

# **Hochschule Osnabrück**

University of Applied Sciences

## **Fakultät Ingenieurwissenschaften und Informatik**

Schriftliche Ausarbeitung zum Thema:

### **PC Part API**

im Rahmen des Moduls  
Software-Architektur – Konzepte und Anwendungen,  
des Studiengangs Informatik-Medieninformatik

Autor:	Jannis Welkener
Matr.-Nr.:	966265
E-Mail:	jannis.welkener@hs-osnab- rueck.de

Autor:	Daniel Graf
Matr.-Nr.:	914743
E-Mail:	daniel.graf@hs-osnabrueck.de

Themensteller:	Prof. Dr. Rainer Roosmann
----------------	---------------------------

# Inhaltsverzeichnis

1	Einleitung .....	7
1.1	Vorstellung des Themas .....	7
1.2	Ziel der Ausarbeitung .....	7
1.3	Aufbau der Hausarbeit .....	8
2	Darstellung der Grundlagen .....	9
3	Planung & Systemmodellierung .....	10
3.1	Schnittstellen .....	11
3.2	Module .....	11
3.2.1	Modul: Computerteile .....	12
3.2.2	Modul: Computerkonfigurationen .....	14
3.2.3	Modul: Nutzerverwaltung .....	14
3.2.4	Modul: Kommentare und Reviews .....	15
4	Persistierung & Zugriffskontrolle .....	17
4.1.1	Vergleich: SQL und MongoDB .....	17
4.1.2	Auswahl der Datenbanken .....	19
4.1.3	Auswahl des Datenbankzugriffsframeworks .....	20
4.2	Rollenbasierter Zugriff und Nutzerverwaltung .....	20
4.3	Nutzerinteraktion .....	22
4.4	Bereitstellung von Bilddaten .....	23
4.4.1	Verifizierung der Eingabedateien für Bilder .....	23
5	Konfiguration und Laufzeitüberwachung .....	25
5.1.1	Konfiguration in Quarkus .....	25
5.1.2	PostgreSQL Konfiguration .....	25
5.1.3	MongoDB Konfiguration .....	26
5.2	Logging .....	27
6	Anwenden der PC Part API .....	29
6.1	Usability & Schemata .....	29
6.2	Input Validation .....	29
6.3	Antwortnachrichten .....	30
6.4	Filtering & Queries für Listenressourcen .....	31
7	Programmresistenz & Fehlertoleranz .....	32
7.1	Fault Tolerance .....	32
7.1.1	Retry .....	32
7.1.2	Timeout: .....	33
8	Unit- & Integration Testing .....	34
8.1.1	Testarten .....	34
8.1.2	Unit Tests .....	34
8.1.3	Integration Testing: .....	35
8.1.4	Testing – Musskriterien .....	35
8.1.5	Testing – Wunschkriterien .....	35
8.1.6	Testing Sicherheitsfunktionen .....	36
9	Zusammenfassung und Fazit .....	37
9.1	Zusammenfassung .....	37
9.2	Fazit .....	37

10 Referenzen .....	38
---------------------	----

## Abbildungsverzeichnis

Abbildung 1: <i>UML</i> -Auszug für das Modul "ComputerConfiguration" .....	10
Abbildung 2: Auszug aus <i>SwaggerUI</i> - Übersicht der <i>API</i> -Definition, Ressourcen sowie deren Hilfsbeschreibungen.....	11
Abbildung 3: PC-Komponentenressource in <i>SwaggerUI</i> .....	12
Abbildung 4: Computerkomponenten nach Typ in <i>SwaggerUI</i> .....	13
Abbildung 5: Bilder für Komponenten in <i>SwaggerUI</i> .....	13
Abbildung 6: Private Computerkonfigurationen in <i>SwaggerUI</i> .....	14
Abbildung 7: Öffentliche Computerkonfigurationen in <i>SwaggerUI</i> .....	14
Abbildung 8: Nutzerverwaltung in <i>SwaggerUI</i> .....	15
Abbildung 9: Kommentare & Bewertungen für Computerkonfigurationen in <i>SwaggerUI</i> .....	15
Abbildung 10: Kommentare & Bewertungen für Computerkomponenten in <i>SwaggerUI</i> .....	16
Abbildung 11: SQL-Abhängigkeiten ([@] <a href="https://elearn.inf.tu-dresden.de/sqlkurs/lektion02/02_99_zusatz_beziehungen.html">https://elearn.inf.tu-dresden.de/sqlkurs/lektion02/02_99_zusatz_beziehungen.html</a> ).....	18
Abbildung 12: MongoDB-Beziehungen ([@] <a href="https://www.researchgate.net/figure/Abbildung-3-Aufbau-der-MongoDB-MongoDB-stellte-einige-Methoden-bereit-um-Relationen_fig3_321797933">https://www.researchgate.net/figure/Abbildung-3-Aufbau-der-MongoDB-MongoDB-stellte-einige-Methoden-bereit-um-Relationen_fig3_321797933</a> ) .....	19
Abbildung 13: User Entität.....	21
Abbildung 14: Validierung der <i>Magic-Number</i> .....	24
Abbildung 15: Auszug aus "application.properties" .....	25
Abbildung 16: Instanziierung eines <i>Logger</i> -Objektes .....	27
Abbildung 17: Filtern einer Listenressource mit Performance-Messung in Millisekunden .....	28
Abbildung 18: Konsolenausgabe bei der Abfrage einer Listenressource .....	28
Abbildung 19: <i>Data Transfer Object</i> für die Computerkomponente: RAM .....	29
Abbildung 20: <i>Input Validation</i> für eine ID innerhalb einer <i>DTO</i> .....	30
Abbildung 21: Erhaltene Antwort auf den Pfad <i>/computers/private</i> mit <i>GET</i> im <i>HATEOAS</i> Format .....	30
Abbildung 22: Listenressource für Computerkomponenten mit Filteroptionen .....	31
Abbildung 23: REST-Endpunkt Methode mit Fault Tolerance Annotationen .....	32
Abbildung 24: Testklasse für private Computerkonfigurationen .....	34
Abbildung 25: Testing - Zugriffskontrolle .....	36

## Tabellenverzeichnis

Tabelle 1: PostgreSQL Einstellungen in "application.properties" .....	26
Tabelle 2: MongoDB Einstellungen in "application.properties" .....	27

## Abkürzungsverzeichnis

CDI	Context and Dependency Injection for the Java EE Plattform
ECB	Entity-Controller-Boundary Pattern
EJB	Enterprise Java Beans
Java EE	Java Enterprise Edition, in der Version 7
JSF	Java Server Faces
SWA	Software-Architektur
SFLB	Stateful Session Bean
SLSB	Stateless-Session Bean
SQL	Structure Query Language
API	Application Programming Interface
UML	Unified Modelling Language
HTTP	Hypertext Transfer Protocol
DTO	Data Transfer Object
JSON	Javascript Object Notation
UUID	Universally Unique Identifier

# 1 Einleitung

*Autor Jannis Welkener*

Computer sind in unserem Leben nicht mehr wegzudenken. Umso schwerer ist es, den richtigen Computer zu finden oder zusammenzustellen. Der Grund dafür ist die große Auswahl an Teilen sowie wenig einheitliche Plattformen, diese zu vergleichen und seriöse Bewertungen zu diesen zu finden.

