» fragt die Reply-Queue kontinuierlich ab speichert die Rückmeldung zusammen mit dem ursprünglichen Command ab. Somit ist protokolliert, welche Commands gesendet wurden und wie vom DeviceController darauf reagiert wurde.

4.3 Deploymentübersicht

Abbildung 6 zeigt das Deployment der Projekte und Komponenten in der Azure Cloud. Die Nodes und Artefakte sind anhand der Visual Studio Projekte benannt. Für den Service Bus und die Storage Tabellen gibt es kein Äquivalent in der Visual Studio Solution. Die Nummerierung im Diagramm bezieht sich auf die Systemabläufe aus dem Abschnitt 3.2.

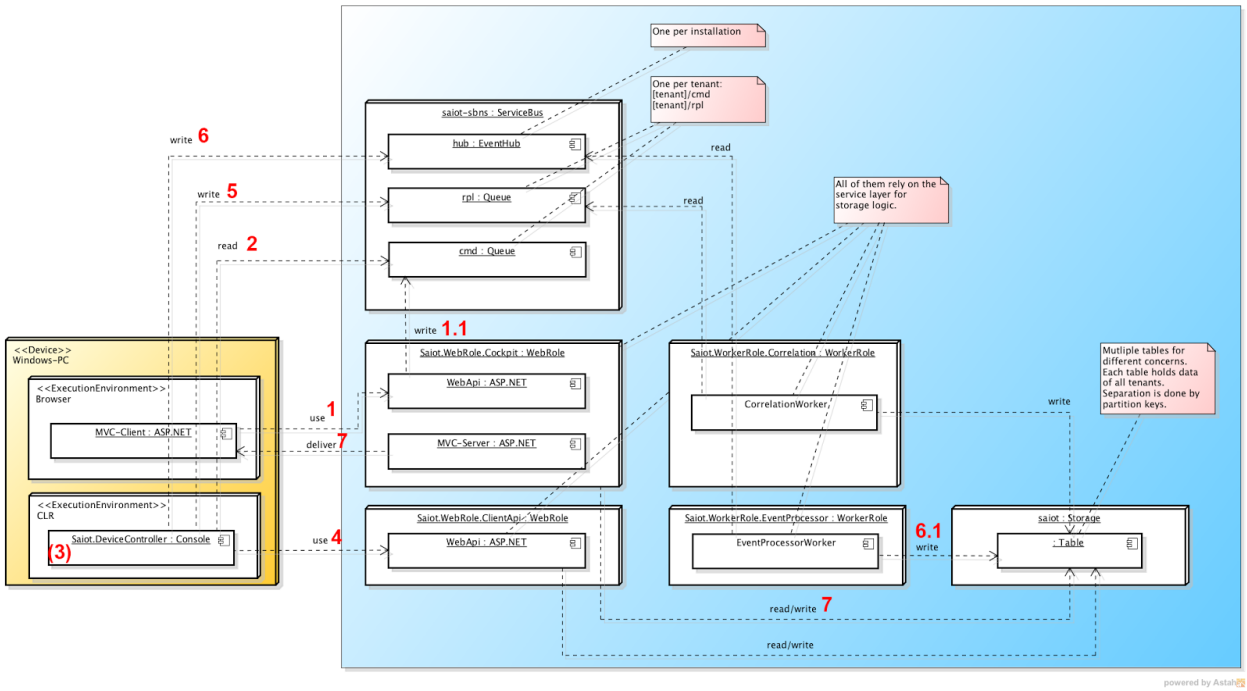

Abbildung 6 – Deploymentübersicht

1	User sendet einen Command-String an die Cockpit WebApi
1.1	Service sendet den Command auf die cmd-Queue
2	DeviceController liest den Command aus der cmd-Queue
3	DeviceController verarbeitet den Command
4	DeviceController speichert den Zustand des Actors
5	DeviceController sendet Status des Commands rpl-Queue
6	DeviceController sendet Event an EventHub
6.1	EventProcessor speichert Event in Storage Table
7	Cockpit Website lädt Events und Storage Table und zeigt sie dem User an

Tabelle 11 – Legende zu Abbildung 6

4.4 Multitenancy

Da nicht für jeden Tenant eine eigene, isolierte Installation des Systems aufgesetzt wird, müssen Massnahmen ergriffen werden, um die Tenants voneinander abzuschirmen. Der erste Schritt dabei ist, dass jedem Tenant eine systemweit eindeutige GUID zugewiesen wird, die im weiteren «TenantId» genannt. Anhand der TenantId können die Ressourcen eindeutig adressiert werden.

