Architektur

Studienarbeit FS-2020

7. April 2020

Autoren:

Mike Schmid
mike.schmid@hsr.ch

Janik Schlatter
janik.schlatter@hsr.ch

Supervisors:

Prof. Stettler Beat
beat.stettler@hsr.ch

Baumann Urs
urs.baumann@hsr.ch

Zweck

Dieses Dokument beschreibt die Architektur, die Möglichkeiten für die Auswahl der Technolodien und die Entscheidungen zum Design und der Architektur des Projektes.

Änderungsgeschichte

Datum	Version	Änderung	Autor
24.03.2018	1.0	Initial Setup	Janik Schlatter
26.03.2020	1.0	Designentscheidungen	Mike Schmid
02.04.2020	1.1	Anpassungen gem. Besprechung vom 01.04.2020	Mike Schmid

Inhaltsverzeichnis

1 Designentscheidungen

1.1 Architekturentscheidungen

1.1.1 Anwendung von Patterns

Patterns in der Softwareentwicklung sind Vorschläge, wie mit bekannten Problemen umgegangen werden soll. Verschiedene Patterns können auf unterschiedliche Probleme angewandt werden und helfen, verschiedene Probleme zu lösen. Es gibt eine Vielfalt von Patterns mit vielen Anwendungsmöglichkeiten und Kombinationen. Für das Zu entwickelnde System wurden zwei Patterns ausgewählt, die die Erweiterbarkeit gewährleisten sollen.

Entscheidung 1: Anwendung von Patterns für die Testerstellung

- Im Kontext der Erweiterbarkeit des zu entwickelnden Systems,
- damit weitere Netzwerktest einfach hinzugefügt werden können,
- wurde entschieden, dass für die Testerstellung das Strategy und Factory-Method Pattern angewandt wird,
- um die Instanzierungslogik von den einzelnen Testklassen zu entkoppeln.
- Dabei wird akzeptiert, dass die Implementation des Systems alles in allem komplexer, und die Erstellung umfangreicher wird.

Entscheidung 2: Anwendung von Patterns für die Kommunikationsschnittstellen

- Im Kontext der Erweiterbarkeit um weitere Netzwerkschnittstellen,
- damit das Hinzufügen von weiteren Schnittstellen vereinfacht wird,
- wurde entschieden, dass die Auswahl und Instanzierung von Verbindungen über ein Factory Method- und Strategy Pattern realisiert wird.
- um die Verbindungsauswahl vom System zu entkoppeln.
- Es wird dabei akzeptiert, dass das System komplexer wird und die Erstellung mehr Aufwand benötigt.

Das Strategy Pattern ist ein Verhaltenspattern und ermöglicht eine erhöhte Austauschbarkeit einzelner Softwarekomponenten. Eine gruppe von Algorithmen wird zu einer Strategie zusammengefasst und können untereinander leicht ausgetauscht werden.

Das Factory Method Pattern erlaubt die instanzierung von Objekten, ohne die spezifische Klasse des Objekts kennen zu müssen. Somit wird die Flexibilität der Software und die Austauschbarkeit der einzelnen Komponenten erhöht.

Weitere Software-Pattern werden im Verlauf der Arbeit bei Bedarf verwendet, sind aber nicht Teil der Architekturentscheidungen.

1.1.2 Testdefinitionssprache

Die Testdefinitionen benötigen ein einheitliches Format, damit sie vom Programm interpretiert werden können und dass daraus Tests erstellt werden können. Für diese Aufgabe gibt es eine Vielzahl von verschiedenen Beschreibungssprachen.

XML (Extensible Markup Language) ist eine Auszeichnungssprache, die in der Informatik oft angewandt wird, wenn Daten in einer Hierarchischen Struktur in einer Textdatei beschrieben werden. Der grösste Vorteil in der Verwendung von XML ist die hohe Verbreitung und Platformunabhängigkeit. Nachteile sind eine höhere Redundanz als in anderen Formaten was zu höherem Speicherverbrauch und grösserem Bandbreitenverwendung führt, wenn Daten im XML-Format gespeichert oder über das Internet versendet werden.

JSON (Javascript Object Notation) ist Menschenlesbar, Leichtgewichtig und wird zum effizienten Austausch von Informationen in einem geordneten Format auszutauschen. Das Format baut auf einer Attribut : Attributwert Logik auf und ist im Vergleich zu XML effizienter über das Internet austauschbar. Im Gegensatz zu XML bietet JSON aber keine Schemas, ein einheitliches Format für den Datenaustausch, was die Verwendung in verschiedenen Systemen komplizierter macht.