Aus diesem Grund beschäftigt sich diese Arbeit im Rahmen des Moduls Software-Architektur mit der Entwicklung einer *API* ausgerichtet an Softwareentwickler, welche Informationen und Funktionen bietet, um diese Lücke zu füllen.

Ein besonderer Schwerpunkt liegt hierbei auf der Software-Architektur des Programms, um eine möglichst große Erweiterbarkeit und Wartbarkeit zu bieten.

## 1.1 Vorstellung des Themas

*Autor Jannis Welkener*

Im Rahmen der Entwicklung wird ein großer Wert auf die Themen Sicherheit, Korrektheit und persistente Speicherung gelegt. Genauer soll eine rollenbasierte Zugriffskontrolle auf bestimmte Funktionen umgesetzt werden, sodass diese von unbefugtem Zugriff geschützt werden. Ebenfalls wichtig ist die Korrektheit der verwalteten Daten, sodass diese nicht von unbefugten verfälscht, oder versehentlich falsch eingegeben werden. Diese Daten werden nicht wie oft in einer *SQL*-Datenbank gespeichert, sondern in der dokumentenbasierten Datenbank namens *MongoDB*.

## 1.2 Ziel der Ausarbeitung

*Autor Jannis Welkener*

Ziel dieser Arbeit ist es, eine *API* für andere Entwickler zu schaffen, sodass diese eine eigene grafische Anwendung entwickeln können, welche für den Endnutzer zur Verfügung steht. Das bedeutet, dass die tatsächlichen Nutzer der *API* andere Entwickler sind, was bedeutet, dass die äußere Struktur der Schnittstelle einfach und übersichtlich für andere Programmierer sein, und allgemeine gängige Konventionen erfüllen muss.

### 1.3 Aufbau der Hausarbeit

*Autor Jannis Welkener*

In dieser Arbeit wird zuerst ein grobes Architekturkonzept im Sinne des *Entity-Control-Boundary-Patterns* erarbeitet. Zu diesem werden sinnvolle Schnittstellen vordefiniert und entwickelt. Für den Nutzer soll die Funktionsweise der Schnittstellen und die Nutzung einfach zu verstehen sein und Paradigmen der Schnittstellenentwicklung erfüllen.

Die *API* soll ein System anbieten, aus dem Informationen zu Computerteilen entnommen werden können. Auch soll es möglich sein, Computerkonfigurationen zu erstellen und Bewertungen anzulegen.

Sind die Schnittstellen definiert, wird eine sinnvolle Lösung der Umsetzung nach aktuellen Standards erarbeitet. Dies umfasst die persistente Speicherung der Daten und Sicherheitsaspekte, um das System gegen böswillige Akteure zu schützen.

Durch diesen Aufbau sind die Entwicklungsziele bereits offensichtlich und die Erarbeitung erschließt sich einfacher.

Daraufhin wird die Schnittstelle möglichst auf Nutzerfreundlichkeit verbessert, sodass es ohne weitere Hilfe möglich sein soll, die *API* zu nutzen. Eine *API* kann zwar funktionieren, ist jedoch nutzlos, wenn niemand versteht, wie sie zu benutzen ist.

Schließlich wird mithilfe von Tests die Korrektheit der Funktionen bewiesen, sodass sichergestellt ist, dass das System dauerhaft konsistent bleibt.



## 2 Darstellung der Grundlagen

*Autor Jannis Welkener*

Die Umsetzung der Schnittstelle erfolgt über das *Java-Framework Quarkus*, welches von *Red Hat* entwickelt wurde. *Quarkus* bietet eine Vielzahl von Bibliotheken, die es erlauben, spezifische Features in das Projekt zu implementieren [[@Quarkus](#)].

Um unabhängig von der jeweiligen Computerdomäne zu sein, wird bei *Quarkus* meist mit *Docker*-Containern gearbeitet. Benötigte Container werden automatisch vom *Framework* aus gestartet, sofern *Docker* installiert ist. Mithilfe dieser Container und den Bibliotheken ist es ebenfalls möglich, von *Quarkus* aus Datenbanken zu erstellen und zu verwalten.

Hierbei ist es möglich, über standardisierte Bibliotheken aus *SQL*- und *NoSQL*-Datenbanken zuzugreifen. Im Rahmen dieser Arbeit wird der *Quarkus*-spezifische Standard „*Panache*“ für die Persistierung von *SQL*- als auch *NoSQL*-Daten genutzt.

Nutzerdaten werden hierbei in *SQL*-Tabellen in einer *PostgreSQL*-Datenbank verwaltet und globale Daten in einer *MongoDB*-Datenbank. Diese speichert Daten in Dokumenten und bietet gegenüber *SQL*-Datenbanken den Vorteil, dass die Daten schneller verarbeitet werden und das System skalierbarer ist.

### 3 Planung & Systemmodellierung

*Autor Daniel Graf*

Zu Beginn der Entwicklung wird das Design- & Modellierungskonzept der Anwendung geklärt. Zur Modellierung des Systems sowie der zugehörigen Module wird das Programm *Visual Paradigm* genutzt. Mit diesem Werkzeug wird vor der eigentlichen Programmierung die allgemeine Systemstruktur als *UML*-Diagramm modelliert (siehe Abbildung 1). So lassen sich allgemeine Beziehungen zwischen den verschiedenen Klassen darstellen. Im Laufe der Entwicklung wird dieses Dokument stückweise auf die Anforderungen angepasst und weiter verfeinert.

