



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	IV
Tabellenverzeichnis	V
Akronyme	VI
Glossar	VIII
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
3.5 Funktionale Anforderungen	22
3.6 Nichtfunktionale Anforderungen	24
4 Entwurf	26
4.1 Entwurf des GUI	26
4.1.1 Grunddesign	26
4.1.2 Entwurf einer kompakten Jira-Ticket Komponente	27
4.1.3 Dashboard	28
4.1.4 Ressourcendetailansicht	30
4.2 Entwurf der Softwarearchitektur	32
4.2.1 Systemarchitektur	32
4.2.2 Jira Ticket Modellierung	34
4.2.3 Komponentenarchitektur	38
5 Entwicklung	40
5.1 Projektaufbau	40
5.2 Implementierung des Frontends	41
5.2.1 Asynchrones Laden aktiver Jira-Tickets	41

5.3	Implementierung des Backends	43
5.3.1	Kategorisieren der Jira Labels	43
5.3.2	Jira Kommunikation	44
5.3.3	REST API Dokumentation	47
6	Literaturverzeichnis	48
7	Eigenständigkeitserklärung	49
8	Anhang	50

Abbildungsverzeichnis

1	Jira Ticket	2
2	HOD Lenkräder in einer Klimakammer	6
3	Diagramm des Beginns einer TCP-Verbindung [Wik10]	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	Finales Kartendesign	27
10	Kompakte Jira-Ticket Komponente	28
11	MockUp des Dashboards	29
12	MockUp des Modals zur anzeigen von Jira-Ticket Details	30
13	MockUp der Ansicht einer spezifischen Ressource	31
14	UML-Komponentendiagramm zur Systemarchitektur	32
15	UML-Sequenzdiagramm zur Anfrage der aktiven Tickets bei Initialisierung	34
16	UML-Klassendiagramm zur Jira-Ticket Struktur	35
17	UML-Klassendiagramm zur Jira-Ticket Struktur	38
18	Auszug eines Tagesplans vom 02.03.2022	52
19	Übersichtsseite der Ressourcenliste vom 26.03.2022	52

Tabellenverzeichnis

1	REST API HTTP Anfragetypen	9
2	HTTP Statuskategorien	10
3	Ø Antwortzeit zwischen Go und Python Webservers	13
4	Aufschlüsselung des regulären Ausdrucks zum Identifizieren einer Erdungskabelstruktur in einem String	44

Akronyme

API Application Programming Interface. 9, 12, 13, 15, 32, 40, *Glossar:* API

CSS Cascading Style Sheets. 32, 38, *Glossar:* JQL

DOM Document Object Model. 33, *Glossar:* DOM

ECU Electronic Control Unit. 6

GUI Graphical User Interface. 32

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

HTML Hypertext Markup Language. 7, 16, 28, 32, 38, 39, *Glossar:* HTML

HTTP Hypertext Transfer Protocol. 8, 15, 33, 44

JQL Jira Query Language. 17, 46, *Glossar:* JQL

JSON Javascript Object Notation. 7, 14, 34–37, *Glossar:* JSON

JSS Joyson Safety Systems. 3, 16, 44

MVC Model View Controller. 38

MVP Minimal Viable Product. 4, 23, *Glossar:* MVP

PDF Portable Document Format. 1, 16

REST Representational State Transfer. 9, 12, 13, 15, 32, 40, *Glossar:* REST

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

TCP Transmission Control Protocol. 10

UI User Interface. 4, 25, 26

URI Uniform Resource Identifier. 7, 42, *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, 26, 32, 40

DOM ein Interface um die Struktur, den Inhalt oder das Aussehen einer Webseite anzupassen. 33

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, 40

Garbage Collector eine automatische Speicherverwaltung. 12

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

HTML eine Sprache zum strukturellen Aufbau eines elektronischen Dokuments (z.B. eine Webseite). 7

JIRA Jira (entwickelt von Atlassian) ist eine Webanwendung zur Fehlerverwaltung, Problembehandlung und zum operativen Projektmanagement. 9, 33

JQL eine Sprache zum filtern der Jira Datenbank. 17, 32

JSON 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**]. 4

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

threadsicher eine Softwarekomponente, welche das gleichzeitige Bearbeiten durch mehrere Programmteile ermöglicht, ohne dass diese sich gegenseitig behindern, nennt man threadsicher. 46

URI ein Identifikator, welcher zur Bezeichnung von Webseiten etc. verwendet wird (z.B. *www.bht-berlin.de*). 7

Codebeispiele

1	Beispielhafte HTTP Anfrage	11
2	Beispielhafte HTTP Antwort	11
3	Go Strukturdefinition mit JSON Tags	36
4	gekürzte beispielhafte JSON Repräsentation eines Jira-Tickets	36
5	TypeScript Funktion zum asynchronen Laden aktiver Tickets (<i>frontened/src/ts/dashboard.ts</i>)	41
6	Go Umsetzung von asynchronen HTTP Anfragen (<i>backend/api/api_jira.go</i>)	45
7	Python Flask Webserver	50
8	Go Webserver	50
9	Python test Skript	51

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 Joyston 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 <small>(View Workflow)</small>
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 in den Labels des Tickets 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 Webbrowsersn 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.

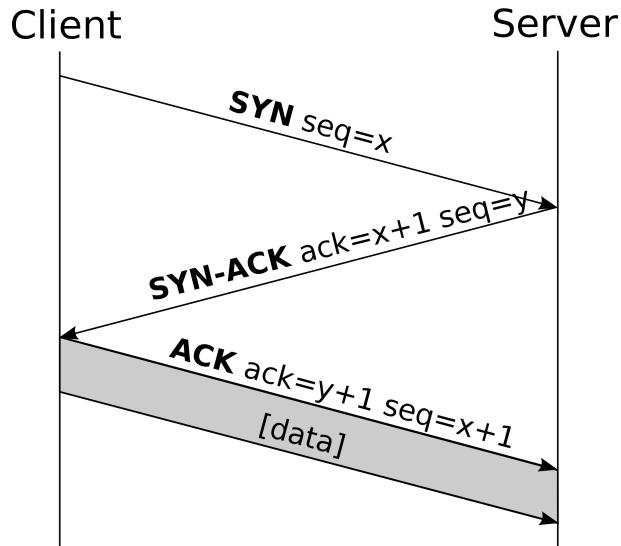


Abbildung 3: Diagramm des Beginns einer TCP-Verbindung [Wik10]

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 2.3.2 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
```

Codebeispiel 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)
```

Codebeispiel 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 [Gou+02].

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

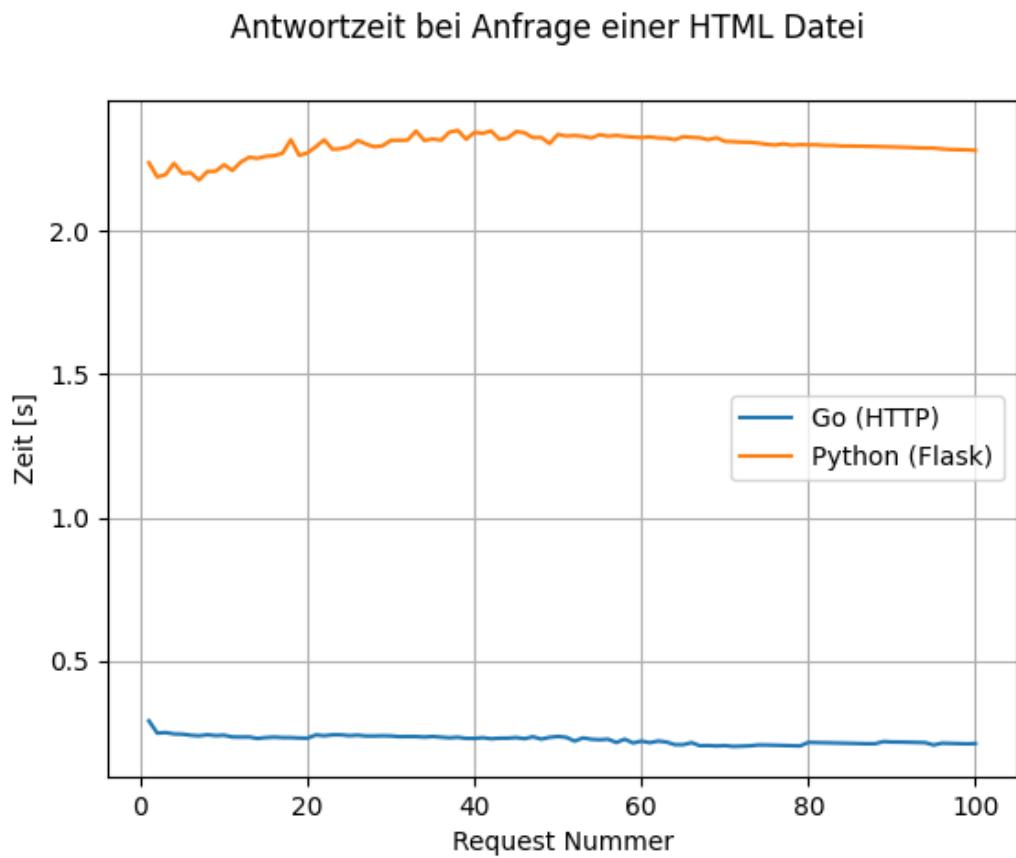


Abbildung 4: Geschwindigkeitsvergleich eines Webservers: Python vs. Go

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 18). 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 19). 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:

