Webapplikation zur Steuerung eines Automatisierungsframeworks

Studienarbeit

Studiengang Informatik
OST – Ostschweizer Fachhochschule
Campus Rapperswil-Jona

Herbstsemester 2020

Autoren:	Felix Kubli, Jérôme Gygax
Betreuer:	Beat Stettler
Co-Betreuer:	Matthias Gabriel, Urs Baumann
Projektpartner:	INS Institute for networked Solutions, Rapperswil

ABSTRACT

Nornir ist ein Netzwerk-Automatisierungs-Tool, mit welchem Netzwerkgeräte programmatisch konfiguriert oder nach Daten abgefragt werden können. Dies verlangt Kenntnisse der Programmiersprache Python, um einen Automatisierungstask zu schreiben und anschliessend auszuführen und auswerten zu können. Netzwerkadministratoren kommen aber häufig aus der Ecke der Systemtechnik und haben entsprechend keine grossen Erfahrungen im Programmieren, was die Benutzung solch praktischer Tools wie Nornir erschwert. Mit der Nornir Webapp sollen diejenigen Netzwerkadministratoren die Vorteile von Nornir nutzen können, ohne sich mit Python auszukennen. In der Webapp können Netzwerkadministratoren Tasks starten sowie die Resultate der Tasks einsehen, ohne eine Zeile Python zu schreiben. Ein Task kann über einen geführten Wizard zusammengeklickt werden und direkt oder zu einer gewünschten Zeit ausgeführt werden. Um die Details der Task-Ausführung kümmert sich das in Python geschriebene Backend, das Frontend wird mittels React Framework und der Material UI für eine moderne UI realisiert. Die Applikation ist erweiterbar ausgelegt, so dass Python-versierte Netzwerkadministratoren die Webapp um weitere Automatisierungs-Tasks erweitert werden können. Mittels Unit-, Integration- und Usability-Tests wurde die Software getestet. Zukünftig könnte die Applikation mit einem Skript-Builder erweitert werden, mit diesem Administratoren ihre Tasks mit vorgefertigten Code-Blöcken zusammenklicken können. So kann die Anwendung um weitere Skripts erweitert werden, ebenfalls ohne Python-Kenntnisse zu besitzen.

INHALT

1 MANAGEMENT SUMMARY

1.1 KONTEXT, PROBLEMSTELLUNG

In der Netzwerktechnik wird noch viel manuell konfiguriert. Die Netzwerkadministratoren kennen deshalb hauptsächlich wie ihre Geräte über eine GUI oder Konsole konfiguriert werden. In der immer schneller ablaufenden Netzwerkwelt ist manuelle Konfiguration ein Bremsklotz, weshalb diese Arbeit immer häufiger automatisiert wird.

Die am Markt verfügbaren Lösungen zur Netzwerkautomation sind häufig kostenpflichtig oder nur auf die Geräte eines Herstellers zugeschnitten. Es gibt eine Fülle an Open-Source Anwendungen, die viel flexibler einsetzbar sind. Diese verlangen jedoch häufig Programmierkenntnisse, um eine Automatisierung zu entwickeln. Häufig fehlt dem klassischen Netzwerkadministrator dieses Wissen.

Um die Welt der Netzwerkautomation einem breiteren Kreis an Netzwerkadministratoren zur Verfügung zu stellen, ist das Ziel dieser Arbeit eine Benutzeroberfläche für ein bekanntes Netzwerk-Automationsframework zu erstellen. Mit dieser sollen Netzwerkadministratoren ohne Programmierkenntnisse im Stande sein, ihr Netzwerk automatisiert zu verwalten.

1.2 ZIEL

Das Hauptziel ist eine Applikation, welche das Python Netzwerkautomationsframework «Nornir» um eine Benutzeroberfläche erweitert. In dieser Oberfläche soll ein Netzwerkadministrator die im System vorgefertigten Tasks auf seinen Geräten ausführen können und die Resultate einsehen können.

Als optionale Ziele soll es einen «Task-Builder» geben, der es dem programmierunerfahrenen Administrator ermöglicht, eigene Tasks zu erstellen, ohne eine Zeile Code dafür zu schreiben. Ebenso soll es Möglichkeiten geben, bereits bestehende Lösungen zur Inventarverwaltung als Quelle für das Netzwerkinventar zu verwenden.

1.3 RAHMENBEDINGUNGEN

Die Entwicklung der Applikation soll keine Kosten verursachen. Verwendete Frameworks und Tools sollen kostenfrei oder Open-Source sein. Die Applikation soll als Teil einer Studienarbeit geschrieben werden.

1.4 VORGEHEN

Die Lösung wird nach der agilen Methode Scrum erarbeitet. Das Entwicklerteam arbeitet iterativ in einem zweiwöchigen Zyklus. Mit dem betreuenden Institut wird wöchentlich über den aktuellen Status diskutiert.

1.5 RESULTATE

Die in der Arbeit entstandene Lösung kann etwas mehr als das definierte Minimum Viable Product (MVP). Die festgelegten Kernelemente der Applikation wurden alle umgesetzt.

Alle definierten Kern-Use Cases wurden implementiert. Ein Netzwerkadministrator kann auf einer von ihm bestimmten Auswahl an Netzwerkgeräten Tasks ausführen und deren Resultate anzeigen lassen. Die Aktionen eines Task, ein sogenanntes Job Template, werden vom System mitgeliefert. Das System ist erweiterbar ausgelegt. Das bedeutet, dass Administratoren, die sich mit Python und Netzwerkautomatisierung auskennen, eigene Job Templates hinzufügen können.

Ein Administrator kann einen Task entweder direkt ausführen oder zu einem gewünschten Zeitpunkt einplanen. Die Resultate kann der Netzwerkadministrator ebenfalls direkt in der Applikation einsehen. Ist ein Task fehlgeschlagen, kann ein Netzwerkadministrator diesen mit den gleichen Einstellungen erneut ausführen oder auch Einzelheiten vor der Ausführung anpassen.

Als Komfortfunktion wurden «vorkonfigurierte Tasks» implementiert. Diese ermöglichen das Ablegen eines konfigurierten Task um diesen bei Bedarf sofort ausführen kann, ohne diesen mit den benötigten Daten abzufüllen, da diese bereits gesetzt sind.

Des Weiteren verfügt die Anwendung über ein Berechtigungssystem, so dass unterschiedliche Benutzergruppen entweder nur Daten einsehen können oder komplette Kontrolle über die im System verwalteten Daten haben. So kann zum Beispiel auch der First Level Support Task Resultate anschauen und entsprechend Auskunft geben.

1.5.1 AUSBLICK

Die Anwendung kann in mehreren Bereichen erweitert werden. Einerseits betrifft dies die Inventarverwaltung. Aktuell wird nur das in Nornir eingesetzte Inventar-Plugin unterstützt, könnte jedoch um weitere Quellen erweitert werden. So könnte die Applikation etwa direkt mit Daten aus dem Asset Management oder dem IPAM abgefüllt werden.

Zusätzlich zu den mitgelieferten Job Templates kann auch ein interaktiver Vorlagen-Builder eingebaut werden, der es erlaubt, aus vorgefertigten Elementen eigene Automatisierungen zusammen zu klicken. Dies kann im Stil von «Wenn dies, dann führe das aus»-Blöcken oder mit «Konfiguriere [Funktion]»-Blöcken gemacht werden.

2 TECHNISCHER BERICHT

2.1 AUSGANGSLAGE & PROBLEMBESCHREIBUNG

Bis vor einigen Jahren wurden grosse Netzwerk in mühsamer Handarbeit manuell konfiguriert. Mittlerweile hat man erkannt, dass dies ein grosser Zeitfresser ist und es wurden in der Netzwerkwelt mehrere Lösungen geschaffen, um seine Netzwerke automatisiert zu provisionieren, konfigurieren oder zu überwachen. Bekannte Lösungen sind etwa Ansible, Napalm, Nornir, oder auch kommerzielle Software von Cisco oder Solarwinds.

Jede Lösung hat unterschiedliche Ansätze. So ist zum Beispiel Ansible ein Tool, welches verwendet wird, um über die für Menschen lesbare Konsole Aktionen automatisiert durchzuführen. Dass dies für Maschinen keine leichte Aufgabe ist, erschwert manchmal die Erstellung von schlauen Automatisierungskonzepten. Andere Lösungen verfolgen andere Ansätze, sind dann möglicherweise nur mit Geräten eines spezifischen Herstellers kompatibel oder kostenpflichtig.

In dieser Arbeit wird auf das Automatisierungs-Tool Nornir fokussiert, da sich dieses auf «Network as Code» konzentriert. Merkmal eines «Network as Code» ist, dass sämtliche Aktionen, die auf den Netzwerkgeräten ausgeführt werden, sei es nun Konfiguration oder Auslesen von Daten, immer als Code geschrieben wird. Im Fall von Nornir ist diese Sprache Python. Nornir erfindet hierbei nicht das Rad neu, sondern setzt auf Plugins, welche sich für die Netzwerkautomation bewährt haben, für eine einfache Erweiterbarkeit. So können in mit Nornir verwalteten Netzwerken Aktionen auf den Geräten sowohl mit Netmiko oder Napalm oder weiteren Tools ausgeführt werden. Dies macht Nornir zu einem mächtigen und vielseitigen Programm, um sein Netzwerk damit zu verwalten.

Einen Haken hat diese Software leider auch: Viele Netzwerkadministratoren haben eine Ausbildung als Informatiker in Fachrichtung Systemtechnik oder sind sogar Quereinsteiger. Gemein ist, dass viele Administratoren Berührungsängste mit dem Schreiben von Code haben, da dies bisher noch nie oder selten von Ihnen verlangt wurde.

Für all jene Administratoren bleiben die Vorteile von Nornir und Network as Code verwehrt. Am INS ist Network as Code ein grosses Thema und ein aktuell heisser Trend in der Branche. Deshalb hatte der INS-Mitarbeiter Urs Baumann die Idee, für das Tool Nornir eine Benutzeroberfläche zu entwickeln, so dass ein Netzwerkadministrator so wenig Code wie möglich zu schreiben hat und ihm der Einstieg in die Netzwerkautomatisierung erleichtert wird.

2.2 LÖSUNGSKONZEPT

Beim Design des Konzeptes zur Umsetzung stellte sich unter anderem die Frage, wie man eine intuitive Benutzeroberfläche gestalten kann. Dies betraf vor allem die Auswahl der Geräte, auf welchen die Netzwerkaktionen stattfinden sollen, und das Erstellen eines Tasks mit Geräteauswahl, der Wahl des Automationstemplates, sowie die Eingabe potenzieller Variablen. Für die Geräteauswahl wurde für eine sortierbare und filterbare Tabelle entschieden, in der man die Geräte mit Checkboxen selektieren kann. Aufgrund dieser Vielfalt an Möglichkeiten wurde entschieden, die Webanwendung nur für Desktops zu entwickeln und nicht mobiltauglich zu machen.

Um einen ausführbaren Task erstellen zu können, wurde für einen Task Wizard entschieden, in dem man Schrittweise alle benötigten Informationen angeben kann. Die Auswahl der Geräte ist in diesen Wizard integriert.

Wichtig für eine solche Applikation ist, dass diese Tasks asynchron ausgeführt werden, um die Webseite nicht zu blockieren. Ausserdem ist es wichtig, dass Operationen wie Softwareupdates oder kritische Konfigurationsänderungen zu einem Zeitpunkt ausgeführt werden, wenn niemand die Systeme verwendet. Deswegen soll man auch eine Ausführungszeit für einen Task festlegen können.

Eine weitere Entscheidung betraf, wie man den Code für Taskdurchführungen speichert, erfasst und in der Applikation registriert. Speichert man alle diese Templates in einem Ordner, oder versucht man es mit der Datenbank? Es wurde für eine Kombination aus beidem entschieden. Der Speicherort und weitere Metadaten des Codes werden in der Datenbank gespeichert und der Code wird am definierten Speicherort abgelegt.

Ein weiterer wichtiger Punkt war die Benutzerverwaltung und Authentifizierung. Ein Netzwerk stellt das Rückgrat einer modernen IT dar und entsprechend muss kontrolliert werden, welche Personen auf die Anwendung Zugriff haben und welche nicht. Es wurde entschieden, die Anwendung nur für angemeldete Personen zur Verfügung zu stellen. Ebenfalls wurden drei Benutzergruppen erarbeitet mit unterschiedlichen Berechtigungsstufen, um so fein steuern zu können, wer alles darf, wer nur lesen darf oder wer Tasks ausführen darf.

2.3 UMSETZUNG

Die Anwendung wurde als Webapplikation mit separatem Front- und Backend umgesetzt.

Für das Frontend wären Frameworks wie Angular, React oder Vue in Frage gekommen. Es wurde schlussendlich das Framework React eingesetzt. Die Auswahl fiel darauf, weil das Framework am INS auch bei anderen Projekten einsetzt worden ist und deren Team das Projekt so nach der Übergabe besser unterhalten kann.

Im Backend wurde aus demselben Grund für die Programmiersprache Python entschieden. Ausserdem eignet sich Python perfekt für die Zusammenarbeit mit Nornir, da dieses ebenfalls in Python geschrieben ist und somit direkt auf die Schnittstellen von Nornir zugegriffen werden kann. Als Framework wurde für Django in Kombination mit dem Django Rest Framework entschieden, da es einfach und schnell zu verwenden ist und einem Entwickler viel Arbeit abnimmt.

Eine mögliche Alternative für das Backend wäre Flask gewesen. Das INS hat das Django Framework empfohlen und gleichzeitig darauf aufmerksam gemacht, dass sie bereits Tutorials für Django in Kombination mit dem Django Rest Framework produziert haben. Diese Tutorials zeigten, dass mit wenig Aufwand sehr viel erreicht werden kann, weshalb schlussendlich für Django mit Django Rest entschieden wurde.

Um Tasks asynchron und zu geplanten Zeiten ausführen zu können, wird Celery verwendet. Auch hier war bereits gute Dokumentationen vorhanden, welche zum Entschluss führten, dass damit am komfortabelsten die gewünschten Funktionen umgesetzt werden können.

2.4 ERGEBNISDISKUSSION

Herausgekommen ist bei dieser Arbeit eine Webapplikation, mit der auf ausgewählte Geräte Nornir-Tasks ausgeführt werden können und die Resultate nach der Ausführung in der Weboberfläche angezeigt werden. Dafür wurde ein Wizard entwickelt, der einen durch die Erstellung eines Tasks führt. Zuerst werden die Geräte oder das Job Template ausgewählt, anschliessend kann dem Task einen Namen vergeben werden sowie der Ausführungszeitpunkt festgelegt und allfällige Variablen ausgefüllt werden.

Tasks können sofort oder zu einer definierten Zeit ausgeführt werden. Der im Backend integrierte Scheduler kümmert sich darum, dass der Task zur geplanten Zeit ausgeführt wird. Häufig benötigte Tasks können auch als sogenannte «vorkonfigurierte Tasks» abgespeichert werden. In diesem Fall sind bereits alle Parameter wie ausgewählte Geräte, Job Template, Name und Variablen gesetzt, jedoch mit keinem Ausführungsdatum versehen. Solche Tasks können zum Beispiel für häufig benötigte Aktionen wie Informationsbeschaffung verwendet werden. So wird das Durchklicken des Wizards erspart.

Schlägt ein Task fehl, kann der Tasks wiederholt werden. In solchen Fällen wird der Wizard mit den Daten und Variablen des fehlgeschlagenen Tasks neu befüllt. Allfällige Fehler können korrigiert werden und danach kann der Task erneut ausgeführt werden. Dabei wird ein neuer Task in der Datenbank erstellt, so dass die Fehlermeldung oder Ergebnisse des vorherigen Tasks nicht verloren gehen.

Für eine bessere Übersicht unterstützt die Applikation mehrere Inventories. So können zum Beispiel die Inventories von verschiedenen Lokationen separat in der Applikation angesteuert werden. Des Weiteren verfügen alle Übersichtsseiten eine Filter-, Sortier-, und Suchfunktion. Damit die Seiten jeweils nicht zu überladen werden, haben alle Tabellen eine Pagination mit einer definierbaren Anzahl an maximalen Elementen pro Seite.

All diese Funktionen wurden über die API des Backend implementiert, womit es möglich ist, eigene Clients für die Applikation zu schreiben. Das Backend besitzt ein Swagger-UI, womit die API automatisch dokumentiert wird.

2.5 AUSBLICK

Um automatische Updates bei der Statusänderung eines Tasks zu erhalten, bieten sich Websockets an. Für diese Arbeit wurde entschieden diese nicht zu implementieren, da sie potenziell viel Zeit zur Umsetzung brauchen und nicht zwingend notwendig für die Effektivität der Anwendung sind und andere Issues höher priorisiert wurden.

Des Weiteren kann man bis jetzt noch keine periodischen Tasks erstellen, die in einem bestimmten Zeitabstand ausgeführt werden. Damit könnte man beispielsweise regelmässige Backups der Netzwerkgeräte erstellen.

Bis jetzt kann man als Verwalter der Webseite Templates registrieren, welche dann von der Oberfläche aus ausgeführt werden können. Eine Erweiterung dafür wäre die Möglichkeit, mehrere Templates aneinander zu hängen, um beispielsweise das Resultat eines vorgegangenen Tasks direkt weiterverwenden zu können. Dies hat das Potential in der Umsetzung sehr komplex zu sein.

Damit auch programmierunerfahrene Netzwerkadministratoren eigene Job Templates entwickeln können, kann ein Task-Builder implementiert werden. In diesem können mit einer grafischen Oberfläche einzelne Task-Blöcke, welche Aktionen darstellen, zusammengefügt werden und so komplexere Tasks erstellt werden, ohne dafür zu programmieren. Man kann sich das etwa so vorstellen wie ein «Scratch» für die Netzwerkautomatisierung. Die Herausforderung hierbei besteht in zwei wesentlichen Punkten:

- Zur Verfügung stellen der Task-Blöcke: Die vielen unterschiedlichen Möglichkeiten, die mit den unterschiedlichen Nornir-Plugins möglich sind, müssen erfasst werden und als Task-Block in der Oberfläche abgebildet werden.
- Überprüfen auf Korrektheit: Die Business Logik muss bis zu einem gewissen Grad überprüfen können, ob die vom Benutzer erstellten Kombinationen möglich und ausführbar sind. Dies ist ein in der Computertheorie bekanntes Grundproblem und nicht einfach zu lösen.

3 ANFORDERUNGSSPEZIFIKATION

3.1 USE CASES

Es gibt drei Aktoren in diesem System.

- Der Supporter hat nur Lese-Berechtigungen auf das System und kann das Inventory, die Job Templates sowie die Task-Übersicht über alle laufenden, geplanten und abgeschlossenen Jobs einsehen.
- Der Netzwerkadministrator hat die gleichen Berechtigungen wie der Support und kann zusätzlich Tasks ausführen respektive planen.
- Der Superuser hat die gleichen Berechtigungen wie der Netzwerkadministrator, kann zusätzlich Nornir-Parameter bearbeiten, das Inventory verwalten/exportieren/importieren und Job Templates verwalten sowie neue Job Templates erstellen.

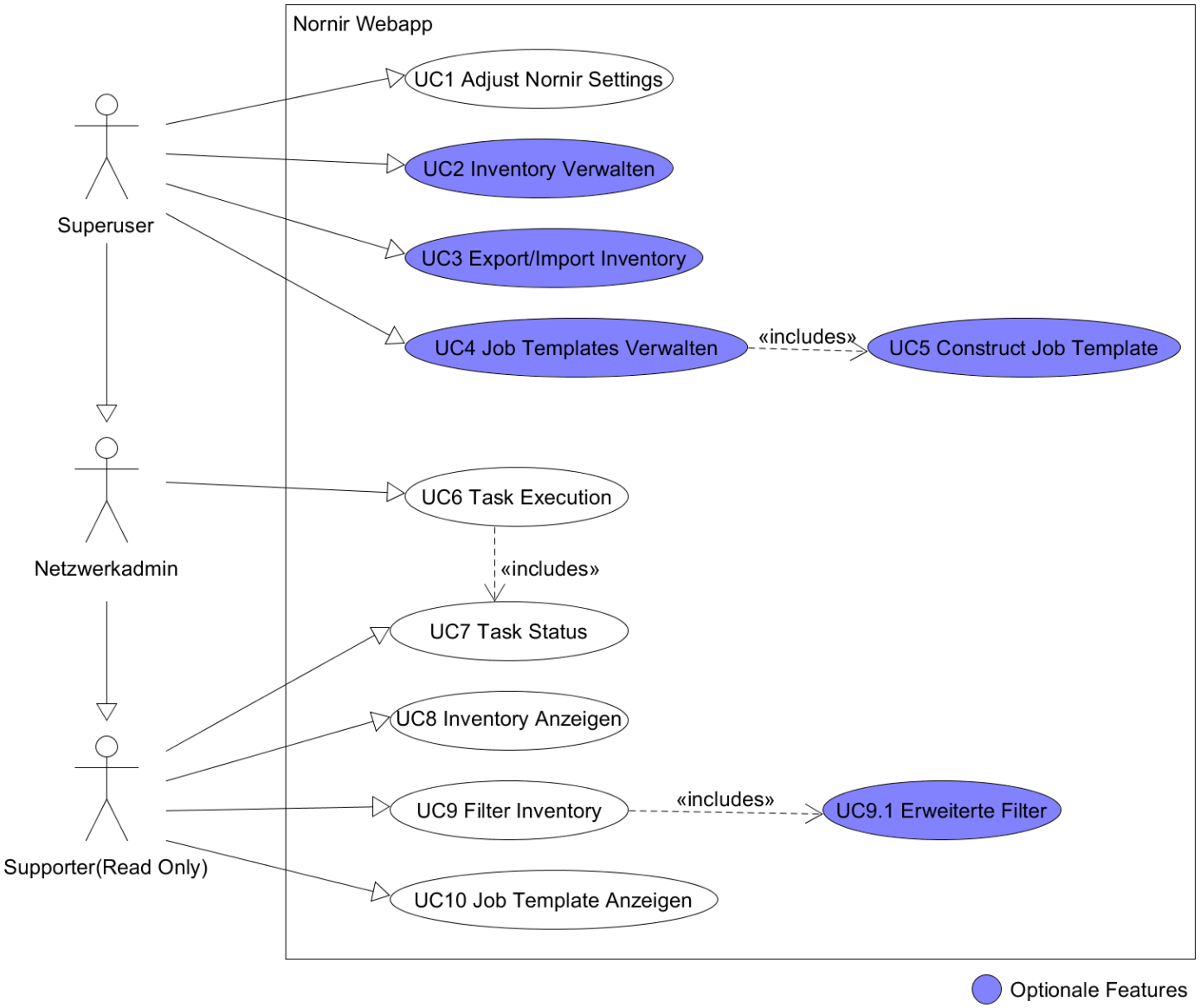


Abb. 3.1: Use Case Diagramm

3.1.1 BRIEF

3.1.1.1 UC1 ADJUST NORNIR SETTINGS

Der Superuser konfiguriert allgemeine Nornir-Parameter (globale Konfiguration) über die Weboberfläche.

3.1.1.2 UC2 INVENTORY VERWALTEN

In der Basisversion verwaltet der Superuser das Inventory über das SimpleInventory Plugin von Nornir.