YAML (YAML ain't Markup Language) ist ein Datenserialisierungsformat mit Fokus auf Menschenlesbarkeit und wird hauptsächlich für die Erstellung von Konfigurationsdaten verwendet. Die grössten Vorteile sind die Datenkonsistenz der Datenmodel und die hohe Portierbarkeit zwischen verschiedensten Programmiersprachen. Zu den Nachteilen gehört die geringere Performanz gegenüber XML und JSON und die geringere Verbreitung von YAML. Ein weiterer Wichtiger Punkt ist, dass Netzwerktechniker in der Lage sein sollten, YAML zu lesen und schreiben.

Entscheidung 3: Auswahl der Testdefinitionssprache

- Um die Testdefinitionen in einer möglichst menschenlesbaren Form zu halten,
- und die Verwendbarkeit im zu entwickelnden System zu gewährleisten,
- haben wir entschieden, die Testdefinitionen in YAML zu verfassen,
- um ein Format zu verwenden, dass auch von Netzwerkleuten verwendet wird.
- Andere Technologien, wie JSON, XML usw. werden dabei voraussichtlich weggelassen,
- auch wenn diese in spezifischen Anwendungen wie die Speicherung von Daten geeignet wären.

1.1.3 Benutzeroberfläche

Damit Benutzer des Systems mit diesem interagieren können, wird eine Benutzeroberfläche benötigt. Das Design der Oberfläche spielt eine entscheidende Rolle für die Benutzbarkeit der Software. Mögliche Umsetzungen sind eine Grafische Benutzeroberfläche (GUI), die es Benutzern erlaubt über eine vordefinierte Oberfläche mit dem Programm zu interagieren. Weiterhin lässt sich das System auch mit einem Kommandozeilen-Userinterface umsetzen, welches weitaus einfacher in der Implementation ist, aber gegenüber dem GUI weniger Interaktionsmöglichkeiten bietet.

Entscheidung 4: Weglassen einer Grafischen Benutzeroberfläche

- Im Rahmen der zeitlichen Beschränkung der Arbeit,
- um uns auf die Implementation der Logik zu konzentrieren,
- werden wir kein GUI entwickeln.
- Stattdessen wird, wo nötig, eine Kommandozeileninteraktion implementiert.
- Es wird akzeptiert, dass die Benutzerfreundlichkeit des zu entwickelnden Systems dadurch eingeschränkt wird.

1.1.4 Testframework

Für die Kommunikation und das automatisierte Testen von Netzwerken gibt es verschiedene Möglichkeiten. Nachfolgend sind einige davon aufgelistet:

Ansible ist ein Framework für automatisierte Netzwerkoperationen wie Konfiguration, Skalierung oder Testing. Das Framework arbeitet, ohne dass auf dem Zielgerät etwas installiert werden muss, das mit der Software kommuniziert, solange das Zielgerät über SSH erreichbar ist. Ansible ist vergleichsweise einfach zu erlernen und in Python geschrieben. Die Netzwerkkonfiguration wird im YAML Format erfasst und das Programm sorgt dafür, dass die gewünschte Konfiguration im Netzwerk umgesetzt wird. Nachteile sind der geringe Windows-Support und das fehlende Auflösen von Abhängigkeiten.

Saltstack ist ein Konfigurations-Management-Framework mit Funktionen für automatische Neztwerktests. Parallele Programmausführung, Skalierbarkeit und eine aktive Community gehören zu den grössten Vorteilen. Allerdings ist der Installationsprozess für Neueinsteiger eher komplex und ausser für Linux ist der Betriebssystemsupport limitiert. Auch Saltstack ist in Python geschrieben, was die Erweiterbarkeit erhöht.

Puppet ist eine konfigurationsautomatisierungs- und Deployment-organisations-Lösung für verteilte Software und Infrastruktur. Geschrieben in Ruby, bietet Puppet eine Vielzahl von Funktionen für die Orchestrierung, Konfigurationsautomatisierung, Visualisierung und Reporting. Stärken sind der Support für eine Vielzahl von Betriebssystemen, hohe Stabilität und Robustheit. Schwierigkeiten können die hohe Anfängerschwierigkeit, schwächere Skalierbarkeit und Flexibilität (verglichen mit den anderen Lösungen).