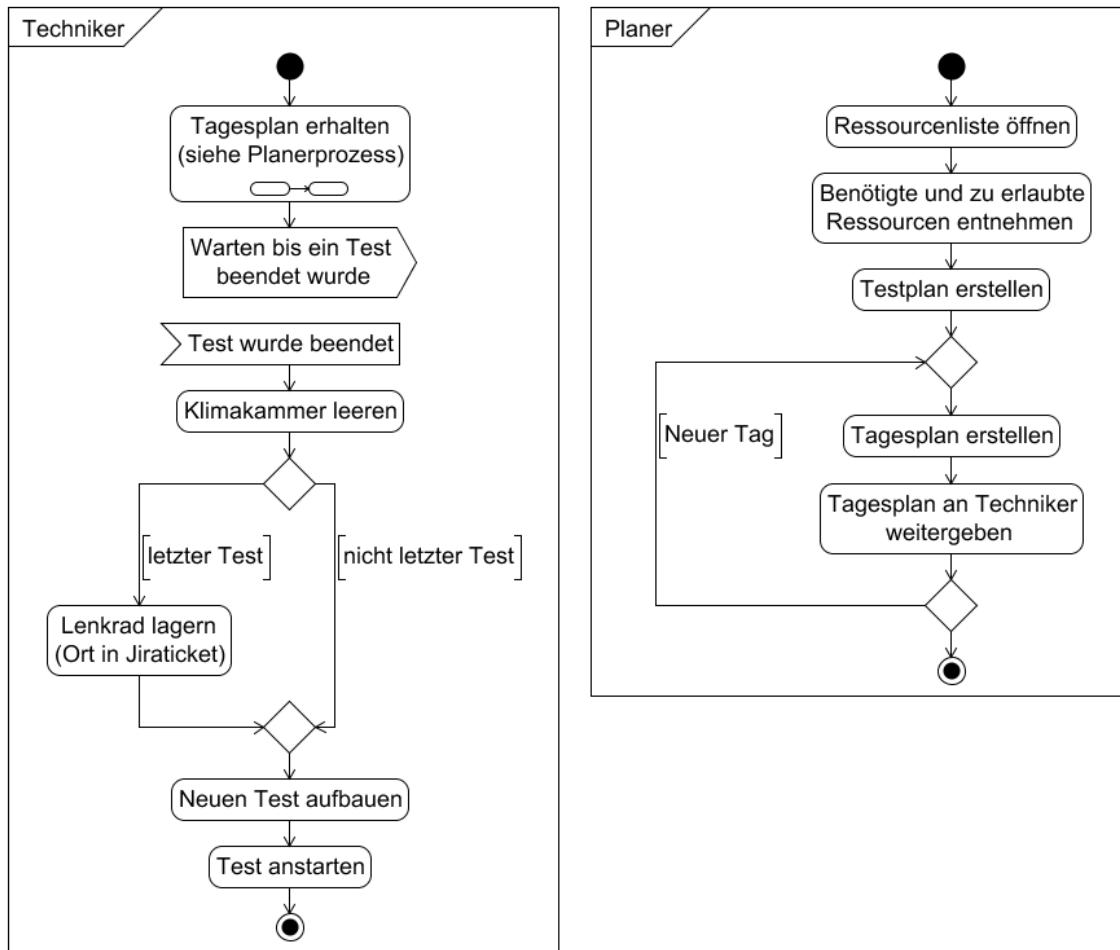


Abbildung 5: UML Aktivitätsdiagramm zum täglichen Testprozess

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.

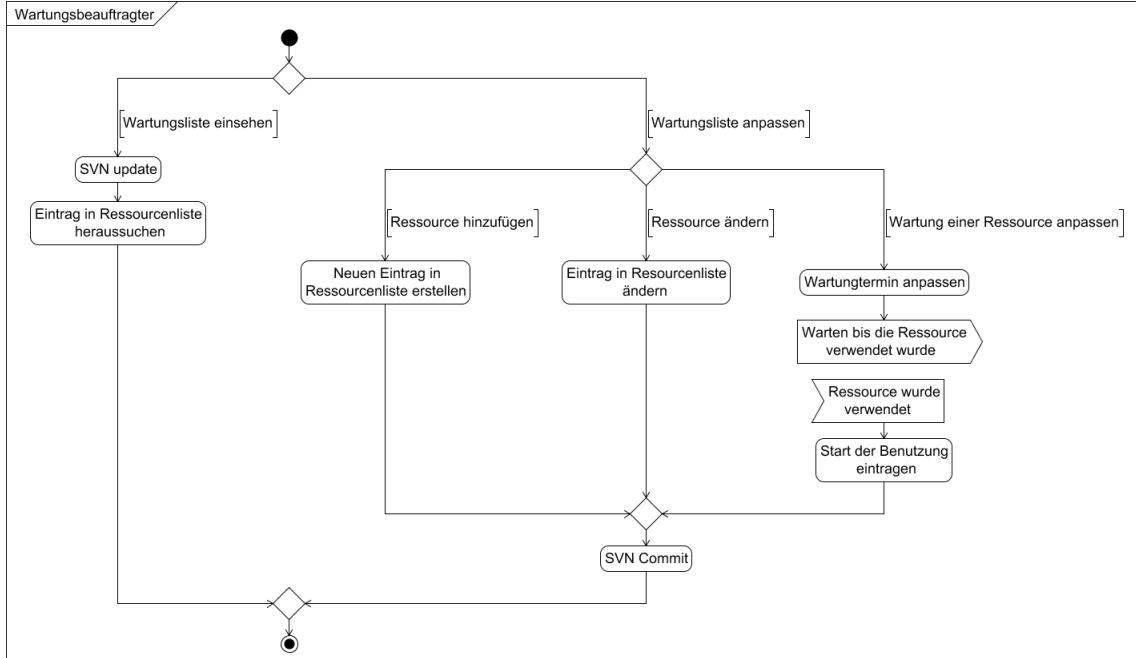


Abbildung 6: UML Aktivitätsdiagramm zum Ressourcenmanagementprozess des Wartungsbeauftragten

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.

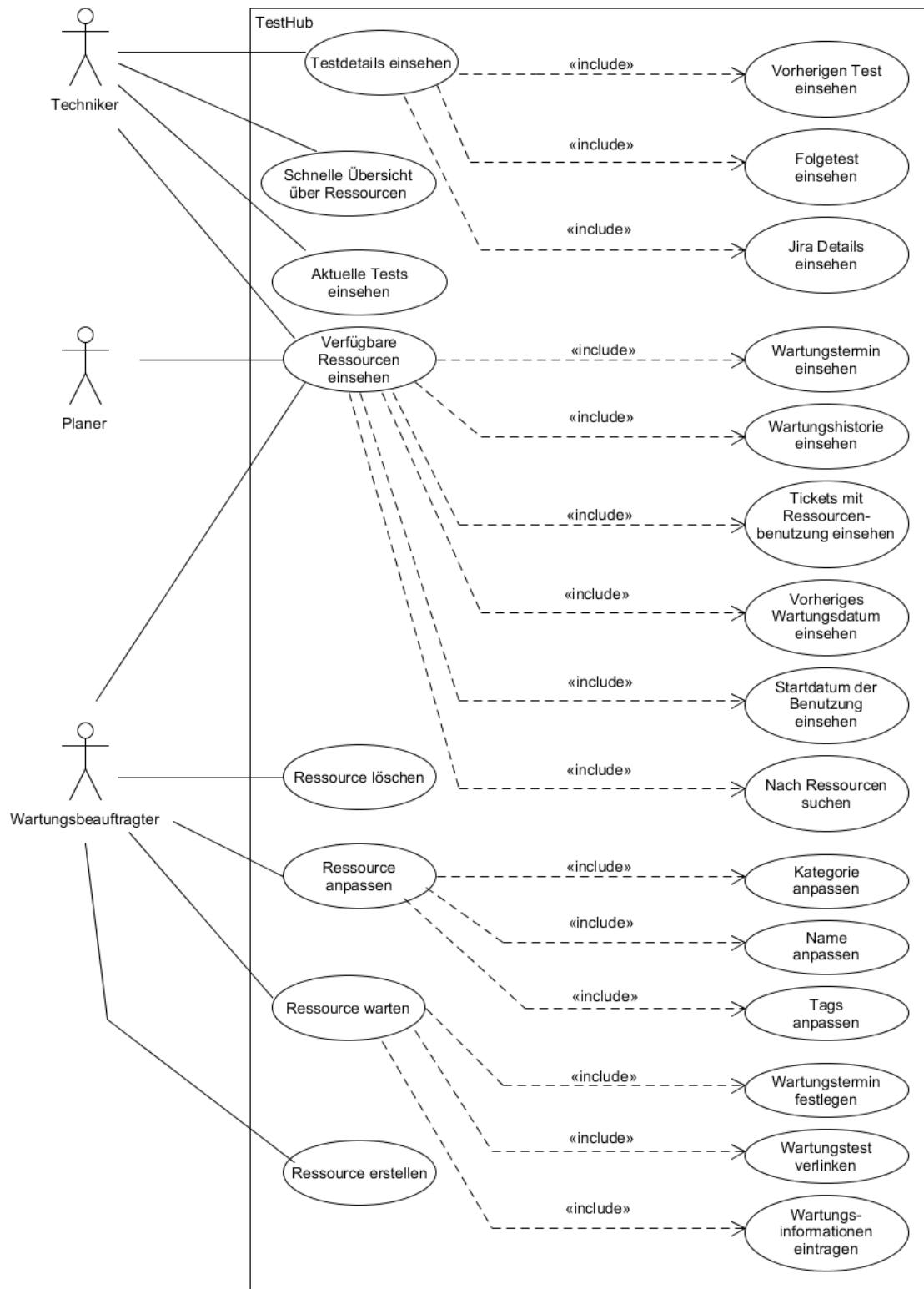


Abbildung 7: UML-Use-Case-Diagramm zu den Anwendungsfällen für die Planer-, Techniker- und Wartungsbeauftragten-Rolle

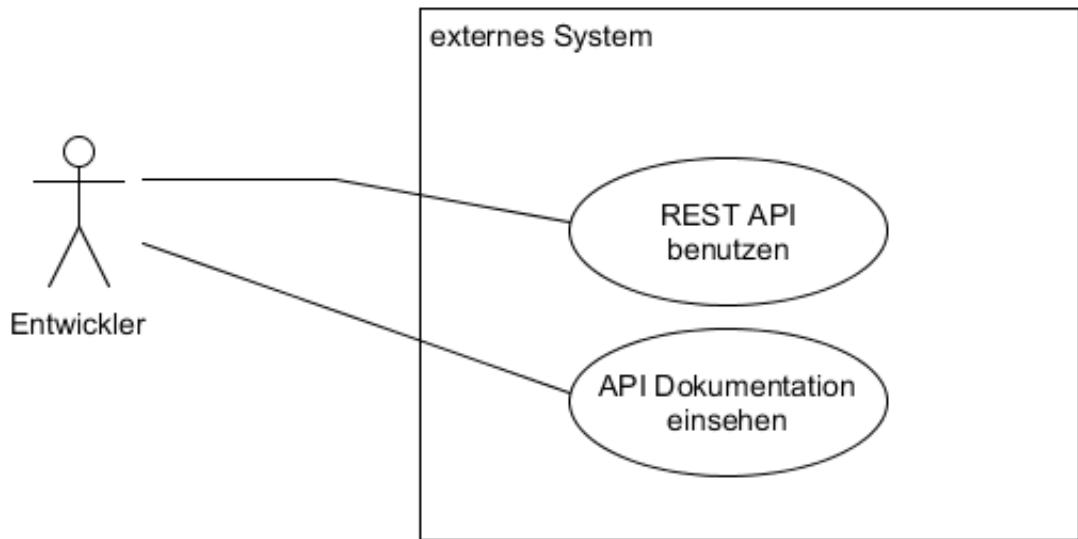


Abbildung 8: UML-Use-Case-Diagramm zu den Anwendungsfällen für die Entwickler-Rolle

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 Jira-Tickets in aufgeschlüsselter Form einsehen*

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

US#05 *Als Techniker möchte ich eine schnelle Übersicht über die wichtigsten Ressourcen erhalten, um den Aufbau von Tests zu planen*

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*

3.5 Funktionale Anforderungen

Um die Funktion der im Umfang dieser Arbeit entwickelten Software validieren zu können, müssen Anforderungen definiert werden. Diese Funktionalen Anforderungen werden in drei Kategorien eingeteilt:

Muss-Kriterien Die Kriterien die “TestHub” als MVP zwingend erfüllen muss

Soll-Kriterien Die Kriterien die für die Funktionalität von “TestHub” nicht zwingend notwendig sind, aber dennoch implementiert werden sollten.