Optionale Features:

- Der Superuser fügt Inventories aus anderen Quellen wie z.B. GitHub hinzu.
- Der Superuser verwaltet das Inventory über das Frontend.

3.1.1.3 UC3 EXPORT/IMPORT INVENTORY

Optionales Feature:

- Der Superuser importiert/exportiert Inventories in gängige portable Formate wie YAML oder JSON.

3.1.1.4 UC4 JOB TEMPLATES VERWALTEN

In der Basisversion schreibt der Superuser Nornir-Skripte in Python und speichert diese im Backend ab. Das Backend stellt diese in der Oberfläche als Job Template zur Verfügung.

Optionales Feature:

- Der Superuser als auch Netzwerkadministrator erstellt, modifiziert oder löscht Job Templates direkt über das Frontend.

3.1.1.5 UC5 CONSTRUCT JOB TEMPLATE

Optionales Feature:

- Der Superuser oder Netzwerkadministrator klickt neue Job Templates aus vorgefertigten Elementen/Befehlen zusammen.

3.1.1.6 UC6 TASK EXECUTION

Der Superuser und Netzwerkadministrator starten einen Task in vier Schritten:

1. Job Template auswählen, welches ausgeführt werden soll.
2. Inventory filtern und festlegen, auf welche Geräte das Template angewendet wird.
3. Allfällige Variablen des Template werden mit Daten angereichert.
4. Zeitpunkt der Ausführung definieren.
    - Der Task wird entweder nur erstellt, sofort oder zu einer definierten Zeit ausgeführt.

Tasks können geklont werden, um erneut ausgeführt zu werden.

Optionales Feature:

- Tasks können nach einem Wiederholungsmuster regelmässig ausgeführt werden.

3.1.1.7 UC7 TASK STATUS

Der Superuser, Netzwerkadministrator, Supporter sieht eine Übersicht über alle geplanten, laufenden und abgeschlossenen Tasks. Bei Geplanten ist das Startdatum sichtbar, bei laufendem Task der aktuellen Status und bei Abgeschlossenen das Resultat der Ausführung.

3.1.1.8 UC8 INVENTORY ANZEIGEN

Der Superuser, Netzwerkadministrator, Supporter zeigt das Inventory an. Die Geräte werden gruppiert aufgelistet.

3.1.1.9 UC9 FILTER INVENTORY

Der Superuser, Netzwerkadministrator, Supporter filtert das Inventory mit den durch Nornir vorgegebenen Attributen.

Der Superuser und Netzwerkadministrator startet auf eine Auswahl des Inventory UC6 «Task Execution».

3.1.1.9.1 UC9.1 ERWEITERTE FILTER

Optionales Feature:

- Der Superuser, Netzwerkadministrator, Supporter filtert mit erweiterten Attributen, die selbst angelegt wurden, mittels der F-Funktion aus Nornir.

3.1.1.10 JOB TEMPLATE ANZEIGEN

Der Superuser, Netzwerkadministrator, Supporter listen alle Job Templates in einer Übersicht auf und können Details eines Template einsehen.

Der Superuser und Netzwerkadministrator startet auf ein ausgewähltes Job Template UC6 «Task Execution».

3.1.2 FULLY DRESSED

Mit Benutzer sind die jeweiligen primären Akteure gemeint.

3.1.2.1 UC5 CONSTRUCT JOB TEMPLATE

Ziel	Der Benutzer fertigt aus einer Auswahl von Elementen ein Job Template.
Niveau	Benutzerziel
Primärer Akteur	Superuser, Netzwerkadministrator
Beteiligte und Interessen	Superuser, Netzwerkadministrator: Erstelle Job Template mit einer grafischen Oberfläche, ohne eine Zeile Python zu programmieren.
Vorbedingungen	Der Benutzer muss angemeldet sein und der Rolle Superuser oder Netzwerkadministrator angehören.
Nachbedingungen	Der Benutzer hat ein fertiges Job Template, das als Task ausgeführt werden kann.
Haupt-Erfolgsszenario	Der Benutzer wählt aus vordefinierten Aktionsblöcken seine Elemente aus. Blöcke sind z.B. «Besorge Informationen», «Konfiguriere [Feature]» Der Benutzer sortiert die Blöcke in der gewünschten Reihenfolge und setzt Variablen für Konfigurationswerte da wo nötig/erwünscht. Der Benutzer speichert sein Template und kann dieses auf ein Inventory ausführen
Spezielle Anforderungen	Der Benutzer bringt ein Grundverständnis mit, welche Möglichkeiten und Einschränkungen in der Kombination der verschiedenen Blöcke existieren.
Auftrittshäufigkeit	Bei Bedarf, ca. 1-2 mal pro Woche
Hinweise	Optionales Feature Dieser Use Case kann schwer feststellen, ob die Kombinationen des Benutzers valide sind und auch fehlerfrei ausgeführt werden können.

Tabelle 3.1: Use Case 5 Construct Task im fully-dressed Format

3.1.2.2 UC6 TASK EXECUTION

Ziel	Der Benutzer startet aus der Inventory-Übersicht oder der Job Template-Übersicht einen Task, der sofort oder zu einer geplanten Zeit ausgeführt wird.
Niveau	Benutzerziel
Primärer Akteur	Superuser, Netzwerkadministrator
Beteiligte und Interessen	Superuser, Netzwerkadministrator: Starte Task, der über eine gewünschte Anzahl Geräte ausgeführt wird.
Vorbedingungen	Der Benutzer muss angemeldet sein und der Rolle Superuser oder Netzwerkadministrator angehören.
Nachbedingungen	Der Benutzer sieht den Task, geplante Ausführung sowie Ausführungsstatus und Resultat, wenn der Task abgeschlossen ist.
Haupt-Erfolgsszenario	1. Der Benutzer startet in der Inventory-Übersicht und filtert auf die Geräte, auf welche er einen Task ausführen möchte. 2. Der Benutzer wählt anschliessend das auszuführende Template aus. 3. Wo nötig reichert der Benutzer Variablen im Template mit Daten an. Z.B. OSPF Router-ID 4. Der Benutzer gibt an, zu welchem Zeitpunkt der Task ausgeführt werden soll (Keine Ausführung, Sofort, geplantes Datum/Zeit) 5. Der Benutzer schliesst den Task-Wizard ab und sieht seinen erstellten Task im Dashboard. ODER 1. Der Benutzer startet in der Job Template-Übersicht und wählt das Template aus, welches er ausführen möchte. 2. Der Benutzer filtert anschliessend das Inventory auf die gewünschten Geräte. 3. Wo nötig reichert der Benutzer Variablen im Template mit Daten an. Z.B. OSPF Router-ID 4. Der Benutzer gibt an, zu welchem Zeitpunkt der Task ausgeführt werden soll (Kein, Sofort, geplantes Datum/Zeit) 5. Der Benutzer schliesst den Task-Wizard ab und sieht seinen erstellten Task im Dashboard.
Erweiterungen	Der Use Case UC5 Construct Job Template kann bei der Template-Wahl eingebaut werden.
Spezielle Anforderungen	Ein Inventar ist in Nornir geladen.
Auftrittshäufigkeit	Ungefähr 5-10 mal pro Tag
Hinweise	

Tabelle 3.2: Use Case 6 Task Execution im fully-dressed Format

3.2 NICHTFUNKTIONALE ANFORDERUNGEN

Die nichtfunktionalen Anforderungen sind in die Prioritäten Muss, Soll, Kann aufgeteilt. «Muss» hat die höchste Priorität und muss zwingend erfüllt sein, «Kann» hat die niedrigste Priorität und ist optional.

3.2.1 FUNKTIONALITÄT

Charakteristik	Definition	Prio.
Konformität	Für die Kommunikation zwischen Frontend wird eine standardisierte Schnittstelle verwendet (REST)	Soll
Sicherheit	Benutzer müssen sich authentisieren und können nur diejenigen Funktionen nutzen, für welche ihre Benutzergruppe autorisiert ist. Siehe dazu die Actors im Kapitel 3.1 Use Cases	Soll

3.2.2 ZUVERLÄSSIGKEIT

Charakteristik	Definition	Prio.
Fehlertoleranz	Tasks laufen weiter, wenn das Frontend die Verbindung zum Backend verliert.	Muss
	Fehlerhafte Eingaben führen nicht zum Absturz der Anwendung.	Muss
Wiederherstellbarkeit	Tasks und deren History sind auch nach Neustart der Anwendung persistent.	Soll
	Der Fortschritt von laufenden Tasks wird überwacht und nach Neustart dort fortgesetzt, wo der Task zuletzt war.	Kann

3.2.3 BENUTZBARKEIT

Charakteristik	Definition	Prio.
Attraktivität	Die Oberfläche wird die Designsprache Material Design verwenden.	Kann
Erlernbarkeit	Der Administrator kann die Anwendung nach zweistündiger Einführung selbständig bedienen.	Soll
Verständlichkeit	Die Oberfläche der Anwendung ist in Englisch.	Muss
	Fehlermeldungen werden mit einer verständlichen Meldung präsentiert.	Soll

3.2.4 EFFIZIENZ

Charakteristik	Definition	Prio.
Zeitverhalten	Interaktionen mit der Anwendung dauern maximal 3 Sekunden bis eine Reaktion der Anwendung erfolgt.	Soll

3.2.5 WARTBARKEIT

Charakteristik	Definition	Prio.
Analysierbarkeit	Klassen werden im Normalfall aufgeteilt, wenn sie mehr als 300 Zeilen Code enthalten.	Soll
	Das Backend wird den Output des Nornir-Frameworks zur besseren Fehleranalyse lokal abspeichern.	Soll
Prüfbarkeit	Sämtliche Änderungen am Code werden bei jedem Pull Request von einem anderen Teammitglied manuell und mit automatisierten Tests über die Build-Pipeline überprüft.	Soll

3.2.6 PORTABILITÄT

Charakteristik	Definition	Prio.
Austauschbarkeit	Die Oberfläche der Anwendung ist Cross-Browser kompatibel: Firefox 80+, Chromium-basierte Browser 85+	Muss
Installierbarkeit	In der Dokumentation wird eine Anleitung zur Installation der Applikation vorhanden sein.	Muss

4 ARCHITEKTUR

4.1 DOMAINMODELL

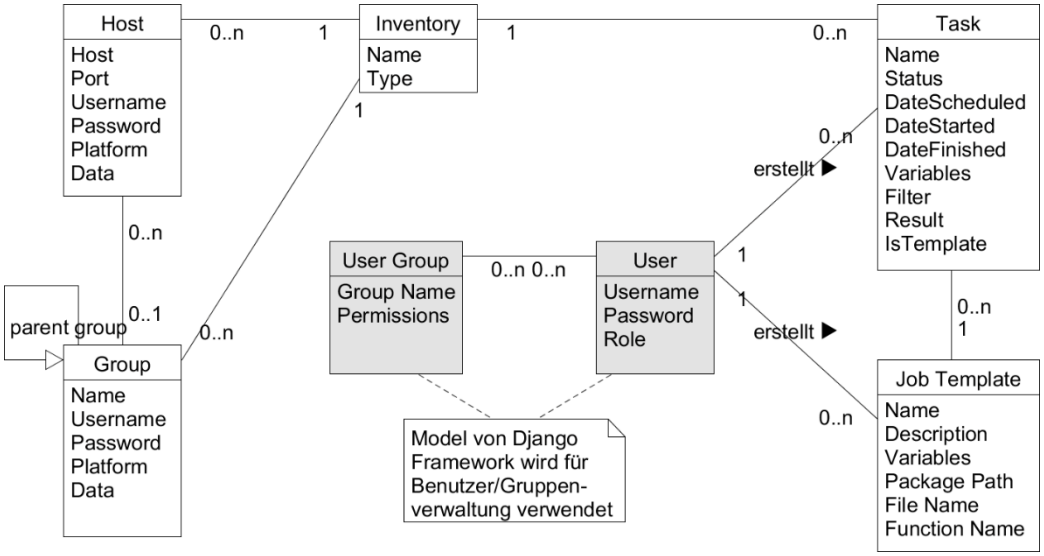


Abb. 4.1: Domainmodell

4.1.1 HOST

Hosts werden in Nornir mit SimpleInventory verwaltet.

Hosts können zu einer Gruppe gehören und erben deren Attribute. Hosts gehören immer einem Inventory an.

4.1.2 GROUP

Group ist ein weiteres Element des SimpleInventory Plugins für Nornir. Das Inventory erlaubt das Verschachteln von Gruppen. Dies wird auch abgebildet, weshalb die Klasse Gruppe eine Verbindung auf sich selbst hat.

Eine Gruppe gehört immer nur einem Inventory an.

4.1.3 USER

Die Benutzer werden mit den im Django Framework integrierten Benutzerklassen implementiert. Ein User einer Gruppe angehören, welche die Berechtigungen des Benutzers bestimmen.

4.1.4 USER GROUP

Jeder Benutzer ist einer User Group zugeteilt. Diese legt die Berechtigungen für die Rollen «Superuser», «Netzwerkadministrator», «Support» fest.

4.1.5 INVENTORY

Die Applikation kann mehrere Inventories verwalten, weshalb eine separate Inventory-Klasse vorhanden ist. Jede Instanz eines Inventory verbindet zu einem SimpleInventory.

4.1.6 TASKS

Jeder Task generiert einen Output (Resultat oder Fehlermeldung), wenn er abgeschlossen ist. Ein Task läuft über die Geräte in einem Inventory. Um die Anzahl Geräte einzuschränken wird ein Filter im Task gespeichert, der auf das Inventory angewendet wird.

Optional – sofern die Zeit im Projekt ausreicht – werden die folgenden zwei Verhalten dem Task-Model ebenfalls hinzugefügt:

- Schlägt ein Task fehl, kann dieser erneut ausgeführt werden. Dies klont den fehlgeschlagenen Task und startet den neuen Task, damit dessen Resultat nicht verloren geht.
- Tasks können als «Template» markiert werden. Darin können bereits alle Filter und Parameter konfiguriert werden, wird jedoch nicht ausgeführt. Wird ein Task «Template» gestartet, so wird dieses geklont und der neue Task (ohne Template-Attribut) wird ausgeführt.

4.1.7 JOB TEMPLATE

Die Job Template Klasse ist simpel und enthält nebst Namen und Beschreibung den Pfad zum Python Skript. Ein Job Template ist die Basis, auf welcher ein Task erstellt wird.

4.2 LOGISCHE ARCHITEKTUR

4.2.1 FRONTEND

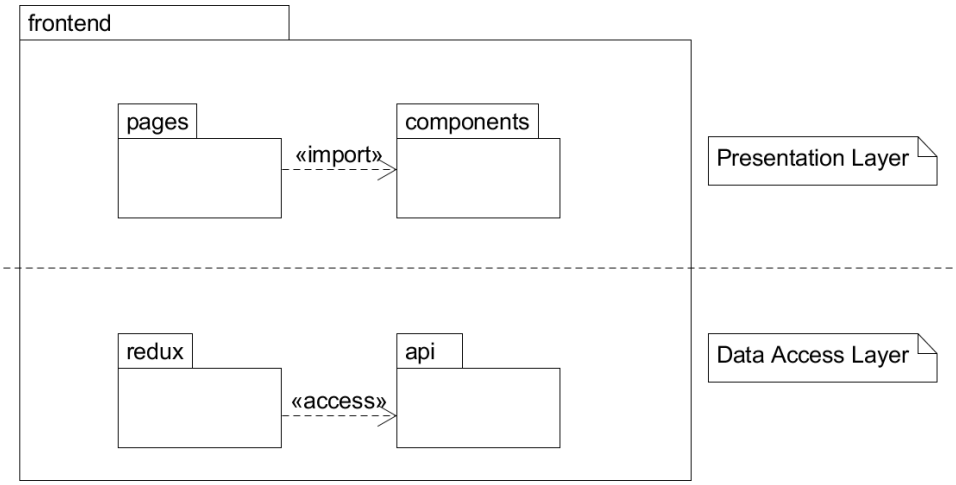


Abb. 4.2: Architekturdiagramm Frontend

In den Packages «pages» und «components» sind React-Komponenten enthalten. Pages sind die Seiten, die im Frontend per URL aufgerufen werden (Inventory View, Task View, …). Um die Wiederverwendbarkeit von einzelnen Komponenten zu erhöhen und weniger duplizierten Code zu schreiben, werden einzelne Komponenten in separate components ausgelagert. Die pages greifen auf die benötigten components zu.

Beide Packages pages und components können sowohl auf den Redux-State als auch auf das API zugreifen. Im Redux-State sind global relevante Informationen zum aktuellen Stand im Task-Wizard, zum Benutzer oder zum ausgewählten Inventory gespeichert.

4.2.2 BACKEND

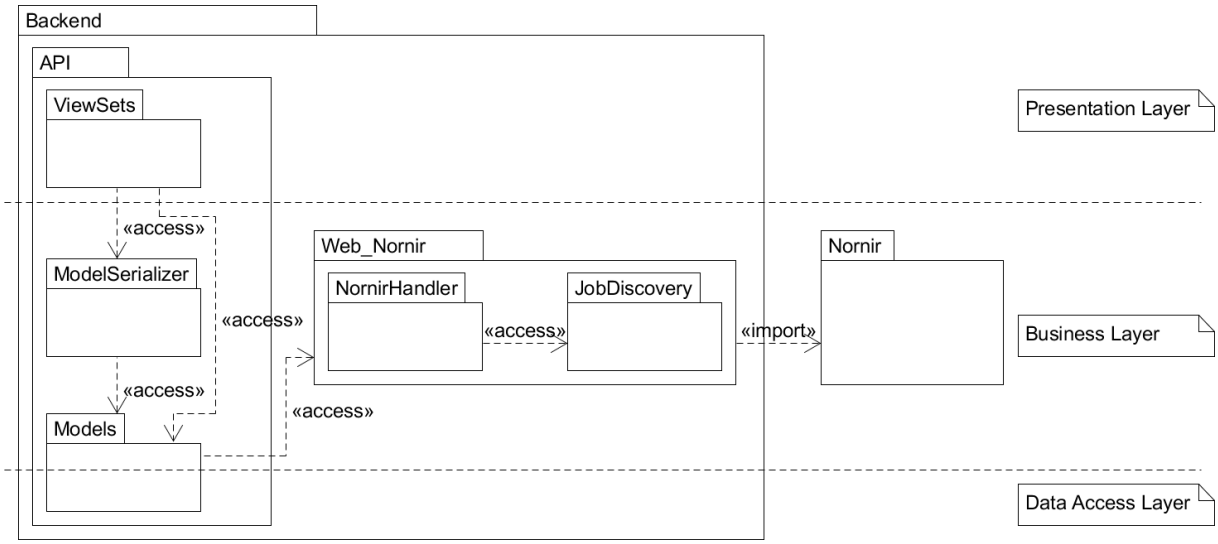


Abb. 4.3: Architekturdiagramm Backend

Durch das Django Framework ist der logischen Architektur des Backends eine gewisse Struktur vorgegeben. Dies betrifft das Package «API», welche die RESTful http-Schnittstelle zur Verfügung stellt. Diese besteht aus den «ViewSets» welche die API-Endpoints darstellen, den «Serializers» welche die Objekte in webtaugliche Austauschformate serialisieren, sowie den «Models». Die Models enthalten Daten über Tasks, Inventories und Job Templates sowie auch Business-Logik, welche die (a-)synchrone Ausführung von Tasks ermöglicht.

Als Schnittstelle zwischen Nornir und dem API dient das Package «Web_Nornir». Im Package ist der «NornirHandler» enthalten, welcher die Aufrufe vom API in korrekte Aufrufe an Nornir umwandelt und anschliessend die Nornir-Ausführung triggert. Ebenfalls übernimmt der NornirHandler die Formatierung der von Nornir erhaltenen Resultate, um diese in webtaugliche Formate zu serialisieren.

4.3 DEPLOYMENT

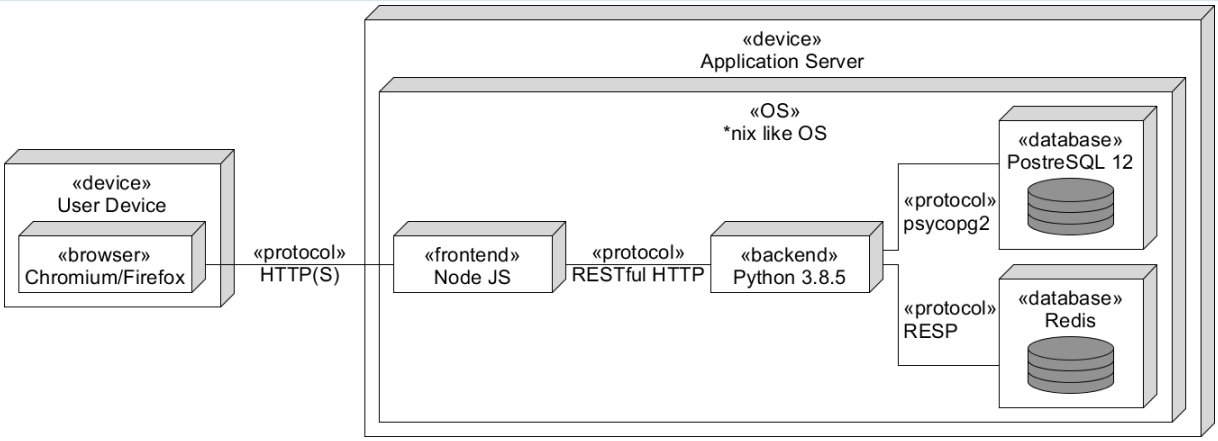


Abb. 4.4: Deploymentdiagramm

Um die Anwendung zu skalieren, können die vier Komponenten Frontend, Backend, PostgreSQL DB und Redis DB auf eigene Hosts ausgelagert werden. Zusätzlich kann zur Skalierung im Backend die Anzahl Worker erhöht werden damit mehr Tasks zeitgleich verarbeitet werden können.

Redis dient in dieser Applikation als Message Broker und wird für die asynchrone Taskausführung verwendet.

4.4 SYSTEMSEQUENZDIAGRAMME

5.4.1 SCHEDULE TASK

Dieses Diagramm zeigt die Verwendung der eingebauten Komponenten bei der Ausführung eines Tasks.

Beim Erstellen eines Tasks wird im Task-Model ein Celery-Task erstellt. Celery kommuniziert von sich aus mit Redis und startet dann sein Task-Scheduling. Aus Sicht des Entwicklers hat man keinen Kontakt mit Redis, dies wird vollautomatisch von Celery übernommen.