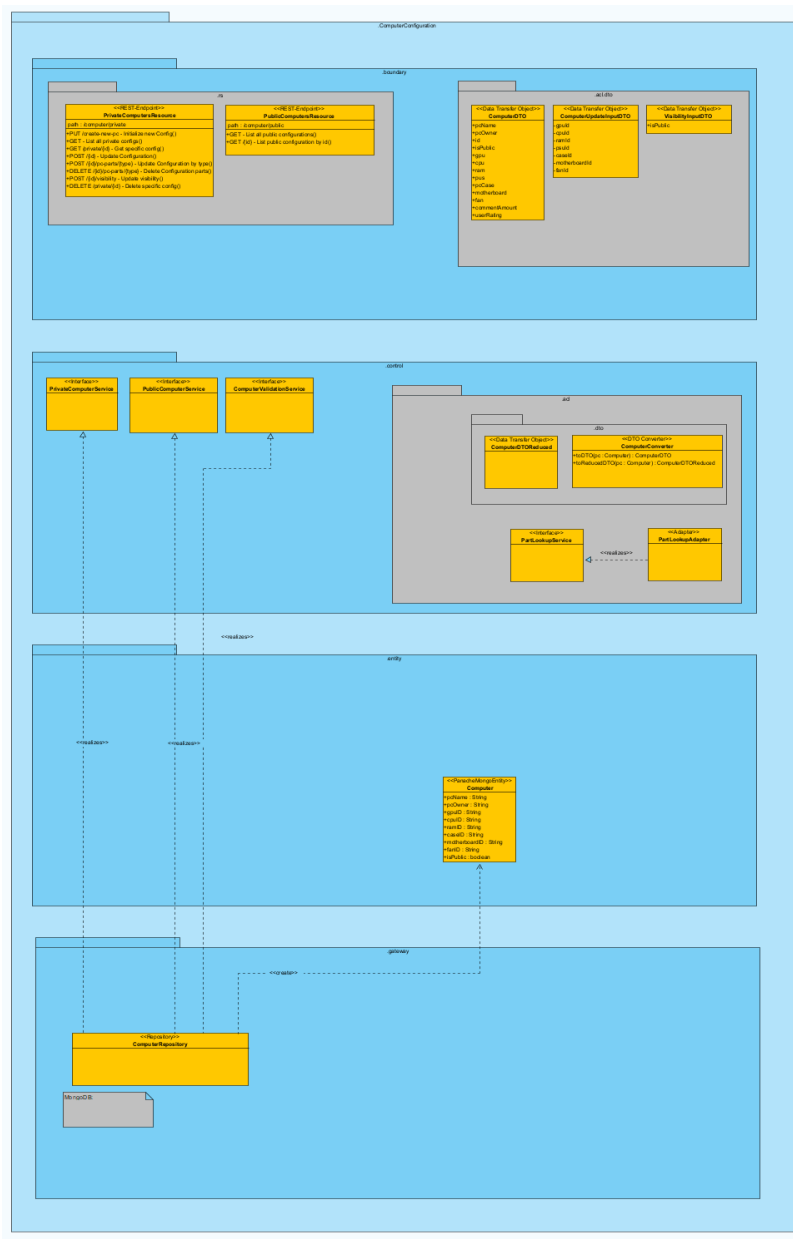


Abbildung 1: UML-Auszug für das Modul "ComputerConfiguration"

### 3.1 Schnittstellen

Autor Jannis Welkener

Der vermutlich wichtigste Teil einer *API* ist die öffentliche Schnittstelle. Diese definiert alle Methoden, die vom Nutzer abrufbar sind und sollten vom Namen her aussagekräftig über ihre Funktion sein.

Bei einer *API* werden *HTTP*-Methoden in Routenform verwendet. Diese Routen sollten sinnvoll Listenstrukturen und Einzelteile von Listen darstellen.

Eine Übersicht aller Methoden lässt sich über die *OpenAPI*-Definition oder über *SwaggerUI* einsehen. Mithilfe letzterem lassen sich Schnittstellen ebenfalls leicht testen. Ein Beispiel dieser Übersicht sieht man in Abbildung 2.

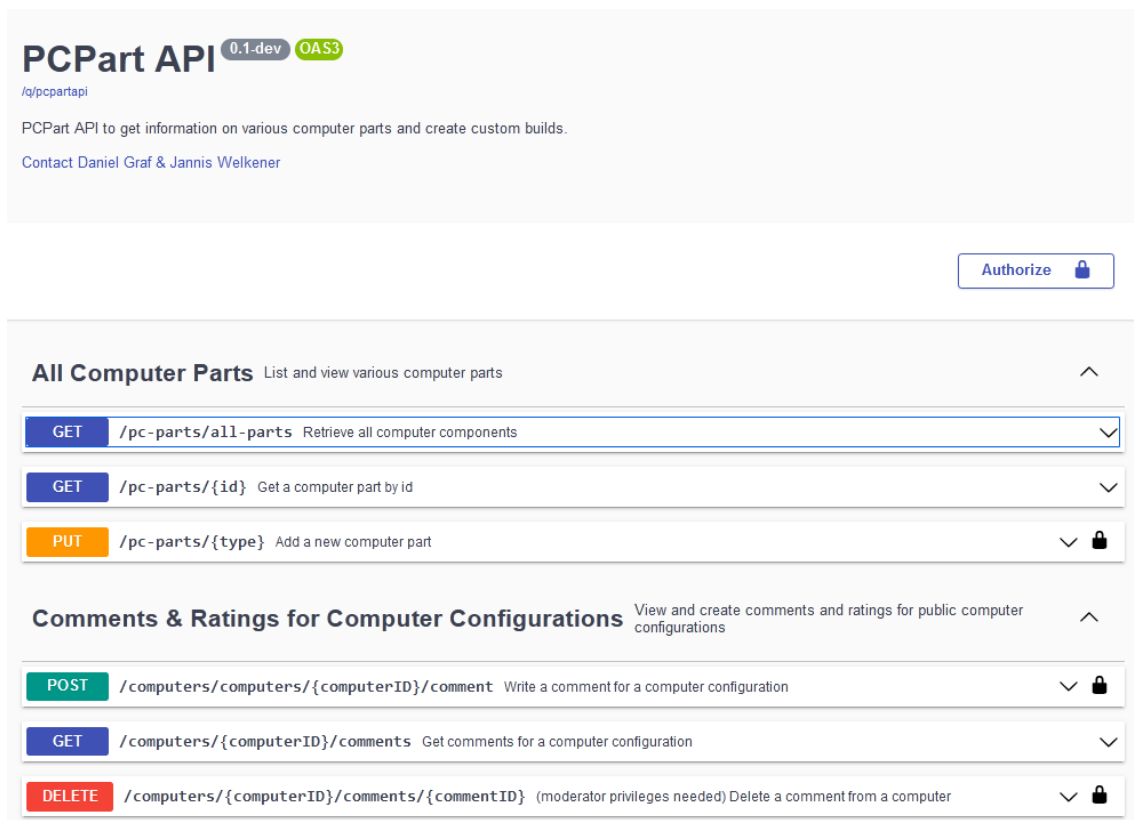


Abbildung 2: Auszug aus *SwaggerUI* - Übersicht der *API*-Definition, Ressourcen sowie deren Hilfsbeschreibungen

### 3.2 Module

Autor Jannis Welkener

Grundlegend lässt sich die Aufgabenstellung in vier Teilbereiche aufteilen:

1. Computerteile, welche hochgeladen und angeboten werden.
2. Computer, die von Nutzern erstellt und geteilt werden.
3. Kommentare die Nutzerbewertungen widerspiegeln.
4. Eine Nutzerkontrolle, die es überhaupt erlaubt, Nutzer anzulegen und zu verwalten.

Diese Teilbereiche werden als fachliche Module im Rahmen des *Domain Driven Design* umgesetzt und sollen möglichst vollständig voneinander unabhängig sein.

Jedes Modul besteht aus derselben *Entity-Control-Boundary*-Struktur, um die Wartbarkeit zu erhöhen.

### 3.2.1 Modul: Computerteile

*Autor Jannis Welkener*

Ziel des Moduls der Computerteile ist es, Computerteile anlegen und abrufen zu können. Beim Abrufen wird zwischen einer Liste von Teilen und einem einzelnen Teil unterschieden. Ein einzelnes Teil kann über die Teil-ID abgerufen werden. Für Listen besteht die Möglichkeit, alle Teile unabhängig des Typs abzurufen und mit *Query*-Parametern zu filtern oder alle Teile eines Typs zu erhalten.