Kann-Kriterien Die Kriterien, die rein optional sind. Da es sich bei “TestHub” um ein MVP handelt, werden diese Kriterien nicht berücksichtigt.

Die folgenden Anforderungen ergeben sich aus den User Stories und den in Abschnitt 3.3 aufgezeigten Use Cases:

FA#01 *TestHub muss den vorherigen und folgenden Test eines aktuell ausgeführten Tests anzeigen können (US#01 & US#02).*

Ein Test kann vorherige und folgende Tests haben, es gibt jedoch auch Tests, bei denen dies nicht der Fall ist.

FA#02 *TestHub muss die Informationen eines Jira-Tickets in übersichtlicher Form anzeigen (US#03).*

FA#03 *TestHub muss die Labels eines Jira-Tickets kategorisiert anzeigen (US#03).*
Die Labels eines Jira-Tickets liegen als simple Liste von Text vor. Dieser Text soll analysiert und entsprechend kategorisiert werden, um eine genauere Zuordnung zu ermöglichen.

FA#04 *TestHub muss alle aktiven Tests nach Klimakammer gruppiert anzeigen (US#04).*

Ein Test gilt als aktiv, wenn der Status seines Jira-Tickets “Read for Test” oder “In Execution” ist.

FA#05 *TestHub muss eine schnelle Übersicht über die wichtigsten Ressourcen anzeigen (US#05).*

Die wichtigsten Ressourcen sind: aktive Tests, Ressourcen mit einem Wartungstermin in den nächsten 30 Tagen.

FA#05 *TestHub muss den nächsten Wartungstermin einer Ressource anzeigen (US#10).*

FA#06 *TestHub muss die Wartungshistorie einer Ressource anzeigen (US#11).*

Die Wartungshistorie enthält alle Änderungen des Wartungstermins.

FA#07 *TestHub muss alle aktiven Tickets anzeigen, welche eine spezifische Ressource verwenden (US#12).*

FA#08 *TestHub muss das vorherige Wartungsdatum anzeigen (US#13 & US#11).*

FA#09 *TestHub muss das Startdatum der Verwendung einer Ressource anzeigen (US#14).*

Das Startdatum ist das Datum des Jira-Tickets welches nach dem Wartungstermin der Ressource aktualisiert wurde und die Ressource verwendet.

FA#10 *TestHub muss die Suche nach Ressourcen unterstützen (US#15).*

FA#11 *TestHub muss das Löschen einer Ressource unterstützen (US#20).*

FA#12 *TestHub muss das Anpassen der Kategorie einer Ressource unterstützen (US#21).*

FA#13 *TestHub muss das Anpassen des Namens einer Ressource unterstützen (US#22).*

FA#14 *TestHub muss das Anpassen der Tags einer Ressource unterstützen (US#23).*

FA#15 *TestHub muss Festlegen eines neuen Wartungstermins unterstützen (US#24).*

FA#16 *TestHub muss das Verlinken eines Wartungstests unterstützen (US#25).*

FA#17 *TestHub muss das Speichern weiterer wartungsspezifischer Informationen unterstützen (US#26).*

FA#18 *TestHub muss das Erstellen einer Ressource unterstützen (US#23).*

Die Ressource benötigt mindestens einen Namen und sie muss nicht zwangsläufig einen Wartungstermin haben.

3.6 Nichtfunktionale Anforderungen

Es ergeben sich folgende nichtfunktionale Anforderungen aus den Interviews und den User Stories:

NFA#01 *Die REST API von TestHub muss innerhalb der Firma verwendbar sein (US#90).*

NFA#02 *Die REST API von TestHub soll dokumentiert und erklärt werden (**US#91**).*

Es soll eine Website oder ein Dokument geben, in dem alle Endpunkte der REST API aufgelistet und erklärt sind. Des Weiteren soll es Beispiele zum Verständnis geben.

NFA#03 *TestHub soll die Verteilung der Jira-Tickets auf die einzelnen Projekte aufzeigen (**US#05**).*

NFA#04 *Das UI von TestHub soll ein einheitliches Design haben.*

NFA#05 *TestHub muss in englischer Sprache verfügbar sein, da dies die Firmensprache von JSS ist.*

NFA#06 *TestHub muss unabhängig von den Jira Benutzerrechten für das HOD Projekt funktionieren*

Da das System eine schnelle Übersicht schaffen soll und auch keine Möglichkeit bietet, Jira Daten anzupassen, soll das System die Jira Informationen unabhängig der Jira Rechte anzeigen.

NFA#07 *TestHub muss mindestens auf den Webbrowsern “Chrome” und “Edge” funktionieren*

Diese zwei Webbrower werden innerhalb der Firma verwendet.

4 Entwurf

Im folgenden Kapitel soll der Prozess der Gestaltung des UIs erläutert werden. Des Weiteren wird die Architektur des Backends und der Komponenten genauer beschrieben.

4.1 Entwurf des GUI

Im folgenden Abschnitt wird das Grundlegende design kurz erläutert. Anschließend wird auf den Aufbau und Entwurf der 2 Hauptseiten eingegangen.

Da “TestHub” im Browser auf verschiedenen Bildschirmen laufen wird, muss das Design auch eine gewisse “Responsiveness” haben, was bedeutet, dass sich die GUI dem Viewport anpasst.

4.1.1 Grunddesign

Um das in **NFA#04** angesprochene einheitliche Design umzusetzen, wurde ein simples Design erstellt, welches sich im gesamten Projekt wiederfinden lässt. Das Design basiert auf “Karten” welche sich leicht entwerfen lassen und zudem übersichtlich und skalierbar sind.



Abbildung 9: Finales Kartendesign

Das Kartendesign wurde mit dem Designtool Figma erstellt. Jede Karte besitzt einen Header welcher die Hintergrundfarbe `#1f2937` besitzt. In diesem Header lassen sich der Karten Titel und ein Button wiederfinden. Der Titel der Karte sollte dabei immer eine kurze Beschreibung des Inhalts sein. Der Button in der rechten Ecke des Headers ist optional und kann beliebig angepasst werden. Der Body der Karte lässt Platz für alle beliebigen Elemente. Durch das simple Design sind die Höhe und Breite der Karte variabel, wodurch jedes beliebige Element im Body Platz findet.

4.1.2 Entwurf einer kompakten Jira-Ticket Komponente

Da es Sinn macht, vorherige und folgende Tests zusammen mit dem aktiven Test anzuzeigen, wurde ein Element entworfen, welches die wichtigsten Informationen des Tickets in kompakter Weise anzeigt und trotzdem die Möglichkeit bietet, den vorherigen

und folgenden Test einzusehen. Durch die Verwendung von TailwindCSS konnte die Komponente schnell und einfach direkt in HTML entworfen werden. Dadurch lässt sie sich auch leichter in das fertige Projekt einbauen.

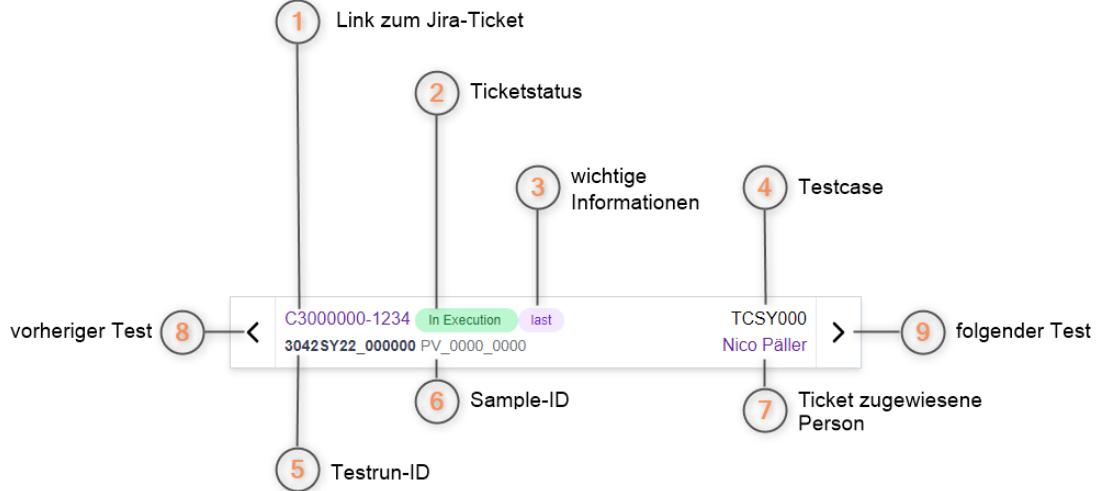


Abbildung 10: Kompakte Jira-Ticket Komponente

In dem Design dieser Komponente lassen sich die wichtigsten Ticket Informationen auf einen Blick erkennen, wie in **FA#02** gefordert. Falls man dennoch das ganze Jira-Ticket ansehen möchte, kann dies über die Verlinkung der Ticket-ID geschehen. Des Weiteren sieht man in dieser Komponente auch schnell, falls ein Test der letzte Test ist (Abbildung 10 Nummer 3). Klickt man auf eine freie Stelle innerhalb der Komponente, soll sich die Detailansicht des Tickets öffnen, klickt man auf den Knopf des vorherigen oder folgenden Tests soll sich die Detailansicht des entsprechenden Tickets öffnen. Weiterhin lässt sich durch die geringe Höhe der Komponente eine kompakte Auflistung mehrerer Tickets ermöglichen.