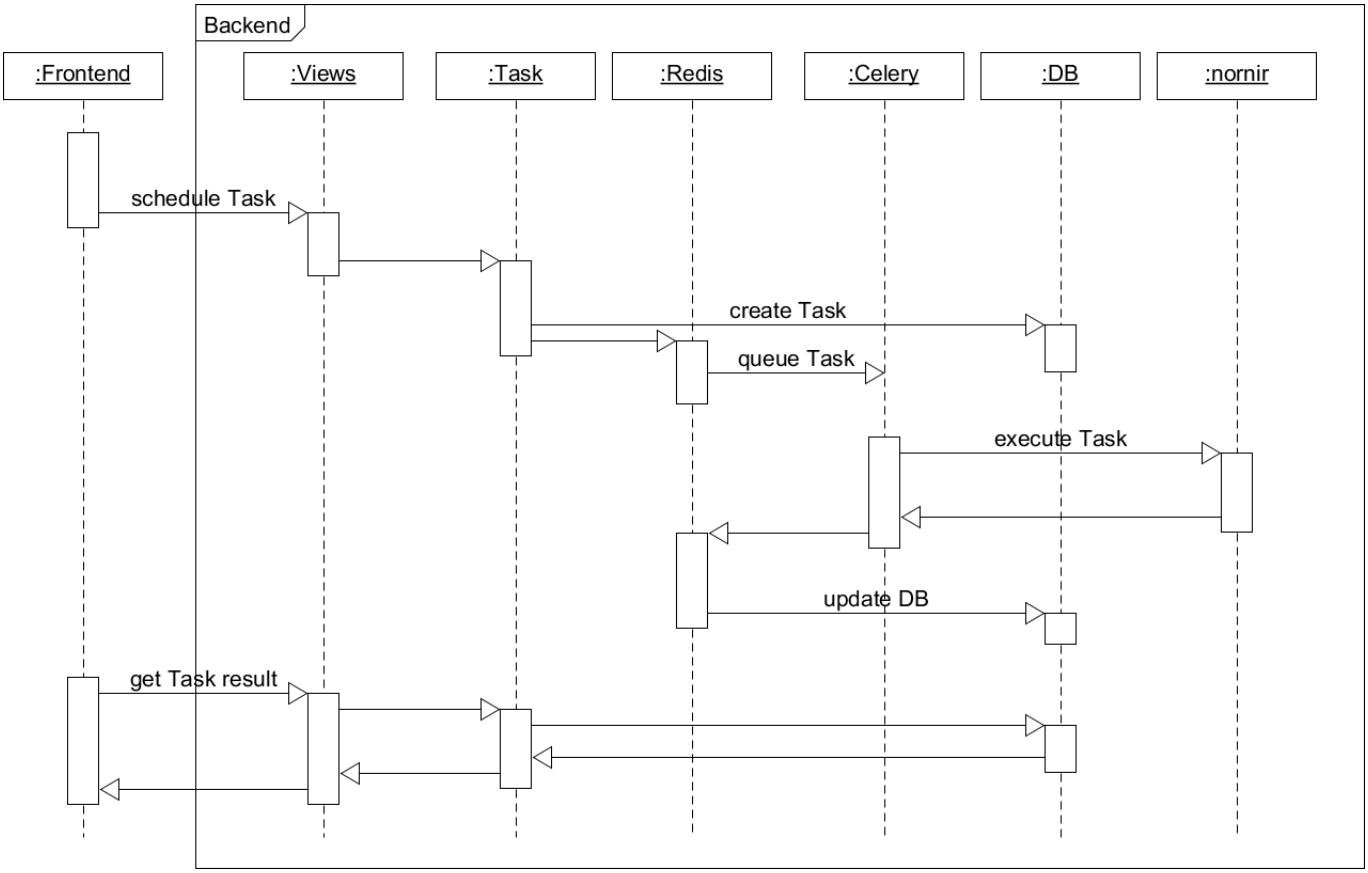


Abb. 4.5: Systemsequenzdiagramm beim Ausführen eines Task

4.5 DATENBANKMODELL

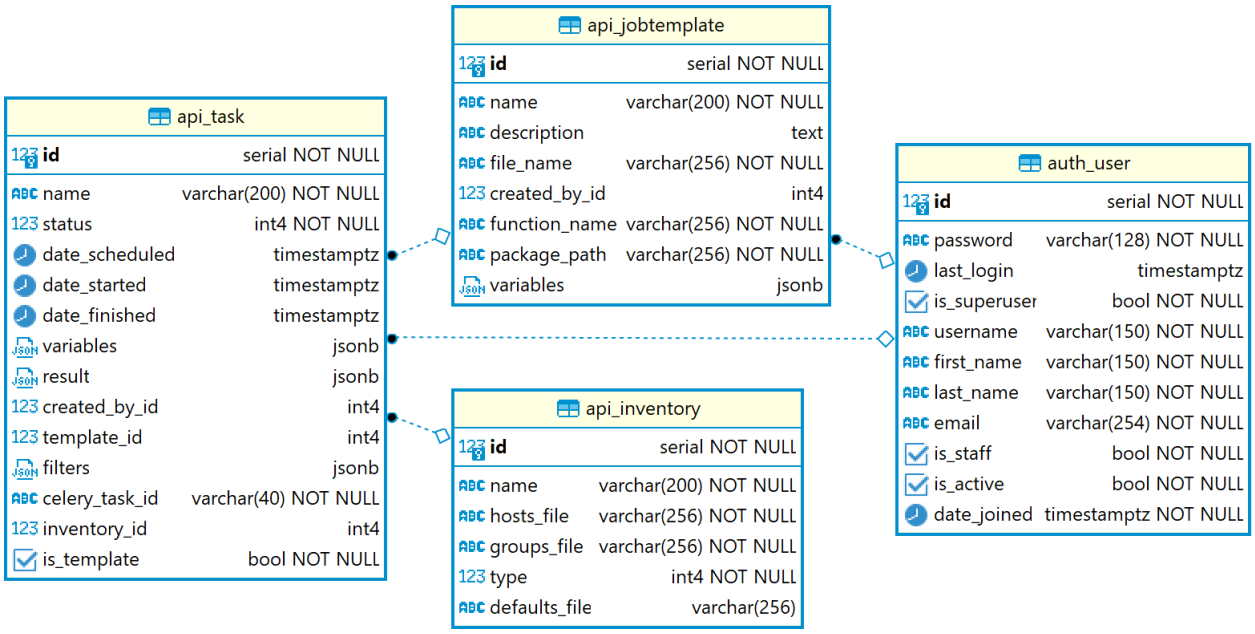


Abb. 4.6: Datenbankmodell

Das Datenbankmodell wird dank dem Django Framework direkt aus dem im Code abgebildeten Domainmodell generiert. Dazu werden weitere Tabellen, wie etwa für die Benutzerverwaltung und Berechtigungen, generiert. Diese Tabellen sind mit Ausnahme der «auth_user» nicht im obigen Datenbankmodell aufgezeichnet, um ein übersichtlicheres Modell zu zeigen. Die Tabelle «auth_user» ist in diesem Diagramm enthalten, um die Verknüpfungen zwischen Benutzern und Job-Template oder Task aufzuzeigen.

5 PROGRAMMIERSPRACHEN, FRAMEWORKS, TOOLS

Gemäss Aufgabenstellung soll eine Webapplikation für das Automationsframework Nornir entstehen. Nornir wird in Python entwickelt, weshalb es naheliegend ist, das Backend ebenfalls in Python zu schreiben. So können die bestehenden Schnittstellen direkt verwendet werden.

Für die Weboberfläche werden die gängigen Sprachen JavaScript, CSS und HTML zum Einsatz kommen.

Die Anwendung wird nach Abschluss der Arbeit an das «INS Institute for Networked Solutions» übergeben. In Absprache mit dem INS wird deshalb im Backend das Python-Framework Django mit der Erweiterung Django Rest Framework verwendet, um das Backend als RESTful http API aufzubauen.

Für die Web-Oberfläche wird das Web-Framework React zum Einsatz kommen, da für dieses am INS ebenfalls schon Knowhow vorhanden ist. Als Design-Bibliothek wird Material-UI verwendet.

Als Datenbank wird PostgreSQL verwendet, da das Entwicklerteam damit bereits Erfahrung hat. Dadurch, dass Django Datenbankzugriffe abstrahiert, kann ohne grössere Anpassungen am Code die verwendete Datenbanktechnologie ausgetauscht werden.

Zusätzlich zur PostgreSQL Datenbank, welche Daten persistiert, wird noch die In-Memory-Datenbank Redis verwendet. Diese wird als Zwischenspeicher und Message-Broker für die Task-Scheduling-Software Celery benötigt. Mithilfe von Celery kann man Tasks asynchron ausführen und einen Task zu einem späteren Zeitpunkt ausführen lassen.

6 IMPLEMENTIERUNG

6.1.1 BACKEND

6.1.1.1 RESTFUL HTTP API

Für das RESTful HTTP API wird das Django Rest Framework verwendet, welches viel Funktionalität anbietet, dem Entwickler einiges an Arbeit abnimmt und sich gut erweitern lässt. Ausserdem ist es das meistgenutzte Framework, um mit Django RESTful HTTP APIs zu entwickeln.

Das API hat Komfortfunktionen wie Sortierung, Filter und Suche eingebaut. Um die Menge an übertragenen Daten zu reduzieren verfügt das API zusätzlich über eine Pagination und es können Datenfelder von der Übertragung ausgeschlossen werden. Dies ist vor allem wichtig, wenn bereits ausgeführte Tasks ein umfangreiches Resultat gespeichert haben. All diese Features können per URL-Parameter verwendet werden.

Um die API automatisch zu dokumentieren, wird ein Open-API Schema generiert und Swagger verwendet, um dieses schön darzustellen. Diese Definitionen können vom Server über eine URL dynamisch abgerufen werden. Die angezeigten Möglichkeiten widerspiegeln immer die Berechtigungen des aktuell angemeldeten Benutzers, d.h. ein Benutzer mit mehr Rechten, sieht entsprechend auch mehr Endpoints in der API-Dokumentation.

6.1.1.2 NORNIR AUSFÜHRUNG

Sämtlicher Code, der mit Nornir direkt interagiert, ist im Package «web_nornir» enthalten.

Die wichtigste Klasse des Pakets ist der «NornirHandler». In dieser Klasse wird eine Nornir Instanz mit den YAML-Dateien für das Inventar sowie der globalen Konfiguration initialisiert. Die Referenzen zu den Inventar-Dateien erhält der NornirHandler aus der Datenbank, der Pfad zur globalen Nornir-Konfiguration ist hart codiert. Die Konfigurationsdatei selbst kann aber bearbeitet werden. Dafür stellt der NornirHandler Methoden bereit, von welchen das API Gebrauch macht.

Die Klasse stellt Methoden zum Abrufen von Daten bereit, wie zum Beispiel Informationen zu einem Host, oder einer Liste von Hosts, sowie gewissen Standardkonfigurationen.

Zudem wird die Methode «execute_task» bereitgestellt, über die ein Task ausgeführt werden kann. Diese Methode nimmt ein Job Template, ein Parameter-Dictionary und ein Filter-Dictionary entgegen. Wenn das Filter-Dictionary ein «host» Array beinhaltet, wird das Nornir-Objekt mittels «F»-Filter nach den angegebenen Hosts durchsucht. Der Rest des Dictionary wird mit einem normalen «filter»-Aufruf verarbeitet. Das Filtern nach angegebenen Hosts ist vor allem wichtig, da man diese im Frontend per Checkbox selektieren kann.

Der Rückgabewert ist ein serialisiertes Nornir-Resultat. Da Nornir-Resultate von sich aus nicht serialisierbar sind, findet sich in diesem Package auch ein Serializer für diese Objekte vor.

6.1.1.3 NORNIR INVENTAR

Anstatt eine eigene Inventar-Verwaltung zu implementieren wird das von Nornir mitgelieferte SimpleInventory verwendet. Die Datenbank speichert nur die Referenzen zu den entsprechenden YAML-Dateien ab. Wenn man über das Webinterface die Anzeige des Inventory aufruft, wird Nornir mit den entsprechenden Referenzen zu den Host-, Group- und Default-Dateien aufgerufen und liefert das Inventar an das API aus. Die Host- und Group-Datei sind obligatorische Eingaben, die Default-Datei ist optional und kann weggelassen werden. Wenn diese nicht angegeben ist, wird Nornir mit einer im Projekt mitgelieferten leeren Default-Datei geladen.

Damit auch im Inventar die gleichen Funktionen für Sortierung und Filter möglich sind wie bei üblichen Django Ressourcen, wurden Hilfsklassen geschrieben, welche die Pagination und Sortierung nachrüsten. Das SimpleInventory bietet solche

Funktionen nicht an. Für die Suche und den Filter hingegen wird direkt auf Nornir-Funktionen zurückgegriffen. Es wird dafür die F-Funktion verwendet.

Bei der Abfrage nach einer Liste von Hosts können die erhaltenen Objekte zwar direkt serialisiert werden, es gehen jedoch einige Datenfelder der Hostobjekte verloren, die nicht automatisch mit serialisiert werden. Deshalb wurden Hilfsfunktionen entwickelt, welche die passenden Informationen aus dem Inventory extrahieren und über das API bereitstellen.

6.1.1.4 NORNIR KONFIGURATION

Neben den Dateien für das Inventar verlangt Nornir auch eine globale Konfigurationsdatei, ebenfalls im YAML-Format. In dieser Datei sind Konfigurationen fürs Logging, die Ausführung und zu verwendende Plugins enthalten. Damit sich ein Administrator bei Änderungen dieser Konfiguration nicht auf dem Backend-Server anmelden muss, wurde dafür ein API-Endpoint implementiert, der im Frontend von der Konfigurations-Maske verwendet wird. Da die Konfiguration als YAML vorliegt, kann sie automatisch serialisiert werden und es muss kein Serializer geschrieben werden.

Vorsicht ist bei der Benutzung dieser Funktion trotzdem geboten: Es wird direkt die configuration.yaml-Datei überschrieben, es wird keine History der vorherigen Konfiguration gespeichert. Ebenfalls wird die Validität der Konfiguration nicht überprüft. Im schlimmsten Fall kann eine fehlerhafte Konfiguration zum Absturz des Backends führen und es können keine Tasks mehr ausgeführt werden. Ausführende Informationen dazu im Kapitel 9.1.1 Einschränkungen Globale Nornir Konfiguration

6.1.1.5 SERIALISIERUNG DER RESULTATE

Die Resultate der Nornir-Tasks können als unterschiedliche Objekte des Typs «AggregatedResult», «MultiResult» oder «Result» daherkommen. Diese Objekte haben allesamt keinen Serializer oder Methoden, um den gesamten Inhalt des Resultats in einer serialisierbaren Form an das Frontend weiterzugeben.

Deshalb wurde in diesem Projekt einen eigener Serializer entwickelt, der die Resultate so aufbereitet, dass diese von Django automatisch serialisiert werden können. Der Serializer orientiert sich an der Struktur des Nornir-Tools «print_result».

Aufgrund der Wahl der Datenstruktur ist der eigene Serializer nicht so hochflexibel wie sein Vorbild «print_result». Der Serializer verlangt immer ein AggregatedResult und verschachtelte MultiResult können nicht serialisiert werden. Weitere Details dazu im Kapitel 9.1.2 Einschränkungen Result Serializer

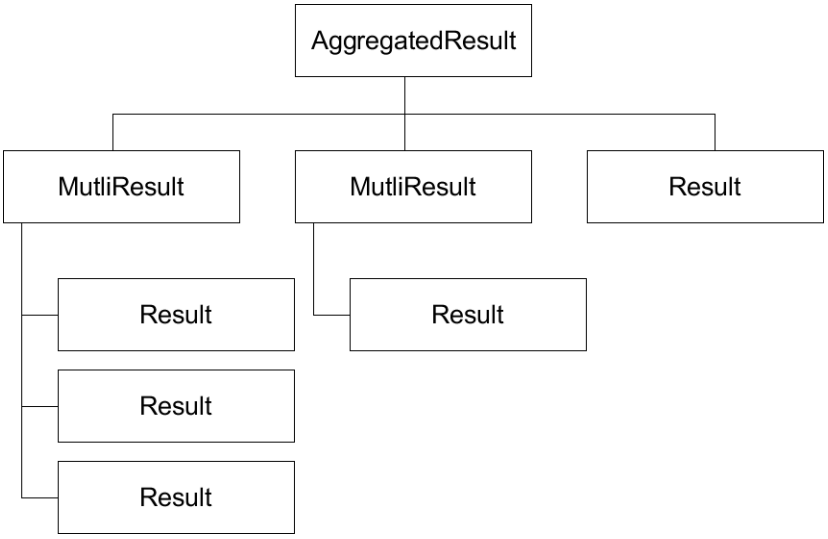


Abb. 6.1: Mögliche Struktur eines Resultats, welches serialisiert werden kann.

Da der NornirHandler Tasks auf eine Liste von Hosts ausführt, kommt in allen Fällen ein AggregatedResult zurück.

Innerhalb eines AggregatedResult unterscheidet der Serializer zwischen einem MultiResult und Result. Beim MultiResult ist immer das erste Resultat der Befehl, der ausgeführt wurde, der Rest die Resultate des Befehls. Diese werden entsprechend extrahiert und in einer Liste gespeichert. Beim Result kann direkt auf das Resultat zugegriffen werden.

Vereinzelt sind die Resultate Objekte und keine Texte, Listen oder Dictionaries. Diese können nicht automatisch serialisiert werden, weshalb sie zuerst als String formatiert werden müssen und erst anschliessend zurückgegeben werden. In diesem Fall wird automatisch die __str__-Methode des Objektes aufgerufen.

Strings, Listen oder Dictionaries werden direkt zurückgegeben. Ein Sonderfall stellt hier wieder das (verschachtelte) MultiResult dar, da dies ebenfalls als Liste gilt, aber nicht automatisch serialisiert werden kann. Hier wird auch ein formatierter String zurückgegeben und es wird in Kauf genommen, dass Teile des Resultats verloren gehen. Ursache ist die __str__-Methode von MultiResult, welche nicht alle Resultate ausgibt.

Während Tests mit Netmiko als auch Napalm als Plugins sind jedoch keine Resultate deswegen verloren gegangen. Es ist davon auszugehen, dass verschachtelte MultiResult nur selten vorkommen.

6.1.1.6 TASK SCHEDULING

Mit Celery wird die asynchrone Taskausführung realisiert. Dazu enthält das Model «Task» zusätzlich zur Methode «run_task» eine Methode «run_task_async». Diese Methode hat den Decorator «@shared_task». Damit wird Celery signalisiert, dass es sich um eine Methode handelt, welche durch Celery ausgeführt werden soll. Beim Schedule des Tasks wird eine «celery_task_id» generiert und das Ausführungsdatum in Celery gespeichert. Zum genannten Zeitpunkt führt dann Celery die Methode run_task_async aus. Mittels dieser celery_task_id kann die Ausführung eines Tasks auch verhindert werden, sofern die Taskausführung noch nicht begonnen hat.

Für den Task gibt es die Status «CREATED», «SCHEDULED», «RUNNING», «FINISHED», «FAILED» und «ABORTED». Wenn ein Task neu erstellt wurde, hat er den Status «CREATED». Wenn man sich für die asynchrone Ausführung entscheidet und eine Ausführungszeit angibt, erhält er den Status «SCHEDULED». Mit diesem Status kann der Task noch abgebrochen werden, infolgedessen er den Status «ABORTED» annehmen würde. Wenn die Ausführung des Tasks gestartet wird, wechselt er in den Status «RUNNING». Um einen Task abzubrechen wird die Celery-Funktion «revoke» aufgerufen. Diese Funktion bricht jedoch keinen laufenden Task ab, da der Prozess der Ausführung nicht gestoppt wird. Man könnte die Ausführung eines laufenden Tasks mit dem Parameter «terminate=True» komplett abbrechen. Dies könnte jedoch zur Folge haben, dass nachfolgende Tasks, die auf diesem Prozess ausgeführt werden sollten, auch nicht ausgeführt werden, da der Prozess komplett terminiert wird. Wegen diesem Verhalten wurde entschieden, das Abbrechen von bereits laufenden Tasks nicht zu unterstützen.

Wenn ein Task erfolgreich ausgeführt wurde, nimmt er den Status «FINISHED» an. Sollte es während der Ausführung Fehler gegeben haben, wird der Status auf «FAILED» gesetzt.

6.1.1.7 JOB TEMPLATES

In den Job Templates ist der Code enthalten, der bei einer Taskausführung ausgeführt wird. Ein Beispiel dafür könnte die Auflistung aller Interfaces eines Netzwerkgerätes sein.

Ähnlich wie beim Inventory enthält die Datenbank für Job Templates nur Referenzen zu den entsprechenden Python-Skripts. Beim Ausführen eines Task wird die entsprechende Datei geladen und durch Nornir ausgeführt.

Durch diesen Aufbau kann die Applikation beliebig mit weiteren Nornir-Tasks erweitert werden. Eine neue Python-Datei wird innerhalb des Backend-Ordner abgelegt, die Datenbank aktualisiert und das neue Job Template kann über das Web-Frontend verwendet werden.

Aus den nachfolgenden Gründen wurde bewusst gegen das automatische Einlesen von Python-Dateien als Alternative zum manuellen Eintragen neuer Templates in die Datenbank entschieden:

- Eine Möglichkeit wäre gewesen, beim Start des Backends alle Python-Dateien innerhalb eines definierten Job Template Ordner einzulesen. Dies hätte aber zur Folge gehabt, dass bei jedem neu hinzugefügten Template das Backend hätte neugestartet werden müssen.
- Eine andere Möglichkeit wäre Polling gewesen. In einem gewissen Zeitabstand hätte ein immer laufender Hintergrundtask das definierte Verzeichnis eingelesen und bei Änderungen die neuen Job Templates der Datenbank hinzugefügt. Auch dies wurde verworfen, da ein zusätzlicher Hintergrundtask die Komplexität unnötig erhöht hätte.

6.1.1.8 AUTHENTICATION UND AUTHORIZATION

Am API wird über ein JWT authentisiert. Dies wird mit der Python-Bibliothek «Simple JWT» implementiert. Der Serializer für das JWT wurde angepasst und erweitert, um im Token auch Informationen über den Benutzer und seine Gruppe mitzugeben. Dies erlaubt das Setzen der richtigen Berechtigungen im Frontend. Im Backend werden diese Gruppen ebenfalls verwendet, um den Zugriff auf gewisse Ressourcen einzuschränken.

Es gibt drei Gruppen, detailliertere Informationen sind im Kapitel 3.1 Use Cases, im Abschnitt «Aktoren» zu finden.

- **Superuser**: Hat Zugriff auf alles und hat keine Einschränkungen.
- **Netadmin**: Hat Leserechte auf die Ressourcen und kann Tasks ausführen.
- **Support**: Hat Leserechte auf alles, jedoch keine Schreibrechte

6.1.1.9 DJANGO ADMIN

Das Django Framework liefert den «Django Admin» mit. Das ist eine Administrationsoberfläche, um Modifikationen an seinen Datenobjekten zu machen. Um das Rad nicht neu zu erfinden und eine eigene Oberfläche zu bauen, um Modifikationen an den Daten in der Datenbank zu machen, wurde Django Admin entsprechend angepasst, um die gewünschte Oberfläche darzustellen. Django Admin wurde so modifiziert, dass nur Benutzer, welche der Gruppe «superuser» angehören darauf Zugriff haben.

Diese Benutzer können in der Admin-Oberfläche neue Inventories, Job Templates oder Tasks hinzufügen, modifizieren oder löschen. Dabei wurden für die einzelnen Models die Admin-Views angepasst, so dass immer die relevanten Informationen zuerst angezeigt werden und Felder die Read-Only sind nicht verändert werden können.

Die Benutzerverwaltung wird ebenfalls über Django Admin geregelt. Hierfür ist ein «Django Superuser» erforderlich. Wie ein solcher Benutzer eingerichtet wird, ist in der Softwaredokumentation oder in der Django-Dokumentation beschrieben.