4.4.1 Azure Storage

Das System verwendet «Azure Storage Tables[4]» um die logischen Entitäten «Actor», «CommandCorrelation», «Event», «Tenant», und «Credential» permanent zu speichern. Global existiert pro logischer Entität nur eine einzige Tabelle und in dieser Tabelle werden die Einträge aller Tenants zur jeweiligen Entität gespeichert. In der Event-Tabelle befinden sich demnach die Event-Einträge aller Tenants. Eine Azure Storage Table schreibt für jeden Eintrag mindestens die Felder «PartitionKey» und «RowKey» vor. Zusammen bilden die beiden Attribute einen Primärschlüssel. In der Tabelle darf der gleiche PartitionKey für beliebig viele Einträge verwendet werden. Der RowKey hingegen darf in Kombination mit einem PartitionKey nur einmal vorkommen. Dazu ein Beispiel:

PartitionKey	RowKey
1	1
1	2
2	1
2	2

Tabelle 12 – Beispiel Schlüsselbildung

Dieses Schema zur Schlüsselbildung eignet sich hervorragend um die Daten mehrerer Tenants zu trennen. Sobald ein Eintrag für einen bestimmten Tenant gespeichert werden muss, wird die zugehörige TenantId als Wert für das Feld PartitionKey eingetragen. Pro Eintrag wird zusätzlich eine weitere GUID als RowKey generiert, sodass der Eintrag innerhalb des Tenants garantiert eindeutig ist. Im äusserst unwahrscheinlichen Fall, dass die gleiche Kombination aus PartitionKey und RowKey schon existiert, wird eine Exception ausgelöst.

Ausser den beiden Werten für den PartitionKey und den RowKey dürfen für jeden Eintrag weitere Felder definiert werden. Das ist für die Multitenancy von Vorteil, da keine Normalform erreicht werden muss. Sollte also ein einzelner Tenant zusätzliche Spalten benötigen, so können diese dynamisch hinzugefügt werden. Azure Storage Tables sind dafür gedacht denormalisierte Daten effizient zu verwalten.

Allein das beschriebene Schema zur Bildung der Schlüssel bietet aber noch keine nennenswerte Vorteile gegenüber einer herkömmlichen relationalen Datenbank, in der ein Fremdschlüssel den Tenant referenziert. Interessant wird es erst, wenn es um die Zugriffsrechte geht. Für jeden Tenant sind letztlich nur diejenigen Einträge sichtbar, die seine TenantId als PartitionKey eintragen haben, ohne dass speziell nach der TenantId selektiert werden muss. Wie das genau implementiert wird, ist im Abschnitt 4.1 beschrieben.
Mehr zur Partitionierung von Azure Storage Tables: http://bit.ly/1wlfGPy

[4] http://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-how-to-use-tables/

4.4.2 ServiceBus

Auf ServiceBus-Ebene wird eine hybride Lösung zur Erreichung der Multitenancy eingesetzt.
Im System existiert nur ein einziger EventHub, der die Events aller Tenants entgegennimmt. Um zu wissen, welche Nachricht im EventHub zu welchem Tenant gehört, kann im Event das Feld «PartitionKey» abgefragt werden. Das Abfüllen des PartitionKeys in das Event-Objekt geschieht automatisch, da die TenantId Teil des Endpoints für den EventHub ist.

Genau wie die Events werden auch Commands und Rückmeldungen von Commands über den Azure ServiceBus verschickt. Das Datenvolumen fällt im Gegensatz zu den Events aber sehr viel kleiner aus. Optimal wäre es, man hätte einen gemeinsamen Buffer für alle Tenants, sodass alle Commands dort zwischengespeichert würden. Am anderen Ende, nämlich dort wo die Commands empfangen werden, müssten die Commands wieder an den richtigen Tenant ausgeliefert werden. Im Azure ServiceBus gibt es genau zwei Konzepte, welche die genannten Anforderungen erfüllen: Topics/Subscriptions und der EventHub. Jedoch haben beide Konzepte gewisse Einschränkungen, sodass weder Topics/Subscriptions noch der EventHub für das Verschicken von Commands in Frage kommt.