Alle Methoden, die mit Computerteilen zu tun haben, können mit dem Präfix *“/pc-parts”* erreicht werden.

Um ein Teil hinzuzufügen, werden Admin-Rechte benötigt. Ein Admin kann dann entweder über die *Route /pc-parts* den Typ des Teils definieren oder direkt ein Teil über die Typ-Route hinzufügen (siehe Abbildung 3 & 4).

Beim Hinzufügen wird ein Korrektes *DTO* erwartet, welches vor dem Einfügen überprüft wird.

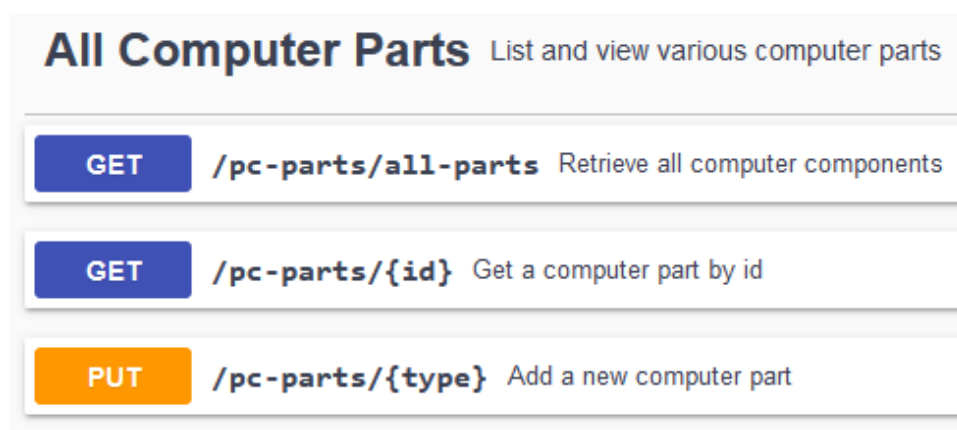


Abbildung 3: PC-Komponentenressource in *SwaggerUI*

Computer Parts by Type <small>List and view computer parts by various types or add new ones (elevated permissions needed)</small>	
GET	/pc-parts/type/cases
PUT	/pc-parts/type/cases
GET	/pc-parts/type/cpus
PUT	/pc-parts/type/cpus
GET	/pc-parts/type/fans
PUT	/pc-parts/type/fans
GET	/pc-parts/type/gpus
PUT	/pc-parts/type/gpus
GET	/pc-parts/type/motherboards
PUT	/pc-parts/type/motherboards
GET	/pc-parts/type/psus
PUT	/pc-parts/type/psus
GET	/pc-parts/type/rams
PUT	/pc-parts/type/rams

Abbildung 4: Computerkomponenten nach Typ in *SwaggerUI*

Ein Teil kann ebenfalls ein Bild besitzen. Ob es eins besitzt oder nicht wird in der Rückgabe eines Teils angegeben. Ist ein Bild vorhanden, kann es über eine Route mithilfe der Teil-ID abgerufen werden.

Über dieselbe *PUT*-Route kann ein Admin auch ein Bild setzen oder Löschen. Hier wird eine korrekte Bilddatei vom Typ *PNG* oder *JPG* erwartet. Wird keine Datei mitgesendet, wird das zugehörige Bild des Teils gelöscht.

Computer Part Images <small>Upload and look at the images of computer parts</small>	
GET	/pc-parts/{id}/image <small>Get the image of a computer part by id</small>
PUT	/pc-parts/{id}/image <small>Upload an image for a computer part</small>

Abbildung 5: Bilder für Komponenten in *SwaggerUI*

### 3.2.2 Modul: Computerkonfigurationen

Autor Jannis Welkener

Das Computerkonfigurationsmodul verwaltet private und öffentliche Konfigurationen von Nutzern. Demnach muss es eine Möglichkeit geben, private und öffentliche Konfigurationen einzusehen, zu bearbeiten und zu löschen, sofern die Rechte des Nutzers es erlauben.

Alle Methoden zum Abrufen einer Funktion, die sich auf Computerkonfigurationen bezieht, beginnt mit dem Präfix `"/computers"`.

Ein eingeloggter Nutzer kann sich alle Konfigurationen ansehen, die er erstellt hat (siehe Abbildung 6 & 7).

Private Computer Configurations <small>Create and configure your personal computer configurations</small>	
GET	<code>/computers/private</code> List all private computer configurations
PUT	<code>/computers/private/create-new-pc</code> Create a new empty computer configuration
PUT	<code>/computers/private/create-new-pc/pre-filled</code> [DEV] Creates a new computer configuration with randomly selected parts
GET	<code>/computers/private/{id}</code> Show Information of a specific private computer configuration
POST	<code>/computers/private/{id}</code> Add or update parts of the computer configuration
DELETE	<code>/computers/private/{id}</code> Delete a private computer configuration permanently
POST	<code>/computers/private/{id}/pc-parts/{type}</code> Add or update parts of the computer configuration
DELETE	<code>/computers/private/{id}/pc-parts/{type}</code> Delete parts of a computer configuration
POST	<code>/computers/private/{id}/visibility</code> Set to 'public' or 'private' to change if other users can see the computer configuration

Abbildung 6: Private Computerkonfigurationen in *SwaggerUI*

Public Computer Configurations <small>List and view all public listed computer configurations sourced by various users on this platform</small>	
GET	<code>/computers/public</code> Retrieve all public computer configurations
GET	<code>/computers/public/{id}</code> Retrieve a specific public computer configuration by id

Abbildung 7: Öffentliche Computerkonfigurationen in *SwaggerUI*

### 3.2.3 Modul: Nutzerverwaltung

Autor Jannis Welkener

Mit der Route `"/users"` lässt sich die Accountverwaltung ansprechen. Über die weitere Route `"/admin-commands"` sind Methoden erreichbar, die nur für einen Admin ansprechbar und interessant sind. Ein solcher kann alle registrierten Nutzer und ihrer Rechte zur Verwaltung einsehen und neue Moderatoraccounts erstellen, welche erweiterte Rechte im Kommentarbereich besitzen.

Mithilfe der “/me”-Route kann jeder Nutzer seinen aktuell angemeldeten Accountnamen einsehen (siehe Abbildung 8).

Unter “/register” kann jeder einen eigenen Account erstellen, sofern ein eindeutiger Nutzername und ein Passwort mitgeschickt werden. Damit wird dann im System ein neuer Nutzer mit User-Rechten erstellt, welcher sofort nutzbar ist.