4.1.3 Dashboard

Um die in **FA#05** geforderte schnelle Übersicht zu entwerfen, wurde zuerst analysiert, welche Daten angezeigt werden müssen. Wie in der Anforderung beschrieben müssen sowohl die aktiven Jira-Tickets als auch aufkommende Wartungen angezeigt werden. All diese Informationen sollen auf dem “Dashboard” übersichtlich dargestellt werden. Das Dashboard soll gleichzeitig die Hauptseite des Projekts sein.

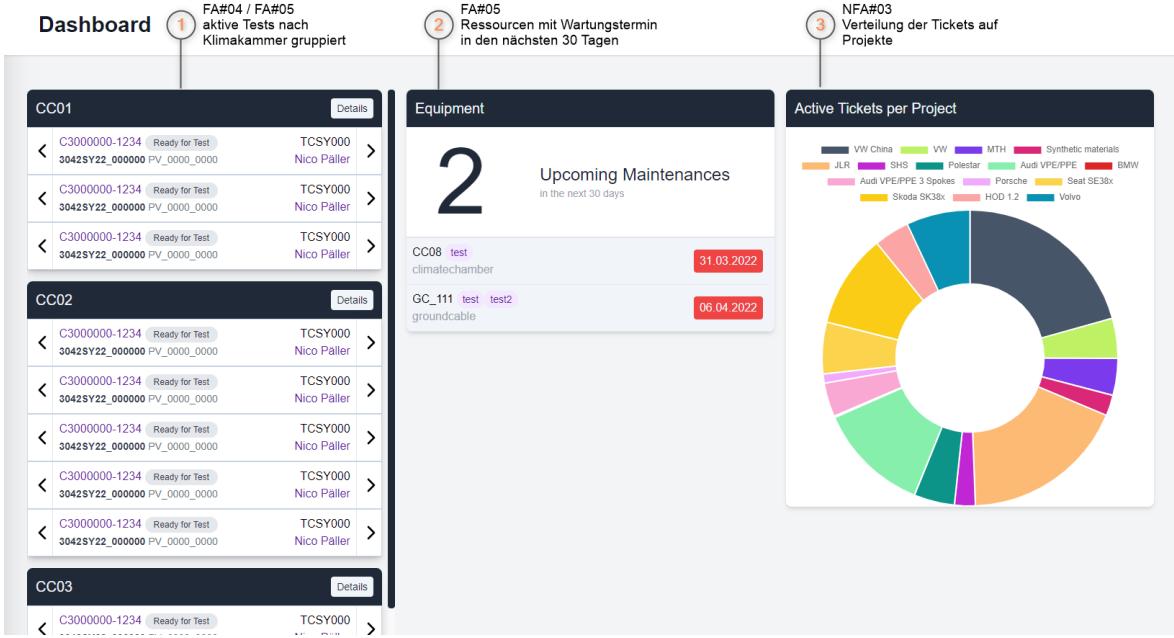


Abbildung 11: MockUp des Dashboards

In Abbildung 11 ist ein Mockup des Dashboards zu sehen. Die Entwicklung des Dashboard folgte einem iterativen Prozess. Das Mockup wurde den betroffenen Personen vorgestellt und in mehreren Schritten verfeinert und die Rückmeldungen der Befragten implementiert.

Das Dashboard benutzt das in Abschnitt 4.1.1 gezeigte Kartendesign. Dabei wird das Dashboard in drei Spalten aufgeteilt, welche die aktiven Tests nach Klimakammer gruppiert (11, Nummer 1), die aufkommenden Wartungen (11, Nummer 2) und eine Projektübersicht (11, Nummer 3) bietet. In den Klimakammerkarten lässt sich auch die in Abschnitt 4.1.2 gezeigte kompakte Jira-Ticket Komponente wiederfinden.

Da es für die Jira-Tickets viele weitere interessante Informationen gibt, welche jedoch sich nicht in einer kompakten Ansicht anzeigen lassen, gibt es für jedes aktive, vorheriges und folgendes Ticket eine Detailansicht. Damit wird sowohl die funktionale Anforderung **FA#01** als auch **FA#03** erfüllen. Die Detailansicht wird durch ein Modal umgesetzt. Ein Modal ist ein Dialog, welcher der rest der Anwendung sperrt, bis das Modal wieder geschlossen wird.

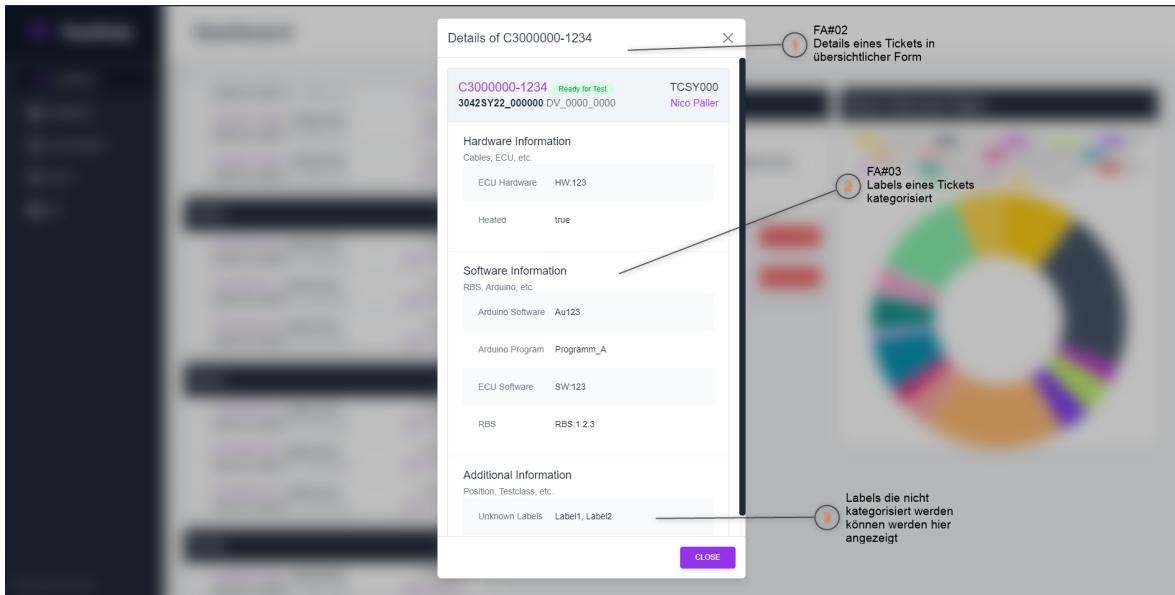


Abbildung 12: MockUp des Modals zur anzeigen von Jira-Ticket Details

In Abbildung 12 werden nur einige der Labelkategorien angezeigt. Falls es nicht möglich ist, ein Label zu kategorisieren, wird dieses ganz unten unter "Unknown Labels" angezeigt.

4.1.4 Ressourcendetailansicht

Um die Details einer Ressource einzusehen und diese zu bearbeiten muss es eine extra dafür angefertigte Seite geben.

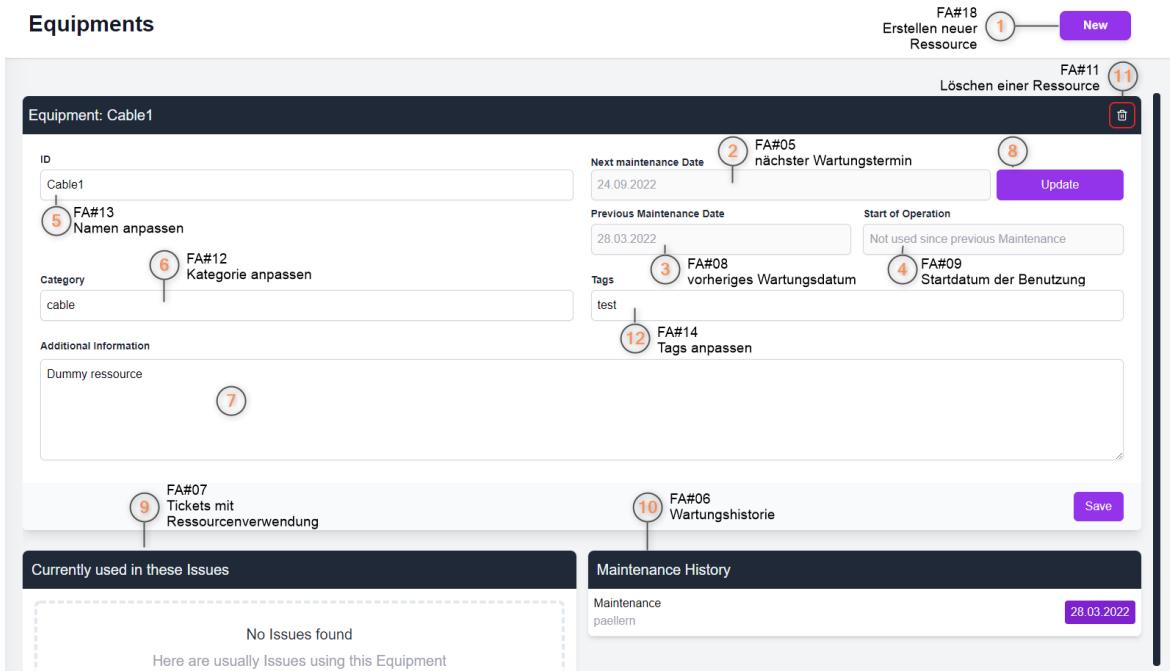


Abbildung 13: MockUp der Ansicht einer spezifischen Ressource

Auch in dieser Ansicht lässt sich das Design aus Abschnitt 4.1.1 wiederfinden. Das MockUp der Ansicht einer spezifischen Ressource liefert alle wichtigen Informationen auf einen Blick. Viele dieser Informationen lassen sich auch interaktiv anpassen und mit dem “Save” Button übernehmen. Die Kategorien und Tags zeigen bei Benutzung des jeweiligen Eingabefeldes, die schon vorhandenen Kategorien oder Tags des Servers an, um die Eingabe so bequem wie möglich zu machen.