Konzept	Vorteile	Problem
Topics/Subscriptions	Das Topic ist eine Queue, die Commands von allen Tenants entgegennimmt. Für jeden Tenant kann eine Subscription so konfiguriert werden, dass nur Commands, die einem Filterkriterium entsprechen, in die Subscription gelangen. Jeder Tenant fragt anschliessend seine Subscription nach Commands ab.	Damit die Commands gefiltert werden können, muss ein Flag für die TenantId beim Abschicken des Commands manuell gesetzt werden. Es gibt keine Möglichkeit zu verhindern, dass einfach eine falsche TenantId eingetragen wird und dadurch Integrität einer Nachricht verloren geht.
EventHub	Die Commands können als Events an den gemeinsamen EventHub geschickt werden. Durch eigene Endpoints für jeden Tenant kann der Absender der Nachricht identifiziert werden.	Wenn eine Nachricht auf dem EventHub gelesen wird, wird die Nachricht nicht auf dem EventHub gelöscht. Das Auslesen des EventHubs kann nur für 32 Tenants so erfolgen, dass keine fremden Nachrichten sichtbar sind. Damit ist auch der EventHub für die Commands ungeeignet.
Queue pro Tenant	Dank eigener Endpoints zum Senden und Empfangen von Commands sind alle Tenants voneinander isoliert. Einfache und solide API.	Das Volumen an Commands pro Tenant bzw. pro Queue wird in den meisten Fällen eher gering sein. Dadurch liegen möglicherweise grosse Teile der Queue brach.

Tabelle 13 – Service Bus Konzepte

Im Anschluss an die Überlegungen aus der obigen Tabelle wurde entschieden, dass die Lösung mit eigenen Queues pro Tenant zum Einsatz kommt, denn die sicherheitsrelevanten Eigenschaften müssen in jedem Fall erfüllt sein.

4.5 Security

4.5.1 Authentifizierung am Azure Active Directory

Der Zugriff auf die Webapplikation wird durch die Authentifizierung am Azure Active Directory sichergestellt. Dazu wurde das WIF (**W**indows **I**dentity **F**oundation) Framework eingesetzt. Folgende Grafik zeigt den Ablauf der Authentifizierung:

