



Entwicklung einer Ressourcenübersicht auf Basis von Jira

**zwölfwöchige Abschlussarbeit im Rahmen der Prüfung
im Studiengang Elektromobilität (B.A.)
an der Berliner Hochschule für Technik**

vorgelegt am: 14.04.2022

von: Nico Päller

Matrikelnummer: 892613

1. Gutachter: Prof. Dr. Sven Graupner
2. Gutachter: Alan Graf

Berliner Hochschule für Technik

Inhaltsverzeichnis

Abbildungsverzeichnis	III
Tabellenverzeichnis	IV
Akronyme	V
Glossar	VI
1 Einführung	1
1.1 Problemstellung	1
1.2 Zielsetzung	3
1.3 Vorgehensweise	4
2 Grundlagen	6
2.1 HOD Systemtest	6
2.2 REST API	6
2.3 HTTP Kommunikation	9
2.3.1 Allgemeines	9
2.3.2 Kommunikation	10
2.4 Programmiersprachen zur Backend Entwicklung	12
2.4.1 Python	12
2.4.2 Go	12
2.4.3 Gegenüberstellung	13
2.4.4 Fazit	15
3 Anforderungsanalyse	16
3.1 Analyse des aktuellen Testprozesses	16
3.2 Einbindung in vorhandene Programme	19
3.3 Use Cases	19
3.4 User Stories	21
4 Anhang	23

Abbildungsverzeichnis

1	Jira Ticket	2
2	HOD Lenkräder in einer Klimakammer	6
3	Diagramm des Beginns einer TCP-Verbindung	10
4	Geschwindigkeitsvergleich eines Webservers: Python vs. Go	14
5	UML Aktivitätsdiagramm zum täglichen Testprozess	17
6	UML Aktivitätsdiagramm zum Ressourcenmanagementprozess des Wartungsbeauftragten	18
7	UML-Use-Case-Diagramm zu den Anwendungsfällen für die Planer-, Techniker- und Wartungsbeauftragten-Rolle	20
8	UML-Use-Case-Diagramm zu den Anwendungsfällen für die Entwickler-Rolle	21
9	Auszug eines Tagesplans vom 02.03.2022	25
10	Übersichtsseite der Ressourcenliste vom 26.03.2022	25

Tabellenverzeichnis

1	REST API HTTP Anfragetypen	9
2	HTTP Statuskategorien	10
3	Ø Antwortzeit zwischen Go und Python Webservers	13

Akronyme

API Application Programming Interface. 9, 12, 13, 15, *Glossar:* API

ECU Electronic Control Unit. 6

HOD Hands On Detection. 1, 6, 12, 16, *Glossar:* HOD

HTML Hypertext Markup Language. 7, 16, *Glossar:* HTML

HTTP Hypertext Transfer Protocol. 8, 15, *Glossar:* HTTP

JQL Jira Query Language. 17, *Glossar:* JQL

JSON Javascript Object Notation. 7, 14, *Glossar:* JSON

JSS Joyson Safety Systems. 3, 16

MVP Minimal Viable Product. 4, *Glossar:* MVP

PDF Portable Document Format. 1, 16

REST Representational State Transfer. 9, 12, 13, 15, *Glossar:* REST

SVN Apache Subversion. 3, 16, 18, *Glossar:* SVN

TCP Transmission Control Protocol. 10

UI User Interface. 4

URI Uniform Resource Identifier. 7, *Glossar:* URI

Glossar

API Eine Schnittstelle, die von verschiedenen Programmen benutzt werden kann um die zur Verfügung gestellten Funktionen zu nutzen. 9

Backend Das in einer Server-Client-Architektur, auf dem Server ausgeführte Programm. 6

dynamisch typisiert Datentypen einer dynamisch typisierten Programmiersprache können sich zur Laufzeit ändern. 12

Framework Programmiergerüst mit einsatzbereitem Code oder Softwareplattform [Dud22c]. 12

Frontend Das in einer Server-Client-Architektur, auf dem Client ausgeführte Programm, welches das User Interface (UI) beinhaltet. 4

Garbage Collector eine automatische Speicherverwaltung. 12

HOD Hands On Detection; ein System das einem Lenkrad ermöglicht Griffe zu erkennen. 1

HTML Hypertext Markup Language; eine Sprache zum strukturellen Aufbau eines elektronischen Dokuments (z.B.: eine Webseite). 7

HTTP Hypertext Transfer Protocol; ein Protokoll zur Übertragung von Daten. 8

JIRA Jira (entwickelt von Atlassian) ist eine Webanwendung zur Fehlerverwaltung, Problembehandlung und zum operativen Projektmanagement. 9

JQL Jira Query Language; eine Sprache zum filtern der Jira Datenbank. 17

JSON Javascript Object Notation; ein Datenformat zum Austausch von Daten zwischen Anwendungen. 7

MVP Minimal Viable Product (deutsch: minimal funktionsfähiges Produkt); Implementierung von lediglich den essenziellen Funktionen eines Produkts [Rie11].

objektorientiert ein Programmierparadigma, mit dem die Konsistenz von Datenobjekten gesichert werden kann und das die Wiederverwendbarkeit von Quellcode verbessert [EK20]. 12

Open-Source Software, deren Quellcode frei zugänglich ist und die beliebig kopiert, genutzt und verändert werden darf [Dud22b]. 12

Planer Eine Person, die die Durchführung der Tests plant. 2

REST ein Paradigma für die Softwarearchitektur von verteilten Systemen, insbesondere für Webservices. 9

statisch typisiert Datentypen einer statisch typisierten Programmiersprache werden zur Kompilierzeit festgelegt und sind nicht änderbar. 12

SVN Apache Subversion; eine freie Software zur zentralen Versionsverwaltung von Dateien und Verzeichnissen. 3

Techniker Eine Person, die Tests aufbaut und durchführt. 1, 2, 16

URI Uniform Resource Identifier; ein Identifikator, welcher zur Bezeichnung von Webseiten etc. verwendet wird. 7

Listings

1	Beispielhafte HTTP Anfrage	11
2	Beispielhafte HTTP Antwort	11
3	Python Flask Webserver	23
4	Go Webserver	23
5	Python test Skript	24

1 Einführung

Die Effizienz in einem Unternehmen spielt immer mehr eine wichtige Rolle, da es häufig viele Deadlines gibt, die eingehalten werden müssen. Effizientes Arbeiten benötigt jedoch nicht nur eine genaue Planung, sondern auch eine Möglichkeit für alle Beteiligten, die vorgesehenen Prozessschritte einzusehen und einzuhalten. Gerade im Bereich der Digitalisierung, gibt es viel Potential, Prozesse effizienter zu gestalten. Circa ein Drittel der Maschinenbau- und Anlagenbau Unternehmen sehen eine Effizienzsteigerung durch die Digitalisierung [Bre17]

1.1 Problemstellung

Bei Joyson Safety Systems in der Abteilung ‘Hands On Detection (HOD) System Test’ werden die durchzuführenden Tests mittels der Ticketsoftware Jira geplant. Diese beinhaltet meistens hunderte einzelne Tests, welche in einer gewissen Reihenfolge durchgeführt werden. Anschließend wird jeden Morgen ein Tagesplan erstellt, in welchem genau beschrieben ist, welcher Test auf welche Weise durchzuführen ist. Dieser Plan liegt in Form einer Portable Document Format (PDF) Datei vor. Dadurch sind die Informationen, die die Techniker erhalten auf das Wichtigste begrenzt.

Um die Zeit des Umbaus zu minimieren und die verfügbare Zeit einer Klimakammer für Tests zu maximieren, muss ein Techniker jedoch wissen, was er für den nächsten Test umbauen muss. Diese Information steht ausschließlich im JIRA Ticket zur Verfügung, weshalb das Nachsehen im Ticket für jeden Test einen erheblichen Aufwand darstellt.