Unter der Equipment Karte, befinden sich noch zwei weitere Karten. Die linke Karte zeigt dabei die in **FA#07** geforderten Tickets, welche die entsprechende Ressource verwenden. In Abbildung 13 sieht man die Ansicht, wie sie aussieht, wenn es keine solcher Tickets gibt. Wenn es doch Tickets gibt, wird die Klimakammeransicht aus Abbildung 11 Nummer 1 erneut verwendet, um die Tickets anzuzeigen.

Mit dem “Update” Button kann ein neuer Wartungstermin erstellt werden. Es öffnet sich erneut ein Modal wo die in **FA#15**, **FA#16** und **FA#17** geforderten wartungs-spezifischen Informationen eingetragen und abgespeichert werden können.

4.2 Entwurf der Softwarearchitektur

Im Kapitel 4.2 wird die Architektur der Software detailliert erläutert und

4.2.1 Systemarchitektur

Das System ist in dem klassischen Server-Client-Prinzip aufgebaut, in dem der Server, als Backend, die nötigen Daten zur Verfügung stellt und der Client diese abfragen kann um seine spezifischen Aufgaben zu lösen [NF13]. In diesem Fall ist der Client der Webbrower mit dem der Benutzer interagiert. Die Server-Komponente, stellt die Daten aus der Datenbank oder dem JiraServer zur Verfügung. Der Server dient auch als Webserver, indem er ebenfalls die HTML, CSS und Javascript Dokumente zur Verfügung stellt.

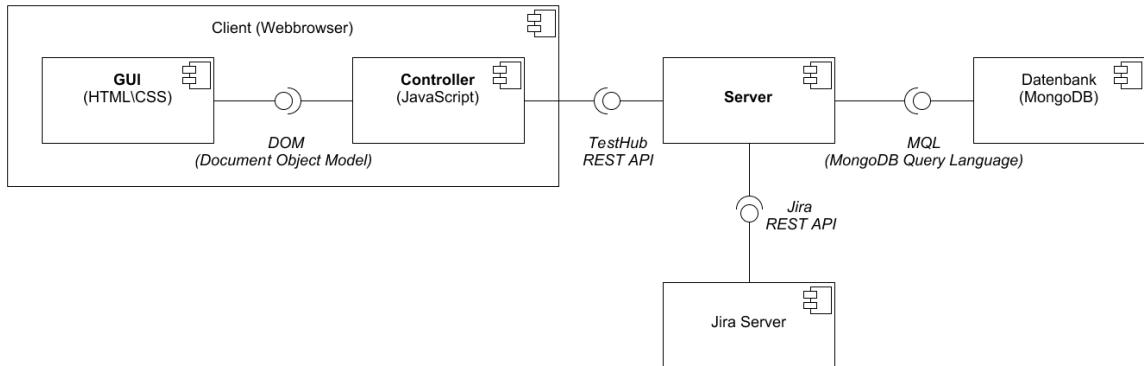


Abbildung 14: UML-Komponentendiagramm zur Systemarchitektur

In Abbildung 14 sind die genannten Komponenten visuell dargestellt. Die dick geschriebenen Komponenten sind in dieser Arbeit entwickelt worden, alle anderen werden nur als Tool genutzt, um Daten zu erhalten oder Interaktionen zu ermöglichen. Dadurch ergeben sich 3 Hauptkomponenten

GUI

Die Grafische Benutzeroberfläche mit der der Benutzer interagieren kann. Die GUI wird durch strukturell durch HTML Dokumente ermöglicht. Das Design wird dabei durch das in Abschnitt ?? erwähnte TailwindCSS umgesetzt.

Server

Der Server bietet über eine REST API (siehe Abs. 2.2) die vom Client benötigten Daten an. Der Server kann dabei eine Vielzahl an Clients gleichzeitig bearbeiten.

Controller

Der Controller verarbeitet die Daten des Servers, sodass sie in der GUI dargestellt werden können. Er ist ein eigenes JavaScript Programm, welches vom Browser ausgeführt wird.

Zusätzlich gibt es zwei weitere Komponenten mit denen der Server interagiert:

Jira Server

Der Jira Server ist der Server auf dem das Programm JIRA ausgeführt wird, welches die Informationen zu allen Tests beinhaltet.

Datenbank

Die Datenbank dient als zweite Informationsquelle. In ihr speichert der Server die Daten der Ressourcen, welche üblicher Weise nicht durch JIRA verwaltet werden.

Die Kommunikation der Komponenten erfolgt auf unterschiedliche Weisen. Während der Controller mit der GUI über das vom Browser zur Verfügung gestellte DOM Interface interagiert, kommuniziert es mit dem Server via HTTP (siehe Abs.2.3). Der Server fragt seine Daten ebenfalls über HTTP vom Jira Server an. Um Einträge in der Datenbank zu speichern, anzupassen oder abzufragen wird die Query-Sprache der Datenbank verwendet.

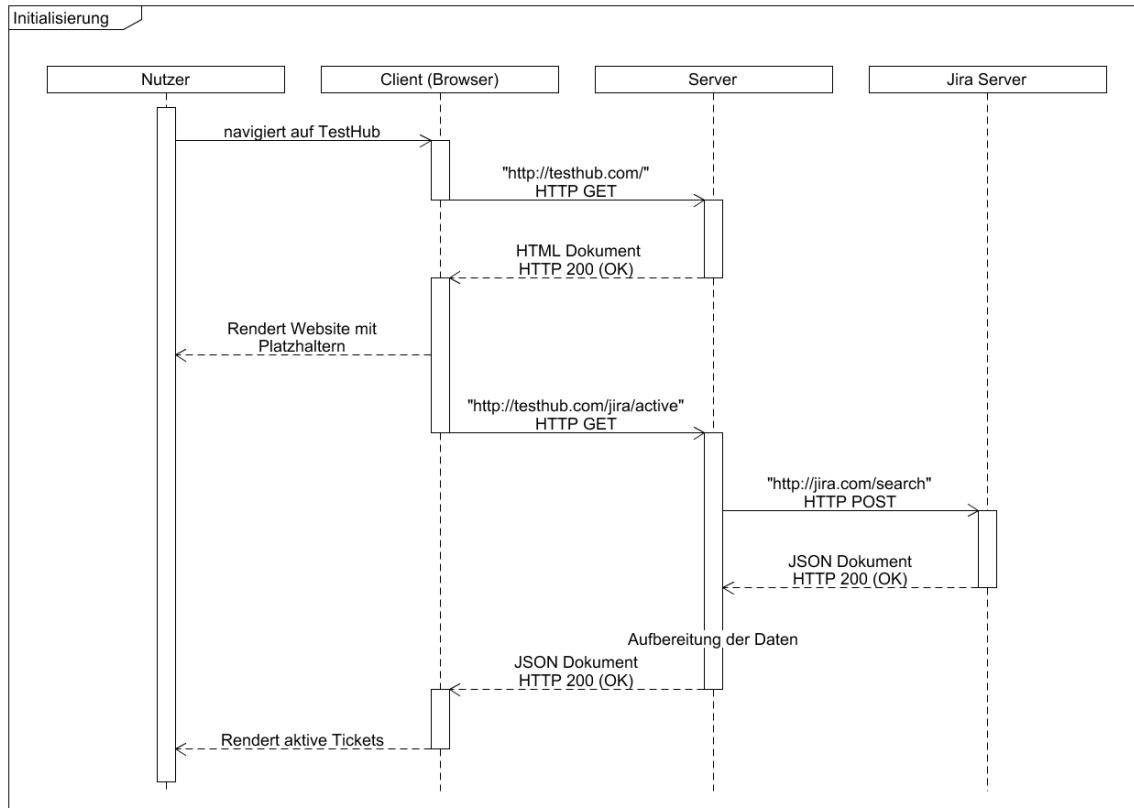


Abbildung 15: UML-Sequenzdiagramm zur Anfrage der aktiven Tickets bei Initialisierung

In Abbildung 15 ist exemplarhaft die Kommunikation beim Laden des Dashboards dargestellt. Da sich der Ablauf immer ähnelt, nur der rechte Teilnehmer, je nach Anfrage die Datenbank ist, kann man dieses Prinzip auf die Kommunikation des gesamten Systems beziehen. Der Server ist demnach nur eine Zwischenkomponente, welche die einheitlich, gesammelt und aufbereitet wiedergibt.

Aus Übersichtlichkeitsgründen wurde die GUI und der Controller als Client zusammengefasst.

4.2.2 Jira Ticket Modellierung

Ein JiraTicket ist im Backend wie folgt strukturiert. Die Struktur ist abgeleitet von dem JSON-Dokument, welches vom Jira Server empfangen wird. Der Jira Server sendet noch viele weitere Informationen zu einem Ticket, welche aufgrund der Relevanz nicht beachtet werden.