6.1.2 FRONTEND

6.1.2.1 AUTHENTICATION UND AUTHORIZATION

Das Frontend verwendet die JWT-Tokens, welche vom Backend bereitgestellt werden. Dabei werden vom Backend jeweils zwei Tokens versendet. Eines ist für die Authentifizierung beim Backend (Access-Token) und das andere kann gesendet werden, um das Authentifizierungstoken zu erneuern (Refresh-Token). Das Authentifizierungstoken ist für nur wenige Minuten gültig, das Erneuerungstoken 24 Stunden. Das Token wird im Session Storage des Browsers gespeichert. Wird die Browser Session geschlossen oder das Token läuft aus, wird der Benutzer automatisch ausgeloggt.

Die im Token befindlichen Informationen zum Benutzer und seinen Gruppen werden im Frontend verwendet um Steuerelemente zu aktivieren, deaktivieren oder verstecken. So können zum Beispiel nur Benutzer der Gruppen Superuser und

Netadmin den Button «Create Task with Selection» sehen und nur für Superuser werden die Felder in der Configuration-Page aktiviert.

6.1.2.2 DESIGN

Für das Design im Frontend werden grösstenteils die Elemente der Material-UI Bibliothek für React verwendet. Diese kommt mit einer Reihe von vorgefertigten UI-Elementen und bietet einen konsistenten und sauberen Stil an. Eigenes CSS wurde nur verwendet, wenn nötig und es wurde das Standard Farbschema verwendet.

6.1.2.3 REDUX

Um im Frontend besser mit State umgehen zu können, wird die Redux-Bibliothek für React verwendet. Mithilfe dieser Bibliothek kann jede Komponente gewisse Informationen abrufen und verändern. Zudem kann man diesen State auch mittels API-Aufrufen aktualisieren. Diese Technologie wird in drei Fällen verwendet: für den Benutzer, den Task Wizard und für das Inventar.

Beim Benutzer ist vor allem wichtig, dass von überall auf das Token zugegriffen werden kann, um mit dem Backend kommunizieren zu können und dass man Zugriff auf die Berechtigungen des Benutzers hat, um die Autorisierung im Frontend umsetzen zu können.

Der Task Wizard spannt sich auch über mehrere Komponenten. Deswegen ist es von Vorteil, wenn jeder Teil des Wizards seine Anpassungen gleich selbst speichern kann, ohne dass er auf seinen Parent-Component angewiesen ist. Da die Verwendung von mehreren Inventories unterstützt wird, ist es wichtig, dass man im Frontend nachschauen kann, welches gerade verwendet wird.

6.1.2.4 KOMPONENTEN

Bei einem mit React gebauten Frontend stehen die Komponenten im Zentrum. Diese können geschrieben und dann beliebig kombiniert und geschachtelt werden. Es wurde dabei eine Trennung in Pakete unternommen, um Seiten und Komponenten zu trennen. Die Komponenten im Paket «pages» sind alle an eine URL gebunden, über welche sie aufgerufen werden. Die Komponenten des Pakets «components» sind hingegen nicht gebunden und können wiederverwendet werden.

6.1.2.5 SUCHE UND FILTER

Für die Listen der Tasks, Job Templates und des Inventars gibt es die Möglichkeit, diese nach verschiedenen Attributen zu durchsuchen und zu filtern. Bei der Suchfunktion kann man einen Begriff eingeben, der dann in mehreren Tabellenspalten gesucht wird. Im Gegensatz dazu wird beim Filter jeweils nach einer Tabellenspalte gefiltert. Zudem kann man mehrere Filter gleichzeitig aktiv haben, während man nur nach einem Begriff suchen kann. Diese beiden Funktionen können auch in Kombination angewendet werden.

Für die Suche, den Filter, die Sortierung, sowie die Pagination werden jeweils API-Aufrufe gemacht. Das heisst, es wird für diese Funktionen kein JavaScript verwendet, sondern all diese Parameter werden gesammelt und dann in die URL des Aufrufs integriert. Dieser Ansatz wurde gewählt, weil damit ein möglichst konsistentes Verhalten erzielt wird, welches auch mit grösseren Datenmengen skaliert, da auf diese Weise alle Operationen auf der Datenbank ausgeführt werden.

7 TESTING

7.1.1 BACKEND

Im Backend wurden die Tests mit Pytest durchgeführt. Es wurde eine Mischung aus Unit- und Integration-Tests gemacht, um einerseits die Programmlogik selbst sowie die Zusammenarbeit mit dem Framework zu überprüfen. Als Testdatenbank wird die konfigurierte PostgreSQL Datenbank verwendet. Um fehlschlagende Tests zu vermeiden, weil die Datenbank andere Daten enthält wie im Test erwartet, sollte vor jedem Test die Datenbank zurückgesetzt und mit dem Befehl «python manage.py create_testdb» befüllt werden. Im Continous Integration wird hingegeben eine SQLite Datenbank verwendet, welche bei jedem Test neu aufgesetzt wird. Mit der Umgebungsvariable DB_CONNECTION_STRING kann ein eigener String für die Datenbank hinterlegt werden.

7.1.1.1 TESTABDECKUNG

Mittels Pytest-Cov wird die Testabdeckung überprüft. Die Testabdeckung erreicht 87%.

Um diesen hohen Testabdeckungsgrad zu erreichen, mussten einige Module von der Messung ausgenommen werden. Dazu gehören zum Beispiel die komplette virtuelle Phyton-Umgebung sowie Django-spezifische Python-Dateien wie die settings.py, asgi.py, wsgi.py und viele weitere. Diese werden jeweils durch das Django Projekt getestet und deswegen sind keine weiteren Tests dafür nötig.

Die Ausnahmen sowie weitere Konfigurationen für pytest-cov sind in der Datei «.coveragerc» hinterlegt.

Dateien wie views.py oder models.py sind Teil des Django Frameworks. Da der Code in diesen Modulen viel Autowiring-Funktionen von Django verwendet, wurde von detailreichen Tests abgesehen. Dies erklärt auch, weshalb manche Module in der Detailübersicht keine Testabdeckung von 80% erreichen. Um für diese Module mehr Testabdeckungsgrad zu erreichen, hätten mehr Integration-Tests geschrieben werden sollen. Diese machten jedoch Probleme mit dem Zugriff auf die Datenbank sowie mit der User-Authentifizierung. Aus unerklärlichen Gründen haben diese Tests nicht wie gewünscht funktioniert, weil entweder der Benutzer abgelehnt wurde oder die gewünschten Daten in der Datenbank nicht auffindbar waren. Deshalb wurden weitere Tests für diese Module weggelassen.

Es ist davon auszugehen, dass die Django Entwickler eine genügend hohe Testabdeckung dieser Module erreichen, sodass eine fehlerfreie Funktion angenommen werden kann.

Die einzelnen Job-Template Module wurden ebenfalls nicht explizit getestet, da auch dort auf weitere Pakete wie Napalm oder Nornir zugegriffen wird. Auch von diesen Modulen wird angenommen, dass diese ausreichend getestet sind. Bei einigen Job-Templates wird trotzdem eine 100%-Testabdeckung erreicht. Der Grund dafür ist, dass diese Module für einige Unit-Tests des NornirHandler verwendet wurden und deshalb auch eine Testabdeckung dafür erreicht wird.

7.1.2 FRONTEND

Es wurde bewusst am Anfang des Projekts dazu entschieden die Tests im Frontend auf ein Minimum zu beschränken, weil diese Tests erfahrungsgemäss nicht sehr kosteneffizient sind. Dies lässt sich auf die hohe Flexibilität und Änderbarkeit des Frontends zurückführen, da sich dort Anforderungen viel öfter ändern als bei einer Backend-Anwendung.

7.1.3 USABILITY TESTS

Während Meilenstein 4 wurden Usability Tests mit der Anwendung durchgeführt. Es wurden vier gängige Testszenarien ausgearbeitet, welche von insgesamt vier Testpersonen durchgeführt wurden. Von allen Testpersonen wurde viel wertvolles Feedback erhalten, welches zur Verbesserung der Oberfläche geführt hat.

Dank diesen Tests konnte die Anwendung bedienerfreundlicher und konsistenter gestaltet werden.

8 ERGEBNISSE

Alle erforderlichen Use Cases konnten umgesetzt werden. Das Ergebnis geht über das Minimum Viable Product hinaus.

Die Anwendung besitzt folgende Funktionen:

- Geräte eines Inventars auflisten
- Details eines Geräts anschauen
- Zwischen mehreren Inventaren hin und her wechseln
- Job Templates auflisten und Details anzeigen
- Tasks geführt im Wizard erstellen und (geplant) ausführen
- Tasks auflisten und Details sowie Resultate eines Task anzeigen
- Tasks mit den gleichen Einstellungen erneut ausführen.
- Tasks vorkonfiguriert aber ohne Ausführungsdatum als Schnellzugriff speichern
- Filter-, Sortier-, Suchfunktion für Inventare, Tasks sowie Job Templates
- Globale Nornir Konfiguration anzeigen und verändern

8.1.1 OFFENE PUNKTE

8.1.1.1 WEBSOCKETS

Wird ein Task direkt ausgeführt, so erhält das Frontend keine automatischen Updates. Als Überbrückungsfunktion wurde ein Refresh-Button eingebaut. Der Einbau von Websockets mittels dem Python Modul Channels hätte erheblichen Mehraufwand verursacht. Nicht nur das Backend hätte in einigen Bereichen umgebaut werden müssen, auch im Frontend hätten umfangreichere Arbeiten getätigt werden müssen. Deshalb wurde von Anfang an entschieden, andere Arbeiten höher zu priorisieren.

8.1.1.2 ERWEITERTE FILTER

Die Filter-Funktion im Backend verwendet schon für die simplen Filter die F-Funktion von Nornir. Um Code Injection zu verhindern, erlaubt das Backend sowie das Frontend nur bestimmte Filterkriterien. Diese Filterkriterien werden direkt an die F-Funktion weitergegeben. Um die Filter-Funktion zu erweitern, müssen jeweils im Backend und im Frontend die erlaubten Kriterien erweitert werden.

Im Frontend müsste zusätzlich eine neue Eingabemaske für den erweiterten Filter erstellt werden.

8.1.1.3 JOB TEMPLATE MANAGEMENT

Das Backend hat bereits die nötigen Funktionen eingebaut, um neue Job Templates hinzuzufügen, zu bearbeiten oder zu löschen. Da im Frontend die Arbeit auf Task-Ausführung und die Aufarbeitung der Resultate konzentriert wurde, konnte die Funktion nicht im Frontend umgesetzt werden. Da das Backend jedoch schon vorbereitet ist, muss nur das Frontend angepasst werden.

Als Übergangslösung kann das Django Admin-Panel verwendet werden.

8.1.1.4 TASK BUILDER

Der Task Builder war als optionales Ziel vorgesehen. Dieser sollte es dem Netzwerkadministrator ermöglichen, aus vorgefertigten Blöcken eigene Job Templates zu erstellen, anstatt diese in Python zu schreiben und im Backend zu hinterlegen. Dies ist eine sehr umfangreiche Aufgabe, da zum einen diese Blöcke definiert werden müssen und es hier eine schier unendliche Menge gibt und es sehr schwierig ist, daraus die wichtigsten oder häufigsten Blöcke auszuwählen.

Die nächste Hürde ist das Überprüfen dieser Block-Kombinationen. Es müsste ein Validator entwickelt werden, der die Kombinationen der einzelnen Blöcke interpretieren kann und entscheiden kann, ob ein gültiges Job Template vorliegt. Aus der Computertheorie ist allgemein bekannt, dass solche Probleme nur sehr schwer bis unmöglich zu lösen sind.

8.1.1.5 INVENTAR MANAGEMENT

Die Anwendung unterstützt nur Inventare vom Typ SimpleInventory. Das optionale Feature, dass weitere Inventar-Typen unterstützt werden, konnte nicht umgesetzt werden. Die Datenstruktur ist jedoch bereits dafür vorbereitet, dass das Inventar um weitere Optionen wie z.B. Daten aus IPAM oder GitHub erweitert werden kann.

Ebenfalls kann das Inventar nicht über das Frontend bearbeitet werden. Das Backend ist zu einem gewissen Grad dafür bereits vorbereitet. Das Backend erlaubt schon jetzt das Hinzufügen, Bearbeiten oder Löschen von Inventories. Um im Frontend das Inventar komfortabel zu bearbeiten fehlt im Backend die Option einzelne Hosts oder Gruppen zu bearbeiten. Das Frontend hingegen kann aktuell nur anzeigen und müsste noch um entsprechende Bearbeitungs-Funktionen erweitert werden.

Auch hier kann als Überganslösung Django Admin verwendet werden.

9 EINSCHRÄNKUNGEN

9.1.1 GLOBALE NORNIR-KONFIGURATION

Nornir bietet keine Möglichkeit an, die Konfigurationsdatei auf Fehler zu überprüfen. Die Implementierung eines eigenen Prüfmechanismus hätte den Scope gesprengt, weshalb auf eine Fehlerüberprüfung der Konfiguration gänzlich verzichtet wurde. Änderungen der Konfiguration über das Frontend werden dementsprechend direkt und ohne Prüfung in die Konfigurationsdatei geschrieben.

Es ist darauf zu achten, bei der Eingabe der Konfiguration keine fehlerhaften Werte mitgeben und der konfigurierende Administrator weiss, was er tut.

Eine Referenz welche Konfigurationsparameter Nornir anbietet, ist hier zu finden: https://nornir.readthedocs.io/en/latest/configuration/

9.1.2 RESULT SERIALIZER

Um die Komplexität der Implementierung kleiner zu halten, können im Moment nur AggregatedResult mit Result oder MultiResult ohne Verschachtelung aus Tasks verarbeitet werden.

Dies liegt einerseits an der Art wie Resultate miteinander kombiniert und unendlich verschachtelt werden können, anderseits am Implementierungsvorbild «print_result» welches mittels Rekursion diese Daten extrahiert. So sind zum Beispiel folgende Kombinationen möglich: AggregatedResult enthält MultiResult, ein Resultat innerhalb des MultiResult kann wiederum ein MultiResult sein, welches mehrere Results enthält.

Der selbstentwickelte Serializer arbeitet ohne Rekursion, da innerhalb der Methode mehrere Werte dem Attribut «result» zugewiesen werden - siehe Beispiel unten.

```
for hostname, host_result in sorted(aggregated_result.items()):
    host_dict = {
        'hostname': host_result.host.hostname,
        'name': host_result.host.name,
        'failed': host_result.failed,
        'result': []
    }
```

Würde dieser Serializer rekursiv aufgerufen werden, so würde ein verschachteltes MultiResult die Folge haben, dass anschliessend im serialisierten JSON unnötig tief mit dem Key «result» verschachtelt wird. Der selbstentwickelte Serializer gibt als Resultat ein Dictionary mit Informationen zum Host sowie den Key «Result» zurück, in welchem dann das Resultat gespeichert ist. Würde man dies nun rekursiv aufrufen, hätte jedes verschachtelte Resultat wieder die Informationen des Hosts sowie eine weitere Verschachtelungsstufe des Resultats. Angenommen, ein Resultat hätte drei Verschachtelungen eines MultiResult, dann würde das resultierende Dictionary so aussehen:

```
{'hostname': '10.20.0.201', 'name': 'spine1', 'failed': False, 'result': {
    'hostname': '10.20.0.201', 'name': 'spine1', 'failed': False, 'result': {
        'hostname': '10.20.0.201', 'name': 'spine1', 'failed': False,
        'result': 'Resultat des verschachtelten Task'}
    }}
```

Der Umbau auf des Serializers auf eine rekursiv aufrufbare Methode hätte den Zeitrahmen gesprengt, weshalb entschieden wurde, nur die Unterscheidung zwischen MultiResult und Result ohne mehrfache Verschachtelung zu respektieren.

9.1.3 PYTHON KOMPATIBILITÄT

In diesem Projekt wird als Dependency Napalm in der Version 3.2.0 verwendet. Diese Version enthält Code, welcher mit 3.9 nicht mehr funktionieren wird. Das Projekt wurde nur mit Python 3.8.5 getestet. Soll eine neuere Python-Version zum Einsatz kommen, muss überprüft werden, ob es bereits einen neueren Release von Napalm gibt:

https://github.com/napalm-automation/napalm/releases

9.1.4 WINDOWS KOMPATIBILITÄT

Aufgrund von Einschränkungen des Python Moduls «Celery» läuft die Applikation nur auf Unix-basierten Betriebssystemen und ist nicht mit Windows kompatibel.

10 PROJEKTMANAGEMENT

10.1 ÜBERSICHT

Am Ende der Studienarbeit existiert eine Webapplikation, die das Steuern des Netzwerkautomations-Werkzeug Nornir nach den Wünschen der Auftraggeber erlaubt.

10.1.1 LIEFERUMFANG

- Dokumentation mit Anhängen
- Technischer Bericht
- Gitlab Repository mit Programmcode, Issues
    - Online-Link: https://gitlab.dev.ifs.hsr.ch/sa-gygax-kubli/web-app-nornir/
- Zeitauswertung

ANNAHMEN UND EINSCHRÄNKUNGEN

Diese Arbeit wird im Rahmen einer Studienarbeit durchgeführt. Gemäss Reglement haben beide Projektmitglieder einen Gesamtaufwand von je 240 Stunden über 14 Wochen zu leisten.

Nornir 3.0, Python, Django und React.js sollen verwendet werden, da diese Technologien bereits im INS eingesetzt werden.

10.2 PROJEKTORGANISATION

An diesem Projekt sind mehrere Parteien beschäftigt, welche in der folgenden Übersicht aufgelistet werden:

Name	Funktion
Beat Stettler	Betreuer
Matthias Gabriel	Ansprechperson Software-Entwicklung
Urs Baumann	Ansprechperson Netzwerkautomation
Felix Kubli	Student an Studienarbeit
Jérôme Gygax	Student an Studienarbeit

Tabelle 10.1: Projektorganisation

10.3 MANAGEMENTABLÄUFE

10.3.1 ZEITPLANUNG

Für die Zeitplanung wird das im Modul Software Engineering gelernte «Scrum+» verwendet, welches zum einen die «Gates» aus Unified Process übernimmt, aber innerhalb dieser Phasen werden wie von Scrum gewohnt Sprints durchgeführt.

Sprints haben eine Dauer von 2 Wochen. Der erste Sprint für Meilenstein 1 «Projektplan» wird nur 1 Woche dauern, so dass genügend Zeit für die Architekturspezifikation, Requirements und Construction-Phase vorhanden ist.

Abb. 10.1: Mischform Unified Process mit Scrum

10.3.1.1 PHASEN / ITERATIONEN

Inception	In der Inception Phase wird mit dem Betreuer den Umfang des Projektes besprochen, wie in etwa die Lösung aussehen soll und welches Tooling wünschenswert ist.
Elaboration	Anhand der Wünsche des Betreuers wird in der Elaboration-Phase der Projektplan erstellt, die Anforderungen spezifiziert sowie ein erster Architekturprototyp erstellt
Construction	In der Construction-Phase wird die Applikation entwickelt. Diese teilt sich in drei Meilensteine auf: • Meilenstein Core Construction in welcher die wichtigsten Kernkomponenten entwickelt werden; • Im Meilenstein Optionale Features werden optionale Funktionen entwickelt, sofern es die Zeit erlaubt; • Mit Erreichen des Meilenstein Optionale Features wird ein Feature Freeze gemacht. Während der Construction Phase werden Usability Tests durchgeführt.
Transition	Es werden keine neuen Features mehr eingebaut, es wird nur noch Bugfixing durchgeführt. Abgabe der Projektarbeit sowie Installation der Applikation auf dem Zielsystem.

Dauer der einzelnen Phasen:

Phase	Datum von	Datum bis
Inception	11.09.2020	11.09.2020
Elaboration	12.09.2020	16.10.2020
Construction	17.10.2020	27.11.2020
Transition	28.11.2020	18.12.2020

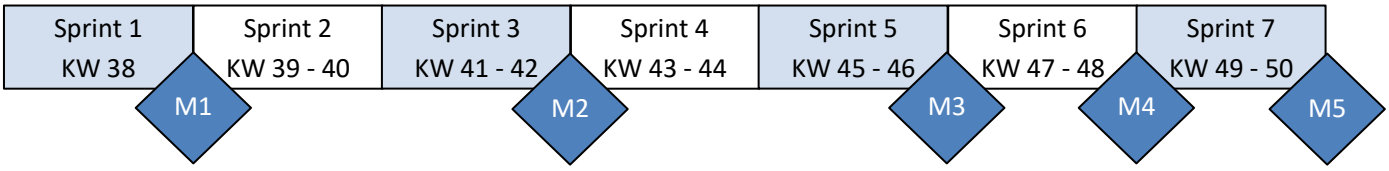
Tabelle 10.2: Dauer der Phasen

10.3.1.2 MEILENSTEINE

Bezeichnung	Datum	Beschreibung
M0 Kick-Off Studienarbeit	11.09.2020	Mit Betreuer Rahmen der Arbeit besprochen
M1 Projektplan	18.09.2020	Projektplan erstellt und mit dem Betreuer besprochen
M2 End of Elaboration	16.10.2020	Prototyp erstellt, Arbeitspakete geplant und geschätzt
M3 Core Construction	13.11.2020	MVP erstellt nach den Wünschen des «Kunden»
M4 Optionale Features	27.11.2020	Optionale Features implementiert, gemäss Priorisierung des Kunden. Feature-Freeze mit Erreichen des Meilensteins
M5 Final Submission	11.12.2020	Feature Freeze. Der Code ist Feature-mässig komplett und es werden nur noch Korrekturen vorgenommen. Die Dokumentation ist zu 80% fertig
M6 Projektpräsentation	18.12.2020	Präsentation des Projekts Abschliessende Arbeiten: Finalisierung des Codes und Dokumentation

Tabelle 10.3: Meilensteine

10.3.2 SPRINTPLANUNG



Sprint	Datum von	Datum bis	Geplante Tätigkeiten
1	12.09.2020	18.09.2020	• Projektplan erstellen • Risikoanalyse
2	19.09.2020	02.10.2020	• Use Cases erstellt und mit INS besprochen • Anforderungsspezifikation • Domainmodell erstellt und abgenommen • Pipelines konfiguriert • Projekte mit Leerprojekt erstellt • Server bestellt und aufgesetzt
3	03.10.2020	16.10.2020	• Prototyp erstellen • Domainmodell aktualisieren • Wireframes anpassen • Risikoanalyse aktualisieren mit neuer Beurteilung

4	17.10.2020	30.10.2020	• Architekturdesign finalisieren • Issues für Core Construction erstellen, priorisieren, schätzen, umsetzen • Verbindung zu Test-Netzwerkgeräten testen • Server in Betrieb nehmen
5	31.10.2020	13.11.2020	• Task Wizard und Detail Views sind fertig • Tasks kann im UI gestartet werden und Resultat wird angezeigt. • Mehrere Inventories können angezeigt werden
6	14.11.2020	27.11.2020	• Authentication mit JWT • Tasks können erneut ausgeführt werden (Ohne Verlust des Resultats → Task klonen) • Tasks können konfiguriert und als Template gespeichert werden • Usability Tests
7	28.11.2020	11.12.2020	• Detailverbesserungen im UI • Refactoring & Überarbeitung der Codebasis • Dokumentation überarbeiten und erweitern • Gestaltung Plakat
8	12.12.2020	18.12.2020	• Finalisierung Dokumentation • Abgabe & Projektpräsentation

Tabelle 10.4: Sprint-Planung

10.3.3 BESPRECHUNGEN

Besprechungen finden jeweils freitags von 10:30 – 11:30 Uhr statt.