C3003889 HOD Audi VPE / C3003889-2888 MASTER TCSY105 TG_B_High/Low temperature storage / C3003889-2983
RUN TCSY105 TG_B_High/Low temperature storage

[Edit](#) [Comment](#) [Assign](#) [More](#) [Declare Executed](#)

Details

Type:	 Test Request Subtask	Status:	IN EXECUTION (View Workflow)
Priority:	 Medium	Resolution:	Unresolved
Affects Version/s:	None	Fix Version/s:	None
Component/s:	System Testing		
Labels:	DPV HW:H10 Nappa PPE Round SW:X11 erster_Test heated v1.3.2		
Test Case:	TCSY105		
Test Location:	304 - Electronics Testing - 2		
Test Run ID:	3042SY22_002356		
Test Class:	PV		
Sample ID:	PV_3889_0889		
Test Environment:	CC05		
Testing Remarks:	ablegen		

Description

Please perform TCSY105 TG_B_High/Low temperature storage

Attachments

 Drop files to attach, or browse.

...

Issue Links

+

has to be done before

 [C3003889-2986 RUN TCSY145 TG_B_PRE_Parameter Test \(large\)](#)

 **SCHEDULED**

Abbildung 1: Jira Ticket

Solch ein Jira Ticket ist zu unübersichtlich um schnell alle wichtigen Informationen zu erhalten. Wichtige Daten, wie etwa das zu verwendende Zubehör sind nur in Unstrukturierter Form vorhanden. Außerdem bringt das Heraussuchen jedes Tickets einen gewissen Zeitaufwand mit sich.

Zusätzlich werden die Wartungsinformationen momentan lediglich in einer Excel Tabelle gespeichert und vom Wartungsbeauftragten gepflegt. Somit weiß der Techniker oder der Planer nicht genau, ob das geplante Zubehör auch wirklich verwendet werden darf bzw. er muss sich darauf verlassen, dass es rechtzeitig gewartet wurde.

Aus den zuvor genannten Situationen treten folgende Probleme hervor:

1. Heraussuchen der Jiratickets für Informationen

Wenn der Techniker beispielsweise beim Umbau eines Tests wissen möchte, welcher Test folgt, muss er sich in Jira anmelden, das Ticket über die Testrun-ID aus dem Tagesplan suchen, und im Ticket gucken, ob der nächste Test verlinkt ist. Da in einer Klimakammer meistens 3 Muster getestet werden muss dieser Prozess für jedes weitere Muster wiederholt werden.

2. Wartungsdatum eines Zubehörs einsehen und anpassen

Wer wissen möchte wann ein gewissen Zubehör gewartet werden muss, bzw. ob es verwendet werden darf, muss, in einer Excel Datei nach dem entsprechenden Zubehör suchen. Da diese Excel Datei im Apache Subversion (SVN) abgelegt ist, wird sie auch nur darüber mit allen weiteren Benutzern synchronisiert. Dabei kann es auch zu unterschiedlichen Versionen der Datei kommen, wenn beispielsweise jemand vor dem einsehen die Datei seine lokale Working Copy aktualisiert.

1.2 Zielsetzung

Das Programm wird unter dem Namen ‘TestHub’ innerhalb der Firma Joyson Safety Systems (JSS) veröffentlicht. Die Arbeit hat folgende Ziele:

- Entwicklung einer interaktiven Übersicht zum schnellen Überblick über Wartungstermine, aktuelle, vorherige und folgende Tests
- Aufbereitung und Visualisierung der unstrukturierten Daten in den Labels eines Jira Tickets
- Zentrale Speicherung, Einsicht und Anpassung von Testzubehör Informationen
- Entwicklung einer offenen Schnittstelle, der zuvor genannten Punkte, zur Integration in andere Programme

Die Entwicklung von TestHub lässt sich in zwei grobe Punkte unterteilen:

Entwicklung des Backends

Das Backend ist die zentrale serverseitige Software, welche die nötigen Daten von Jira oder der Datenbank sammelt und aufbereitet über das Internet versendet.

Entwicklung des UI

Um die Daten visuell und kompakt darzustellen, gibt es ein Interface, welches die vom Backend empfangenen Daten übersichtlich auf einer Webseite anzeigt

Durch eine Trennung des Backends und des Frontends entstehen verschiedene Vorteile. Zum einen können andere Programme das Backend ebenfalls benutzen und die nötigen Daten bei sich integrieren. Es lässt sich außerdem leicht erweitern, Zum Anderen lässt sich das UI leicht austauschen bzw anpassen, da es lediglich die vorhandenen Daten anzeigt.

Testhub ist als Minimal Viable Product (MVP) entwickelt, es werden daher nur die wichtigsten in Abschnitt

1.3 Vorgehensweise

Die vorliegende Arbeit wird in mehrere Kapitel unterteilt

Kapitel 1: Einleitung: In der Einleitung werden die Problemstellung und die Ziele mitsamt der Motivation der Arbeit beschrieben

Kapitel 2: Grundlagen: Im Kapitel über die Grundlagen werden Implementierungsmöglichkeiten diskutiert und abgewogen. Zusätzlich wird dort die Strukturierung von unstrukturierten Daten besprochen.

Kapitel 3: Anforderungsanalyse: Im Kapitel 3 werden die aktuellen Probleme analysiert und Anforderungen an das zu entwickelnde Programm definiert.

Kapitel 4: Entwurf: Im Entwurf wird sowohl die Softwarearchitektur des Programms als auch Gestaltung des UI beschrieben

Kapitel 5: Entwicklung: Im Entwicklungsteil der Arbeit werden sowohl die Implementierung als auch der Softwarelebenszyklus des gesamten Projekts beschrieben.

Kapitel 6: Verifizierung und Validierung: Die in Kapitel 3 ermittelten Anforderungen werden in diesem Kapitel den tatsächlichen Funktionen des Programms gegenübergestellt und verifiziert.

Kapitel 7: Fazit: Im Fazit Kapitel werden die Ergebnisse der Arbeit zusammengefasst und weitere mögliche Schritte hinsichtlich der Weiterentwicklung besprochen.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der im HOD System Test erläutert. Anschließend wird auf das zentrale Element der Arbeit, die REST API eingegangen. Zusätzlich werden Programmiersprachen zur Entwicklung vom Backend gegenübergestellt und eine potenzielle Verwendung abgewogen.

2.1 HOD Systemtest

Die Abteilung Hands-On-Detection entwickelt ein Lenkrad, welches durch eine kapazitive Matte im Lenkrad erkennen kann, ob und wie ein Lenkrad berührt wurde. Dieses System muss nicht nur entwickelt sondern auch ausgiebig getestet werden. Da Joyson nach dem V-Modell testet, werden die Tests auf mehrere Ebenen verteilt. Die Ebene des Systemtests, testet das ganze System, das heißt es wird das ganze Lenkrad mit der von Joyson entwickelten ECU getestet.



Abbildung 2: HOD Lenkräder in einer Klimakammer

Die verschiedenen Funktionen der Lenkräder wird bei -40°C bis 80°C getestet. Dabei werden meistens 3 Lenkräder in einer Klimakammer getestet. Alle, für diese Tests verwendeten Teile, wie Kabelbäume, Messhardware, die Klimakammer etc., werden Ressourcen genannt.

2.2 REST API

Seit der ersten Website von Tim Berners-Lee 1991 wächst das Web teilweise exponentiell. Allerdings war das Web nicht für solch ein rasantes Wachstum gemacht. Es

gab weder einheitliche Kommunikationsprotokolle, noch konnte die Infrastruktur den Datenverkehr standhalten. Alarmiert von diesem Problem, entwickelte Roy Fielding eine Web Architektur, welche die Bedingungen des Webs einheitlich lösen soll. Diese Bedingungen lassen sich in 6 Kategorien zusammenfassen [Mas11]:

1. Client-Server

Die Client-Server Struktur stellt eine Trennung der Anliegen dar. Der Client fordert dabei einen Dienst oder eine Information an, welche der Server erfüllt. Die Technologie welche zum Entwickeln von Client und Server verwendet wird, spielt dabei keine Rolle, solange sie das einheitliche Interface implementieren.

2. Einheitliches Interface

Mark Massé fasst in seinem Buch *Rest Api: Design Rulebook* [Mas11] das einheitliche Interface in 4 Anforderungen zusammen:

1. Identifikation von Ressourcen

Jedes webbasierte Konzept (Ressource) muss über einen einzigartigen Uniform Resource Identifier adressiert werden können.