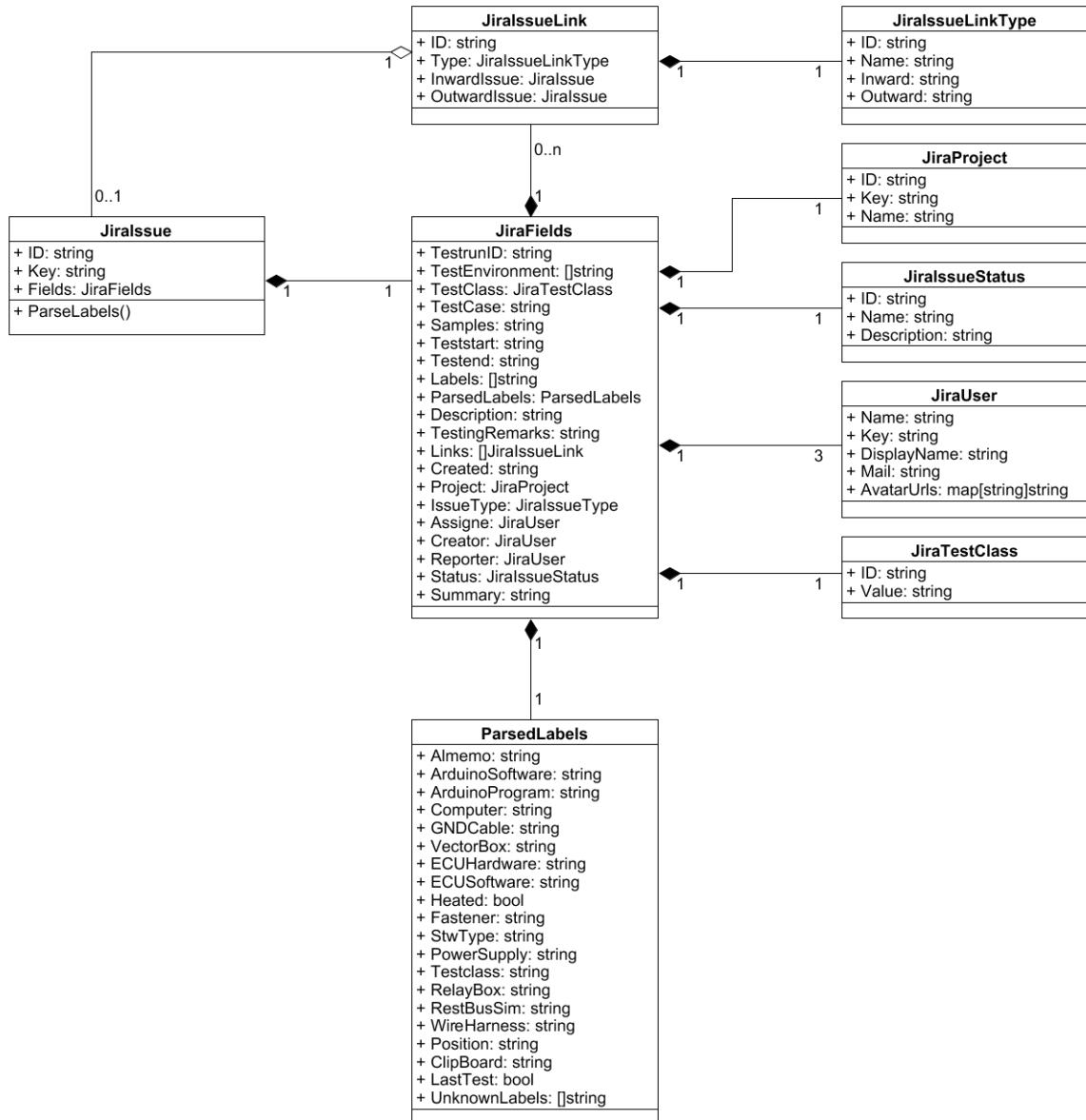


Abbildung 16: UML-Klassendiagramm zur Jira-Ticket Struktur

In Abbildung 17 ist zu sehen, dass ein JiraIssue hauptsächlich aus seinen “Field”-Attributten besteht. Diese decken sich größtenteils mit der JSON Repräsentation des Jira Tickets, sie wurde lediglich um die ParsedLabels Struktur erweitert, um die Labels in aufgeschlüsselter Form anzuzeigen (siehe FA#03). Dafür wird auch die öffentliche Methode “ParseLabels” eingeführt, welche das Parsen, also das Analysieren und Kategorisieren durchführt.

In den “Field”-Attributten lassen sich auch die Verlinkungen wiederfinden, falls es welche gibt. Diese Verlinkungen werden durch eine eigene Struktur realisiert. Dabei erhält

jeder Link eine ID, die Art der Verlinkung und einen Inward- *oder* einen Outward-Issue. “Inward” bedeutet in dem Fall, dass ein anderes Ticket auf dieses Ticket verlinkt, “Outward” beschreibt hingegen die ausgehende Verlinkung von diesem Ticket aus. Um riesige Baumstrukturen durch die Verlinkungen zu vermeiden, ist zwar der Inward- bzw. Outward-Issue ein Objekt der JiraIssue Klasse, jedoch werden in dem verlinken Ticket nur eine geringe Anzahl der Felder mit Werten befüllt. Des Weiteren hat das verlinke Ticket nie weitere Verlinkungen.

Die jeweilige Art der Verlinkung wird in der separaten “JiraIssueLinkType” Struktur festgehalten. Dort findet man auch die Beschreibung der Beziehung zwischen den Tickets in Worten. Durch diese Verlinkungsstruktur lassen sich die vorherigen und folgenden Tests, wie in **FA#01** gefordert, ermitteln.

Nicht nur für die Verlinkung gibt es extra Strukturen, auch der Status, das Projekt, die Testklasse und Accounts werden in eigenen Strukturen betrachtet, um weitere Informationen zu dem jeweiligen Feld zu liefern.

Das Parsen zu oder von der JSON Repräsentation kann in GO sehr leicht über sogenannte “Tags” realisieren. Somit kann man bei der Deklaration der Struktur direkt den JSON-Key mit angeben, welcher zur (De)Serialisierung verwendet wird.

```

1 type JiraIssue struct {
2     ID      string      `json:"id"`
3     Key     string      `json:"key"`
4     Fields JiraFields `json:"fields"`
5 }
```

Codebeispiel 3: Go Strukturdefinition mit JSON Tags

Ein in JSON serialisiertes Jira-Ticket, beispielsweise im Body einer Antwort des Servers, sieht wie folgt aus:

```

1 {
2     "id": "12345",
3     "key": "C3000000-1234",
4     "fields": {
5         "labels": [
6             "Au123",
7             "HW:123",
8             "RBS:1.2.3",
9             "SW:321",
```

```

10         "heated"
11     ],
12     "parsed_labels": {
13         "arduino_sw": "Au123",
14         "ecu_hw": "HW:123",
15         "ecu_sw": "SW:321",
16         "heated": true,
17         "restbus_sim": "RBS:1.2.3",
18         "last_test": false,
19         "unknown_labels": null,
20         // ...
21     },
22     "description": "Please perform TCSY000",
23     "created": "2022-03-09T08:27:41.000+0100",
24     "updated": "2022-03-23T13:34:59.000+0100",
25     "assignee": {
26         "name": "paellern",
27         "key": "paellern",
28         "displayName": "Nico Paeller",
29         "emailAddress": "nico.paeller@joysonsafety.com",
30         "avatarUrls": {
31             // ...
32         }
33     },
34     "status": {
35         "id": "10900",
36         "name": "Ready for Test",
37     },
38 }
39 }
```

Codebeispiel 4: gekürzte beispielhafte JSON Repräsentation eines Jira-Tickets

Diese JSON-Repräsentation eines Jira-Tickets wurde gekürzt, um die Übersichtlichkeit dieses Dokuments zu erhalten.

Hier kann man auch die Labels im Vergleich zu den “ParsedLabels” sehen.

4.2.3 Komponentenarchitektur

Die einzelnen Komponenten wurden nach der Model-View-Controller-Architektur entworfen. Unter Komponenten sind hier die Views, also die einzelnen Webseiten (Dashboard, etc.) von TestHub gemeint.

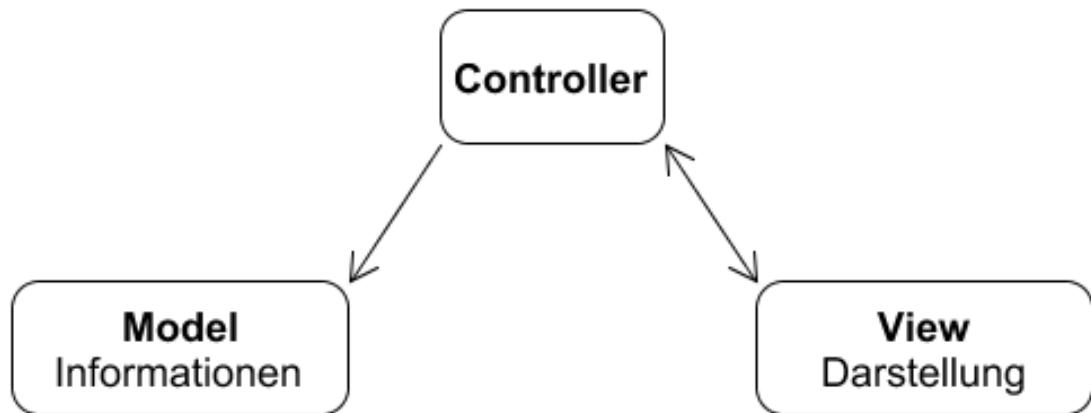


Abbildung 17: UML-Klassendiagramm zur Jira-Ticket Struktur

Das Projekt ist also in drei Teile unterteilt:

Model

Das Model beinhaltet die Informationen, welche dargestellt werden sollen. Das Model ist im Fall von "TestHub" der Server, welcher die Daten zur Verfügung stellt.

View

Das View übernimmt die Anzeige der Daten an den Benutzer. Es kann zusätzlich verschiedene Methoden zur Verfügung stellen, um Benutzereingaben an den Controller weiter zu geben. Bei TestHub ist das View das Zusammenspiel aus HTML und CSS.

Controller

Der Controller stellt die Schnittstelle zwischen View und Model her. Er kann auf Eingaben vom View reagieren und entsprechende Models laden und anzeigen. Der Controller wird durch die JavaScript/TypeScript Programme umgesetzt.

Durch diese Architektur bleibt das Projekt übersichtlich gegliedert und erweiterbar, da einzelne Views, Controller oder Models ausgetauscht werden können und somit eine andere Darstellung oder Funktionalität bieten können. Des Weiteren ist das Projekt so aufgebaut, dass der einzige Punkt, wo View und Controller zusammengeführt werden, das Laden des jeweiligen Skripts im HTML Heads ist. Da es keine weiteren Funktionsaufrufe oder Ähnliches im HTML Dokument gibt, kann der Controller aufgebaut und Design werden wie es nötig ist.

Die schemenhafte Umsetzung dieser Architektur lässt sich in Abbildung 14 wiederfinden.

5 Entwicklung

In diesem Kapitel wird die Umsetzung des in Kapitel 14 besprochenen Entwurfs erläutert. Es wird nicht nur auf den Projektaufbau und die Implementierung von Frontend und Backend eingegangen, sondern auch auf den Softwarelebenszyklus.

5.1 Projektaufbau

Das Projekt wurde als “Monorepo”, also als allumfassendes Repository, entwickelt. Dadurch sind alle für TestHub benötigten Dateien am gleichen Ort. Weiterhin muss keine Synchronisation von Frontend und Backend Versionen stattfinden, was den Entwicklungs- und Update-Prozess erleichtert [PL16].