10.3.4 WICHTIGE TERMINE

Termin	Datum
Abgabe Projekt und Dokumentation	18.12.2020
Präsentation Projekt	18.12.2020

Tabelle 10.5: Wichtige Termine

10.4 RISIKOMANAGEMENT

Die Risikoanalyse ist im Kapitel 12 Risikoanalyse zu finden und wird während des Projekts laufend aktualisiert.

In der Detailplanung der Arbeitspakete wird darauf geachtet, dass innerhalb der Sprints genügend Zeit als Puffer reserviert wird, um allfällige Rückstände durch eingetretene Risiken auszugleichen.

10.5 QUALITÄTSMASSNAHMEN

Um die Qualität des Python-Codes zu gewährleisten wird Pylint als Linting-Tool verwendet. Im Frontend wird ESLint für React für denselben Zweck verwendet.

Zudem wird für jedes Feature ein Feature Branch erstellt, für den dann jeweils ein Codereview gemacht wird.

Gegen Ende der Construction Phase werden Usability Tests durchgeführt, um sicherzustellen, dass die Applikation gut bedienbar und verständlich ist.

10.5.1 TEST COVERAGE

Im Backend wird eine Mischform von Integrations- und Unit-Tests angewendet. Es soll eine Test Coverage von mindestens 80% erreicht werden.

Das Frontend ist nur schwer testbar, weshalb auf Integration-/Unit-Tests verzichtet wird, dafür aber Usability Tests durchgeführt werden.

10.6 INFRASTRUKTUR

Folgende Technologien sollen bei der Umsetzung des Projekts zur Anwendung kommen:

- React als Frontend-Framework mit JavaScript
- Django mit Python als Backend-Framework
- Nornir 3.0 als Automatisierungsframework, für welches die Applikation geschrieben wird.
- PostgreSQL als Datenbank
- Pytest als Testing Library

Für die Versionierung wird Git verwendet, zusammen mit der Gitlab-Instanz der HSR. Dabei wird mit einem master, einem develop und Feature-Branches gearbeitet. Nach dem Erreichen eines Meilensteins wird jeweils der develop in den master Branch gemergt werden und alle Feature-Branches werden in den develop gemergt.

10.6.1 CI/CD

Über Gitlab werden bei jedem Push die Tests und der Linter ausgeführt. Pull Requests werden nur gemergt, wenn die Tests und der Linter erfolgreich durchgelaufen sind.

10.7 ARBEITSPAKETE

Arbeitspakete werden als Issues erfasst und mit einem der unten aufgelisteten Labels markiert, so dass eine Zeitauswertung über die unterschiedlichen Tätigkeiten möglich ist.

Tätigkeit	Beschreibung
Planung	Besprechungen des Projekts über weiteres Vorgehen im Team
Dokumentation	Dokumentieren des Projektfortschritt und der Arbeitspakete
Design & Architektur	Design sowie auch Architektur-Tätigkeiten
Infrastruktur	Konfiguration, Deployment der Infrastruktur
Entwicklung Frontend	Entwicklung des Frontend
Entwicklung Backend	Entwicklung des Backend
Testen	Usability Testing und weitere Testing-Aktivitäten
Besprechungen	Besprechungen mit dem Auftraggeber/Betreuer
Bugs	Fehlerbehebung von bereits abgeschlossenen Arbeitspaketen
Recherche	Zeit die für die Recherche und Wissensaneignung von neuen Technologien aufgewendet wird.

Tabelle 10.6: Arbeitspakete

11 PROJEKTMONITORING – IST

11.1 ZEITAUSWERTUNG

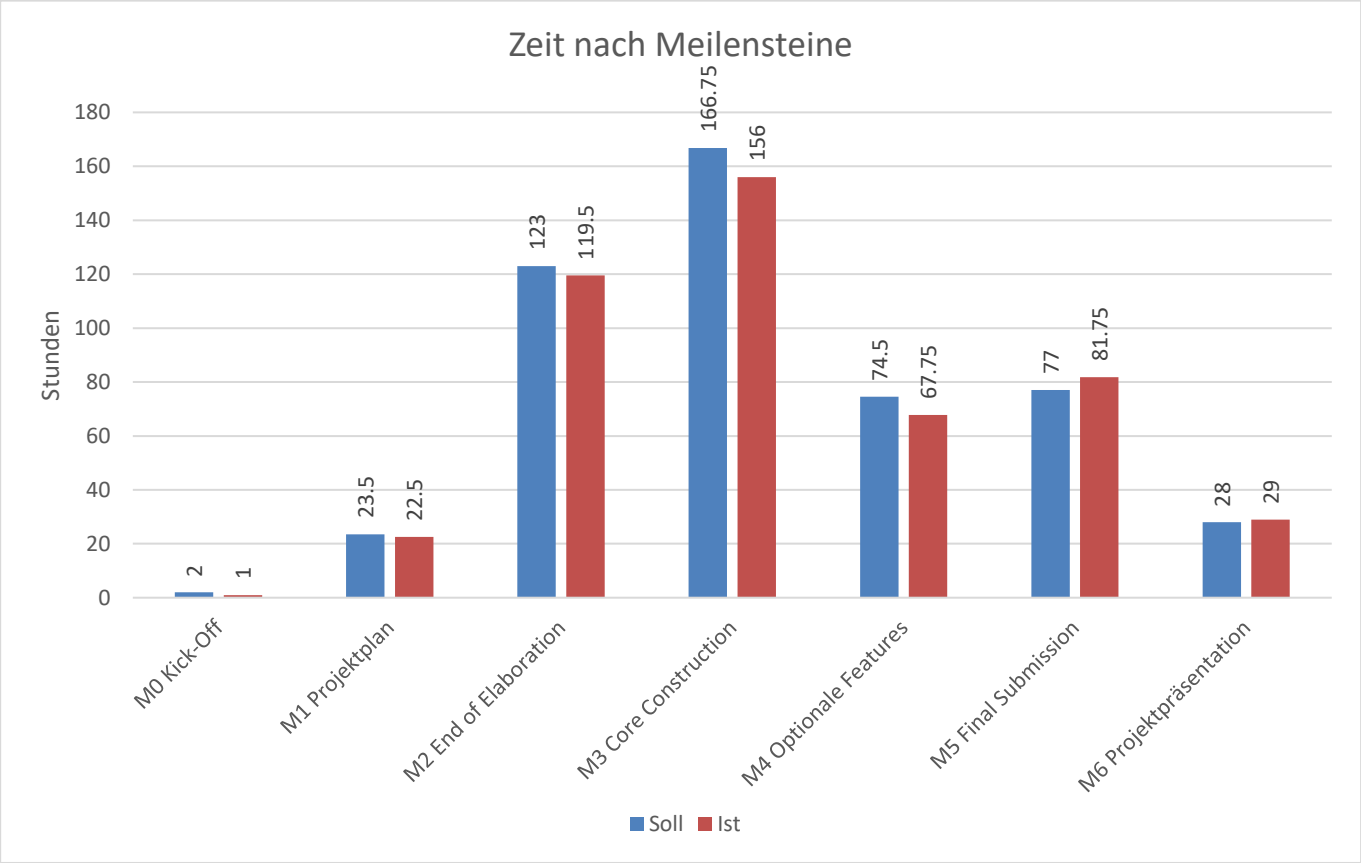


Abb. 11.1: Zeitauswertung auf Meilensteine

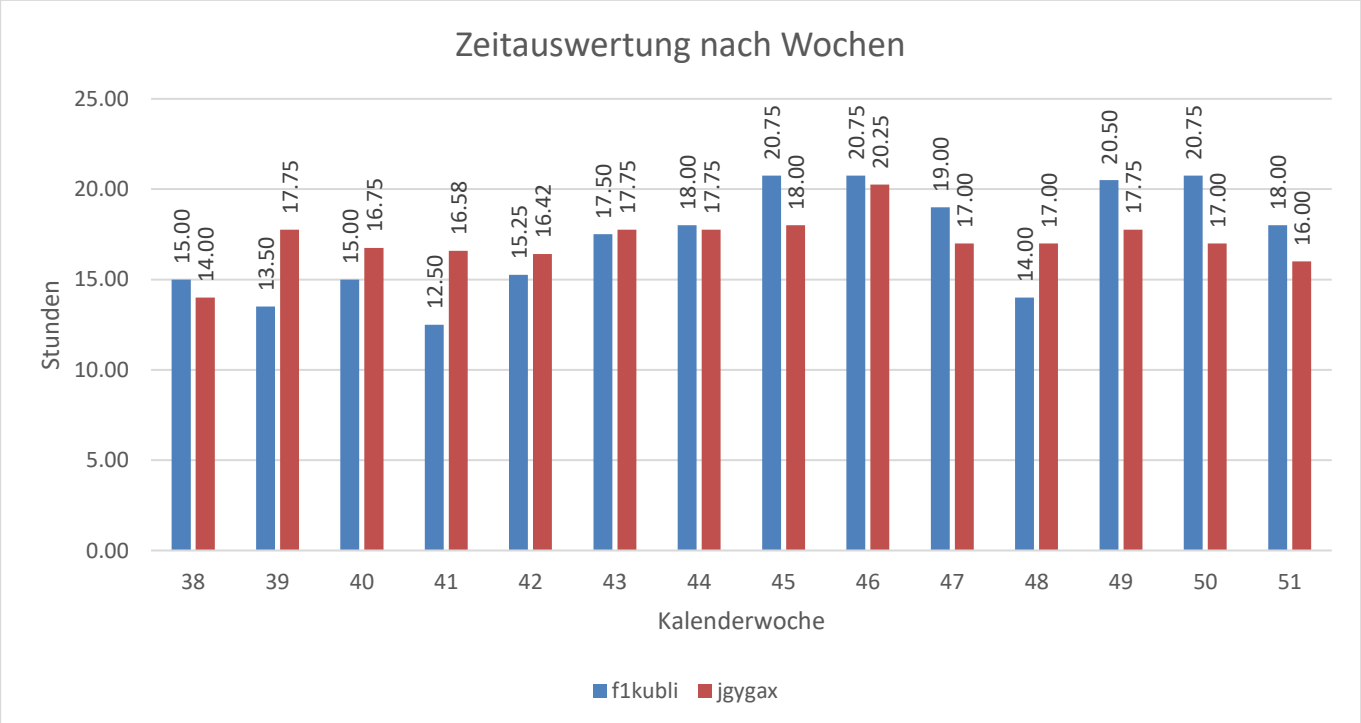


Abb. 11.2: Zeitauswertung nach Wochen

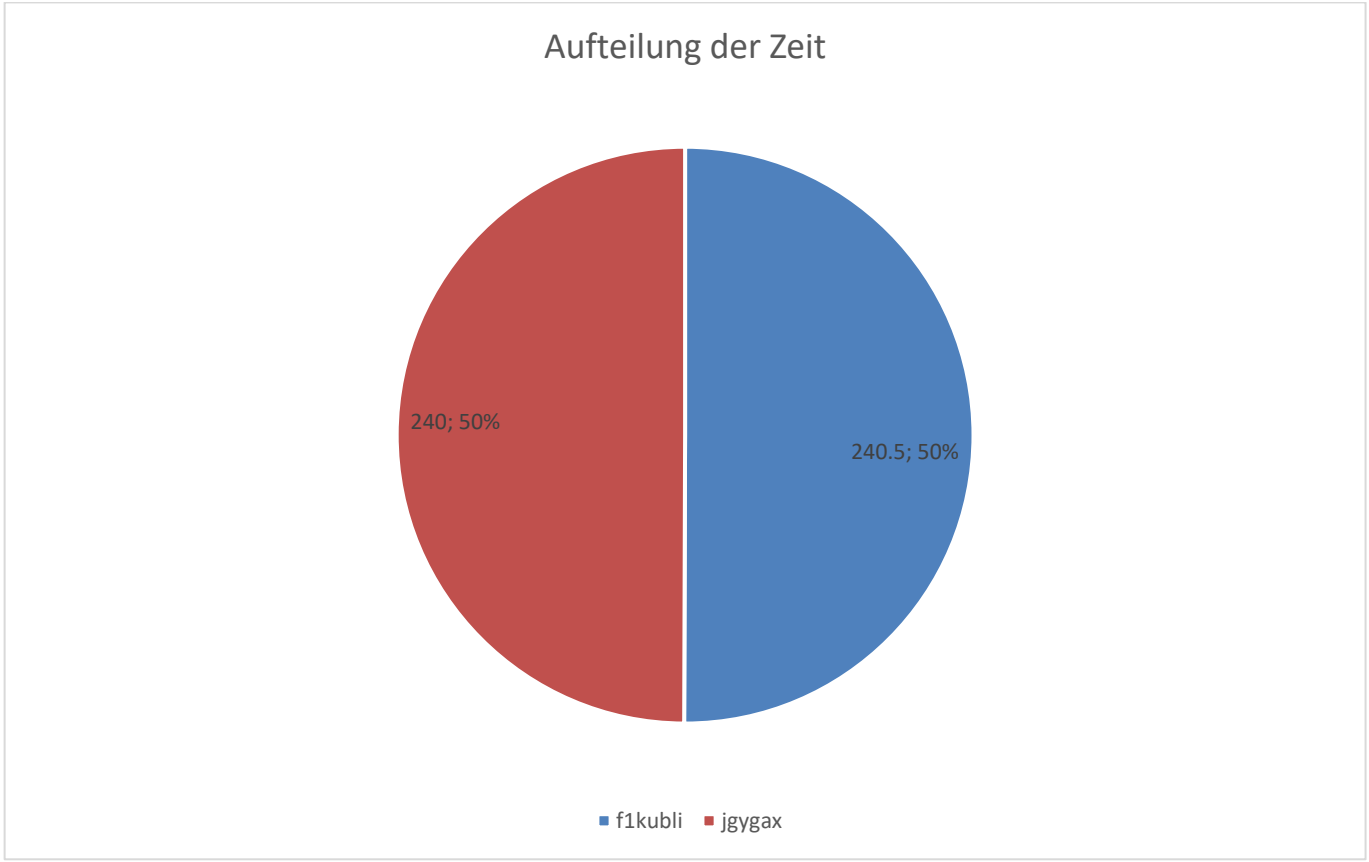


Abb. 11.3: Aufteilung der Zeit zwischen den Projektmitgliedern

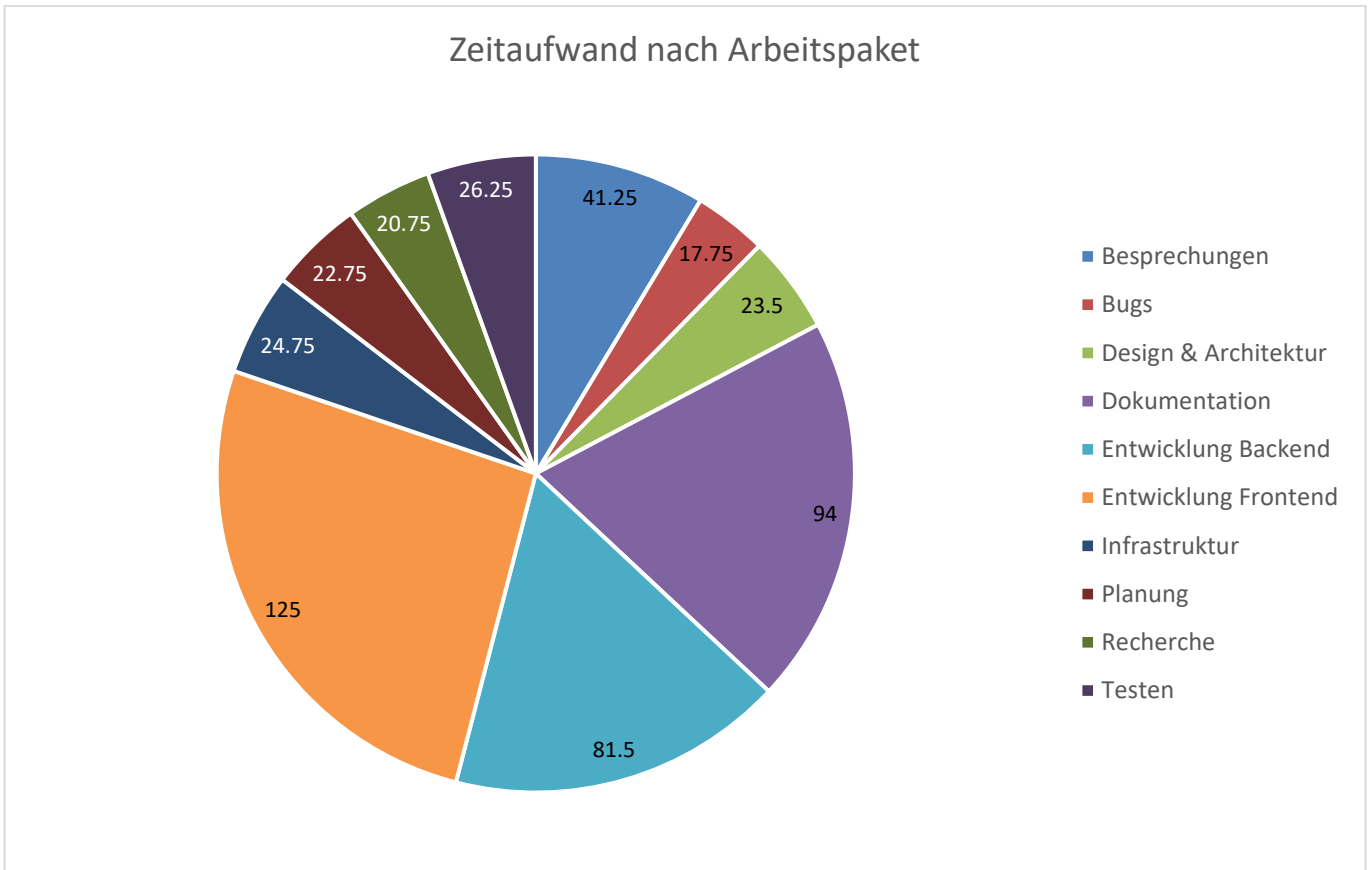


Abb. 11.4: Zeitaufwand aufgeschlüsselt nach Tätigkeiten

11.2 CODESTATISTIK

11.2.1 VERTEILUNG DER PROGRAMMIERSPRACHEN

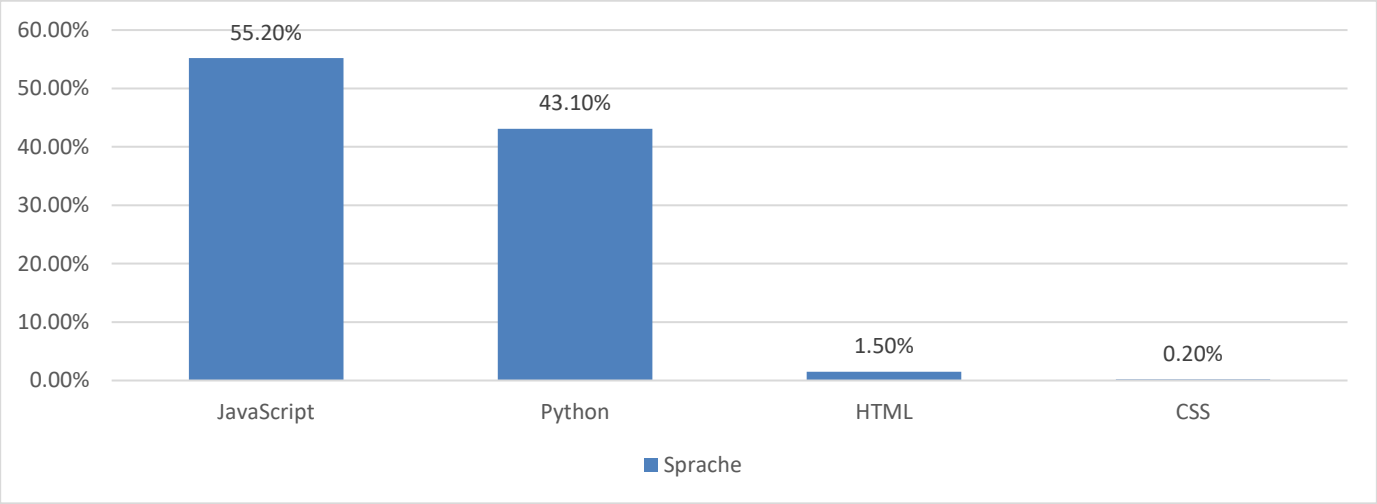


Abb. 11.5: Verteilung der Programmiersprachen

11.2.2 LINES OF CODE

	Lines of Code	Davon Sourcecode	Davon Tests
Backend	1530	1020	510
Frontend	3289	3223	66
Total	**4819**	**4243**	**576**

Tabelle 11.1: Lines of Code

11.2.3 METRIKEN

	Klassen bzw. Komponenten	Methoden
Backend	28	48
Frontend	31	68

Tabelle 11.2: Code Metriken

11.3 ERREICHTE USE CASES

Optionale Use Cases sind kursiv.

Use Case	Erreicht	Bemerkung
UC1 Adjust Nornir Settings	☑	
UC2 Inventory Verwalten	☐	Funktionalität im Backend zu gewissen Teilen bereits vorhanden, Maske im Frontend fehlt.
UC3 Export/Import Inventory	☐	Funktionalität konnte aus Zeitgründen nicht implementiert werden.