User Account Register and login to your created user account		
GET	/users/admin-commands/get-users	Retrieve all user-accounts (admin privileges needed)
POST	/users/admin-commands/register-moderator	Register a moderator account (admin privileges needed)
GET	/users/me	Retrieve your personal account info (requires login first)
POST	/users/register	Register a new user

Abbildung 8: Nutzerverwaltung in *SwaggerUI*

### 3.2.4 Modul: Kommentare und Reviews

Autor Jannis Welkener

Computer und Computerteile können beide Kommentare besitzen. Das Kommentar-Modul erweitert beide Schnittstellen so, dass Kommentare erstellt, gelesen und von Moderatoren gelöscht werden können.

Ein Kommentar bezieht sich immer auf ein Computerteil oder einen Computer, weshalb eine Zuweisung über die Objekt-ID besteht. So hat jeder die Möglichkeit, zu jedem Objekt eine Liste an Kommentaren abzurufen (siehe Abbildung 9 & 10).

Für das Erstellen eines Kommentares muss ein Nutzer eingeloggt werden, sodass Kommentare einem Nutzer zugeordnet werden können. Ein Kommentar besteht aus einer Bewertung von 1-5 und einem verfassten Text.

Comments & Ratings for Computer Configurations View and create comments and ratings for public computer configurations		
POST	/computers/computers/{computerID}/comment	Write a comment for a computer configuration
GET	/computers/{computerID}/comments	Get comments for a computer configuration
DELETE	/computers/{computerID}/comments/{commentID}	(moderator privileges needed) Delete a comment from a computer

Abbildung 9: Kommentare & Bewertungen für Computerkonfigurationen in *SwaggerUI*

Comments & Ratings for Computer Parts		Write and look at ratings for computer parts
POST	/pc-parts/{partId}/comment	Write and publish a comment for a computer part
GET	/pc-parts/{partId}/comments	Get comments for a computer part
DELETE	/pc-parts/{partId}/comments/{commentId}	[Elevated rights] Delete a comment from a computer part

Abbildung 10: Kommentare &amp; Bewertungen für Computerkomponenten in SwaggerUI



## 4 Persistierung & Zugriffskontrolle

*Autor Jannis Welkener*

Um Nutzern Daten zu Computerteilen anzubieten, sowie Kommentierung zu erlauben ist es sinnvoll, alle Daten persistent in einer Datenbank zu speichern. Da es bisweilen sehr viele Computerteile geben kann, wurde für die Verwaltung die Datenbank *MongoDB* gewählt. Hierbei handelt es sich um eine *NoSQL*-Datenbank, welche die Daten nicht in Tabellen speichert.

Nutzerdaten sollten jedoch von den restlichen Daten getrennt werden. Aus diesem Grund werden die Login-Daten für alle Accounts in einer *PostgreSQL*-Datenbank gehalten und verwaltet.

### 4.1.1 Vergleich: SQL und MongoDB

*Autor Jannis Welkener*

Das Konzept von einer *SQL*-Datenbank und einer *MongoDB*-Datenbank ist sehr verschieden. *SQL*-Datenbanken speichern die Daten in relationalen Tabellen, sodass ein Speicherobjekt andere Objekte referenzieren kann. Eine *MongoDB* speichert Daten unabhängig voneinander in *JSON*-Dokumenten. Diese Konzepte bringen einige Vor- und Nachteile mit sich.

Eine *SQL*-Datenbank erlaubt es, komplexe Analysen und *Queries* zu erstellen. Mit Hilfe von *Joins* lassen sich Daten vereinen und so durch die Relationalität tiefere Verbindungen erzeugen.

*MongoDB* bietet dagegen nur simple *Queries*, mit denen Objekte bloß abgerufen und grob gefiltert werden können.

Stattdessen bietet *MongoDB* eine deutlich bessere Skalierbarkeit. Wie die meisten *NoSQL*-Datenbanken ist *MongoDB* horizontal skalierbar. Das bedeutet, dass wenn der Speicherplatz nicht mehr ausreicht, einfach weitere Speichermedien hinzugefügt werden können. Aufgrund der nicht vernetzten Dokumentstruktur sind die Dateien unabhängig vom lokalen System und können auf mehrere verteilt werden.

*SQL*-Datenbanken sind jedoch nur vertikal skalierbar, alle Daten müssen sich also auf einem System befinden, da die Tabellen untereinander abhängig sind. Wenn mehr Speicherplatz benötigt, muss das ganze System vergrößert, oder sogar komplett ausgetauscht werden (siehe Abbildung 11).

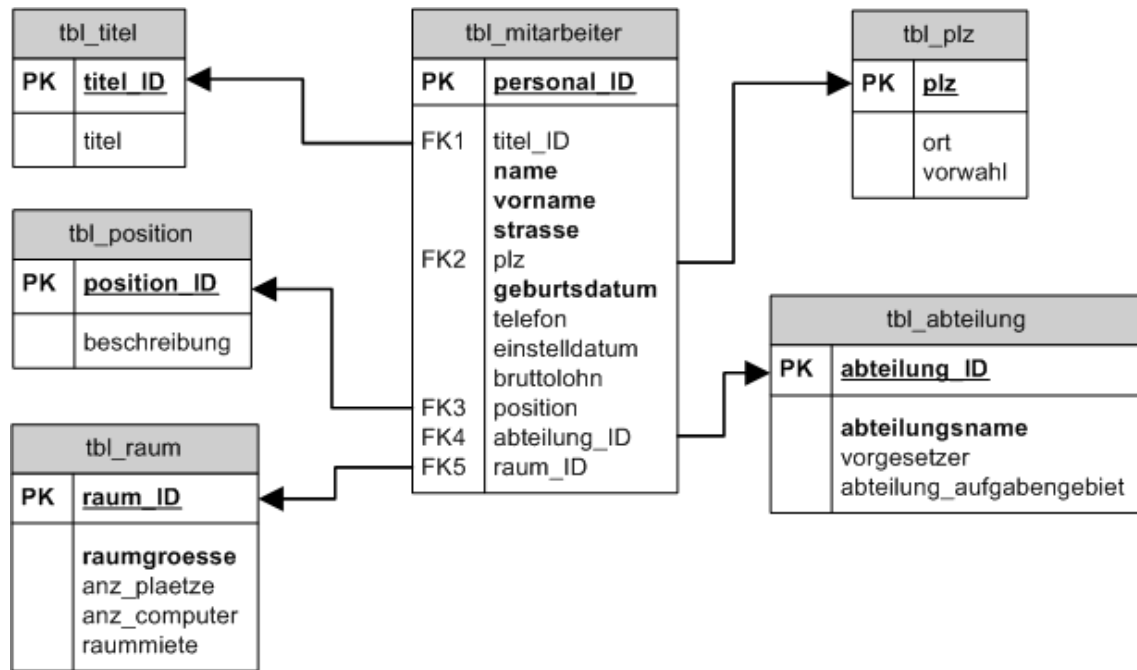


Abbildung 11: SQL-Abhängigkeiten ([@] [https://elearn.inf.tu-dresden.de/sqlkurs/lektion02/02\\_99\\_zusatz\\_beziehungen.html](https://elearn.inf.tu-dresden.de/sqlkurs/lektion02/02_99_zusatz_beziehungen.html))

In Verbindung dazu steht auch die Resilienz eines Datenbank-Systems. Bei *SQL*-Datenbanken muss das vollständige System jederzeit voll einsatzbereit sein. Fällt ein Teil des Systems aus, funktioniert das restliche System auch nicht mehr.