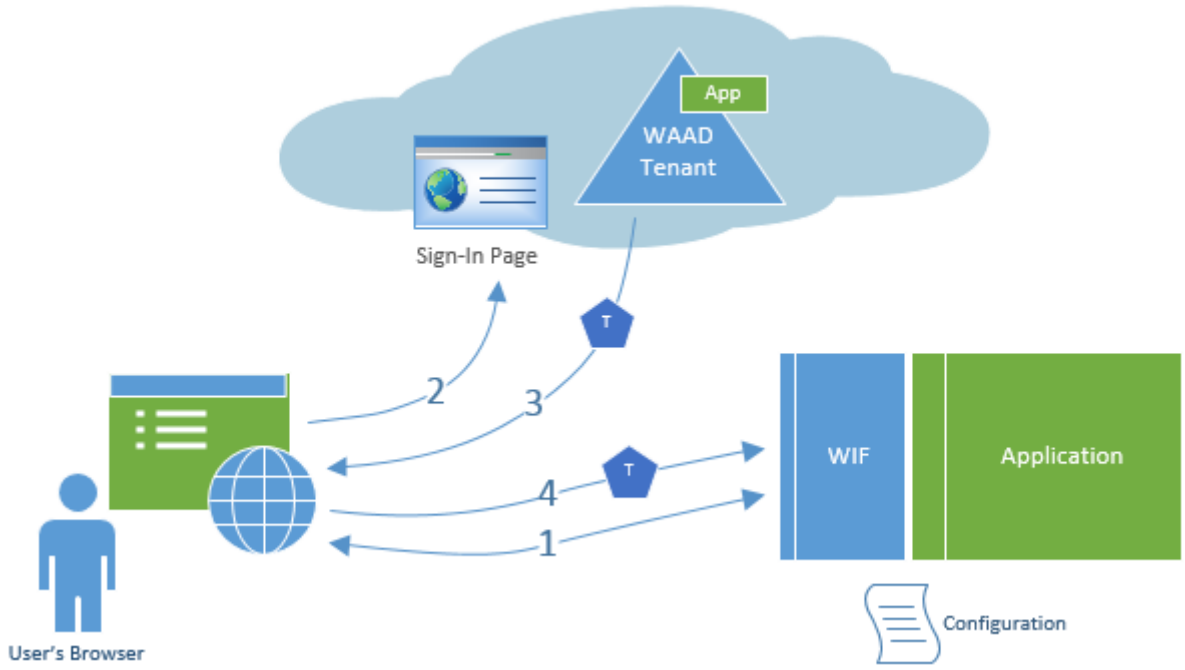

Abbildung 7 – Übersicht Authentifizierung

Der Benutzer meldet sich bei der Webapplikation an und wird an eine Sign-In Page von Microsoft weitergeleitet. Diese vergleicht die Credentials mit dem Active Directory und sendet bei positivem Resultat einen Token zurück. Mit diesem Token kann sich nun der Client beide Web Applikation anmelden. Die Webapplikation speichert den Token in einer SQL Datenbank mit dem zugehörigen Tenant.

Die Kommunikation zwischen der Webapplikation und dem Active Directory wird über TLS verschlüsselt.
Dazu muss dem Cloud Service im Management Portal ein Zertifikat hochgeladen werden. Wir haben ein Self Signed Certificate eingesetzt. Es stellt die gleiche Sicherheit zur Verfügung, wird aber nicht öffentlich verifiziert. Daher wird es auch vom Browser als nicht vertrauenswürdig eingestuft und muss manuell akzeptiert werden.

Das Zertifikat kann auf verschieden Arten[5] erzeugt werden. Es muss jedoch Schlüsselaustausch unterstützen und daher einen Public Key sowie einen Private Key besitzen. Es muss mindestens 2048 Bit verschlüsselt unterstützen und der Subject Name muss mit der Domain der Website übereinstimmen.

Bei der Erzeugung der MVC Web Applikation ist es wichtig, dass der Typ der MVC-Applikation richtig gewählt wird, sodass dieser Multitenancy unterstützt. So erstellt VisualStudio die nötigen Komponenten mit deren Dependencies.

[5] http://azure.microsoft.com/en-us/documentation/articles/web-sites-configure-ssl-certificate/

Dabei wird auch die gesamte Logik für die Persistierung des Tenants und des Tokens vorgegeben. Dabei muss dem DB-Context nur noch der Connection String der SQL-Datenbank mitgegeben werden.
Wenn sich nun ein Benutzer anmeldet, wird der Tenant mit dem Token in der Datenbank abgelegt, zusammen mit dem Ablaufdatum des Tokens. Wenn der Token nicht mehr gültig ist, wird die Datenbank aufgeräumt und der Token gelöscht.

4.5.2 Sicherung der Web API

Die API ist gegen unautorisierten Zugriff geschützt, in dem man die Controller mit dem Attribut «Authorize»[6] anreichert. Dadurch wird gewährleistet, dass nur Benutzer, die den Authentifizierungs- und Authorisierungs-Ansprüchen gerecht werden.

4.5.3 Sicherung des Service Bus

Der Zugriff auf den Service Bus wird über Shared Access Signatures[7] (SAS) geregelt. Die Signatur ist ein String und enthält alle nötigen Informationen für die Authentisierung als auch für die Autorisierung. Alle Informationen, die zur Generierung einer solchen Signatur benötigt werden sind in der Storage Tabelle «Credentials» hinterlegt. Die Service-Klasse «TenantService» kann auf die Tabelle zugreifen und die Signatur erzeugen.

```
string sas = tenantService.GetServiceBusSignature("xy", "hub",
"DeviceController");
```

Beim EventHub erhält jeder Tenant einen eigenen Endpoint, der in die Signatur einfliesst[8].