2. Manipulation von Ressourcen durch Repräsentationen

Clients müssen die Ressourcen manipulieren können, indem Sie mit verschiedenen Repräsentationen arbeiten. Beispielsweise kann ein Dokument sowohl in JSON für ein automatisiertes Programm als auch als HTML für einen Webbrower repräsentiert werden. Dadurch lässt sich laut Massé eine Interaktion mit dem Dokument gewährleisten, ohne das Dokument und seinen Identifier zu verändern.

3. Selbstbeschreibende Nachrichten

Wenn der Client eine Anfrage schickt, ist dies nur der gewünschte Zustand der Ressource. Der tatsächlich aktuelle Zustand, ist repräsentativ in der Antwort des Servers enthalten. Wenn also jemand einen Kommentar bei YouTube schreibt, schlägt er nur den Inhalt des Kommentars vor. Ob der Kommentar tatsächlich übernommen und angezeigt wird, hängt allein vom Server ab. Um diese selbstbeschreibenden Nachrichten mehr Informationen über die versendete oder angefragte Ressource enthalten zu lassen, können Metadaten in den Nachrichten enthalten sein. Diese Metadaten können zum Beispiel die

Art der Repräsentation, die Länge der Daten oder eine Authentifizierung sein.

Bei einer HTTP-Nachricht werden diese Metadaten in die “Header” geschrieben, welche vordefinierte Zwecke besitzen.

4. Hypermedien als Antrieb des Applikationsstatus

Laut Massé, sind Links die “Fäden die das Netz zusammennähen” [Mas11, S. 4]. Daher sollte es in dem einheitlichen Interface die Möglichkeit einer Navigation der Informationen durch Links geben.

3. Schichtsystem

Durch ein Schichtsystem soll ermöglicht werden, Zwischensysteme, wie einen Proxy Server oder ein Gateway zu etablieren. Diese Systeme werden benötigt, um beispielsweise Sicherheitsstandards zu erzwingen oder um viele gleichzeitige Anfragen auf die vorhandene Hardware zu balancieren (load balancing).

4. Caching

Caching ist das Zwischenspeichern von Informationen. Um den Server zu entlasten und somit Geld zu sparen und zusätzlich die Latenz des Clients zu verringern, sollten Ressourcen zwischengespeichert werden, sodass bei einer erneuten Anfrage die zwischengespeicherte Version geladen wird und keine neue Datenübertragung initialisiert werden muss. Caching wird von allen modernen Webbrowsern automatisch betrieben, kann aber auch von den oben genannten Zwischensystemen umgesetzt werden.

5. Zustandslosigkeit

Die Anforderungen der Zustandslosigkeit beschreibt, dass alle kontextuellen Informationen in der Anfrage des Clients enthalten sein muss, sodass der Server keine Informationen zum Client speichern muss. Dadurch kann der Server wesentlich mehr Anfragen bearbeiten, das der Aufwand pro Anfrage gering gehalten wird. Die Zustandslosigkeit trägt erheblich zur Skalierung der Architektur des Webs bei, laut Massé [Mas11, S. 4].

6. Code-On-Demand

Code-On-Demand ist die Möglichkeit, ausführbare Skripte oder Programme vom

Server an den Client zu schicken. Da der Client jedoch den empfangenen Code verstehen und ausführen muss, ist dies die einzige optionale Anforderung an die Architektur des Webs.

Diese zuvor genannten Anforderungen, werden elegant durch eine REST API erfüllt. REST steht für Representational State Transfer und beschreibt eine Ansammlung von Regeln, nach welchem man seine API architektonisch aufbauen sollte. Diese Regeln lassen sich jedoch auf die unterschiedlichsten Weisen implementieren.

2.3 HTTP Kommunikation

Das Hypertext Transfer Protokoll (kurz HTTP) regelt die Datenübertragung zwischen Anwendungen. Viele REST APIs verwenden HTTP, da dieses gewisse Anforderung, wie die Zustandslosigkeit, bereits implementiert. Im Folgenden soll die Funktionsweise vom HTTP erläutert werden, da dies grundlegend für sowohl “TestHub” als auch die JIRA Kommunikation ist.

2.3.1 Allgemeines

HTTP implementiert verschiedene Anfragetypen und Headertypen. Die wichtigsten, welche auch von der REST API von “TestHub” verwendet wird, werden hier aufgelistet:

Anfragetyp	Beschreibung
GET	erhalten einer Ressource
POST*	erstellen einer Ressource oder Query-Abfrage
PUT*	editieren einer Ressource
DELETE	löschen einer Ressource

Tabelle 1: REST API HTTP Anfragetypen
* darf Daten im Body der Anfrage versenden

Außerdem hat jede Antwort des Servers auch einen entsprechenden Status Code, welcher verschiedene Aussagemöglichkeiten hat:

Zusätzlich gibt es eine Vielzahl an Headern, welche verwendet werden können. “TestHub” verwendet dabei nur die Standardheader wie *Content-Length*, *Date* und *Content-Type*, um dem Client mitzuteilen, um welche Art der Repräsentation es sich handelt.

HTTP Statuscode	Kategorie
1XX	Information
2XX	Erfolg
3XX	Weiterleitung
4XX	Client Error
5XX	Server Error

Tabelle 2: HTTP Statuskategorien

2.3.2 Kommunikation

HTTP verwendet TCP, welches für eine akkurate Übertragung der Daten sorgt. Wenn der Client nun mit dem Server kommunizieren möchte, muss er zuerst eine TCP Verbindung aufbauen.

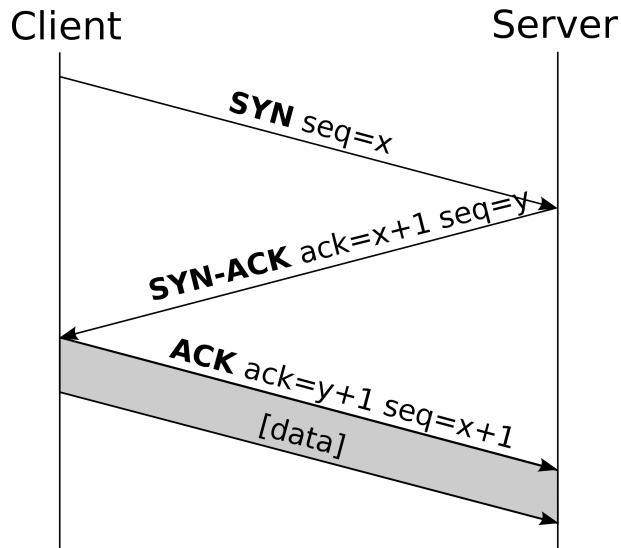


Abbildung 3: Diagramm des Beginns einer TCP-Verbindung [Wik10]

SYN

Der Client erstellt eine zufällige Zahlenfolge x und versendet diese (und eventuell weitere TCP Optionen) in einem *SYN* Packet an den Server.

SYN-ACK

Der Server inkrementiert x um 1 und erstellt selbst eine zufällige Zahlenfolge y . Der Server kann an dieser Stelle seine eigenen TCP Optionen zusammen mit den Zahlenfolgen x und y an den Client zurückschicken.

ACK

Der Client inkrementiert sowohl x als auch y um 1 und sendet das Packet wieder an den Server zurück.

Sobald der Server die *ACK* Nachricht empfangen hat, ist der Handshake abgeschlossen. Von nun an können Daten versendet werden. Dies geschieht über eine HTTP Nachricht, welche wie folgt aussehen kann:

```
1 GET / HTTP/1.1
2 Host: domain.com
3 Accept-Language: de
```

Listing 1: Beispielhafte HTTP Anfrage

In dieser Nachricht sind Anfragetyp, und die HTTP Version enthalten. *Host* und *Accept-Language* sind dabei Header, welche gewisse Metadaten zur Anfrage enthalten. HTTP/1.1 ist dabei noch in Klartext gestaltet, und somit von Menschen lesbar. HTTP/2 verwendet das gleiche Prinzip, jedoch sind die Nachrichten als Frames verpackt und somit nicht mehr einfach so lesbar.