Chef war ursprünglich ein Tool für internes End-zu-End Server Deployment und wurde später als Open Source Lösung für Cloud und Infrastruktur Automatisierung veröffentlicht. Es wird als eine der flexibelsten Lösungen für OS und Middleware Management beschrieben und wurde für Programmierer entwickelt. Chef hat eine ausführliche Dokumentation und ist sehr stabil und verlässlich für grosse Systeme. Die grössten Nachteile sind die steile Lernkurve und ein komplizierteres Inbetriebnehmen gegenüber den anderen Systemen.

Nornir wurde in der Studienarbeit von Anfang an als mögliche Lösung für die Testautomation genannt. Das Framework wurde in Python für die Verwendung mit Python entwickelt und bietet eine Vielzahl von Möglichkeiten für die Ausführung von Tests auf verschiedensten Systemen. Nornir führt die Tests als serielle oder parallele Tasks aus und verwendet YAML für die Datenverwaltung. Allerdings lässt sich Nornir nicht einfach ausführen, sondern muss in ein Python-Programms implementiert werden. Deshalb kann Nornir nicht einfach Installiert und danach verwendet werden, sondern ist ein Tool, welches es erlaubt, eigene Netzwerktasks zu definieren und auszuführen.

Entscheidung 5: Verwendung von Nornir als Testframework

- Für die Kommunikation mit verschiedenen Netzwerken,
- um die Implementation möglichst zu vereinfachen,
- wurde entschieden, dass das System das Nornir-Automations-Framework verwendet wird,
- weil Nornir dafür gedacht ist, in einem Python-Programm eingesetzt zu werden
- und eine umfangreiche Sammlung von nützlichen Funktionen bietet, mit denen man Netzwerktests ausführen kann.

Die gesamte Arbeit wurde von Anfang an darauf ausgelegt, mit Nornir als Automatisierunsframework zu arbeiten. Die Vielseitigen Methoden, um mit Netzwerkgeräten zu kommunizieren und die Vielzahl von möglichen Befehlen erlauben es, einfachen und effizienten Code zu schreiben, welcher die Netzwerke testet.

1.1.5 Datenhaltung

Für die Verwaltung von Daten, insbesondere das persistente Speichern und Laden, gibt es diverse Möglichkeiten. Da die Testdefinitionen in YAML verfasst werden, wäre für die Verwaltung der YAML-Files ein0e Dokumentdatenbank geeignet. Dokumentdatenbanken entspringen aus dem NoSQL Paradigma und haben kein fixes definiertes Schema, was eine flexiblere Verwendung ermöglicht, aber auch Fehleranfälliger ist. Üblicherweise werden Dokumentdatenbanken in einem Key-Value Format gehalten, was die Verwendung mit YAML vereinfacht.

Weiterhin liesse sich eine Relationale Datenbank implementieren, wobei dafür ein Datenbankschema entworfen werden müsste. Die Verwendung einer SQL-Datenbank würde zu einer stärkeren Konsistenz der Daten führen, da die Datenbank eine strenge Konsistenz erzwingt. Allerdings sind relationale Datenbanken im Vergleich zu NoSQL Datenbanken komplexer und lassen sich schwieriger Skalieren.

Eine weitere Möglichkeit für die Speicherung der Daten wäre eine Graphdatenbank. Graphdatenbanken speichern nicht nur die Daten in Form von Knoten, sondern auch die Beziehungen zwischen den Daten in Form von Kanten zwischen den Knoten. Graphalgorithmen erlauben es, sehr effiziente Abfragen auf die Daten auszuführen und Verknüpfungen zwischen den Daten abzufragen, für die eine Relationale Datenbank viel komplexere und langsamere Queries verwenden würde.

Ausserdem besteht noch die Möglichkeit, keine Datenbank zu implementieren und die Daten auf dem Clientsystem zu verwalten. Hierbei werden die YAML-Dokumente direkt auf der Dateiablage des Clients gespeichert. Der Ansatz ist definitiv simpler als die Verwendung einer Datenbank, allerdings kann dies dazu führen, dass Konsistenzprobleme auftreten. Die Anwendung ohne Datenbank legt somit mehr Verantwortung in die Hände der Benutzer, welche sich selber darum kümmern müssen, dass die Daten in der richtigen Form persistent gespeichert werden. Es ist trotzdem möglich, gemeinsam einen Datensatz zu halten, beispielsweise liesse sich die Testkonfiguration über ein Git-Repository verwalten und von verschiedenen Standorten her verwenden. Auch die Verwendung mit einem zentralen Filesystem, z.B. ein Ordner auf dem Server ist grundsätzlich möglich, erfordert aber geringfügige Anpassungen am Programm, damit die Pfade korrekt verwendet werden.