4.5.4 Sicherung des Storage Tables

Ähnlich zu den Shared Access Signatures vom Service Bus gibt es auch für Storage Tables vergleichbare Signaturen[9] in Form von Strings. Mit den Signaturen kann gesteuert werden, auf welche Teile einer Tabelle zugegriffen werden kann. Zusätzlich wird auch definiert, ob gelesen, geschrieben, verändert oder gelöscht werden darf. Azure Storage Tables können anhand eines SAS Tokens instanziert werden, sodass der Instanz nur die eingeschränkten Operationen zur Verfügung stehen.

4.5.5 Sicherung des Tenants

Wie bereits erwähnt werden die Web Roles mit Hilfe von Azure Active Directory geschützt. Die Authentisierung an sich regelt aber noch nicht den Zugriff auf die Daten des jeweiligen Tenants. Jeder Tenant darf nur auf seine eigenen Daten zugreifen, die aber in derselben Tabelle gespeichert sind wie die Daten anderer Tenants. Dazu wird der Zugriff direkt mittels SAS Tokens auf die entsprechenden Bereiche der Tabelle eingeschränkt.

Die Repositories im Data Access Layer benötigen eine Tabelleninstanz, die aber nicht von den Repositories selbst erzeugt werden kann. Die Instanz kann jedoch über ein «IStorageProvider» Objekt angefordert werden. Beim Anfordern der Instanz muss das Repository dem IStorageProvider die benötigte Tabelle in Form eines TableClaims beschreiben.

[6] http://msdn.microsoft.com/en-us/library/system.web.mvc.authorizeattribute(v=vs.118).aspx
[7] http://msdn.microsoft.com/en-us/library/azure/dn170477.aspx
[8] http://msdn.microsoft.com/en-us/library/azure/dn789974.aspx
[9] http://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-shared-access-signature-part-1/

Ein TableClaim enthält den Tabellenname, Berechtigungen und den PartitionKey (TenantId). Das TableClaim-Objekt hat zudem eine Property «IsAuthorized». Nur falls diese Property true ist kann der TableClaim zum Erzeugen der Signatur verwendet werden. Damit der Claim autorisiert wird, muss man in der Methode TableClaim.AuthorizeWith ein spezielles Objekt mitgegeben werden, welches das Interface IClaimAuthorizer implementiert. Das ClaimAuthorizer-Objekt muss dafür qualifiziert sein zu entscheiden ob der TableClaim gültig ist oder nicht.

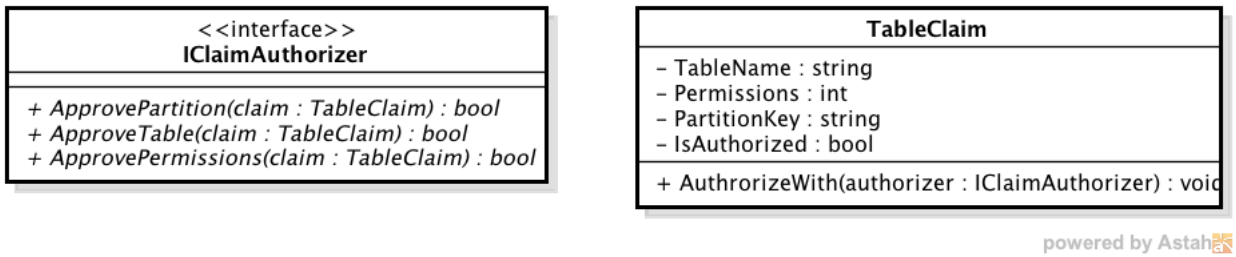

Abbildung 8 – IClaimAuthorizer und TableClaim

```
claim.AuthorizeWith(qualifiedObjectThatImplementsIClaimAuthorizer);
```

Ein qualifiziertes Objekt zum Autorisieren eines Claims ist typischerweise ein ASP.NET MVC Controller. Dieser kann anhand des angemeldeten Benutzers überprüfen, ob der Zugriff legitim ist.

Durch dieses Design wird verhindert, dass ein unautorisierter Zugriff auf die Daten eines Tenants technisch überhaupt möglich ist. Solange man sich an diesen Workflow hält, kann es nicht passieren, dass eine unautorisierte Anfrage versehentlich ausgeführt wird.