Sobald der Server alle vom Client angefragten Ressourcen gesammelt hat, sendet er eine Antwort:

```
1 HTTP/1.1 200 OK
2 Date: Sat, 09 Oct 2010 14:28:02 GMT
3 Content-Length: 29769
4 Content-Type: text/html
5
6 <!DOCTYPE html ... (29769 Bytes der angefragten Seite)
```

Listing 2: Beispielhafte HTTP Antwort

Üblicherweise sendet der Server zu seiner Antwort, für den Client wichtige Header, wie der zuvor genannte *Content-Type* Header, wodurch der Client weiß, wie er die empfangenen Daten interpretieren soll. Die tatsächlichen Daten stehen dabei ganz unten im sogenannten Body. Der Server sendet zusätzlich einen Status, in diesem Fall ist das der Status 200, welcher für einen unspezifischen Erfolg steht.

Schlussendlich kann die Verbindung geschlossen werden, um Platz für andere Anfragen von anderen Clients zu schaffen oder die Verbindung kann für Folgeanfragen genutzt

werden.

2.4 Programmiersprachen zur Backend Entwicklung

Die Grundlage und demnach auch die Performance des Backends bildet die verwendete Programmiersprache. Joyson verwendet in der Firma hauptsächlich Python, C# und Go. Da mit C# nicht in der Abteilung HOD System Test entwickelt wird, werden lediglich die Sprachen Python und Go gegenüber gestellt und ein Fazit zur verwendeten Sprache gebildet.

Sowohl Python als auch Go sind Open-Source und somit frei verwendbar. Es gibt für beide Sprachen mehrere Bibliotheken zur Entwicklung einer REST API.

2.4.1 Python

Python wurde von Guido van Rossum 1989 in Amsterdam entwickelt. Es ist eine dynamisch typisierte Skriptsprache, welche auch objektorientiert nutzbar ist. Python ist allerdings eine interpretierte Sprache, das bedeutet, dass der compilierter Byte-Code in einer virtuellen Maschine ausgeführt wird. Diese virtuelle Maschine nennt man Interpreter [EK20].

Trotz der umfangreichen Standardbibliothek von Python kann ein Webserver nicht ohne Weiteres entwickelt werden. Hierfür wird ein Framework benötigt, wie zum Beispiel Django, Flask oder FastAPI. Da Flask laut *StackOverflow Developer Survey 2021* unter den genannten 3 Frameworks, das ist, welches am meisten genutzt wird [Sta21], bezieht sich die folgende Gegenüberstellung auf diese Bibliothek.

2.4.2 Go

Die Programmiersprache Go wurde 2012 von der Firma Google veröffentlicht. Go wirbt damit, effizient und übersichtlich zu sein. Weiterhin kann in Go Concurrency, also das parallele ausführen von Code, sehr leicht umgesetzt werden. Im Gegensatz zu Python ist Go statisch typisiert, beide Sprachen besitzen jedoch einen Garbage Collector.

Go bietet in seiner Standardbibliothek schon das ‘net/http’ package an, welches verwendet werden kann um einen Webserver zu implementieren.

2.4.3 Gegenüberstellung

Sowohl Python als auch Go sind bekannte Sprachen, wobei Python wesentlich beliebter ist. Python wird von ca. 48% der an der *StackOverflow Developer Survey 2021* teilgenommenen Entwickler verwendet, Go hingegen nur ca. 10%. Allerdings ist Go auf Platz 4 der Sprachen, die die meisten dieser Entwickler lernen wollen [Sta21]. Demnach gibt es für beide Sprachen eine ausreichende Community für Fragen oder Probleme. Außerdem werden beide Sprachen wahrscheinlich in den nächsten Jahren weiterhin verwendet werden.

Da ‘TestHub’ so schnell und responsive wie möglich sein sollte, wurde ein Geschwindigkeitstest durchgeführt. Es wurden jeweils ein Webserver in Go mit dem ‘net/http’ Package und in Python mit Flask entwickelt. Beide Webserver stellen einen REST API Endpunkt bereit, welcher eine Test Website [Bra22] verschickt. Es wurden anschließend 100 asynchrone Request an diesen Endpunkt geschickt und die jeweilige Zeit gemessen.

Framework	\varnothing Antwortzeit [s]
Go (net/http)	0,2242
Python (Flask)	2,2979
Leistungsunterschied	10,2493

Tabelle 3: \varnothing Antwortzeit zwischen Go und Python Webservers

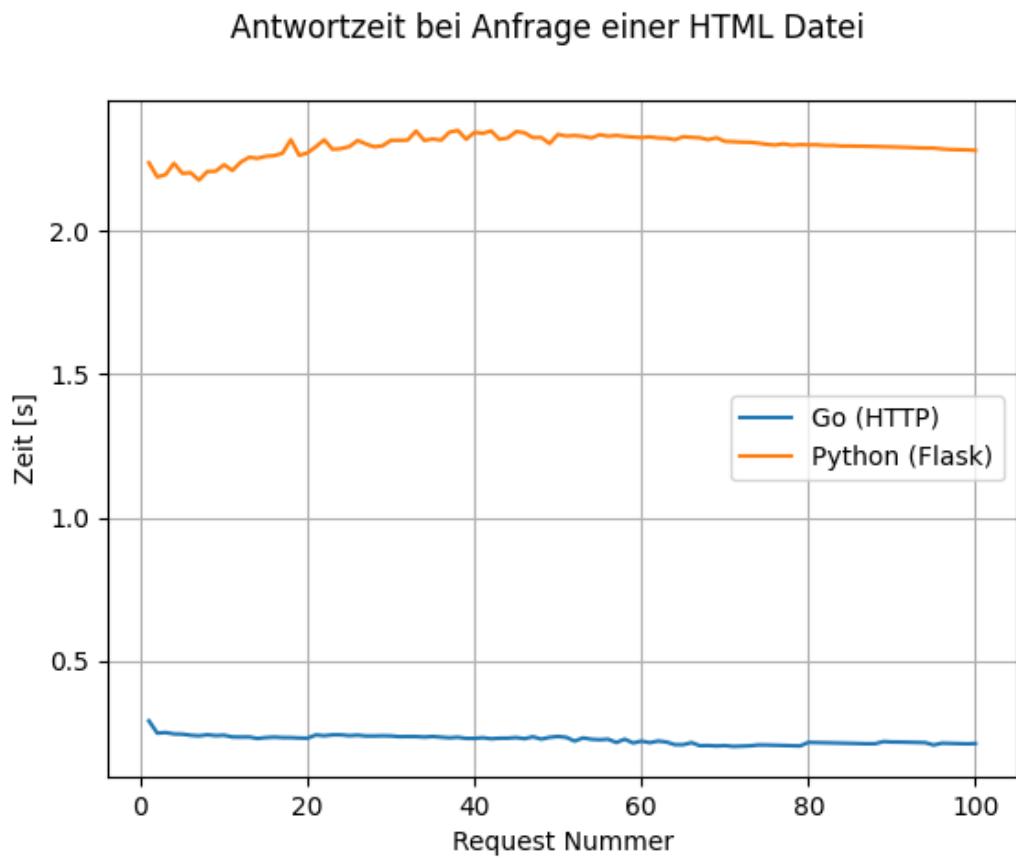


Abbildung 4: Geschwindigkeitsvergleich eines Webservers: Python vs. Go

Tests wurden lokal auf einem Dell Latitude 5590 (Intel Core i5-8350U CPU @ 1.70GHz; 8GB RAM) durchgeführt, der Code lässt sich im Anhang finden

In Abbildung 4 ist die Antwortzeit in Sekunden über der Nummer der Anfrage dargestellt. Es ist schnell ersichtlich, dass Go ca. 10 mal schneller als Flask ist, zumindest beim verschicken einer HTML Datei. Diese Ergebnisse decken sich mit den Web Framework Benchmarks von TechEmpower, wo Go beim versenden von Text auf Platz 18 vor Flask auf Platz 55 ist. Auch bei der JSON Serialisierung liegt Go (Platz 22) weit vor Flask (Platz 59) [Tec21]. Weiterhin ist die Performance des Go Servers mehr konsistent, als die des Python Servers

2.4.4 Fazit

Da die Geschwindigkeit der Webseite und der REST API essentiell für die Benutzererfahrung ist, sowohl beim Laden der Seite, als auch beim bearbeiten von HTTP Anfragen, beispielsweise Suchanfragen, ist Go hier klar die bessere Alternative. Des weiteren werden nur weniger Abhängigkeiten benötigt, da Go schon viele benötigten Webserver Funktionen in seiner Standardbibliothek hat. Somit ist das Programm, langfristig gesehen, weniger anfällig für Fehler bei Aktualisierungen der Bibliotheken.