Anders ist es bei *MongoDB*. Fällt hier ein Cluster aus, sind die Daten auf diesem Cluster zwar nicht mehr abrufbar, die Daten auf anderen Clustern jedoch schon (siehe Abbildung 12).

Allgemein unterscheiden sich auch die Transaktionseigenschaften. *SQL*-Datenbanken verwenden das *ACID*-Prinzip, während *MongoDB* das *CAP-Theorem* implementiert.

*SQL*-Datenbanken haben atomare Transaktionen, die in sich Konsistent und isoliert sein müssen. Transaktionen dürfen also nicht andere Transaktionen beeinflussen. Außerdem muss eine Aktion vollständig oder gar nicht durchgeführt werden. Nachdem eine Transaktion ausgeführt wurde, muss das System die Veränderungen persistent speichern, sodass die Daten auch bei einem Systemfehler erhalten bleiben.

Das *CAP-Theorem* legt stattdessen mehr Wert auf die Zuverlässigkeit des Systems. Daten müssen immer gleich sein, egal aus welchem Cluster sie abgerufen werden. Die Zuverlässigkeit des Systems ist sehr wichtig, sodass bei einem Teilausfall das System trotzdem weiterhin erreichbar sein soll.

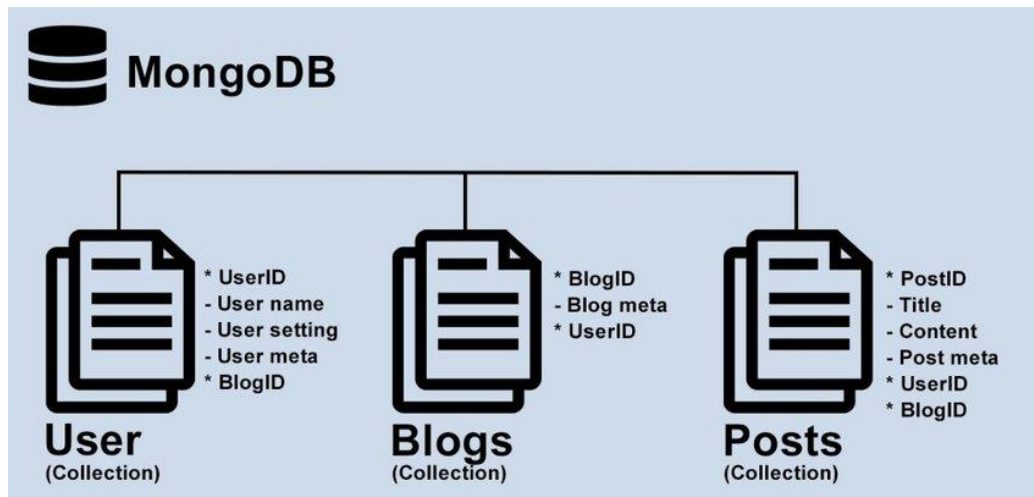


Abbildung 12: MongoDB-Beziehungen ([@] [https://www.researchgate.net/figure/Abbildung-3-Aufbau-der-MongoDB-MongoDB-stellte-einige-Methoden-bereit-um-Relationen\\_fig3\\_321797933](https://www.researchgate.net/figure/Abbildung-3-Aufbau-der-MongoDB-MongoDB-stellte-einige-Methoden-bereit-um-Relationen_fig3_321797933))

#### 4.1.2 Auswahl der Datenbanken

Autor Jannis Welkener

Computerteildaten sind unabhängig voneinander und daher werden nur geringe Relationen benötigt. Außerdem ist es wichtig, dass die Daten immer erreichbar sind und es einfach ist, die Speicherkapazität zu erhöhen, da jedes Jahr neue Teile entwickelt und veröffentlicht werden. Da es nur einen Admin-Nutzer gibt, ist es auch unwahrscheinlich, dass es beim Verändern von Daten zu Race-Konditionen kommt. Demnach sind *ACID*-Eigenschaften vernachlässigbar. Außerdem kann es bisweilen viele Aufrufe auf die Datenbank geben, weshalb eine große Resilienz benötigt wird.

Demnach ist es sinnvoll, keine *SQL*-Datenbank für das Speichern der Hauptdaten der *API* zu verwenden. *MongoDB* erfüllt alle Eigenschaften, die für dieses System erwünscht sind, und wird deswegen in diesem Projekt verwendet.

Anders ist es bei den Nutzerdaten. Diese müssen nicht zwangsweise immer erreichbar sein, da es einige Methoden gibt, bei denen das Einloggen nicht nötig ist. Ein großer Teil des Systems funktioniert also trotzdem weiter. Sollte es zu Problemen mit den Nutzerdaten kommen, ist es auch sinnvoll, dass alle Nutzer ausgeschlossen werden, da möglicherweise Admin- und Moderatoraccounts betroffen sein können, während böswillige Nutzer im System Chaos verursachen können. Die gespeicherten Daten zu einem beliebigen Nutzer sind ebenfalls sehr klein, weshalb es zu keinen Speicherplatzproblemen kommen sollte.

Aus diesen Gründen ist eine SQL-Datenbank ideal, um die Nutzerdaten dieses Projektes zu speichern und zu verwalten.

### 4.1.3 Auswahl des Datenbankzugriffsframeworks

*Autor Jannis Welkener*

Um auf die Hauptdatenbank zugreifen zu können, wird eine Zugriffsbibliothek benötigt, die *MongoDB* unterstützt. Hierbei gibt es für Quarkus zwei Standards: *Panache* und *JNoSQL*. Während es *Panache* überwiegend, aber nicht exklusiv für SQL-Datenbanken gibt, ist *JNoSQL* eine Bibliothek für viele NoSQL-Datenbanken.

Bei der Persistierung mit *Panache* muss das zu speichernde Objekt eine Unterklasse des Typs *PanacheEntity* sein. Durch diese Vererbung gibt es Objektmethoden wie *persist()* oder *update()*, mit denen Speicheraktionen auf ein Objekt ausgeführt werden können. Um Daten abzurufen, werden Klassenmethoden von *PanacheEntity*, wie *list()* oder *find()* verwendet. Hier ist es möglich, simple *Queries* zu erstellen. Die *Panache-MongoDB*-Bibliothek verwendet eine spezialisierte Klasse namens *PanacheMongoEntity* und basiert auf manuellen *MongoDB-Client* von *Quarkus*.

Bei *JNoSQL* wird stattdessen ein injiziertes *Repository* für zu speichernde Objekte genutzt. Dieses bietet Methoden zum Speichern und Abrufen an.

Die *Quarkus-JNoSQL*-Bibliothek sind jedoch kaum Dokumentationen zu finden. Die Bibliothek ist für Quarkus nicht sehr verbreitet. Aus diesem Grund wird *Panache-MongoDB* verwendet, da *Panache* die für Quarkus mit meist verbreitetste Persistierungsbibliothek ist und ebenfalls für SQL verfügbar ist. Es existiert sogar eine offizielle Dokumentation auf der Quarkus-Webseite.