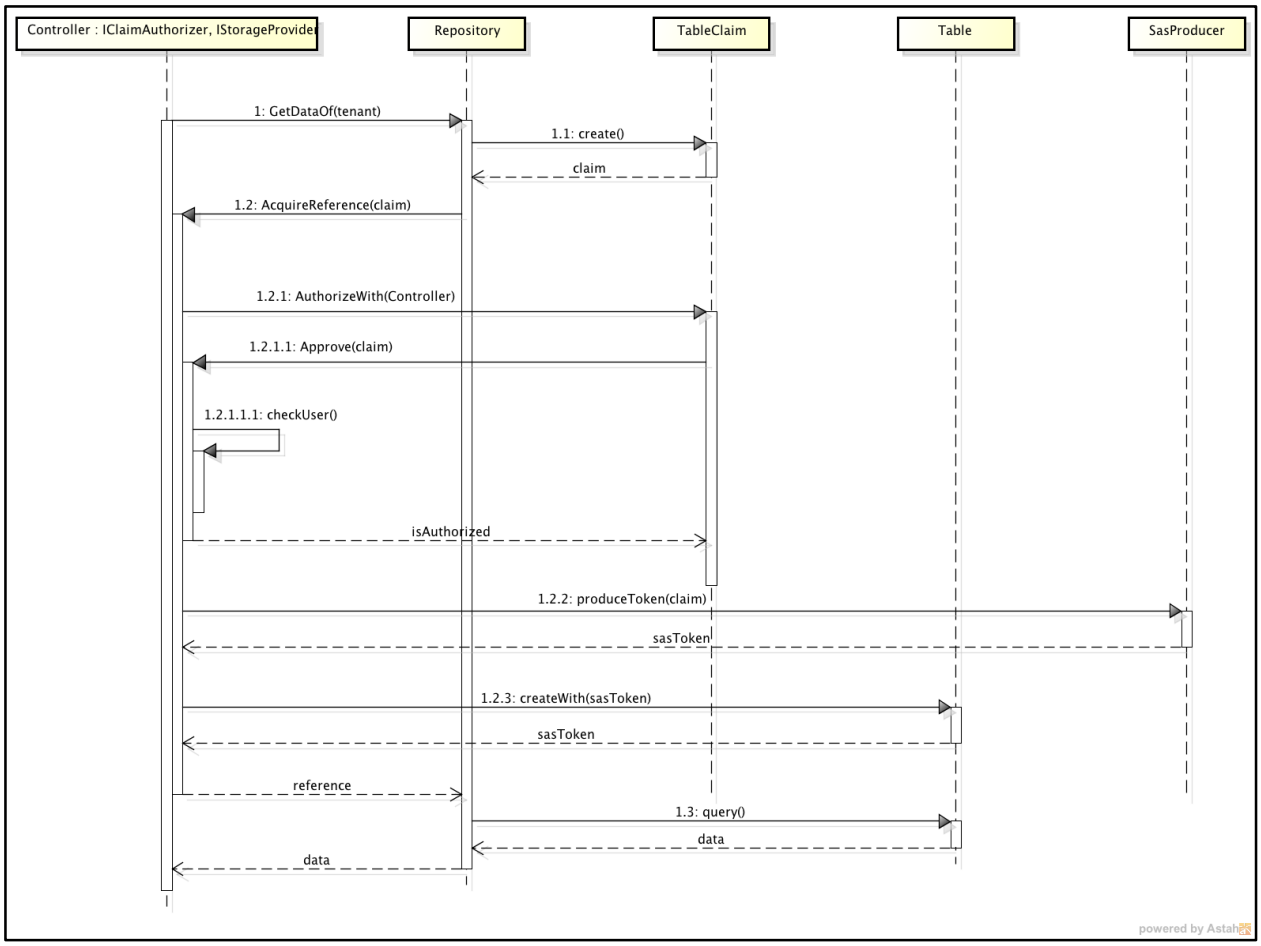

Abbildung 9 – Workflow TableClaim

4.6 Web Role / RESTful API

Für den Zugriff auf die persistierten Events und das Verwalten von Actors werden Web Roles verwendet, die eine RESTful API anbieten. Nachfolgend werden die verschiedenen Typen von Methoden beschrieben, die eingesetzt werden. Die komplette API Spezifikation befindet sich im Anhang.