Dennoch kann das Projekt in zwei interne Teile aufgeteilt werden:

Frontend

Unter das Frontend fallen alle vom Client ausgeführten Dateien. Unter das Frontend fallen nicht nur die Dateien im *frontend/src* Ordner, wo sich die Controller Dateien befinden, sondern auch alle Dateien in *static/* (kompiliertes JavaScript/CSS) und *templates/* (HTML Dateien).

Backend

Das Backend beschreibt das serverseitige Programm, welches auch die REST API implementiert. Der Quellcode dazu liegt im *backend/* Ordner.

Im *frontend/src/* Ordner lassen sich Quellcode für den Style Sheet und die Controller finden. Der finale Style Sheet wird mittels TailwindCSS¹ kompiliert, welches den Quellcode der HTML und TypeScript Dateien analysiert und die darin verwendeten Klassen generiert. Die Controller hingegen, wurden alle mit TypeScript² erstellt, welches, im Gegensatz zu JavaScript, Typisierung, Interfaces und vieles mehr ermöglicht. Die TypeScript Dateien werden ebenfalls in einen einstellbaren JavaScript Standard, wie zum Beispiel “es2016” kompiliert. Alle Kompilate werden im *static/* Ordner unter der gleichen Struktur gespeichert.

Das Projekt wurde gänzlich im Dateieditor Visual Studio Code³ unter Verwendung der Go und der JavaScript und TypeScript Extension entwickelt.

¹<https://tailwindcss.com>

²<https://www.typescriptlang.org>

³<https://code.visualstudio.com>

5.2 Implementierung des Frontends

In diesem Abschnitt werden ausgewählte Teile des Frontendquellcodes vorgestellt. Dabei wird nicht nur auf relevante, sondern auch auf problematische Stellen eingegangen, welche während der Entwicklung zu Problemen geführt oder eine besondere Implementierung gefordert haben.

5.2.1 Asynchrones Laden aktiver Jira-Tickets

Um das Laden der aktiven Tickets so schnell wie möglich zu gestalten, werden die Tickets asynchron, also pseudo-parallel geladen. Dies wird zum einen möglich, durch die intuitiv benutzbare Fetch API⁴ von JavaScript, welche eine einfache asynchrone Umsetzung von Requests ermöglicht. Zum Anderen, kann die Anzahl der angeforderten Tickets durch sogenannte Pagination limitiert und eingestellt werden.

```
1 const issueAmount = 20
2
3 function loadClimateChamberWidget(){
4     var total = 0
5     var request_amount = 0
6
7     api<JiraGroupedCCQueryResult>(
8         '/jira/active?grouped=true&startAt=0&maxResults=${issueAmount}
9     '
10    ).then(res => {
11        // first api request to also get the total amount of issues
12        total = res.totalResults
13        request_amount = res.startAt + res.maxResults
14        storeClimateChamber(res.data)
15
16        // multiple async requests to load the data faster
17        for (
18            let i = issueAmount*2;
19            i < res.totalResults;
20            i+=issueAmount
21        ) {
22            api<JiraGroupedCCQueryResult>(
23                '/jira/active?grouped=true&startAt=' + i + '&maxResults=20'
24            ).then(res => {
25                storeClimateChamber(res.data)
26            })
27        }
28    )
29}
```

⁴https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

```

22         '/jira/active?grouped=true&startAt=${i}&maxResults=${
23         issueAmount}'

24     ) .then( res => {
25         storeClimateChamber( res .data )
26     })
27 }
28 }
```

Codebeispiel 5: TypeScript Funktion zum asynchronen Laden aktiver Tickets (*frontend/src/ts/dashboard.ts*)

In der Funktion *loadClimateChamberWidget* wird zuerst eine GET API Anfrage gemacht. Das Ergebnis ist ein Objekt, welches dem Interface *JiraGroupedCCQueryResult* entspricht. Die Anfrage wird an die URI */jira/active* gesendet, wobei noch einige optionale Formwerte angegeben werden:

grouped Angabe, ob die aktiven Tickets vom Server, so wie in **FA#04** gefordert, nach Klimakammer sortiert werden sollen.

startAt Der Startindex, ab welchem die Tickets in der Antwort enthalten werden sollen. In der ersten Anfrage an den Server (Zeile 10) sollen alle Tickets ab dem ersten empfangen werden.

maxResults Die Anzahl der Tickets, die maximal empfangen werden soll. In der ersten Anfrage an den Server (Zeile 10) sollen die ersten 20 Tickets empfangen werden (weil startAt=0).

Die Antwort des Server enthält demnach nicht nur die Tickets, sondern auch Informationen zum Requests, wie die eben genannten *startAt* und *maxResult* Werte aber auch die Anzahl der gesamten verfügbaren Tickets.

Wenn eine Antwort empfangen wurde, werden weitere Anfragen dieser Art verschickt, mit inkrementiertem *startAt* Wert. Durch die asynchrone Eigenschaft der *fetch* Funktion (Innerhalb der *api* Funktion), wird nicht auf die Antwort einer Anfrage gewartet bis man die nächste startet. Stattdessen können weitere Anfragen gestartet werden, während auf Antworten gewartet wird. Durch diese Umsetzung lässt sich ein wesentlich schnelleres Laden der Jira-Tickets ermöglichen .

5.3 Implementierung des Backends

In diesem Abschnitt werden ausgewählte Teile des Backendquellcodes vorgestellt. Dabei wird nicht nur auf relevante, sondern auch auf problematische Stellen eingegangen, welche während der Entwicklung zu Problemen geführt oder eine besondere Implementierung gefordert haben.

5.3.1 Kategorisieren der Jira Labels

Das Kategorisieren der Labels wird durch reguläre Ausdrücke (englisch: Regular Expressions; kurz: RegEx) umgesetzt. Mit diesen kann der strukturelle Aufbau von Text schnell analysiert werden [Fit12]. Da reguläre Ausdrücke sehr schnell sehr unübersichtlich werden können, wurden zu jedem RegEx String ein entsprechender Link zu Regex101.com erstellt, in welchem der String analysiert, erklärt und getestet werden kann.

Im folgenden wird beispielhaft ein RegEx String für ein Erdungskabel kurz gezeigt und aufgeschlüsselt erklärt:

```
(?i)^gc(_|-)?\d+$
```

Die ID eines Erdungskabel hat das Format: GC_100. Jedoch gibt es, durch das manuelle Eintragen der ID, häufig andere Variationen dieser ID, welche vom Programm dennoch als Erdungskabel kategorisiert werden sollen, wie zum Beispiel: GC100, gc_100, GC-100, GC_0100 Durch den RegEx String können all diese Formate schnell und ohne viel Quellcode erkannt werden. Obwohl der String anfangs recht kompliziert aussieht, lässt er sich sehr leicht erklären⁵:

⁵<https://pkg.go.de/regexp/syntax>

Substring	Erklärung	Veranschaulichung
(?i)	Case-insensitive flag; ignoriert Groß- und Kleinschreibung	-
^	String muss mit folgendem Zeichensatz beginnen	-
gc	akzeptiert nur die Buchstaben “g” und “c” in dieser Reihenfolge	GC_100
(_ -)	akzeptiert entweder “_” oder “-”	GC_100
?	das vorherige Zeichen oder Gruppe darf 0 oder 1 mal vorkommen	-
\d+	akzeptiert nur Zahlen (eine oder mehr durch +)	GC_100
\$	String muss mit vorherigem Zeichensatz enden	-

Tabelle 4: Aufschlüsselung des regulären Ausdrucks zum Identifizieren einer Erdungsabelstruktur in einem String

5.3.2 Jira Kommunikation

Das TestHub Backend kommuniziert mit dem Jira Server via HTTP. Dabei muss jede Anfrage eine Authentifizierung im Header mitschicken. Diese Authentifizierung wird durch den Benutzernamen und das Passwort als Base64 kodierter String ermöglicht. Diese Methode ist zwar nicht sehr sicher, gerade über eine unverschlüsselte Verbindung wie HTTP, allerdings gibt es, da es keine OAuth Authentifizierung auf dem Jira Server von JSS bereitgestellt wird, keine andere Möglichkeit. Allerdings ist der Jira Server nur im Firmennetzwerk erreichbar, was das Sicherheitsrisiko durch Dritte, die eventuell sensible Daten, wie das Passwort, mitlesen, verringert.

Um die in **NFA#06** geforderte Funktionalität umzusetzen, wird ein standard Jira-Account verwendet, der die Rechte zu den benötigten Projekten besitzt. Die Zugangsdaten werden aus Sicherheitsgründen nicht direkt im Quellcode beschrieben, sondern über Umgebungsvariablen gelöst. Eine Umgebungsvariable ist eine Variable, welche Werte zur Laufzeit bereitstellen kann. Sie wird meistens vom Betriebssystem verwaltet. Durch diese Lösung, existieren die Accountdaten nur auf dem Rechner, wo sie auch angelegt wurden und können nicht einfach so geteilt werden. Da es jedoch, je nach Betriebssystem,

etwas mühselig ist, Umgebungsvariablen zu verwalten, verwendet TestHub eine `.env` Datei, welche zur Laufzeit eingelesen wird und die darin enthaltenen Variablen als Umgebungsvariablen setzt. Diese Datei wird demnach auch nie in das Versionskontrollsystem aufgenommen und kann trotzdem leicht unter Entwicklern geteilt werden.

Asynchrone Kommunikation

Das Kreieren einer asynchronen Funktion in Go ist prinzipiell sehr leicht umsetzbar. Dazu muss lediglich das `go` Keyword vor eine Funktion geschrieben werden. Dadurch wird die Funktion in einer Go Routine ausgeführt, welche mit einem Thread des Betriebssystems vergleichbar ist. Jedoch ist das Problem bei der Entwicklung, dass der Server, wie in Abbildung 14 dargestellt, zwischen Client und Server liegt. Daher muss, nachdem der Client eine entsprechende Anfrage geschickt hat, der TestHub Server die Daten vom Jira Server so schnell wie möglich lesen und in der gewünschten Form an den Client zurücksenden. Werden die Anfragen von TestHub an Jira jedoch in einer Go Routine ausgeführt, wird logischer Weise nicht auf die Antwort des Jira Servers gewartet, bis die nächste Zeile ausgeführt wird. Dadurch würde TestHub dem Client mit einem leeren Datensatz antworten.

Daher müssen diese Anfragen weiterhin asynchron verschickt werden, um den Geschwindigkeitsbonus beizubehalten, jedoch muss auf jede Jira-Antwort gewartet werden, bis TestHub an den Client eine Rückmeldung geben kann. Da der Code etwas umfangreicher ist, wird er vorerst an einem Pseudo-Code Beispiel erklärt.