UC4 Job Templates Verwalten	☐	Funktionalität im Backend zu gewissen Teilen bereits vorhanden, Maske im Frontend fehlt.
UC5 Construct Job Template	☐	Feature konnte aufgrund seiner Komplexität und Zeitgründen nicht implementiert werden.
UC6 Task Execution	☑	Implementiert mit Zusatzfeatures wie «Task erneut ausführen» oder «Preconfigured Task»
UC7 Task Status	☑	
UC8 Inventory anzeigen	☑	
UC9 Filter Inventory	☑	
UC9.1 Erweiterte Filter	☐	Backend ist bereits für erweiterte Filter vorbereitet, es fehlt eine entsprechende Oberfläche im Frontend.
UC10 Job Template anzeigen	☑	

Tabelle 11.3: Erreichung Use Cases

11.4 ERREICHTE NICHTFUNKTIONALE ANFORDERUNGEN

Anforderung	Priorität	Erreicht	Bemerkung
Konformität: REST Schnittstelle	Soll	☑	
Sicherheit: Benutzer müssen sich Authentisieren	Soll	☑	
Fehlertoleranz: Tasks laufen weiter	Muss	☑	Sobald Nornir getriggert wurde, braucht es keine aktive Verbindung mehr zwischen Frontend und Backend.
Fehlertoleranz: Fehlerhafte Eingaben führen nicht zu Absturz	Muss	☑	
Wiederherstellbarkeit: Tasks und History nach Neustart persistent	Soll	☑	Tasks werden vor der Ausführung in der Datenbank abgespeichert.
Wiederherstellbarkeit: Nach Neustart wird Task fortgesetzt, wo er war	Kann	☐	Es wird kein Task-Journal geführt, stürzt das Backend ab, während ein Task läuft, so sind diese Daten verloren. Der Task muss manuell erneut ausgeführt werden. Bei einem Neustart von Celery, werden jedoch Tasks mit einem zukünftigen Ausführungszeitpunkt wieder von Celery automatisch zur Ausführung aufgenommen.
Attraktivität: Oberfläche verwendet Material Design	Kann	☑	
Erlernbarkeit: Administrator kann Anwendung nach zweistündiger Einführung bedienen	Kann	☐	Wurde nicht überprüft. Aufgrund der Erfahrungen aus den Usability Tests ist davon auszugehen, dass das Ziel

			realistisch ist und erreicht werden kann, da ein Usability Test jeweils nur etwa eine halbe Stunde dauerte.
Verständlichkeit: Anwendung in Englisch	Muss	☑	
Verständlichkeit: Fehlermeldungen mit verständlicher Meldung	Soll	☑	Benutzer erhält bei fehlgeschlagenen Login Rückmeldung oder erhält visuelles Feedback, wenn ein Task fehlgeschlagen ist.
Zeitverhalten: Interaktionen dauern maximal 3 Sekunden	Soll	☑	
Analysierbarkeit: Klassen maximal 300 Zeilen lang	Soll	☑	Die längste Klasse ist 140 Zeilen lang.
Analysierbarkeit: Backend speichert Nornir-Log	Soll	☑	Im Backend wird das Nornir-Log abgespeichert.
Prüfbarkeit: Sämtliche Pull-Requests von Teamkollegen überprüft	Soll	☑	
Austauschbarkeit: Cross-Browser kompatibel	Muss	☑	Getestet mit Firefox sowie Chromium-basierten Browsern-
Installierbarkeit: Installationsdokumentation vorhanden	Muss	☑	Installationsanleitung ist in Kapitel 17.1 Softwaredokumentation zu finden.

Tabelle 11.4: Erreichung nichtfunktionale Anforderungen

12 RISIKOANALYSE

Nr.	Titel	Beschreibung	Max. Schaden [h]	Eintrittswahrscheinlichkeit	Gewichteter Schaden	Vorbeugung	Verhalten beim Eintreten
R1	Technologien inkompatibel	Die ausgewählten Technologien sind inkompatibel zueinander.	24	10%	2.4	Genaue Planung und Recherche bei der Auswahl der Technologien.	Workaround prüfen, ansonsten Technologiewechsel durchführen.
R2	Domainmodell	Das Domainmodell stellt sich als nicht gut genug aus und muss überarbeitet werden.	16	20%	3.2	Genügend Zeit zur Gestaltung des Domainmodell investieren.	Mit höchster Priorität Anpassungen vornehmen.
R3	Architektur	Software-Architektur wurde falsch aufgebaut und muss umgebaut werden.	8	15%	1.2	Architektur gut planen und reviewen lassen.	Einschränkungen dokumentieren, Lösungsvorschlag falls innerhalb der Arbeit nicht lösbar
R4	Usability	Applikation ist nicht verständlich und schwer bedienbar.	16	20%	3.2	Wireframes früh besprechen. Während der Construction Phase Usability Tests durchführen.	Applikation anpassen basierend auf Input der Tester, detaillierte Bedienungsanleitung verfassen.
R5	Bugs/Fehler	Schwerwiegende Fehler werden erst spät im Projekt entdeckt.	40	10%	4	Unit-Testing, Integration Testing	Analyse & beheben, Fehler Dokumentieren falls nicht gelöst werden kann.
R6	Datenverlust	Dokumentation oder Source Code geht verloren	4	5%	0.2	Dokumentation ist in MS SharePoint abgelegt, wird lokal gesichert. Code ist auf Gitlab gespeichert.	Restore der Daten aus lokalen Backups.
R7	Fehlendes Know-How	Unbekannte Technologien, wir kommen damit nicht auf Anhieb klar	8	20%	1.6	Zeit in Wissensbeschaffung investieren, Webinare des INS sowie offizielle Dokumentationen konsultieren	Ansprechpersonen kontaktieren, Dokumentation konsultieren
R8	Konflikte im Team	Uneinigkeit im Team	5	5%	0.25	Direkte und offene Kommunikation	Eskalation über Betreuer
R9	COVID-19	Teammitglieds muss aufgrund von COVID-19 in Isolation/Quarantäne	20	5%	1	Einhalten der Hygieneregeln des Bundes und Kantons.	Falls gesundheitlich machbar: Arbeit über Teams weiterführen. Features rausstreichen bei Ausfall der Person
Ergebnis			**141**		**17.05**		

Tabelle 12.1: Risikoanalyse

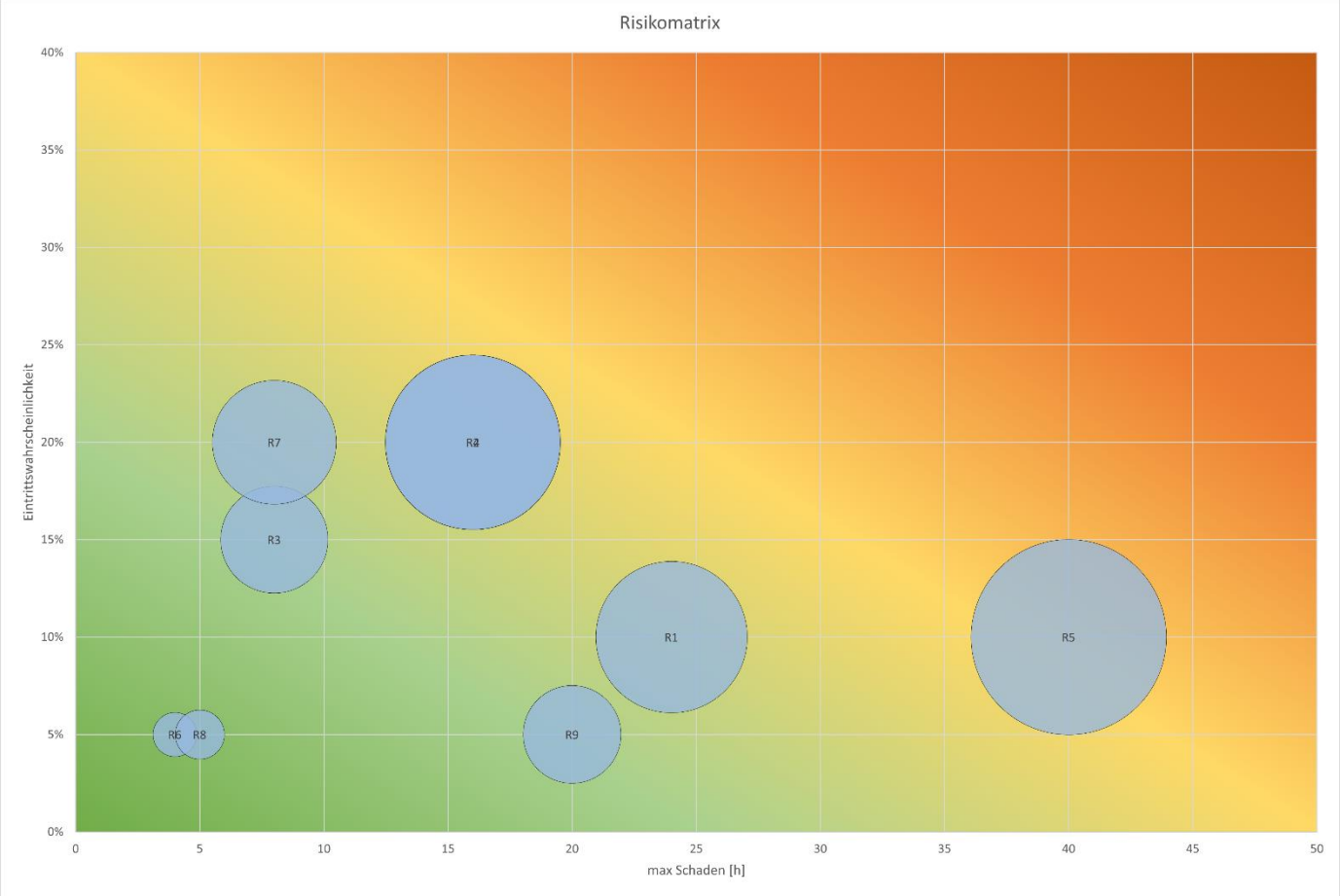


Abb. 12.1: Risikomatrix

12.1 VERLAUF DER RISIKOBEWERTUNG

12.1.1 02.10.2020 MASSNAHMEN STAND MEILENSTEIN 2

12.1.1.1 R1 TECHNOLOGIEN INKOMPATIBEL

Das Framework Nornir ist komplett mit Python gebaut. Da im Backend Python als Sprache mit dem Framework Django zum Einsatz kommt, gibt es da keine Inkompatibilitäten.

12.1.1.2 R2 DOMAINMODELL

Das Domainmodell wurde in der Gruppe und mit den Betreuern angeschaut und Feedback wurde umgesetzt.

12.1.1.3 R6 DATENVERLUST

Der Programmcode ist auf Gitlab verfügbar und wird zur Bearbeitung auf die jeweilige lokale Maschine geklont. Somit sind immer mehrere Klone vorhanden.

Die Dokumentation wird auf MS Teams/SharePoint aufbewahrt.

12.1.1.4 R7 FEHLENDES KNOWHOW

Das Team hat sich mithilfe von Tutorials intensiv mit den Technologien befasst.

12.1.2 16.10.2020 ABSCHLUSS MEILENSTEIN 2

12.1.2.1 R1 TECHNOLOGIEN INKOMPATIBEL

Mit dem Prototyp wurde ein Durchstich erreicht und es wurde bewiesen, dass die ausgewählten Technologien miteinander harmonieren.

Das Risiko kann deshalb auf 0% herabgestuft werden.

12.1.2.2 R2 DOMAINMODELL

Während der Entwicklung des Prototyps sind einige Unstimmigkeiten im Domainmodell aufgefallen, diese wurden intensiv besprochen und die nötigen Änderungen umgesetzt.

Das Risiko, dass das Domainmodell Probleme bereitet, kann auf 10% reduziert werden.

12.1.2.3 R3 ARCHITEKTUR

Die Architektur, die mit dem Prototyp aufgebaut wurde, hat soweit funktioniert und es mussten keine Anpassungen gemacht werden.

Das Risiko kann deshalb auf 10% herabgestuft werden.

12.1.2.4 R4 USABILITY

Da es noch zu früh ist, um Usability Tests durchzuführen, kann dieses Risiko noch nicht bewertet werden. Das Risiko bleibt weiterhin bei 20%.

12.1.2.5 R5 BUGS/FEHLER

Bisher wurden keine schwerwiegenden Fehler entdeckt. Da erst der Prototyp erstellt wurde, kann das Risiko noch nicht herabgesetzt werden. Es verbleibt bei 10% Eintrittswahrscheinlichkeit.

12.1.2.6 R6 DATENVERLUST

Die verwendeten (Cloud-)Dienste haben sich bewährt und keinen Ausfall geleistet.

12.1.2.7 R7 FEHLENDES KNOW-HOW

Das Team hat sich weiterhin mit den nötigen Tutorials und Dokumentationen eingedeckt, um die verwendeten Technologien einzusetzen. Ebenfalls ist das Team im Austausch mit anderen SA-Teams, welche die gleichen Technologien verwenden.

12.1.2.8 R8 KONFLIKTE IM TEAM

Nennenswerte Konflikte gab es keine im Team, bei Unklarheiten konnte man sich jeweils verständigen und die Sache klären.

12.1.2.9 R9 COVID-19

Bisher musste kein Team-Mitglied in Quarantäne. Wegen der sich verschlechternden allgemeinen Lage in der Schweiz wurde darauf geeinigt, sich nicht mehr vor Ort an der HSR zu treffen. Es wird über die bestehenden digitalen Kanäle zusammengearbeitet.

12.1.3 13.11.2020 ABSCHLUSS MEILENSTEIN 3

12.1.3.1 R1 TECHNOLOGIEN INKOMPATIBEL

Auch im weiteren Verlauf konnten keine Probleme mit den ausgewählten Technologien gefunden werden. Das Risiko kann bei 0% verbleiben.

12.1.3.2 R2 DOMAINMODELL

Während der Entwicklung musste einige kleinere Detailverbesserungen am Domainmodell vorgenommen werden. Ansonsten hat sich das Domainmodell bewährt. Das Risiko, dass das Domainmodell Probleme verursacht, kann auf 5% reduziert werden.

12.1.3.3 R3 ARCHITEKTUR

Die Architektur hat sich bewährt. Da im kommenden Meilenstein Features geplant sind, die Anpassungen an der Architektur erfordern, verbleibt das Risiko bei 10%.

12.1.3.4 R4 USABILITY

Die Anwendung ist nun soweit, dass Usability Tests durchgeführt werden. Diese werden jedoch erst im nächsten Meilenstein durchgeführt. Das Risiko bleibt somit bei 20%

12.1.3.5 R5 BUGS/FEHLER

Einige kleine Fehler wurden entdeckt, schwerwiegende Fehler sind keine aufgetreten. Durch intensives Unit- und Integration-Testing konnten schwerwiegende Fehler vermieden werden. Allfällige Einschränkungen wurden dokumentiert. Das Risiko kann auf 5% reduziert werden.

12.1.3.6 R6 DATENVERLUST

Die verwendeten (Cloud-)Dienste haben sich bewährt und keinen Ausfall geleistet.

12.1.3.7 R7 FEHLENDES KNOW-HOW

Bei Jérôme gab es aufgrund fehlender Erfahrung mit React Zeitverlust von etwa 4 Stunden. Mit Hilfe von Felix und mit dem Studium von Tutorials konnten die Probleme trotzdem gelöst werden und Jérôme ist besser gefestigt mit React. Das Risiko kann deshalb auf 10% reduziert werden.

12.1.3.8 R8 KONFLIKTE IM TEAM

Im Team gab es keine Konflikte. Es herrscht ein reger und funktionierender Austausch.

12.1.3.9 R9 COVID-19

Aufgrund der anhaltenden Corona-Massnahmen befinden sich beide Teammitglieder im Home-Office und sind entsprechend einem niedrigeren Ansteckungsrisiko ausgesetzt.

12.1.4 27.11.2020 ABSCHLUSS MEILENSTEIN 4

12.1.4.1 R1 TECHNOLOGIEN INKOMPATIBEL

Bei der Implementierung der Authentisierung mittels JWT gab es im Frontend Probleme mit der Nebenläufigkeit, die doppelt so viel Zeit gekostet hat wie ursprünglich geplant. Der Zeitplan ist dadurch nicht gross durcheinandergeraten, da dafür andere Features einfacher zu implementieren waren wie geplant.

Das Restrisiko kann bei 0% verbleiben.

12.1.4.2 R2 DOMAINMODELL

Das Domainmodell hat sich bewährt und es mussten keine Änderungen gemacht werden. Das Risiko kann auf 0% reduziert werden.

12.1.4.3 R3 ARCHITEKTUR

Die Architektur hat sich ebenfalls als stabil bewährt. Optionale Features, die grössere Änderungen an der Architektur erfordert hätten, werden aus Zeitgründen nicht mehr umgesetzt. Deshalb kann das Restrisiko auf 0% reduziert werden

12.1.4.4 R4 USABILITY

Die Usability Tests wurden durchgeführt und haben viel hilfreiche Rückmeldungen zur Verbesserung der Usability erbracht. Der Grossteil dieser Änderungswünsche wurde umgesetzt, bei einigen Punkten muss der Aufwand zur Implementierung geprüft werden. Das Risiko kann deshalb auf 5% gesenkt werden.

12.1.4.5 R5 BUGS/FEHLER

Es sind weiterhin keine schwerwiegenden Fehler aufgetreten. Mehrere kleinere Bugs wurden gefunden, diese konnten alle innerhalb des Zeitplans behoben werden.

12.1.4.6 R6 DATENVERLUST

Die verwendeten (Cloud-)Dienste haben sich bewährt und keinen Ausfall geleistet.

12.1.4.7 R7 FEHLENDES KNOW-HOW

In diesem Meilenstein gab es keine Probleme mit fehlendem Knowhow, welches zu Zeitverlust führte. Wo benötigt konnte durch Konsultation der Dokumentation das nötige Knowhow angeeignet werden. Das Risiko kann auf 5% reduziert werden.

12.1.4.8 R8 KONFLIKTE IM TEAM

Im Team gab es keine Konflikte. Der Austausch und die Kommunikation funktionieren.

12.1.4.9 R9 COVID-19

Durch die Anhaltende Home-Office Regelung ist das Risiko einer Ansteckung gering, weshalb es auch zu keinen Ausfällen durch Krankheit kam.

12.1.5 11.12.2020 ABSCHLUSS MEILENSTEIN 5

Da nach diesem Meilenstein keine weiteren Features implementiert werden, gibt es auch keine aktuellen Risiken. Das Einzige, was passieren könnte ist, dass weitere Bugs und Fehler entdeckt werden, weshalb nur beim Punkt R5 ein Restrisiko verbleibt.

13 REFLEXION

13.1 FELIX KUBLI

Da ich aus dem Bereich der Softwareentwicklung komme, hatte ich vor dieser Studienarbeit noch sehr wenig mit Netzwerken zu tun, bis auf das Computernetze Modul. Deshalb war es interessant für mich zu sehen, was für Möglichkeiten man mit Automatisierungstools hat und wie sie einem viel Arbeit abnehmen können. Für mich war es deshalb am Anfang schwierig herauszufinden was die Applikation dieses Projekts für Anforderungen erfüllen muss. Unter anderem aus dem Grund war ich froh mit Jérôme zusammenarbeiten zu können, da er schon Erfahrung in diesem Bereich hat.

Was die Technologien betrifft, war ich schon etwas vertraut mit React, was sich als sehr hilfreich erwiesen hat. Dennoch gab es im Frontend einige Herausforderungen, welche wir aber schlussendlich meistern konnten.

Mit Python hatte ich hingegen noch keine Erfahrung vor diesem Projekt. Die Sprache ist jedoch sehr intuitiv, was auch für Django und das Django Rest Framework gilt.

Gerade mit unserer Analyse der Anforderungen bin ich sehr zufrieden, da wir uns ein klares Bild der Situation machen konnten. Dies half uns dann bei der Umsetzung, da wir schon gut wussten, was wir wie umsetzen wollen.

Zudem hat mir die Aufteilung in Core-Features, die zum Minimal Viable Product (MVP) gehören, und den Optional Features sehr gefallen, da wir so die Arbeitspakete priorisieren konnten. Ausserdem haben wir die Grösse dieses MVPs gut gewählt, da wir alle Core-Features umsetzen konnten. Im vorgegangenen Engineering Project (EPJ) haben wir diese Aufteilung ebenfalls gemacht und deshalb war es nicht eine grosse Überraschung für mich, dass wir nicht alle Optionalen Features umsetzen konnten. Im EPJ war es nämlich ähnlich verlaufen, trotz anderen Teammitgliedern.

Positiv bleiben auch die Usability Tests in Erinnerung, da wir mit dem resultierenden Feedback noch gute Änderungen vornehmen konnten, die die Bedienung der Applikation wesentlich angenehmer machen.

Die Zusammenarbeit und Organisation mit Jérôme haben sehr gut funktioniert. Anfangs trafen wir uns noch regelmässig in Rapperswil, aber als dann die Corona Pandemie wieder an Fahrt aufnahm, kommunizierten wir hauptsächlich online. Für die Planungsphase war die lokale Zusammenarbeit sehr hilfreich, aber die online Kommunikation funktionierte auch ohne Probleme.

Alles in allem bin ich zufrieden mit dem entstandenen Produkt und dem Verlauf des Projekts und den gewonnenen Erfahrungen, die daraus resultieren.

13.2 JÉRÔME GYGAX

Mit der Studienarbeit habe ich mein bisher grösstes Softwareprojekt abgeschlossen. Die Arbeit hat mir Spass gemacht, unter anderen da unterschiedliche, für mich auch neue, Technologien zum Einsatz kommen. Ebenfalls konnte ich wieder in bekannten Feldern «wildern», nämlich dem Netzwerk. Für mich war vor allem die Verbindung zwischen der Netzwerkhardware, die ich bei meiner früheren Arbeit häufig manuell konfiguriert habe, und der Software sehr spannend. So durfte ich auch Erfahrungen in der Netzwerkautomatisierung sammeln.

Insgesamt gefiel mir die vielseitige Arbeit, von der Projektplanung, der Gestaltung der Oberfläche über die Ausarbeitung des API im Backend bis hin zur Verbindung auf die Netzwerkgeräte. Im Backend war ich erstaunt, wie schnell die Lernkurve von Python und den Frameworks ist. Man konnte so schnell weit kommen.

Das Web-Frontend hingegen mit seinen Frameworks war komplett neu für mich und ich musste mich da einiges ins Zeug legen, um diese Technologien kennenzulernen. Das war auch immer einfach und ich hatte ab und an doch meine schwierigen Momente damit. Zusammen mit meinem Teamkollegen Felix konnten wir diese Probleme miteinander anschauen und immer lösen, da er im Bereich Webentwicklung bedeutend mehr an Erfahrung hat. Dafür bin ich sehr dankbar.

Trotz Corona konnten wir uns im Team sehr gut organisieren. Anfangs der Arbeit sogar noch in Person, ab der zweiten Hälfte dann nur noch über Telefon. Die Zusammenarbeit schätzte ich sehr und wir konnten immer gut miteinander kommunizieren, trotz der physischen Distanz. Wir haben uns sehr gut verstanden, obwohl dies unsere erste gemeinsame Arbeit war.

Rückblickend mussten wir feststellen, dass wir in der Planungsphase etwas zu optimistisch diverse Features eingeplant haben. Die Kernapplikation hat doch mehr Zeit beansprucht wie erwartet, weshalb einige Zusatzfunktionen nicht mehr umgesetzt werden konnten. Trotzdem bin ich mit der Ansicht, dass wir eine brauchbare Anwendung entwickelt haben, welche im jetzigen Status schon produktiv eingesetzt werden kann und schon manchen Workflow beschleunigen kann.