Diese Methoden liefern Ressourcen eines bestimmten Tenants. Gemäss den RESTful Prinzipien kann jede Ressource adressiert werden. Ob die Ressource jedoch auch zurückgegeben wird, entscheidet dann die Zugriffsberechtigung in den unteren Layern.

4.6.1 GetEvents

Diese Methode gibt die Events eines Tenants zurück. Die Identifikation des Tenants wird als Parameter übergeben.

Controller

```
public IHttpActionResult GetEvents(string tenantId)
        {
            return PerformClaimBasedOperation(() =>
            {
                return Ok(eventService.GetRecentEvents(tenantId));
            });
        }
```

Service

```
public IEnumerable<EventDto> GetRecentEvents(string TenantId)
        {
            return (from row in EventRepository.FindEvents(TenantId, 10)
                    select new EventDto(row)).ToList();
        }
```

Repository

```
public IEnumerable<EventEntity> FindEvents(string tenantId, int limit = -1)
        {
            TableQuery<EventEntity> query = new TableQuery<EventEntity>()
                .Where(TableQuery.GenerateFilterCondition(
                                        "PartitionKey",
                                         QueryComparisons.Equal,
                                         tenantId
                                    )
                        );
            eventTableClaim.PartitionKey = tenantId;
            var table = StorageProvider.AcquireReference(eventTableClaim);
            return table.ExecuteQuery(query)
                        .OrderByDescending(c => c.Timestamp).Take(limit);
         }
```

Tabelle 14 – Web Role API GetEvents

In diesem Beispiel wird der Tenant dem Service übergeben. Der Service ruft die Methode «FindEvents()» aus dem Repository auf. Diese Methode führt die Abfrage auf dem Table Storage aus und gibt die Tabelle mit den entsprechenden Einträgen zurück. Im Service wird daraus ein «EventDto» Objekt erstellt. Aus diesem Objekt wird eine Liste erstellt und dem Controller zurückgegeben. Dieser wiederum gibt die Liste dem Client zurück mit dem Statuscode 200 OK.

4.6.2 PutConfig

Diese Methode aktualisiert die Konfiguration eines Actors. Als Parameter wird die Identifikation des Tenants, des Actors und die Konfiguration übergeben.

Controller

```
public IHttpActionResult PutConfig(
              string tenantId,
              string actorId,
              [FromBody] AbstractActorDto dto
          ) {
            return PerformClaimBasedOperation(() =>
            {
                var actor = actorService.GetActor(tenantId, actorId);
                var command = string.Format(
                        "set-config -name \"{0}\" -Location \"{1}\"
                        -config \"{2}\"",
                        actor.Name,
                        actor.Location,
                        Convert.ToBase64String(
                             Encoding.UTF8.GetBytes(dto.Config)
                        )
                 );
                commandService.SendCommand(tenantId, command);
                return Ok();
            });
        }
```

Tabelle 15 – Web Role API PutConfig

Aus den übergebenen Parametern wird ein Command zusammengesetzt, der dem Command-Service übergeben wird. Sofern der Command gesendet werden kann, wird ein 200 OK zurückgegeben.

Service

```
public CommandCorrelationDto SendCommand(string tenantId, string command)
        {
            var guid = Guid.NewGuid().ToString();
            var helper = getHelper(tenantId, "cmd");
            helper.SendMessage(new BrokeredMessage(command) {
                                        Label = guid
                                    }
            );
            if (helper.SenderState != ConnectionState.Success)
                    throw new Exception("failed sending command");
            var correlation = new CommandCorrelationEntity
            {
                PartitionKey = tenantId,
                RowKey = guid,
                Cmd = command
            };
            CommandRepository.InsertOrUpdate(correlation);
            return new CommandCorrelationDto(correlation);
        }
```

Tabelle 16 – Web Role API PutConfig

Die SendCommand-Methode verwendet eine Helper-Klasse, die den nötigen Kontext liefert. Dazu muss der Tenant und die zu verwendende Queue mitgegeben werden. An die Queue wird nun eine Brokered Message gesendet.
Für die Abgleichung der Ankunftsbestätigung des Commands beim Device Controller wird einen

Eintrag im Correlation Table-Storage erstellt.