3 Anforderungsanalyse

Das folgende Kapitel analysiert den bestehenden Prozess zur Ressourcenübersicht beim HOD Systemtest. Basierend auf dieser Analyse wird der Soll-Zustand des Projekt definiert, indem genaue Anforderungen an das System gestellt werden. Diese Anforderungen werden priorisiert, um anschließend die Funktion des fertigen Systems zu validieren.

3.1 Analyse des aktuellen Testprozesses

Aufgrund der Mitgliedschaft von mehreren Jahren bei JSS und der Entwicklung von mehreren Programmen, die diesen Prozess automatisieren oder unterstützen, Infrastruktur mir der Prozess äußerst bekannt. Dennoch wurden Interviews mit den Hauptbeauftragten der unten beschriebenen Rollen durchgeführt. Um den Prozess besser zu verstehen, wurden UML Aktivitätsdiagramme erstellt. Da die Rollen teilweise wenig miteinander interagieren, wurden die Prozesse der jeweiligen Rollen einzelnen betrachtet.

Der aktuelle Testprozess basiert auf einem täglich erstelltem Tagesplan (Siehe 9). Dieser Tagesplan beinhaltet die Tests, welche an diesem Tag von den Technikern durchgeführt werden sollen, mit den wichtigsten Informationen, wie zum Beispiel das Programm zur Reihenfolge der simulierten Griffe. Da diese Liste als Jira Auszug in Form einer PDF oder statische HTML Datei zur Verfügung gestellt wird, kann sie keinerlei Informationen zu vorherigen oder folgenden Tests liefern, welche jedoch für einen effizienten Umbau benötigt werden.

Weiterhin gibt es keinerlei Informationen zu Wartungszeiten der einzelnen Ressourcen. Diese müssen in einer separaten Ressourcenliste eingesehen werden (Siehe 10). Die Aktualisierung dieser Liste findet nur über das Versionskontrollsystem SVN statt und liegt verborgen in einer Vielzahl an Ordnern. Zusätzlich muss der Start der Benutzung der Ressource eingetragen werden. Dadurch muss die Liste nach einer Wartung erneut angepasst werden. So ergeben sich die folgenden Prozesse:

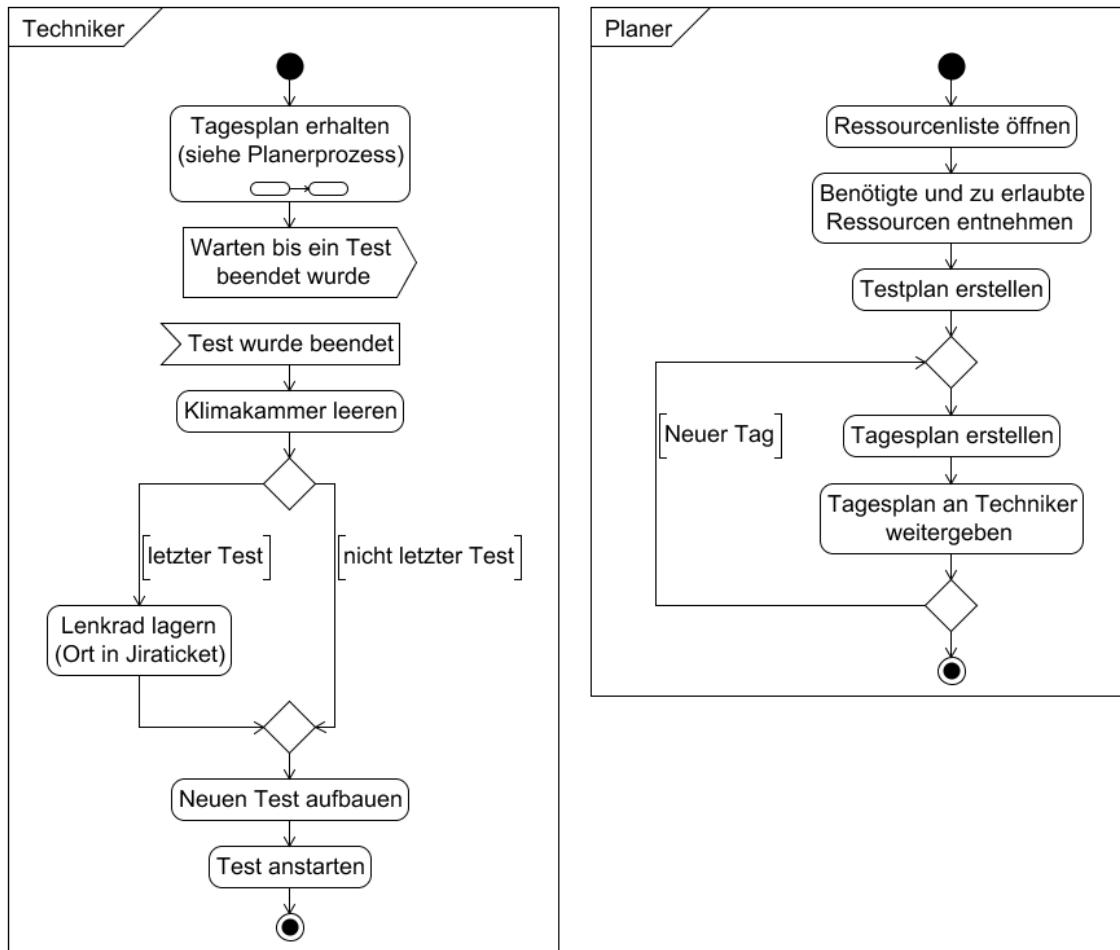


Abbildung 5: UML Aktivitätsdiagramm zum täglichen Testprozess

Aus Abbildung 5 gehen zwei verschiedene Rollen hervor. Der Planer (rechts) ist verantwortlich für das langfristige Planen von Tests. Dabei muss dieser beachten, dass die Wartungstermine eingehalten werden, wenn eine gewisse Ressource eingeplant wird. Welche Ressourcen überhaupt zur Verfügung stehen und wann die Wartungstermine überhaupt sind, kann der Planer aus der Ressourcenliste entnehmen. Zusätzlich erstellt der Planer auch den täglichen Tagesplan, welcher nur das Ergebnis eines JQL Filters ist.

Der Techniker (links) hingegen, ist für den Umbau und die tatsächliche Durchführung der Tests verantwortlich. Er benötigt den Tagesplan um zu wissen, welche Tests als nächstes durchgeführt werden. Da im Tagesplan allerdings nicht vermerkt ist, welcher Test nach einer spezifischen Test durchgeführt werden muss, muss er die Klimakammer so weit es nötig ist leeren, um anschließend den nächsten Test aufzubauen. Zusätzlich

weiß er nicht, welcher Test der letzte einer Testgruppe ist. Diese Information ist nötig, um die anschließende Lagerung des Lenkrads einzuleiten. Um diese Information zu erhalten, muss er im jeweiligen Jira Ticket nachschauen.