Dadurch, dass *Panache* auch für SQL nutzbar ist, kann dasselbe Paradigma auch für die Nutzerverwaltung verwendet werden, sodass die Persistierung im Projekt einheitlich ist. Zusätzlich erlaubt dies, falls nötig die gesamte Speicherung ohne große Veränderung durch eine SQL-Datenbank auszutauschen. Dies macht das Projekt deutlich flexibler und zukunftsorientiert.

## 4.2 Rollenbasierter Zugriff und Nutzerverwaltung

*Autor Jannis Welkener*

Nicht alle Nutzer sollen die Möglichkeit haben, alle Funktionen aufrufen zu können. Außerdem sollen nur angemeldete Benutzer Konfigurationen erstellen können, um sicherzustellen, dass andere Nutzer private Konfigurationen nicht sabotieren können. Hierfür wird eine Nutzerverwaltung mit rollenbasierter Zugriffskontrolle mithilfe der *Security-JPA*-Bibliothek umgesetzt.

Die persönlichen Accountdaten werden hierfür von den Anwendungsdaten getrennt und in einer eigenen *PostgreSQL*-Datenbank verwaltet. Der Zugriff erfolgt dennoch über denselben Zugriffsstandard *Panache*, sodass der Code einheitlich bleibt. Ein Benutzer besitzt einen Nutzernamen, eine Rolle, welche ihm bestimmte Rechte gibt und ein privates Passwort, welches verschlüsselt gespeichert wird (siehe Abbildung 13).

```
@Entity
@Table(name = "test_user")
@UserDefinition
public class User extends PanacheEntity {
    @Username
    public String username;
    @Password
    public String password;
    @Roles
    public String role;

    /**
     * Adds a new user to the database
     * @param username the username
     * @param password the unencrypted password (it will be encrypted with bcrypt)
     * @param role the comma-separated roles
     */
    public static void add(String username, String password, String role) {
        User user = new User();
        user.username = username;
        user.password = BcryptUtil.bcryptHash(password);
        user.role = role;
        user.persist();
    }
}
```

Abbildung 13: User Entität

Wird das Programm initial gestartet, werden Nutzer aus der *insertDefaultUserData.sql*-Datei geladen. Jeder Nutzer besitzt eine der 3 Rollen, welche ihnen unterschiedliche Rechte geben.

Ein Nutzer, der nicht registriert oder angemeldet ist besitzt nur eingeschränkten Zugriff auf die *API*. Er kann alle Teile und öffentliche Computer und zugehörige Kommentare abrufen. Seine Fähigkeiten beschränken sich auf *GET*-Methoden, er kann also keine Daten verändern, außer um sich als Nutzer zu registrieren.

Der Nutzer "User" mit selbigem Passwort besitzt ebenfalls die User-Rolle, mit der er das Recht hat, eigene Computer zu erstellen und andere öffentliche Computer und Teile zu kommentieren.

Der Nutzer "Moderator" besitzt Moderatorrechte, das bedeutet, er hat alle Rechte eines Users und kann ebenfalls unerwünschte Kommentare löschen, um eine zivile Atmosphäre aufrecht zu erhalten.

Der Nutzer "Admin" besitzt alle Rechte und kann zusätzlich Computerteile verwalten, also neue Teile anlegen und löschen, sowie Bilder zu Teilen hinzufügen. Es ist auch möglich, alle

Accounts einzusehen, wobei Passwörter natürlich nicht vom Admin einsehbar sind. Lediglich der Name und die zugeordnete Rolle ist sichtbar.

Mithilfe einer öffentlichen Registrierungsfunktion kann jeder unangemeldete Benutzer einen eigenen Account erstellen, um Nutzerrechte zu erlangen. Hierfür muss er einen Nutzernamen wählen, der noch nicht existiert, sowie ein Passwort festlegen, welches verschlüsselt gespeichert wird.

Der Admin hat auch die Fähigkeit, über einen weiteren Registrierungspfad neue Moderatorenaccount zu erstellen. Demnach kann der Moderatorenrang nur durch Vertrauen durch den Admin erlangt werden und ist somit vor böartigen Personen geschützt.

Der Admin-Account wird nur initial erstellt und existiert nur ein mal. Weitere Admin-Accounts können nicht erstellt werden. Somit ist das Admin-Passwort vollständig geschützt, da nur eine Person es kennt und es ausschließlich über die *insertDefaultUserData.sql*-Datei gesetzt werden kann.

Die Autorisierung erfolgt schließlich mithilfe der Security-JPA. Die Userklasse definiert einen Security-User über die Annotationen *UserDefinition*, *Username*, *Password* und *Roles*. Ein geschützter Pfad muss mit *RolesAllowed* annotiert werden, wo alle erlaubten Rollen für diesen Pfad definiert werden.

Mithilfe des *SecurityContext* kann der Name des angemeldeten Benutzers abgerufen werden. Da dieser einzigartig ist, ist der Nutzer damit vollständig identifizierbar.

### 4.3 Nutzerinteraktion

*Autor Jannis Welkener*

Nutzer haben die Möglichkeit, miteinander durch Kommentare zu interagieren. Diese sind entweder einem Computer oder einem Teil zugeordnet und werden ebenfalls in der *MongoDB* gespeichert. Dies ermöglicht es, gute Teile von schlechten zu unterscheiden und hilfreiches Feedback zu einer Konfiguration zu bekommen, bevor ein Nutzer sich die Teile tatsächlich kauft.

Bei einem solchen System besteht jedoch immer die Gefahr, dass bestimmte Nutzer unerwünschte Dinge schreiben, die möglicherweise anstößig oder anderweitig unangemessen sind. Aus diesem Grund ist es Moderatoren möglich, solche Kommentare zu löschen.

Jeder Kommentar besteht aus einer Bewertung von 1 bis 5 und einem Text. Hierdurch kann die Qualität eingeschätzt und gefiltert werden. So ist es möglich, nur Objekte mit einer Mindestbewertung anzuzeigen.

Kommentare sind zwar einem Objekt zugeordnet, werden aber nicht zusammen gespeichert. Vor dem Hochladen eines Kommentares wird überprüft, ob das zugehörige Objekt existiert. Dadurch ist sichergestellt, dass in Zukunft weitere Objekte mit Kommentaren erweitert werden können. Objekte mit Kommentaren bieten einen *HATEOAS*-Link, um zugehörige Kommentare zu referenzieren. Bei Abruf eines dieser Objekte wird die Durchschnittsbewertung und die Anzahl der Kommentare abgerufen, sodass ersichtlich ist, ob es überhaupt Kommentare gibt und, sofern es sie gibt, wie die aktuelle Bewertung ist.