QueueHelper

```
public void SendMessage(BrokeredMessage message)
        {
            SenderState = ConnectionState.Sending;
            try
            {
                InternalSend(message);
                SenderState = ConnectionState.Success;
            }
            catch
            {
                SenderState = ConnectionState.Failed;
            }
        }
protected virtual void InternalSend(BrokeredMessage message)
        {
            Client.Send(message);
        }
```

Tabelle 17 – Web Role API PutConfig

Die Helper Klasse ruft die Methode «InternalSend()» auf und überlässt dieser das Senden der Message an den Client. Danach wird der entsprechende Status gesetzt.

4.6.3 Verwendung der API

Die Verwendung der API wird hier durch ein JavaScript Beispiel gezeigt.

GetEvents

```
function getEvents() {
    $.getJSON("/api/events/" + viewbagTenant)
        .done(function (data) {
            $.each(data, function () {
                $('#entry').append(formatEventData(this)).data(data);
            });
        })
        .fail(function (jqXHR, textStatus, err) {
           //Error handling
        });
}
```

Tabelle 18 – Verwendung API GetEvents

Die Funktion «getEvents()» ruft über das jQuery Framework die RESTful API auf und übergibt den Tenant über den ViewBag von ASP.NET.
Die erhaltenen Daten werden für eine HTML-Liste (Bootstrap) formatiert und in die Seite eingefügt.
Die Fehlerbehandlung wurde hier Bericht ausgeklammert aufgrund besserer Übersicht und deren Trivialität. Im Code ist sie jedoch einsehbar.

PutConfig

```
function putConfig(actorId) {
    var mode = document.getElementById("on").checked == true ? "on" : "off";

    var config = {
        level: document.getElementById("level").value,
        radius: document.getElementById("radius").value,
        threshold: document.getElementById("threshold").value,
        interval: document.getElementById("interval").value,
        mode: mode
    }
    var data = {
        Location: document.getElementById("position").value,
        Type: document.getElementById("type").value,
        Config: JSON.stringify(config)
    }

    $.ajax({
        url: "/api/actors/" + viewbagTenant + "/config/" + actorId,
        type: 'PUT',
        contentType: 'application/json; charset=utf-8',
        data: JSON.stringify(data),
        success: function () { location.reload(true); },
        error: function (result) {
            //Error handling
        },
    });
}
```

Tabelle 19 – Verwendung API PutConfig

Für die Put-Funktion wird Ajax verwendet. Der API wird der Tenant und die Identifikation des Actors im URL mitgegeben.

Im Demosystem werden Änderung der Konfiguration durch ein HTML Formular gemacht. Daher müssen die Daten vor dem Senden aufbereitet werden. In unserem Fall werden zwei JSON-Objekte erstellt. Das «config» Objekt enthält die Konfiguration. Dazu werden die Daten aus den entsprechenden Feldern gelesen.
Das Konfigurations-Objekt wird dann in ein weiteres Objekt eingefügt, zusammen mit der Lokation und dem Typ des Actors.
Das verschachtelte Objekt wird nun durch Ajax, via API, an den Controller gesendet.

Anhang

1 Abbildungsverzeichnis

2 Tabellenverzeichnis

3 Quellenverzeichnis

Bezeichnung	URL
ASP.NET: MVC	http://www.asp.net/mvc
ASP.NET: Controller	http://www.asp.net/mvc/overview/getting-started/introduction/adding-a-controller
ASP.NET: WEB API Tutorial	http://www.asp.net/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api
Azure: Storage Partition Strategy	http://bit.ly/1wlfGPy
Azure: How to use tables	http://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-how-to-use-tables/
Azure: SSL Certificate	http://azure.microsoft.com/en-us/documentation/articles/web-sites-configure-ssl-certificate/
ASP.NET MVC Authorize Attribute	http://msdn.microsoft.com/en-us/library/system.web.mvc.authorizeattribute(v=vs.118).aspx
Azure: SAS Authentication	http://msdn.microsoft.com/en-us/library/azure/dn170477.aspx
Azure: Event Hub Security	http://msdn.microsoft.com/en-us/library/azure/dn789974.aspx
Azure: SAS	http://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-shared-access-signature-part-1/

Tabelle 20 – Quellenverzeichnis