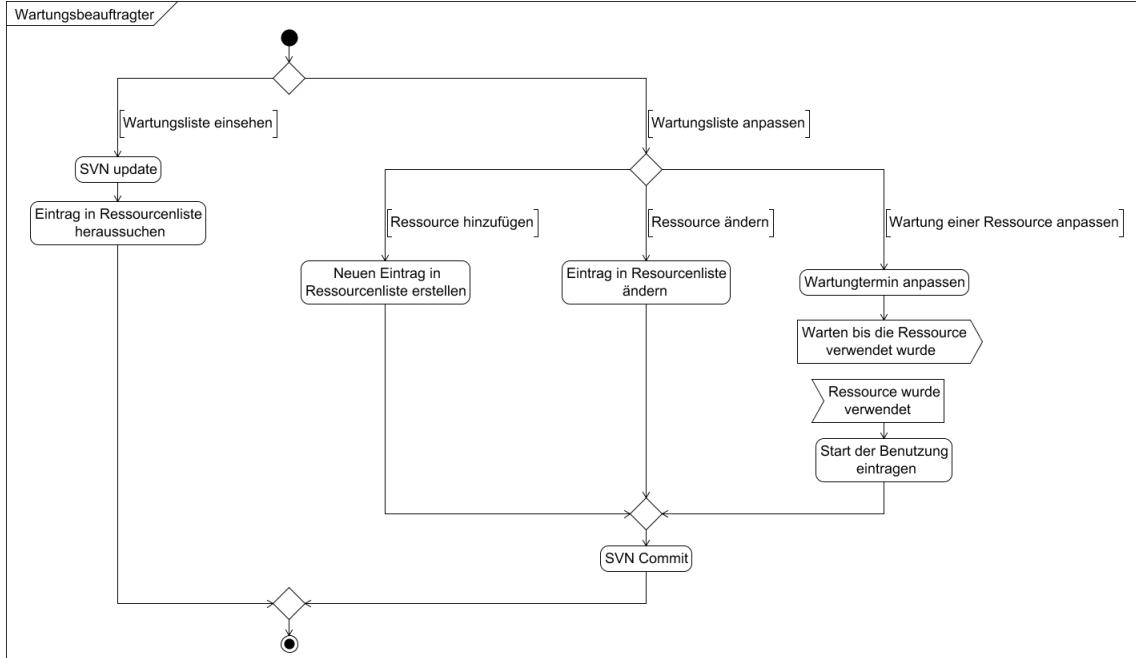


Abbildung 6: UML Aktivitätsdiagramm zum Ressourcenmanagementprozess des Wartungsbeauftragten

Der in Abbildung 6 beschriebene Prozess zeigt den Umgang mit der Ressourcenliste. Der Wartungsbeauftragte muss jedes mal, wenn er die Wartungsliste einsehen will, vorher ein SVN Update machen, um auch tatsächlich die aktuelle Version der Liste zu erhalten. Wird dies nicht getan, kann es bei Änderungen zu SVN Konflikten kommen oder die entnommenen Informationen sind veraltet. Falls er irgendetwas an der Liste anpasst, muss diese anschließend wieder ins SVN hochgeladen werden, um die Informationen mit allen anderen Mitarbeitern zu synchronisieren. Zusätzlich muss zu jedem neuen Wartungstermin, das tatsächliche Datum der ersten Benutzung seit der Wartung eingetragen werden.

Somit ergeben sich die folgenden Rollen:

Planer Die Rolle der Person, die Pläne für kurzfristige oder langfristige Tests erstellt.

Wartungsbeauftragter Die Rolle der Person, die Wartungen überwacht, plant und durchführt.

Techniker Die Rolle der Person, die Tests aufbaut, umbaut und durchführt

Auch wenn es sein kann, dass eine Person mehrere Rollen annimmt, ist es dennoch wichtig die Rollen voneinander zu unterscheiden um den Prozess nachvollziehbarer und übersichtlicher zu gestalten.

3.2 Einbindung in vorhandene Programme

Da gerade die Wartungsinformationen auch in anderen Programmen benötigt werden, beispielsweise um Warnungen bei abgelaufener Wartungen bei der Testvorbereitung anzeigen lassen zu können, ergibt sich eine weitere Rolle:

Entwickler Die Rolle der Person, die die Informationen der offenen Schnittstelle für eigene Programme verwendet

3.3 Use Cases

In Abbildung 7 und Abbildung 8 sind die bereits definierten Rollen als Akteure dargestellt. Die Use Cases ergeben sich aus den Interviews.

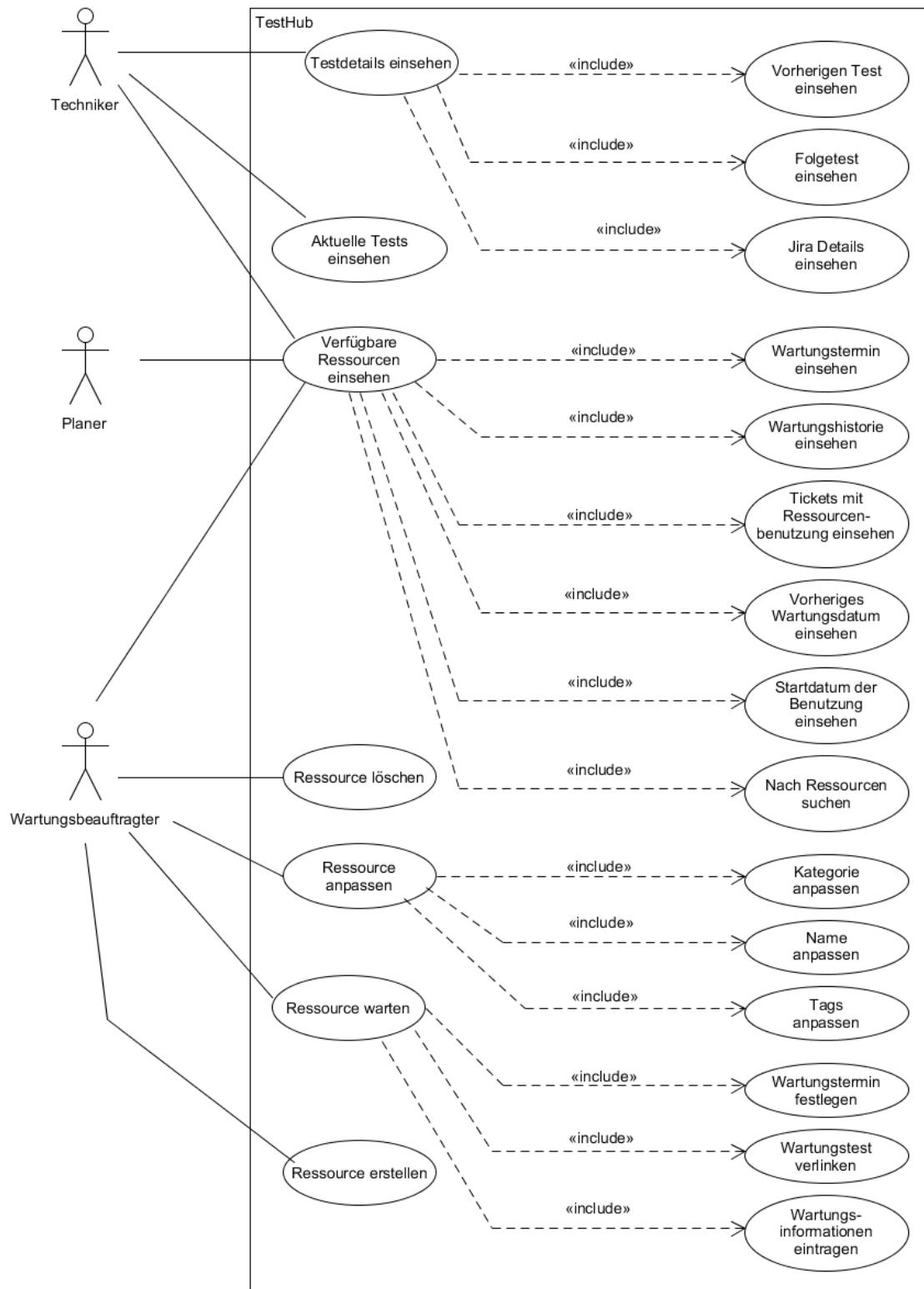


Abbildung 7: UML-Use-Case-Diagramm zu den Anwendungsfällen für die Planer-, Techniker- und Wartungsbeauftragten-Rolle