Die Implementierung dieses Lösungsansatzes wird an einem Auszug der in **NFA#03** geforderten Projektübersicht erklärt.

```
1 // create a channel to retrieve the results of each request
2 c := make(chan *models.JiraAsyncQueryResult)
3 // create a waitgroup to wait for all requests to finish
4 var wg sync.WaitGroup
5
6 // asynchronously query jira server
7 for key := range projects {
8     wg.Add(1)
9     jql := fmt.Sprintf(config.JQL_TICKETS_PER_PROJECT, key)
10
11    go jira.QueryIssuesAsync(key, c, &wg, jql, []string{}, 0, 1)
```

```

12 }
13
14 go func() {
15 // this blocks the goroutine until WaitGroup counter is zero
16     wg.Wait()
17     close(c) // Channels need to be closed
18 }()
19
20 totalResults := 0
21 // parse all received data
22 for res := range c {
23     // do something with results
24 }
25 // respond to client

```

Codebeispiel 6: Go Umsetzung von asynchronen HTTP Anfragen
(*backend/api/api_jira.go*)

In diesem Beispiel das rapide starten mehrere HTTP Anfragen zu sehen. Um dies In Go zu ermöglichen, wird eine *WaitGroup* benötigt, welche eine threadsichere Möglichkeit bietet, das Ausführen weiteren Codes zu verhindern, solange mindestens ein Element in der *WaitGroup* noch nicht beendet wurde. Man kann sich diese *WaitGroup* wie einen Zähler vorstellen, welcher zählt, wie viele Funktionen gestartet und wie viele beendet wurden.

Um eine Kommunikation mit den Go Routinen herzustellen, wird ein von Go bereitgestellter threadsicherer Channel verwendet. Dieser kann *JiraAsyncQueryResult* speichern.

Ab Zeile 9 wird pro Projekt die *WaitGroup* hochgezählt und ein JQL String erstellt. Anschließend wird die *QueryIssuesAsync* Funktion in einer Go Routine ausgeführt, welche die tatsächliche Anfragelogik beinhaltet. Diese Funktion erhält, neben den Standardparametern auch den Key, zur Identifikation der Antwort, den Channel, zum versenden der Antwort an die Haupt-Go-Routine und eine Referenz der *WaitGroup* zum Dekrementieren dieser, bei Termination der Funktion.

Darauf folgend wird in Zeile 18 auf das beenden aller zuvor gestarteten Go Routinen gewartet und anschließend der Channel wieder geschlossen, damit aus diesem die gespeicherten Daten entnommen und weiterverarbeitet werden können.

5.3.3 REST API Dokumentation

Um die Anforderung **NFA#02** zu erfüllen, wurde eine Dokumentation mittels Postman⁶ erstellt. Postman ist ein Programm zum gemeinsamen Testen und Debuggen von APIs. Es bietet zusätzlich an eine Dokumentation zu generieren und bereitzustellen, mit den tatsächlichen Anfragen und realen Antworten vom Server. Dadurch lässt sich das Testen der API Endpunkte und das Dokumentieren dieser effizient zusammenfassen und an einem Ort wiederfinden. Außerdem kann in der Dokumentation automatisch die gewünschte Sprache und somit Codebeispiele für die jeweiligen Anfragen generieren. Durch die zusätzliche Integration von Markdown, lassen sich aussagekräftige und tatsächlich hilfreiche API Dokumentationen erstellen.

⁶<https://www.postman.com>

6 Literaturverzeichnis

Literatur

- [Gou+02] David Gourley u. a. *HTTP: The Definitive Guide*. O'Reilly and Associate Series. O'Reilly Media, 2002. ISBN: 9781565925090.
- [Wik10] Wikimedia. *Tcp-handshake.svg*. 2010. URL: <https://commons.wikimedia.org/wiki/File:Tcp-handshake.svg> (besucht am 25.03.2022).
- [Mas11] M. Masse. *REST API Design Rulebook*. O'Reilly and Associate Series. O'Reilly Media, 2011. ISBN: 9781449310509. URL: <https://books.google.de/books?id=4lZcsRwXo6MC>.
- [Fit12] M. Fitzgerald. *Einstieg in reguläre Ausdrücke : [Schritt für Schritt reguläre Ausdrücke verstehen]*. O'Reilly, 2012. ISBN: 9783868999402.
- [NF13] K.D. Niemann und S. Fedtke. *Client/server-Architektur: Organisation und Methodik der Anwendungsentwicklung*. Zielorientiertes Business Computing. Vieweg+Teubner Verlag, 2013. ISBN: 9783663001706.
- [PL16] Rachel Potvin und Josh Levenberg. *Why Google Stores Billions of Lines of Code in a Single Repository*. 2016. URL: <https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext> (besucht am 29.03.2022).
- [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.

7 Eigenständigkeitserklärung

“Ich versichere, dass ich die Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen oder anderen Quellen entnommen sind, sind als solche kenntlich gemacht. Die schriftliche und die elektronische Form der Arbeit stimmen überein.”

Ort, Datum

Unterschrift

8 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)
```

Codebeispiel 7: 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 }
```

Codebeispiel 8: 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"]))
```

Codebeispiel 9: 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	27.08n in AB starten	C3003932-4588	PV_3932_1886	TCSY128	3042SY21_008692	PV	Au26a_CC_Profil_anpassen_HOD_heat_HW.089_Mainstream_PV_18, RBS 1.0, SW.022_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	C3003932-4589	PV_3932_1890	TCSY128	3042SY21_008693	PV	Au26a_CC_Profil_anpassen_HOD_heat_HW.089_Mainstream_PV_18, RBS 1.0, SW.022_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	C3003932-4590	PV_3932_1956	TCSY128	3042SY21_008688	PV	Au26a_CC_Profil_anpassen_HOD_only_HW.089_Mainstream_PV_18, RBS 1.0, SW.022_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	C3003932-4591	PV_3932_1957	TCSY128	3042SY21_008689	PV	Au26a_CC_Profil_anpassen_HOD_only_HW.089_Mainstream_PV_18, RBS 1.0, SW.022_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	C3003932-4592	PV_3932_1959	TCSY128	3042SY21_008690	PV	Au26a_CC_Profil_anpassen_HOD_only_HODbox10_HW.089_PV_18, RBS 1.0, RBS 1.0, 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 min	C3003932-4597	PV_3932_1811	TCSY128	3042SY21_008691	PV	Au26a_CC_Profil_anpassen_HOD_heat_HW.089_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	TCSYET			A_Dust_chamber	C3003436-12550 Leerzeile_A_Dust_chamber	
A_KPK100	Open	000 - LEERZEILE - Elementais 1215	C3003436-16751	XXXXXXXXXXXXXX	TCSYET			A_KPK1215	C3003436-12550 Leerzeile_A_KPK1215	
A_KPK100	Ready for Test	02.03.22 - starten	C1005862-445	PV_5862_5159	TCSY145	3042SY22_001600	PV	Almemo_Au27_HOD_HW.089_PV_28_Programm_L_RBS 1.10, SW.0023X_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_001600	PV	Almemo_Au27_HOD_HW.089_PV_28_Programm_L_RBS 1.10, SW.0023X_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_Au27_HOD_HW.089_PV_28_Programm_L_RBS 1.10, SW.0023X_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_Au27_HOD_HW.089_PV_28_Programm_A_RBS 1.10, SW.0023X_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_Au27_HOD_HW.089_PV_28_Programm_A_RBS 1.10, SW.0023X_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_Au27_HOD_HW.089_PV_28_Programm_A_RBS 1.10, SW.0023X_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_Au27_HOD_HW.089_PV_28_Programm_L_RBS 1.10, SW.0023X_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_Au27_HOD_HW.089_PV_28_Programm_L_RBS 1.10, SW.0023X_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_Au27_HOD_HW.089_PV_28_Programm_L_RBS 1.10, SW.0023X_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	DW_4533_0004	TCSY507	3042SY22_000425	DV	HOD+HEAT_HW.002_SW.0003	C3004533-19 RUN TCSY507 High_Temperature_storage_504h	

Abbildung 18: Auszug eines Tagesplans vom 02.03.2022

Übersicht: Daten übertragen sich automatisch aus anderen Reihen hier nicht bearbeitbar														
Standort	Klimakammer	Liter Volume	maximale Musteranzahl	Rechner	Vectorbox / NI Case	Relaisbox	Klemmbrett	Netzteil	Netzteil 2	Almemo	Betriebsbereit	Wartung	Defekt	
										Name	Status RB	3	0	2
Aucoteam	KPK1216	300	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Status CB:	3	1	0
	KPK1215	1000	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A				
	KPK6	600	3	N/A	N/A	N/A	N/A	N/A	N/A	N/A				
	KPK1212	300	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A				
	KPK1213	600	3	N/A	N/A	R8 302	N/A	N/A	5758	6593				
	KPK2	800	3	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				
RPH-103		1000	3	N/A	N/A	R8 300	1	6598	6738	N/A				
	KPK104	1000	4	N/A	N/A	N/A	4	6595	6594	N/A				
JSS	DC05	600	3	D304TE147	HOD SysTest	R8 309	7	6731	6737	N/A				
	DC07	600	3	D304TE142	HOD SysTest	N/A	11	1873	6737	N/A				
	DC08	600	3	D304TE148	HOD SysTest	R8321	22	5044	6734	N/A				
	DC09	600	3	D304TE141	HOD SysTest	N/A	30	6732	6724	N/A				
	DC10	600	3	D304TE149	N/A	R8310	17	6725	N/A	N/A				
	DC11	600	3	D304TE150	HOD SysTest	R8328	18	6726	6736	Almemo 6				
	DC12	600	3	D304TE150	HOD SysTest	R8322	2	6598	5747	N/A				
	DC13	600	3	D304TE143	HOD SysTest	N/A	13	6727	6733	Almemo 2				
	DC14	600	3	D304TE138	HOD SysTest	N/A	24	5744	6735	N/A				
	DC15	600	3	D304TE151	HOD SysTest	R8316	16	5743	6735	N/A				
	DC16	600	3	D304TE144	HOD SysTest	R8 303	N/A	6599	1886	N/A				
	DC17	600	3	D304TE032	HOD SysTest	N/A	14	5746	N/A	Almemo 3				

Abbildung 19: Übersichtsseite der Ressourcenliste vom 26.03.2022