Entscheidung 6: Verzicht auf eine Datenbank

- Um Daten persistent zu speichern,
- damit sie vom System wieder abgerufen werden können,
- werden die Daten im YAML-Format abgelegt,
- und nicht in einer Datenbank,
- wobei ein Datenbanksystem für die zentralisierte Speicherung möglicherweise besser geeignet wäre.

1.1.6 Deployment

Die fertige Software muss auf verschiedenen Systemen mit unterschiedlichen Betriebssystemen einsetzbar sein. Die Installation soll dabei so einfach wie möglich sein und eine geringe Anzahl Abhängigkeiten an weitere Software haben. Wenn das Programm beispielsweise fünf weitere Programme benötigt, die alle parallel dazu installiert werden müssen, könnte das für Netzwerktechniker ein Grund sein, eine andere Software zu wählen. Ausserdem führt eine kompliziertere Installation dazu, dass der Einstieg für Unerfahrene Netzwerkleute schwieriger wird. Somit sollte die Auslieferung möglichst simpel erfolgen und geringe/keine Abhängigkeiten haben.

Eine Auslieferung als ausführbare Datei, z.B. im .Exe-Format für Windows-Benutzer, wäre sicher für die Benutzung am einfachsten, allerdings wäre das Programm dann nicht so einfach erweiterbar, weil für jede Änderung das Programm nochmals aktualisiert werden müsste.

Das Deployment als Dockercontainer ist, unabhängig der eigentlichen Auslieferung, auch eine mögliche Option. Diese Möglichkeit hat aber mehr Abhängigkeiten gegenüber einer einfachen ausführbaren Datei, da auf dem Client-Computer zuerst noch Docker installiert werden muss.

Die einfachste Variante ist die Auslieferung als reines Python-Skript. Der grösste Vorteil ist, dass sich ein Python Skript auf jedem System ausführen lässt, welches Python installiert hat. Die meisten Betriebssysteme haben heute per Default Python vorinstalliert, zumindest auf MacOS und Linux-Systemen. Somit liesse sich die Software, ohne weitere Software (ausser evtl Python) installieren zu müssen, verwenden.

Entscheidung 7: Deployment als Python-Skript

- Im Kontext des Deployments, der Auslieferung der Software,
- um das Programm auf möglichst vielen Plattformen einfach einsetzen zu können,
- wird das System in Form eines Python-Scripts ausgeliefert,
- und nicht als ausführbare Datei im .exe Format,
- da Python auf den meisten Linux-Basierten Systemen und einigen modernen Windows-Systemen bereits vorinstalliert ist,
- womit sich die Installation/Ausführung des Programms stark vereinfacht.
- Die Folge, dass auf einigen Geräten für die Programmausführung zuerst Python installiert werden muss, wird akzeptiert.

Eine Auslieferung per Dockercontainer oder über einen Server ist hierbei immer noch möglich. Es werden aber jeweils systemspezifisch Änderungen am Programm/Konfiguration benötigt.

1.1.7 Security

Sicherheitspraktiken schlagen vor, dass Passwörter nur gespeichert werden sollen, wenn die Verwendung eines Systems eine Authentifizierung benötigt. Passwörter sollten in dem Falle mit einem Hash und Salt encoded und in dieser Form gespeichert werden. Das zu entwickelnde System benötigt Passwörter, um mit den Netzwerkgeräten kommunizieren zu können. Jedes Gerät benötigt (üblicherweise) einen Login-Namen und ein Passwort, um darauf die Konfiguration anzusehen, zu verändern oder Befehle auf dem Netzwerkgerät ausführen zu können.

Es wird davon abgeraten, eigene Algorithmen für die Verschlüsselung von Daten zu implementieren, da diese nicht getestet sind. Stattdessen sollen Bibliotheken verwendet werden, die von Kryptographen und Sicherheitsexperten getestet und als sicher eingestuft wurden.

Ein Nachteil für die Verwendung von Sicherheitspraktiken ist die geringere Benutzbarkeit. Es ist nicht möglich, nachzusehen, ob das Passwort, welches für ein Netzwerkgerät gesetzt wurde, immer noch korrekt ist, da das Hashing dies unmöglich macht. Deshalb ist es schwierig, das Programm so zu entwickeln, dass es sich sicher und simpel verwenden lässt, ohne dafür erheblichen Mehraufwand zu betreiben.