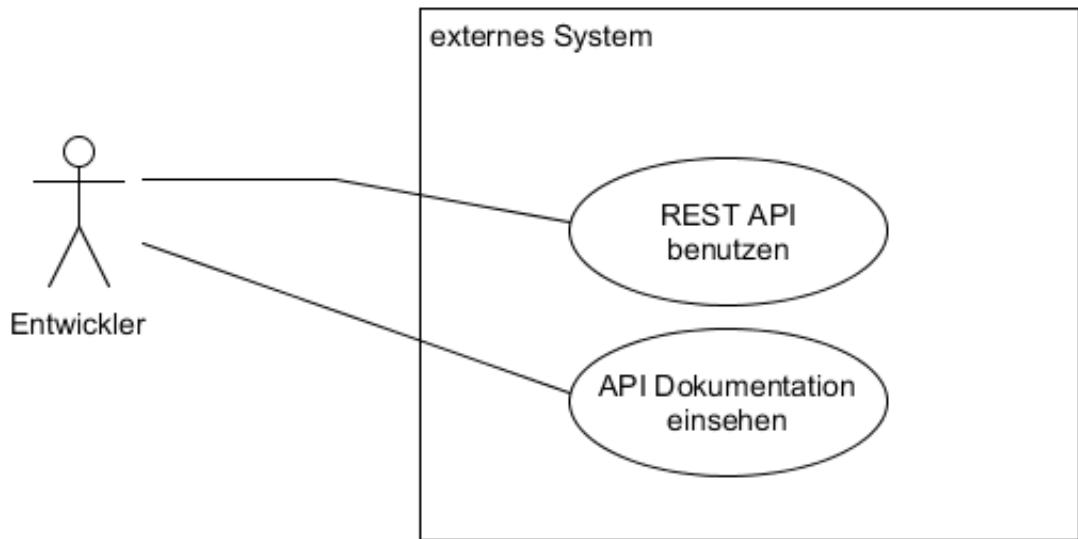


Abbildung 8: UML-Use-Case-Diagramm zu den Anwendungsfällen für die Entwickler-Rolle

3.4 User Stories

Die User Stories ergeben sich aus den Use Cases.

US#01 *Als Techniker möchte ich einen vorherigen Test einsehen*

US#02 *Als Techniker möchte ich einen folgenden Test einsehen*

US#03 *Als Techniker möchte ich die Details eines Jiratickets in aufgeschlüsselter Form einsehen*

US#04 *Als Techniker möchte ich alle aktiven Tests einsehen*

US#10 *Als Planer möchte ich einen Wartungstermin einsehen, um zu erfahren ob ich diese Ressource einplanen kann*

US#11 *Als Planer möchte ich eine Wartungshistorie einsehen, um eventuelle vergangene Probleme aufzudecken*

US#12 *Als Planer möchte ich alle aktiven Tickets sehen, welche eine spezifische Ressource verwenden*

US#13 *Als Planer möchte ich das vorherige Wartungsdatum einsehen, um zu wissen, ob das Startdatum der Benutzung hinter dem Wartungsdatum liegt und ich somit die Ressource verwenden kann*

US#14 *Als Planer möchte ich das Startdatum der Benutzung einsehen, um zu wissen, ob das Startdatum der Benutzung hinter dem Wartungsdatum liegt und ich somit die Ressource verwenden kann*

US#15 *Als Planer möchte ich nach Ressourcen suchen*

US#20 *Als Wartungsbeauftragter möchte ich eine Ressource löschen*

US#21 *Als Wartungsbeauftragter möchte ich die Kategorie einer Ressource anpassen*

US#22 *Als Wartungsbeauftragter möchte ich den Namen einer Ressource anpassen*

US#23 *Als Wartungsbeauftragter möchte ich die Tags einer Ressource anpassen*

US#24 *Als Wartungsbeauftragter möchte ich einen neuen Wartungstermin festlegen*

US#25 *Als Wartungsbeauftragter möchte ich das Ergebnis eines Wartungstest verlinken*

US#26 *Als Wartungsbeauftragter möchte ich Informationen zu einer Wartung speichern*

US#27 *Als Wartungsbeauftragter möchte ich eine Ressource erstellen*

US#90 *Als Entwickler möchte ich die REST API verwenden, um die Informationen von "TestHub" in meine Programm einzubinden*

US#91 *Als Entwickler möchte ich die Dokumentation der API ansehen, um zu verstehen wie ich die API benutzen kann*

Literatur

- [Wik10] Wikimedia. *Tcp-handshake.svg*. 2010. URL: <https://commons.wikimedia.org/wiki/File:Tcp-handshake.svg> (besucht am 25.03.2022).
- [Mas11] M. Massé. *REST API Design Rulebook*. Oreilly and Associate Series. O'Reilly Media, 2011. ISBN: 9781449310509. URL: <https://books.google.de/books?id=4lZcsRwXo6MC>.
- [Bre17] A. Breitkopf. „Wird es in Ihrem Unternehmen durch die Digitalisierung (Industrie 4.0) künftig neue Effizienz- und/oder Flexibilisierungspotenziale geben?“ In: *PLoS medicine* (2017).
- [EK20] Johannes Ernesti und Peter Kaiser. *Python 3: Das umfassende Handbuch*. Rheinwerk Computing, 2020.
- [Sta21] StackOverflow. *2021 Developer Survey*. 2021. URL: <https://insights.stackoverflow.com/survey/2021> (besucht am 22.03.2022).
- [Tec21] TechEmpower. *Web Framework Benchmarks: Round 20 (08.02.2021)*. 2021. URL: <https://www.techempower.com/benchmarks/#section=data-r20&hw=ph&test=json&l=zijmrj-sf> (besucht am 22.03.2022).
- [Bra22] Chris Bracco. *HTML5 Test Page*. <https://github.com/cbracco/html5-test-page>. 2022.

4 Anhang

```
1 from flask import Flask, send_from_directory
2
3 # create Flask App
4 app = Flask(__name__)
5 app.config.from_object(__name__)
6
7 # add file endpoint that serves the testfile
8 @app.route('/file')
9 def send_report():
10     return send_from_directory('..', "testfile.html")
11
12 # run app on Port 3000
13 if __name__ == '__main__':
14     app.run(port=3000)
```

Listing 3: Python Flask Webserver

```
1 package main
2
3 import (
4     "log"
5     "net/http"
6 )
7
8 // create HTTP Handle Func that serves the testfile
9 func fileServe(w http.ResponseWriter, r *http.Request) {
10     http.ServeFile(w, r, "./testfile.html")
11 }
12
13 func main() {
14     // add endpoint
15     http.HandleFunc("/file", fileServe)
16
17     // start server on port 3000
18     log.Println("Listening on :3000...")
19     err := http.ListenAndServe(":3000", nil)
20     if err != nil {
21         log.Fatal(err)
22     }
23 }
```

```
22     }
23 }
```

Listing 4: Go Webserver

```
1 import grequests
2
3
4 ITERATIONS = 100
5 URL = "http://localhost:3000/file"
6 FILENAME = "./results.csv"
7
8 results = (grequests.get(URL) for _ in range(ITERATIONS))
9 results = grequests.map(results)
10
11
12 with open(FILENAME, "w") as f:
13     # write header
14     f.write("Index;Time [s];\n")
15     for i, res in enumerate(results, 1):
16         # check for valid status code
17         if res.status_code != 200:
18             raise Exception(f"Statuscode: {res.status_code}")
19
20         # write results to file
21         f.write(";" .join([str(i), str(res.elapsed.total_seconds()), "\n"]))
```