14 LITERATUR UND QUELLENVERZEICHNIS

Herkunft der Vorlage	Das Dokument wurde auf der Basis einer Vorlage für Technische Berichte erstellt. Die Vorlage ist ein Element des „Werkzeugkastens Technische Berichte“ der Hochschule für Technik Rapperswil. Sie orientiert sich an Prinzipien des Strukturierten Schreibens.
Dokumentation Nornir	https://nornir.readthedocs.io/en/latest/
Dokumentation Django Framework	https://docs.djangoproject.com/en/3.1/
Dokumentation Django Rest Framework	https://www.django-rest-framework.org/
Dokumentation SimpleJWT for Django Rest Framework	https://django-rest-framework-simplejwt.readthedocs.io
Dokumentation React.js	https://reactjs.org/docs/getting-started.html
Dokumentation Material-UI for React	https://material-ui.com/

15 VERZEICHNISSE

15.1 GLOSSAR UND ABKÜRZUNGSVERZEICHNIS

Autowiring	Begriff aus der Software-Entwicklung. Autowiring bedeutet, dass ein Entwickler weniger Code schreiben muss, um Abhängigkeiten zwischen Komponenten zu verbinden. Das Framework übernimmt diese Arbeit, es *verkabelt* die Komponenten *automatisch*.
Celery	Celery ist ein Python Modul, das Task-Queing und Task-Scheduling unterstützt und in dieser Arbeit für das Task-Scheduling verwendet wird. https://docs.celeryproject.org/en/stable/
Channels	Channels ist eine Erweiterung für das Django Framework und stellt WebSockets zur Verfügung. https://channels.readthedocs.io/en/latest/
Dictionary	Datentyp in Python, speichert Daten als Key-Value Paare ab.
Django	Django ist ein High-Level Python Web Framework, um Webapplikationen zu entwickeln. https://www.djangoproject.com/
F-Funktion	In Nornir integrierte Funktion, welche die Suche und Filterung nach bestimmten Kriterien ermöglicht.
INS	Institute for networked solutions. Das Institut, welches die Idee für diese Arbeit hatte und das Projektteam während der Umsetzung unterstützte.
Inventory	Geräte-Inventar von Nornir. Ein Inventar enthält eine Liste von Hosts. Diese Hosts können diverse netzwerkfähige Geräte sein: Server, Netzwerkgeräte, etc…
Job Template	Playbook/Gerüst, welches Anweisungen für Nornir enthält. Diese Anweisungen können Variablen enthalten und können über ein ausgewähltes Inventory ausgeführt werden.
JWT	JSON Web Token. Kann für die Authentifizierung verwendet werden. https://jwt.io/
Network as Code	Das Netzwerk wird nicht manuell konfiguriert, sondern über Frameworks und mit Hilfe von Code «orchestriert». Sämtliche Änderungen, die am Netzwerk vorgenommen werden, geschehen nicht über eine Konfigurationsoberfläche oder Shell, sondern werden mittels Code und Frameworks/APIs gemacht. https://blogs.cisco.com/developer/what-does-network-as-code-mean
Nornir	Nornir ist ein in Python geschriebenes Automationsframework, um in Python Skripte zur Automatisierung verschiedener Aufgaben im Netzwerkbereich zu bewerkstelligen. https://nornir.readthedocs.io/en/latest/
React	React ist ein von Facebook entwickeltes UI-Framework, um einfach komplexe Webseiten zu erstellen. Häufig kommt es bei Single-Page-Webanwendungen zum Einsatz.

Redis	Redis ist eine In-Memory Datenbank mit einem einfachen Key-Value-Store. Er wird in diesem Projekt als Message Broker eingesetzt. https://redis.io/
Redux	Redux ist eine JavaScript-Bibliothek zur Verwaltung von Zustandsinformationen innerhalb einer Webanwendung. https://react-redux.js.org
Scratch	Visuelle Programmiersprache, um das Erlernen der Programmierung zu erleichtern. https://scratch.mit.edu/
Task	Job/Ausführung eines Template auf ein ausgewähltes Inventory. Die Anweisungen für Nornir werden wo nötig mit Daten angereichert und über das ausgewählte Inventory ausgeführt.
Taskergebnis	Resultat eines ausgeführten Task auf ein Inventory mit entsprechendem Output den Nornir zurückgegeben/generiert hat.

15.2 ABBILDUNGEN

15.3 TABELLEN

16 ERKLÄRUNG ZUR URHEBERSCHAFT

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit ohne Hilfe Dritter angefertigt habe. Ich habe nur die Hilfsmittel benutzt, die ich angegeben habe. Gedanken, die ich aus fremden Quellen direkt oder indirekt übernommen habe, sind kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort/Datum

Rapperswil, 15. Dezember 2020

Unterschrift

Felix Kubli

Jérôme Gygax

17 ANHANG

17.1 SOFTWAREDOKUMENTATION

17.1.1 INSTALLATION

Um die beiden Frontend und Backend-Projekte aufzusetzen, bitte die README-Dateien im Repository konsultieren. Dort ist Schritt für Schritt beschrieben, wie die beiden Projekte vorbereitet werden.

In diesem Kapitel wird beschrieben, wie die Applikation als Produktiv-Instanz installiert werden kann.

17.1.1.1 SYSTEMVORAUSSETZUNGEN

Komponente/Software	Mindestens erforderliche Version
Build-essentials	12.8
nginx	1.18.0
NodeJS	12.19.0 LTS
PostreSQL	12
Python	3.8.5
Python3-dev	3.8.5

Um den Server produktiv einzusetzen, müssen folgende Pakete installiert werden:

- Node-Package «serve» für Frontend (über yarn installieren)
- Python-Package «daphne» fürs Backend
- nginx-Server, Konfiguration ist im Repository abgelegt.

17.1.1.2 UMGEBUNGSVARIABLEN

Das Backend liest Umgebungsvariablen aus, um Einstellungen zu überschreiben. Dies betrifft die Datenbank sowie die URL des Celery Brokers, in dieser Applikation der Redis-Server.

Standardmässig nutzt Django PostgreSQL als Datenbank auf der lokalen Maschine, ebenfalls wird ein lokaler Redis-Server erwartet.

Variable	Bedeutung
DB_CONNECTION_STRING	Definiert die zu benutzende Datenbank. Welche Connection-Strings möglich sind, sind der Dokumentation für «dj_database_url» zu entnehmen: https://github.com/jacobian/dj-database-url
CELERY_BROKER_URL	URL des Redis-Servers

17.1.1.3 FRONTEND

Bevor der Build des Frontend erstellt werden kann, muss in der Datei «api.js» die URL zum Backend angepasst werden. Diese muss auf den FQDN des Backend-Servers lauten. Zur Entwicklung ist diese URL auf localhost gesetzt.

Mit den folgenden Befehlen wird der Build generiert und der Server auf Port 3000 gestartet.

```
ins@sr-000105:~/web-app-nornir/frontend$ yarn build
ins@sr-000105:~/web-app-nornir/frontend$ serve -s build -l 3000
```

17.1.1.4 BACKEND

Zuerst müssen alle statischen Dateien gesammelt werden, damit diese auch vom Server ausgeliefert werden. Der Pfad wird in «settings.py» mit «STATIC_ROOT»/«STATIC_URL» konfiguriert und ist im Projekt auf den Projektordner mit Unterverzeichnis «static» eingestellt.

```
(venv) ins@sr-000105:~/web-app-nornir/backend$ python manage.py collectstatic
```

Die nginx-Server-Konfiguration aus dem Repository muss ins Konfigurationsverzeichnis kopiert werden.

```
(venv) ins@sr-000105:~/web-app-nornir/backend$ sudo cp nginx.conf \
/etc/nginx/sites-enabled/django.conf
(venv) ins@sr-000105:~/web-app-nornir/backend$ sudo service nginx restart
```

Die Konfiguration muss allenfalls auf die richtige Adresse und Port des Backends angepasst werden.

Mit dem ersten Befehl kann überprüft werden, ob alle Debug-Optionen in Django deaktiviert sind. Sind alle deaktiviert, wird der Celery Worker und anschliessend der Backend-Server gestartet.

```
(venv) ins@sr-000105:~/web-app-nornir/backend$ python manage.py check --deploy
(venv) ins@sr-000105:~/web-app-nornir/backend$ celery -A backend worker
(venv) ins@sr-000105:~/web-app-nornir/backend$ daphne backend.asgi:application \
 -b 0.0.0.0
```

Das Backend ist unter Port 80 oder 8000 erreichbar.

17.1.1.5 INSTALLATION ALS SERVICE

Im Repository sind drei «.service»-Dateien für das Frontend, Backend und Celery enthalten. In diesen Dateien ist der Benutzer & Gruppe sowie die Pfade anzupassen.

Diese drei Dateien können anschliessend nach `/etc/systemd/system/` kopiert werden. Um diese Services zu aktivieren und starten können die gewohnten Befehle verwendet werden:

```
ins@sr-000105:~/web-app-nornir/$ sudo systemctl enable celeryworker
ins@sr-000105:~/web-app-nornir/$ sudo systemctl enable daphne
ins@sr-000105:~/web-app-nornir/$ sudo systemctl enable serve
ins@sr-000105:~/web-app-nornir/$ sudo systemctl start celeryworker
ins@sr-000105:~/web-app-nornir/$ sudo systemctl start daphne
ins@sr-000105:~/web-app-nornir/$ sudo systemctl start serve
```

17.1.1.6 TLS SUPPORT

Die Applikation hat keinen TLS-Support konfiguriert. Um Verbindungen zwischen Endbenutzer, Frontend und Backend zu verschlüsseln kann entweder Traefik oder ein anderer Application Proxy vorgeschaltet werden.

Django und nginx unterstützen ebenfalls TLS, um dies zu konfigurieren konsultieren Sie die entsprechenden offiziellen Dokumentationen.

17.1.2 BEGRIFFLICHKEITEN

17.1.2.1 INVENTORY

Im Inventory sind alle verwalteten Geräte erfasst. Diese Applikation unterstützt mehrere Inventories, so dass Geräte zur Erhöhung der Übersichtlichkeit in eigene Inventories ausgelagert werden können. So können etwa Geräte von unterschiedlichen Lokationen in eigenen Inventaren verwaltet werden.

Über die Geräte im Inventar werden Tasks ausgeführt. Task können jeweils nur über die Geräte von einem Inventory ausgeführt werden. Inventory-übergreifende Tasks sind nicht möglich.

17.1.2.2 JOB TEMPLATE

Ein Job Template ist ein Python-Skript, das Nornir-Anweisungen enthält. Darin sind weder die Hosts, auf denen es ausgeführt werden soll noch allfällige Variablen enthalten. Diese werden über das Web-Interface konfiguriert und bei der Ausführung eines Job Templates als Task abgefüllt.

Ein Beispiel für ein solches Job-Template kann ein Skript sein, welches auf Cisco Geräten BGP konfiguriert. Entsprechend enthält das Skript den nötigen Code, um BGP zu aktivieren. Anstelle ein AS und andere Parameter hart zu codieren, werden an diesen Stellen Variablen gesetzt, die über das Web-Interface dann konfiguriert werden. Ebenfalls wird im BGP-Skript kein Inventory und keine Hosts angegeben.

17.1.2.3 PRECONFIGURED TASK

Bei Preconfigured Tasks handelt es sich um bereits konfigurierte Job-Templates, das heisst, der Job, die Variablen und das Inventory sind bereits festgelegt. Ein Preconfigured Task hat aber kein Start-Datum. Wird ein solcher Preconfigured Task ausgeführt, wird davon eine Kopie erstellt und die Kopie wird ausgeführt.

Mit Preconfigured Tasks können häufig benötigte Tasks (z.B. Interface-Status der Geräte im Datacenter) bereits fixfertig eingerichtet werden und bei Bedarf müssen diese nur noch ausgeführt werden.

17.1.3 BETRIEB

17.1.3.1 BENUTZERVERWALTUNG

Für den Betrieb der Anwendung sind vier Gruppen vorgesehen mit unterschiedlichen Profilen:

Gruppe	Berechtigungen
Django Superuser (System)	Alle
Superuser	Kann Inventory, Tasks, Job Templates einsehen, erstellen, modifizieren, löschen.
Netadmin	Kann alles einsehen und Tasks erstellen.
Support	Kann nur Daten einsehen, aber nichts modifizieren.

Der **Django Superuser** ist **nicht** zur alltäglichen Benutzung vorgesehen. Dieser soll nur verwendet werden, wenn neue Benutzer dem System hinzugefügt werden müssen. Diese Art von Benutzer kann nur mit dem Django-Management-Befehl `python manage.py creatsuperuser` erstellt werden.

Die nachfolgenden Gruppen können dann über die Weboberfläche von Django Admin erstellt werden:

Zu der Gruppe **Superuser** gehören diejenigen, die Nornir kennen und die Konzepte verstehen sowie fähig sind dafür Jobs in Python zu schreiben. Superuser befüllen über die Django Admin Oberfläche die Datenbank mit Inventories oder Job Templates.

Zur Gruppe **Netadmin** gehören die Netzwerkadministratoren, welche die Vorteile von Nornir verwenden möchten, ohne es im Detail zu verstehen. Netadmins können alle Daten einsehen und zusätzlich dazu eigene Tasks zusammenklicken und ausführen.

Die Gruppe **Support** hat nur Leseberechtigungen und kann so entsprechend Auskunft geben, kann ansonsten aber keine Modifikationen am System ausführen.

17.1.3.2 INVENTORY

Empfohlener Pfad zur Dateiablage: `[Projekt-Pfad]/backend/web_nornir/nornir_config/`

Die Anwendung unterstützt zurzeit nur Inventories vom Typ SimpleInventory. Das System verlangt mindestens eine «hosts.yaml» und «groups.yaml»-Datei. Die «defaults.yaml»-Datei ist optional.

Anschliessend müssen die Pfade zu den Konfigurationsdateien **manuell** über die Django Admin Oberfläche als Inventory hinzugefügt werden.

Das Backend kennt keinen Mechanismus, um Inventories automatisch einzulesen.

17.1.3.3 JOB TEMPLATES

Empfohlener Pfad zur Dateiablage: `[Projekt-Pfad]/backend/web_nornir/job_templates/`

Gleich wie für Inventories kennt das Backend keinen Mechanismus, um Job Templates automatisch einzulesen. Die Job-Templates müssen als Skript in einem normalisierten Format abgelegt werden und anschliessend über die Django Admin Oberfläche in die Datenbank eingetragen werden.

Variablen müssen zwingend in der Datenbank erfasst sein, ansonsten fragt das Frontend diese nicht ab!

Standard-Struktur für Job Templates (Funktionsname darf variieren, muss in der DB entsprechend angepasst werden):

```
def job_function(task: Task) -> Result:
    return Result(
        host=task.host,
        result=task.run(
            # Von Nornir auszuführende Befehle
        )
    )
```