Entscheidung 8: Umgang mit Passwörtern

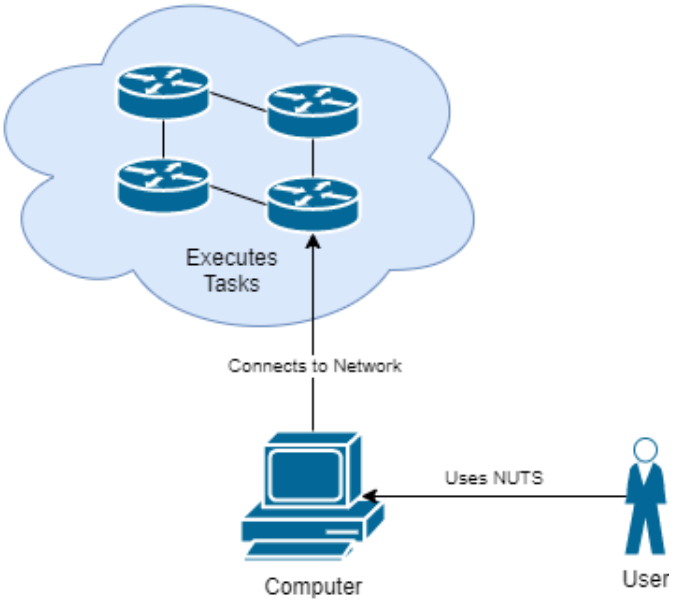
- Im Hinblick auf die Sicherheit des Systems,
- um die Verwendung des Programms möglichst einfach zu halten,
- werden Passwörter für die verschiedenen Netzwerkgeräte als Plaintext gespeichert,
- und nicht in verschlüsselter Form.
- Der sichere Umgang mit den Konfigurationsdaten ist somit in der Verantwortung des Benutzers.

Die Sichere Verwendung der Software würde voraussetzen, dass sich der Benutzer gegenüber des zu entwickelnden Programms authentifizieren muss. Nachdem der User verifiziert wurde, werden die bis dahin verschlüsselten Konfigurationsdaten entschlüsselt und lassen sich bearbeiten/benutzen. Eine weitere Möglichkeit wäre die Verwendung einer Datenbank, um die Konfigurationsdaten zu verwalten. Somit würde die Verantwortung für die sichere Speicherung bei der Datenbank und nicht beim Testprogramm liegen.

Beide Optionen werden als mögliche Erweiterung des Systems in Betracht gezogen.

2 Systemübersicht

Die Systemübersicht gibt einen Überblick über die verschiedenen Komponenten des Systems. Nachfolgend sind die einzelnen Komponenten detaillierter beschrieben.



2.1 Computer

Dies entspricht unserem Client. Auf dem Computer läuft das zu entwickelnde System, welches aus einem Python Programm besteht und mittels Nornir mit einem Netzwerk kommuniziert und Tests darauf ausführt.

2.2 Netzwerk

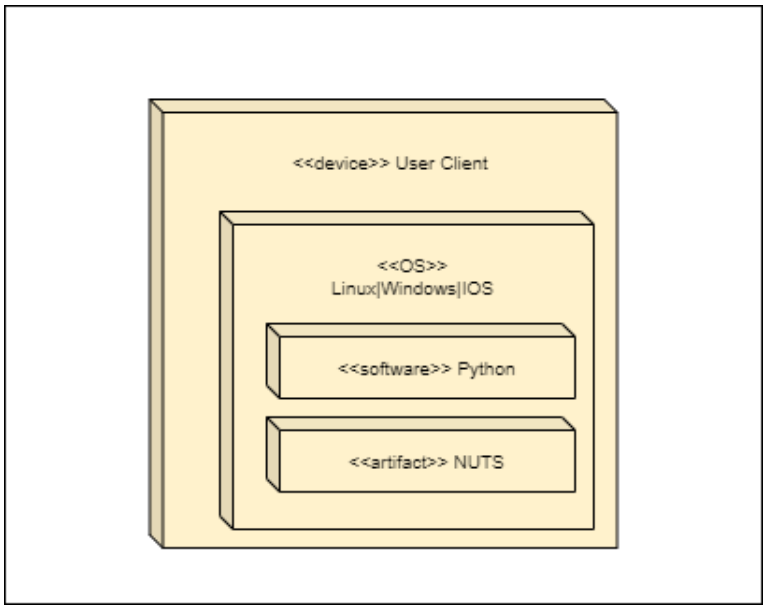
Dies ist das Netzwerk, auf dem die automatisierten Tests ausgeführt werden sollen. Es besteht aus mehreren Netzwerkknoten, z.B. Switches, Router und Clients. Das Netzwerk wird über eine lokale Schnittstelle oder über das Internet angesteuert.