Listing 5: Python test Skript

Tagesplan (EDP Issue Management System)									
Displaying 144 issues at 02/Mar/22 9:20 AM									
Test Environment	Status	Testing Remarks	Key	Sample ID	Test Case	Test Run ID	Test Class	Labels	Summary
AB-Werk1-14	Ready for Test	37.08n in AB starten	C3003932_4858	PV_3932_1886	TCSY128	3042SY21_008602	PV	Au26a_CC_Profil_anpassen_HOD_heat_HW089_Mainstream_PV_18, RBS 1.0, SW_0022_skript_nutzen	C3003932_4325 RUN TCSY128 L_03_Life_time_test retest - Fortsetzung AB
AB-Werk1-14	Ready for Test	27.08n in AB starten	C3003932_4859	PV_3932_1890	TCSY128	3042SY21_008693	PV	Au26a_CC_Profil_anpassen_HOD_heat_HW089_Mainstream_PV_18, RBS 1.0, SW_0022_skript_nutzen	C3003932_4325 RUN TCSY128 L_03_Life_time_test retest - Fortsetzung AB
AB-Werk1-14	Ready for Test	27.08n in AB starten	C3003932_4860	PV_3932_1956	TCSY128	3042SY21_008688	PV	Au26a_CC_Profil_anpassen_HOD_only_HW089_Mainstream_PV_18, RBS 1.0, SW_0022_skript_nutzen	C3003932_4325 RUN TCSY128 L_03_Life_time_test retest - Fortsetzung AB
AB-Werk1-14	Ready for Test	27.08n in AB starten	C3003932_4861	PV_3932_1957	TCSY128	3042SY21_008689	PV	Au26a_CC_Profil_anpassen_HOD_only_HW089_Mainstream_PV_18, RBS 1.0, SW_0022_skript_nutzen	C3003932_4325 RUN TCSY128 L_03_Life_time_test retest - Fortsetzung AB
AB-Werk1-14	Ready for Test	27.08n in AB starten	C3003932_4862	PV_3932_1959	TCSY128	3042SY21_008690	PV	Au26a_CC_Profil_anpassen_HOD_only_HODbox10_HW089_PV_18, RBS03.3_RBS1.0_SW_0022_Sport_skript_nutzen	C3003932_4325 RUN TCSY128 L_03_Life_time_test retest - Fortsetzung AB
AB-Werk1-14	Ready for Test	Ende 10.02.22 - 10:55 Uhr	C3003932_4857	PV_3932_1811	TCSY128	3042SY21_008691	PV	Au26a_CC_Profil_anpassen_HOD_heat_HW089_PV_18, RBS 1.0, SW_0022_Sport_skript_nutzen	C3003932_4325 RUN TCSY128 L_03_Life_time_test retest - Fortsetzung AB
A_Dust_chamber	Queued	000 - LEERZEILE	C3003436_16756	XXXXXXXXXXXXXX	TCSET			A_Dust_chamber	C3003436_12550 Leerzeile A_Dust_chamber
A_KPK100	Open	000 - LEERZEILE - Ehemaals 1215	C3003436_14911	XXXXXXXXXXXXXX	TCSET			A_KPK1215	C3003436_12550 Leerzeile A_KPK1215
A_KPK100	Ready for Test	02 03 22 - starten	C1005862_445	PV_5862_5159	TCSY145	3042SY22_001601	PV	Almemo_Au027_HOD_HW089_PV_28_Programm_L_RBS 1.10, SW_0023_X5_Sport_heat	C1005862_399 RUN TCSY145 TG_A_PRE_Parameter Test (large)
A_KPK100	Ready for Test	02 03 22 - starten	C1005862_446	PV_5862_5160	TCSY145	3042SY22_001602	PV	Almemo_Au027_HOD_HW089_PV_28_Programm_L_RBS 1.10, SW_0023_X5_Sport_heat	C1005862_399 RUN TCSY145 TG_A_PRE_Parameter Test (large)
A_KPK100	Ready for Test	02 03 22 - starten	C1005862_447	PV_5862_5161	TCSY145	3042SY22_001603	PV	Almemo_Au027_HOD_HW089_PV_28_Programm_L_RBS 1.10, SW_0023_X5_Sport_heat	C1005862_399 RUN TCSY145 TG_A_PRE_Parameter Test (large)
A_KPK100	Ready for Test	03 03 22 - starten	C1005862_451	PV_5862_5159	TCSY61	3042SY22_001607	PV	Almemo_Au027_HOD_HW089_PV_28_Programm_A_RBS 1.10, SW_0023_X5_Sport_heat	C1005862_400 RUN TCSY61 TG_A_G1_G5_Fundamental_test_in_series
A_KPK100	Ready for Test	03 03 22 - starten	C1005862_452	PV_5862_5160	TCSY61	3042SY22_001608	PV	Almemo_Au027_HOD_HW089_PV_28_Programm_A_RBS 1.10, SW_0023_X5_Sport_heat	C1005862_400 RUN TCSY61 TG_A_G1_G5_Fundamental_test_in_series
A_KPK100	Ready for Test	03 03 22 - starten	C1005862_453	PV_5862_5161	TCSY61	3042SY22_001609	PV	Almemo_Au027_HOD_HW089_PV_28_Programm_A_RBS 1.10, SW_0023_X5_Sport_heat	C1005862_400 RUN TCSY61 TG_A_G1_G5_Fundamental_test_in_series
A_KPK100	Ready for Test	10 03 22 - starten	C1005862_457	PV_5862_5159	TCSY145	3042SY22_001613	PV	Almemo_Au027_HOD_HW089_PV_28_Programm_L_RBS 1.10, SW_0023_X5_Sport_heat	C1005862_401 RUN TCSY145 TG_A_POST_Parameter Test (large)
A_KPK100	Ready for Test	10 03 22 - starten	C1005862_458	PV_5862_5160	TCSY145	3042SY22_001614	PV	Almemo_Au027_HOD_HW089_PV_28_Programm_L_RBS 1.10, SW_0023_X5_Sport_heat	C1005862_401 RUN TCSY145 TG_A_POST_Parameter Test (large)
A_KPK100	Ready for Test	10 03 22 - starten	C1005862_459	PV_5862_5161	TCSY145	3042SY22_001615	PV	Almemo_Au027_HOD_HW089_PV_28_Programm_L_RBS 1.10, SW_0023_X5_Sport_heat	C1005862_401 RUN TCSY145 TG_A_POST_Parameter Test (large)
A_KPK100	In Execution	Ende 15.02.22 - ca. 13 Uhr	C3004533_57	DV_4533_0004	TCSY507	3042SY22_000425	DV	HOD+HEAT_HW089_SW 0003	C3004533_19 RUN TCSY507 high_temperature_storage_504h

Abbildung 9: Auszug eines Tagesplans vom 02.03.2022

Übersicht: Daten übertragen sich automatisch aus anderen Reitern hier nicht bearbeiten															
Standort	Klimakammer	Liter	Volume	maximale Musteranzahl	Rechner	Vectorbox / NI Case	Relaisbox	Klemmbrett	Netzteil	Netzteil 2	Almemo	Name	Betriebsbereit	Wartung	Defekt
Aucotem	KPK12156	300	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3	0	2
	KPK12151	1000	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3	1	0
	KPK6	600	3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A			
	KPK12112	300	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A			
	KPK100	650	3	N/A	N/A	N/A	RE302	N/A	5740	6500	N/A	N/A			
	KPK2	800	3	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A			
	KPK77	1000	4	N/A	N/A	R8313	9	5745	N/A	N/A					
	KPK103	1000	3	N/A	N/A	RB309	1	6598	6738	N/A					
	KPK104	1000	4	N/A	N/A	N/A	4	6595	6594	N/A					
JSS	CC06	600	3	D304TE147	HOD SysTest	RB309	7	6731	6737	N/A					
	CC07	600	3	D304TE142	HOD SysTest	N/A	11	1873	6728	N/A					
	CC08	600	3	D304TE143	HOD SysTest	RB301	22	5584	6724	N/A					
	CC09	600	3	D304TE141	HOD SysTest	N/A	30	6732	6724	N/A					
	CC10	600	3	D304TE149	N/A	RB310	17	6725	N/A	N/A					
	CC11	600	3	D304TE151	HOD SysTest	RB324	18	6726	6735	Almemo 6					
	CC12	600	3	D304TE150	HOD SysTest	RB332	2	6596	5741	N/A					
	CC13	600	3	D304TE143	HOD SysTest	N/A	13	6727	6733	Almemo 2					
	CC14	600	3	D304TE130	HOD SysTest	N/A	24	5744	6735	N/A					
	CC15	600	3	D304TE033	HOD SysTest	R8316	16	5743	6739	N/A					
	CC16	600	3	D304TE144	HOD SysTest	RB363	N/A	6599	1886	N/A					
	CC17	600	3	D304TE032	HOD SysTest	N/A	14	2746	N/A	Almemo 3					

Abbildung 10: Übersichtsseite der Ressourcenliste vom 26.03.2022