17.2 SCREENSHOTS

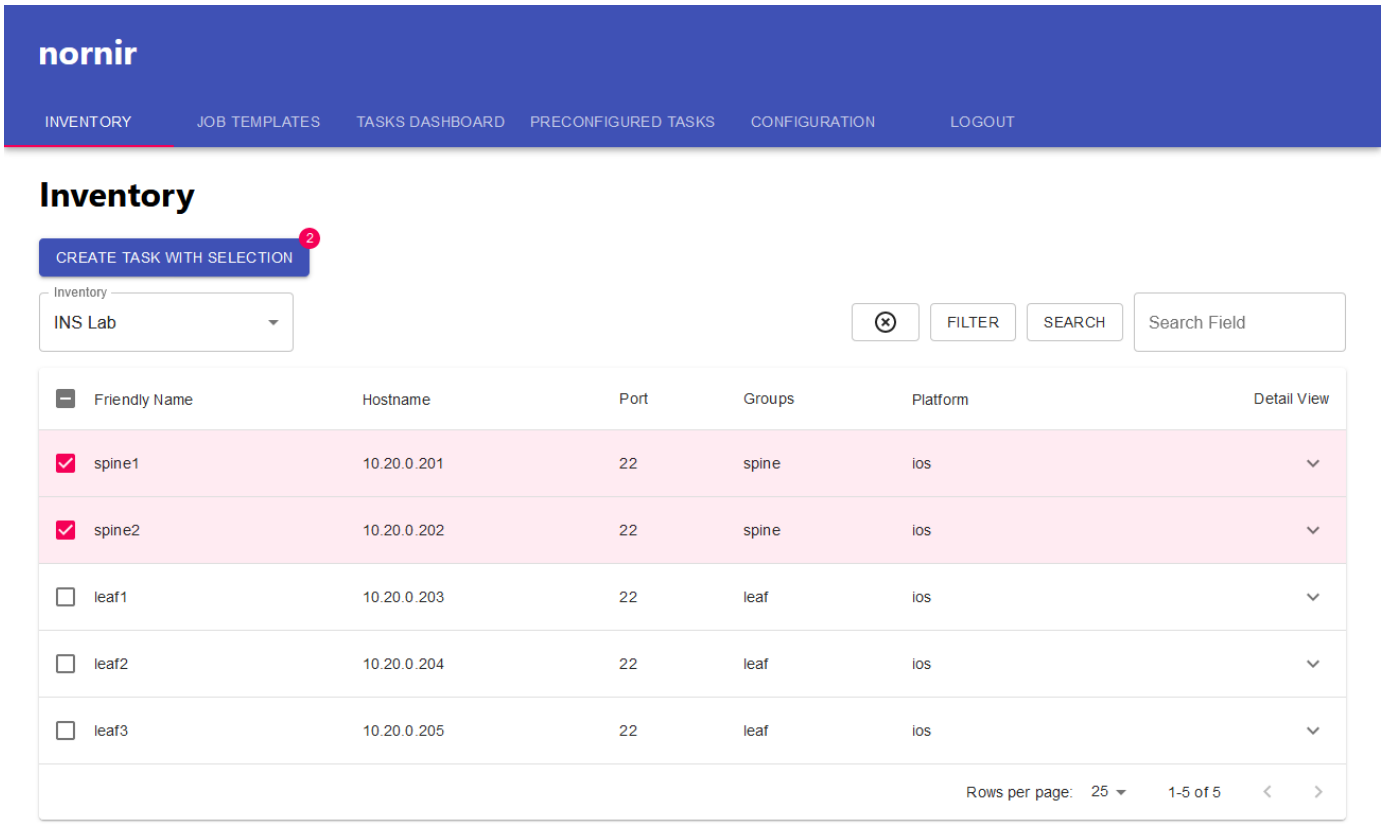


Abb. 17.1: Inventory Seite

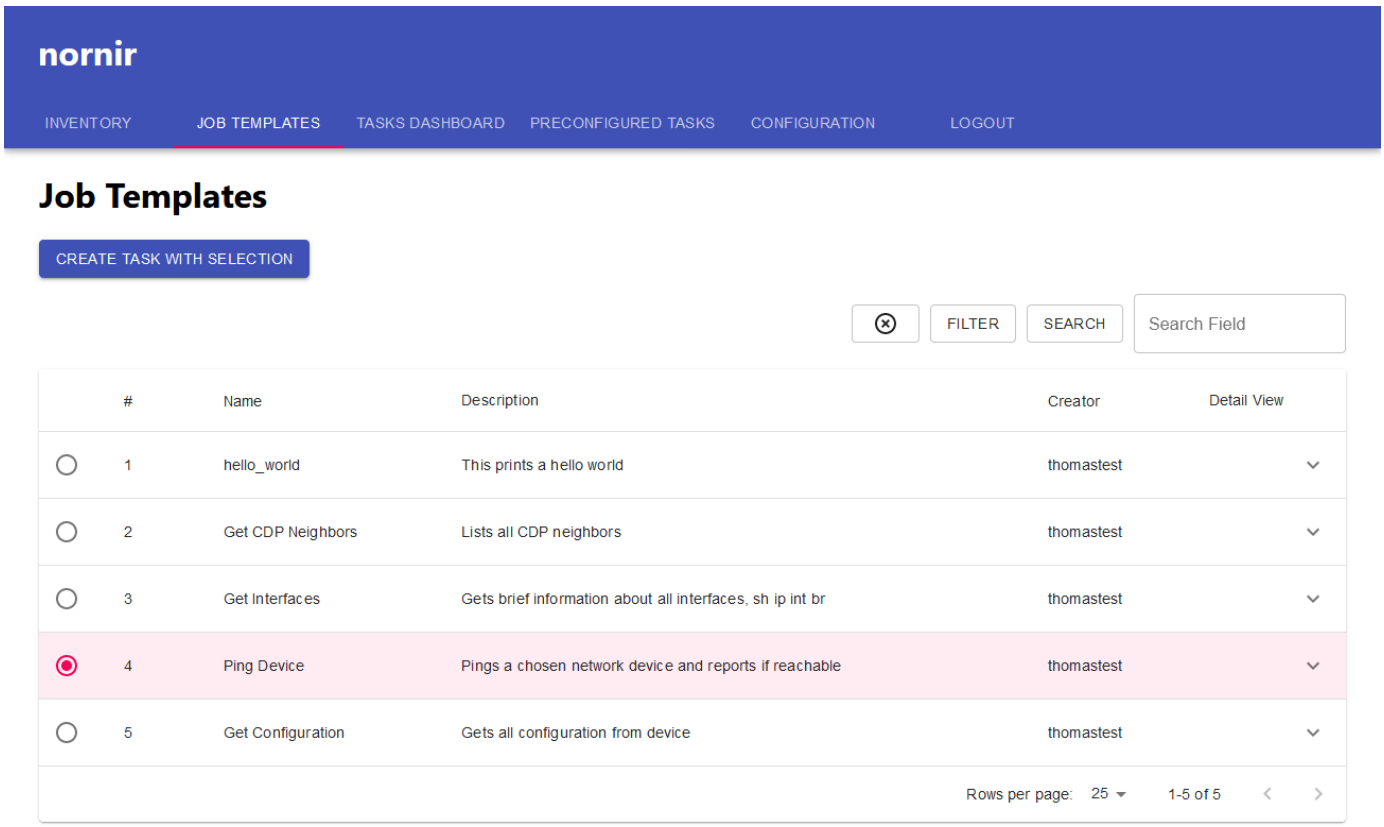


Abb. 17.2: Job Templates Seite

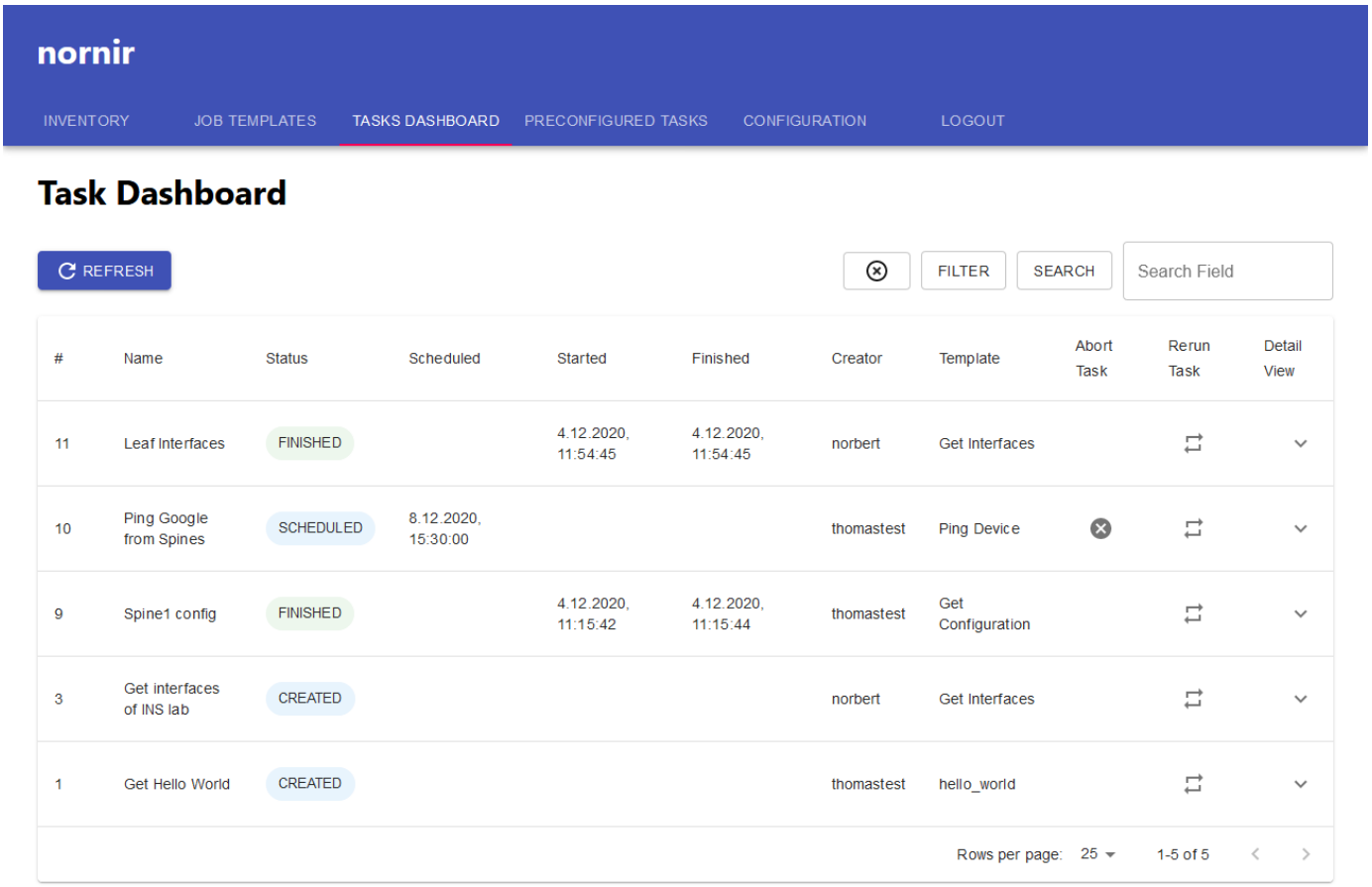


Abb. 17.3: Task Dashboard

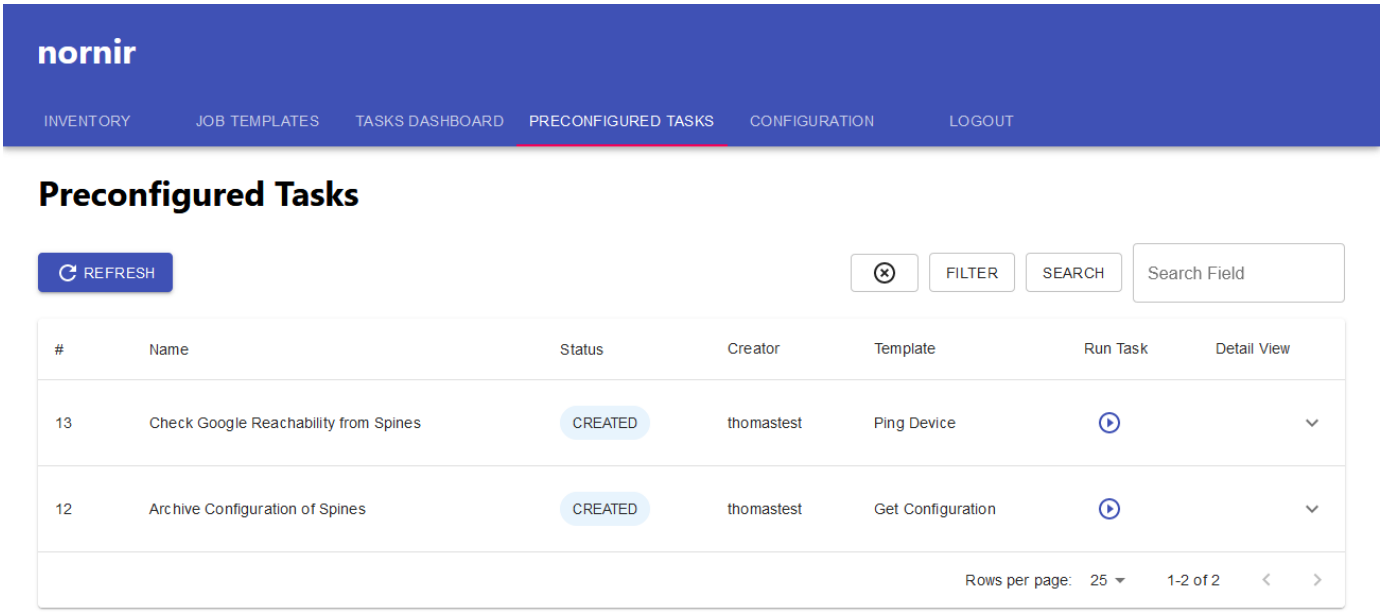


Abb. 17.4: Preconfigured Tasks

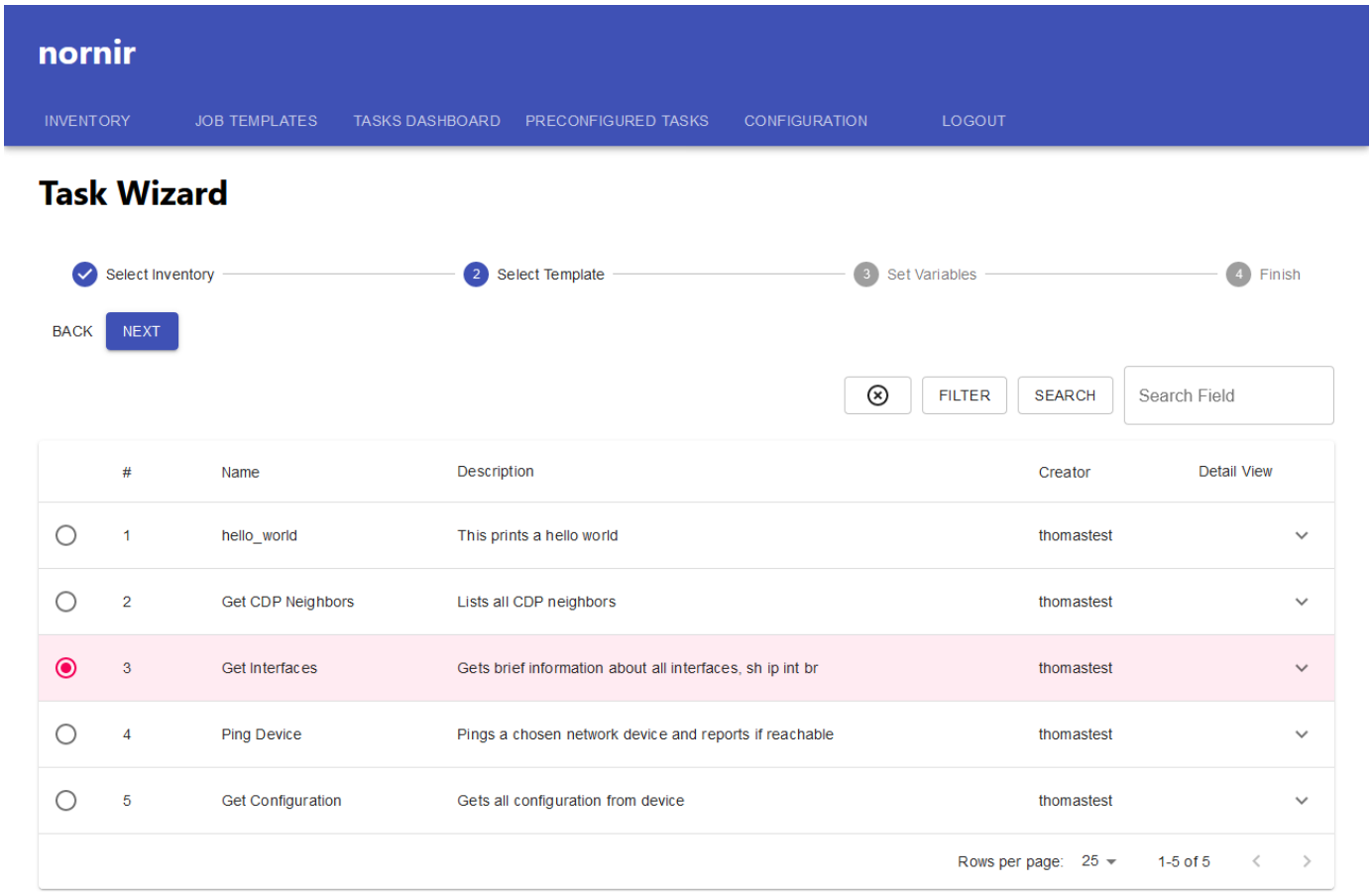


Abb. 17.5: Task Wizard; gestartet mit Selektion aus Inventory

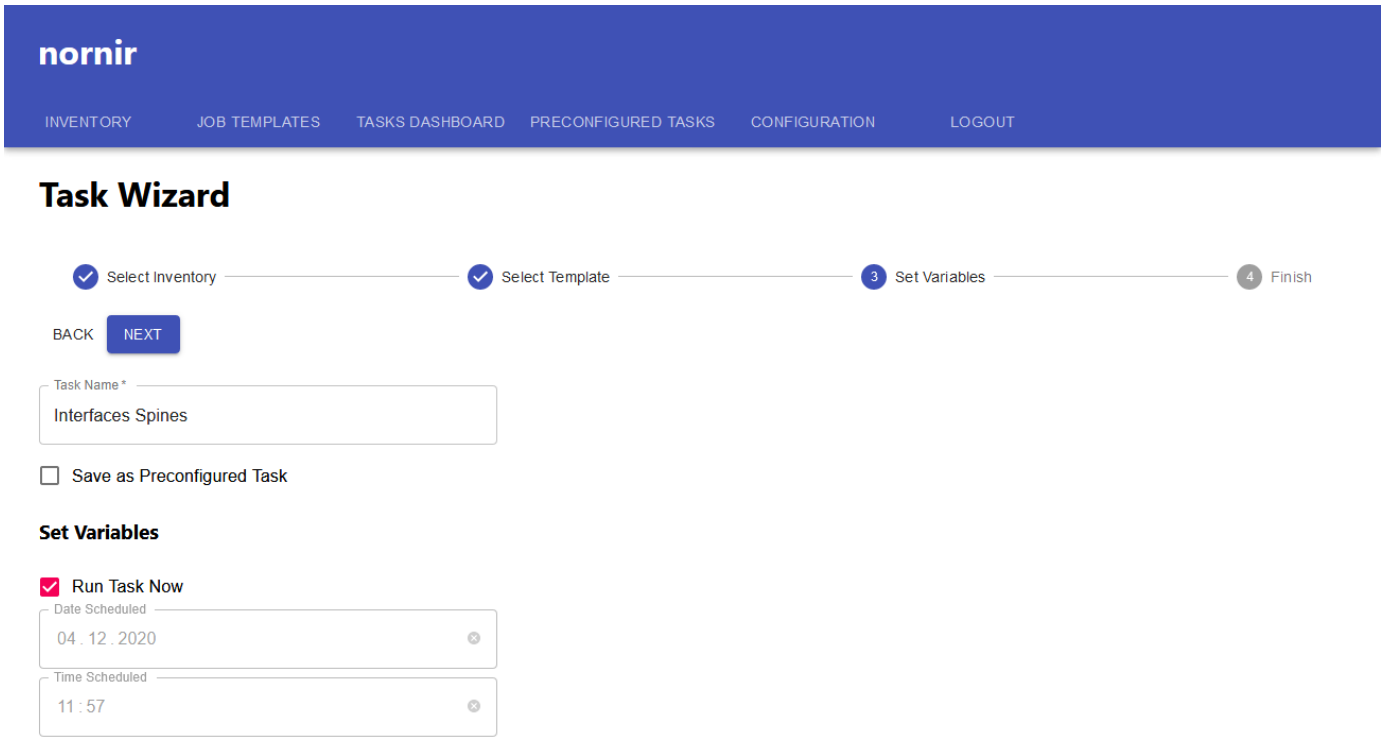


Abb. 17.6: Task Wizard; Eingabe der Variablen und Zeit

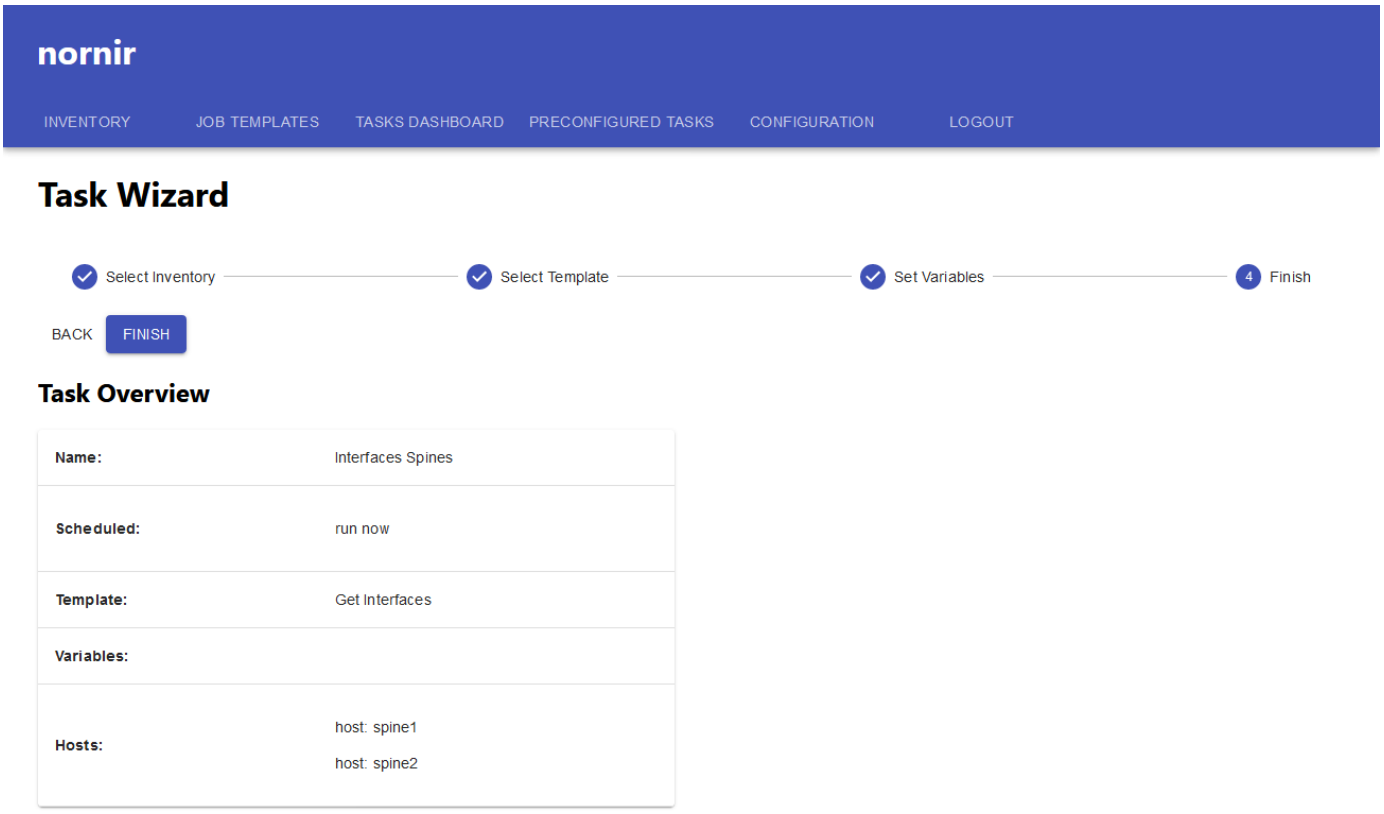


Abb. 17.7: Task Wizard, Übersicht über zu erstellenden Task

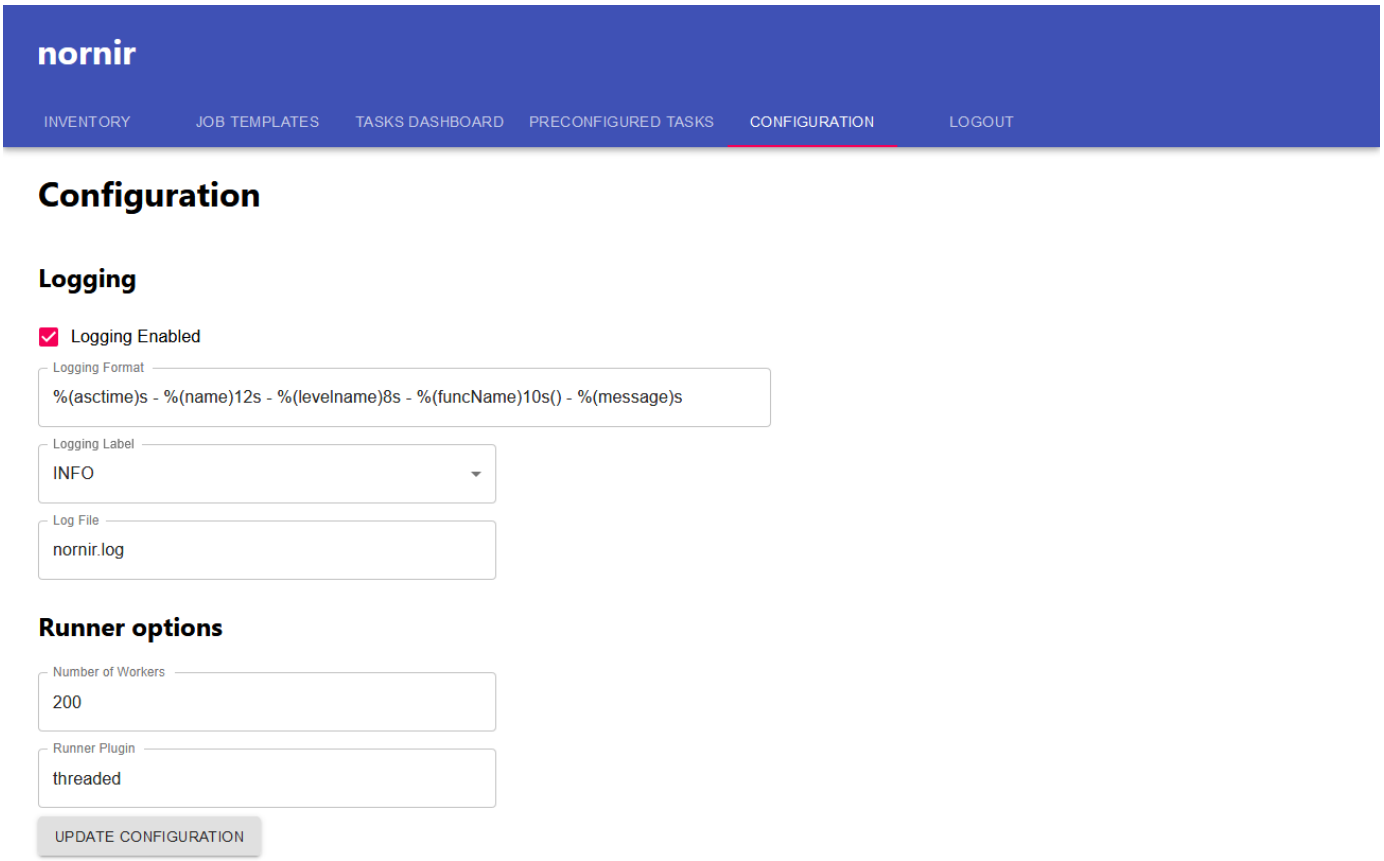


Abb. 17.8: Konfigurations-Seite für globale Nornir Einstellungen

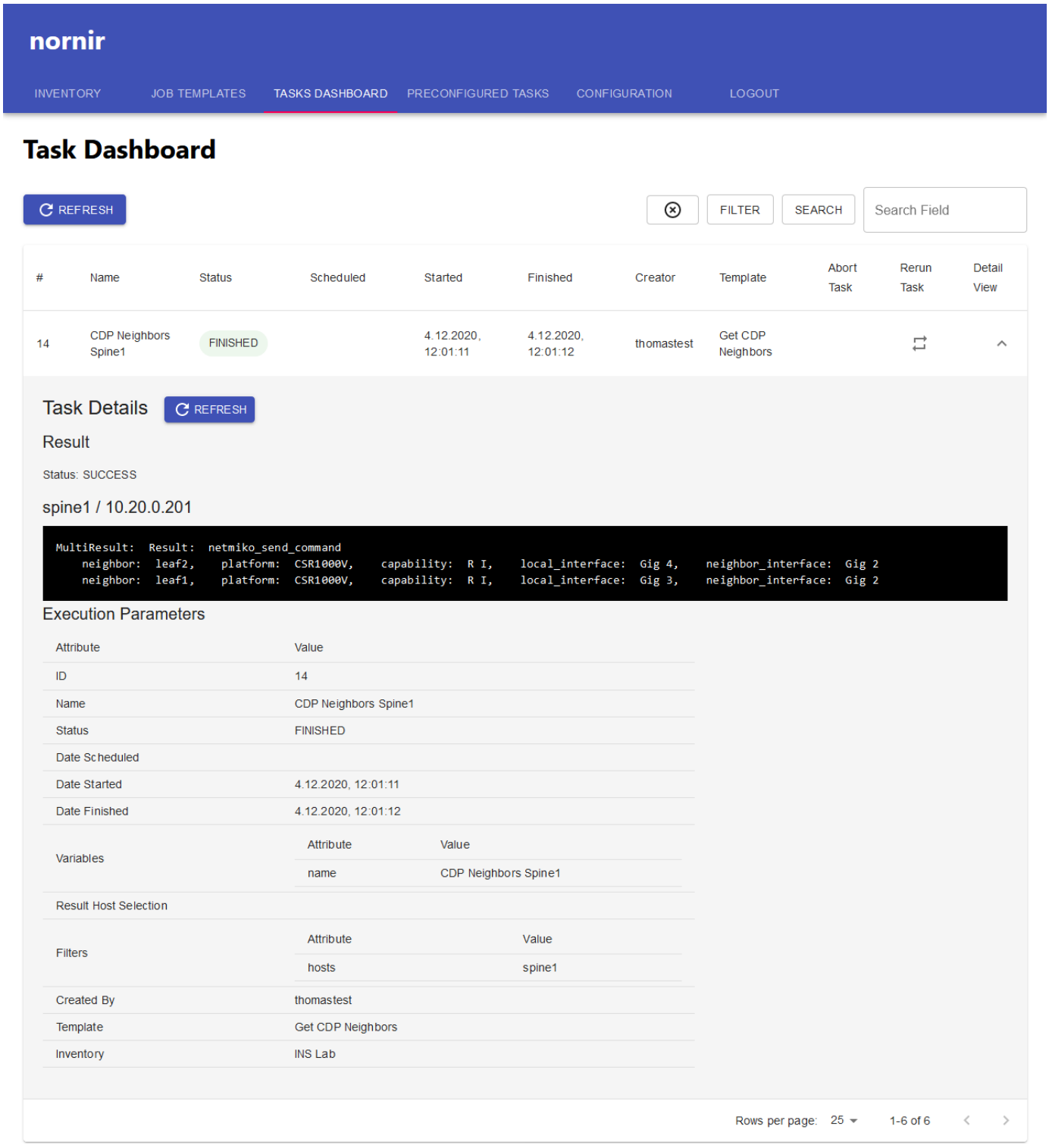


Abb. 17.9: Task Dashboard mit aufgeklapptem Result & Detail eines Task