2.3 Datenablage

Die Datenablage befindet sich auf dem Computer und kann bei Bedarf über ein Verwaltungstool mit anderen Clients geteilt werden. Darin befinden sich sämtliche für das Testsystem benötigte Daten. Diese Daten werden in Form von YAML-Files als Key-Value Store angelegt.

3 Deployment

3.1 Deploymentdiagramm



3.2 Client

Der User Client kann ein Computer oder ein Server sein, welcher die automatisierten Tests ausführen soll. Darauf läuft ein beliebiges Betriebssystem

3.3 Betriebssystem

Es gibt keine spezifischen Anforderungen an das Betriebssystem, welches auf dem Client installiert ist. Python Code lässt sich auf beliebigen Betriebssystemen ausführen.

3.4 Python Installation

Für die Ausführung von Python-Code muss Python auf dem Client installiert sein.

3.5 NUTS2.0

Das zu entwickelnde Programm, welches auf dem Client installiert wird. Es verbindet sich mit einem Netzwerk und führt vorher definierte Tests darauf aus.

4 Ausbauszenarien

4.1 Szenario 1: Erweiterung um weitere Neztwerktests

Die Erweiterung des Systems um weitere Tests wird durch die Architektur so einfach wie möglich gehalten. Es müssen lediglich die Testfactory angepasst und neue Testklassen erstellt werden, um neue Tests durch das System zu ermöglichen. Die Tests können danach in der Testdefinition angelegt werden.

4.2 Szenario 2: Erweiterung um Netzwerkschnittstellen

Durch den Einsatz des Factory- und Strategy-Pattern wird die Erweiterung um neue Schnittstellen vereinfacht. Die Factory muss um die neue Schnittstelle erweitert und die Schnittstelle als Klasse implementiert werden. Danach sollte sich die Schnittstelle über die Testdefinition aufrufen lassen.

4.3 Szenario 3: Erweiterung um eine Graphische Benutzeroberfläche (GUI)

Das System wird mit einem Fokus auf die Funktionalität entwickelt. Dabei wird die Usability durch die Verwendung eines GUI vorerst vernachlässigt. Die Erfassung von Netzwerkgeräten, das Erfassen von Tests und die Orchestrierung der Tests liessen sich alle über eine Benutzeroberfläche ansteuern. Nach der Einschätzung der Autoren entspricht die GUI-Erweiterung ungefähr drei bis sechs Wochen Arbeit. Es ist darauf zu achten, die Prinzipien der nutzerzentrierten Entwicklung (User Centered Design) anzuwenden und regelmässig Usability-Tests durchzuführen.

4.4 Szenario 4: Anbindung einer Datenbank

Eine Datenbank würde dem System diverse Möglichkeiten für die effizientere Datenhaltung ermöglichen:

- Geräte liessen sich in einer Graphdatenbank mit den jeweiligen Verbindungen als Relationen speichern, was eine intuitive Darstellung und effiziente Abfragen ermöglicht.
- Durch die Verwendung einer Datenbank liessen sich das Inventar und die Testdefinitionen einfacher von mehreren Clients verwenden und untereinander austauschen.
- Passwörter von Devices, welche nach der aktuellen Architektur in Plaintext gespeichert werden, wären in einer Datenbank besser gesichert.

Auch hier ist mit Aufwand von drei bis vier Wochen zu rechnen. Ausserdem wird für die Haltung einer Datenbank Hardware, beispielsweise ein DB-Server, benötigt.

4.5 Szenario 5: Erweiterung um Security

Damit Passwörter nicht in Plaintext gespeichert werden und von potentiellen Angreifern einfach ausgelesen werden können, benötigt das System eine Verschlüsselung der Konfigurationsdaten. Am einfachsten lässt sich dies erreichen, wenn sich der Benutzer gegenüber der Softwar mit einem Passwort authentifizieren muss, um zugriff auf die Konfigruation zu erhalten. Dafür müsste der Python-Code mit einer Security-Library erweitert werden und der Code müsste angepasst werden. Deviceinformationen würden dann nur noch in Encrypteter Form gespeichert werden.

Bei dieser Erweiterung ist darauf zu achten, dass das System so robust wie möglich bleibt. Beispielsweise müssen im falle eines Fehlers sämtliche Konfigurationen der Devices wieder Verschlüsselt werden, bevor sie gespeichert werden.