## 4.4 Bereitstellung von Bilddaten

*Autor Jannis Welkener*

Es ist oftmals schwierig, nur anhand von Daten eine Entscheidung zu einem Computerteil zu treffen. Aus diesem Grund soll es möglich sein, ein Bild für ein Teil hochzuladen. Aus Urheberrechtlichen- sowie aus Datenintegritätsgründen sollte dieses Privileg jedoch nur einem Admin erlaubt sein, welches sich diesen Anforderungen bewusst ist.

Da ein Bild zu einem Computerteil zugeordnet ist, befindet sich der *API*-Pfad im selben Modul. Hier gibt es eine Route zum Setzen eines Bildes, welche nur mit Adminrechten abrufbar ist sowie eine Route, um ein Bild zu einem Teil abzurufen, welche für alle erreichbar ist.

Ein Bild wird innerhalb der ihm zugeordneten Teileentität gespeichert. Dies ist jedoch nicht im Java-typischen *BufferedImage*-Format möglich, da dieses nur für lokales speichern innerhalb der Laufzeit eines Programmes vorgesehen ist. Stattdessen wird die Datei als *InputStream* geladen und in Bytes umgewandelt, welche gespeichert werden. Wenn keine Daten hochgeladen werden, wird das aktuell gespeicherte Bild gelöscht. Um eine komplexe Datei hochladen zu können, wird die *MultipartBody*-Hilfsklasse benötigt, welche einen *InputStream* in der Form eines *Oktetstreams* akzeptiert. Aus diesem lässt sich der tatsächliche *InputStream* auslesen.

Um ein Bild auszugeben, reicht es nicht aus, die Bytes zurückzugeben. Hierfür werden die Bytes der Datei in einen *InputStream* und daraus in ein normales *BufferedImage* umgewandelt. Dies wird jedoch nicht korrekt als Bild übertragen. Die Hilfsklasse *BufferedImageBodyWriter* bietet die Möglichkeit, ein Bild vom Typ *BufferedImage* zum *Client* zu schreiben, sodass das Bild als solches angezeigt wird.

### 4.4.1 Verifizierung der Eingabedateien für Bilder

*Autor Jannis Welkener*

Um zu vermeiden, dass eine falsche Datei geladen und gespeichert wird, muss vor der Persistierung überprüft werden, ob die Datei im richtigen Dateiformat ist. Dies wird in der *isImageFile*-Methode getestet. Standarddateitypen besitzen meist eine so genannte "*Magic-Number*",

einen bestimmten Wert in einem bestimmten Bereich der Datei, an der der Dateityp erkannt werden kann. In der Methode wird überprüft, ob die ersten 8 Bytes der Datei mit der *Magic-Number* der Dateitypen *PNG* oder *JPEG* übereinstimmen. Ist dies der Fall, handelt es sich höchst wahrscheinlich um eine Bilddatei und es ist erlaubt, diese zu speichern (siehe Abbildung 14).

```
private static boolean isImageFile(byte[] fileToBeChecked) {  
    if (fileToBeChecked.length < 8) {  
        return false;  
    }  
  
    byte[] magicNumPNG = {(byte)0x89, (byte)0x50, (byte)0x4E, (byte)0x47, 0x0D, (byte)0x0A, (byte)0x1A, (byte)0x0A};  
    byte[] magicNumJPG = { -1, -40, -1, -32, 0, 16, 74, 70 };  
  
    byte[] inputMagicNumber = new byte[8];  
    for (int i = 0; i < 8; i++) {  
        inputMagicNumber[i] = fileToBeChecked[i];  
    }  
    return Arrays.equals(magicNumPNG, inputMagicNumber) || Arrays.equals(magicNumJPG, inputMagicNumber);  
}
```

Abbildung 14: Validierung der *Magic-Number*

Dies heißt jedoch nicht, dass das System vollständig sicher ist. Ein böartiger Akteur kann die falsche Datei so verändern, dass inkorrekterweise die *Magic-Number* eines anderen Dateityps verwendet wird. Dies ist für dieses System jedoch zu vernachlässigen, da nur ein Benutzer mit höchster Berechtigung überhaupt in der Lage ist, eine Datei hochzuladen. Stattdessen dient die Überprüfung mehr der Nutzbarkeit, damit ein Admin nicht versehentlich eine falsche Datei hochlädt.



## 5 Konfiguration und Laufzeitüberwachung

Autor Daniel Graf

Nachdem nun die Module der Anwendung in ihrer Relevanz und Funktion sowie auch das Persistierungsverfahren beschrieben wurden, wird nun deren Konfiguration & Einstellung in Quarkus näher erläutert.

### 5.1.1 Konfiguration in Quarkus

Autor Daniel Graf

Die Konfiguration der Anwendung wird in der Datei „*application.properties*“ festgelegt. Hier lassen sich wichtige Programmeigenschaften definieren. Auch im Rahmen der *OpenAPI* Spezifikation lassen sich hier Informationen bezüglich der *API* festlegen. So sollte in Bezug auf die Dokumentation und Nutzbarkeit der *API* einige wichtige Informationen hinterlegt sein, wie z. B. die Kontakt-E-Mail, Kontaktnamen, Titel der *API*, Beschreibung sowie aktuell hinterlegte Version der *PC Part API*.

```
# API Infos
quarkus.smallrye-openapi.info-contact-email=daniel.graf@hs-osnabrueck.de
quarkus.smallrye-openapi.info-contact-name=Daniel Graf & Jannis Welkener
quarkus.smallrye-openapi.info-title=PCPart API
quarkus.smallrye-openapi.info-description=PCPart API to get information on various computer parts and
create custom builds.
quarkus.smallrye-openapi.info-version=0.1-dev
quarkus.smallrye-openapi.version=1
quarkus.smallrye-openapi.path=pcpartapi
```

Abbildung 15: Auszug aus "application.properties"

Des Weiteren wird in der Konfigurationsdatei beschrieben, wie die Datenbankcontainer zu erstellen bzw. verknüpfen sind. Wie bereits erläutert, verwaltet die Anwendung ihre Daten innerhalb zwei Datenbankcontainer, welche hier die entsprechenden Verbindungswege gesetzt haben müssen.

### 5.1.2 PostgreSQL Konfiguration

Autor Daniel Graf

Für die Einstellung der PostgreSQL Datenbank für das User-Modul sollten in der Konfigurationsdatei einige Werte definiert und festgelegt sein, die Erklärung der wichtigsten wird in der nachfolgenden Tabelle erläutert:

Tabelle 1: PostgreSQL Einstellungen in "application.properties"

Konfigurationsvariable	Wert	Beschreibung
quarkus.hibernate-orm.database.generation	<i>drop-and-create</i>	Legt fest, wie das Datenbank Tabellen-schema beim Starten der Anwendung zu generieren ist. Mit „ <i>drop-and-create</i> “ wird beim Neustart der Anwendung das vorherige Schema gelöscht und ein neues angelegt. Dies ist hauptsächlich während der Entwicklungszeit sinnvoll und sollte beim finalen <i>Release</i> umgestellt werden.
quarkus.hibernate-orm.log.sql	<i>false</i>	Mit diesem Einstellungswert lassen sich die <i>SQL</i> -Befehle mittels <i>Logging</i> ausgeben. Dies ist während der Entwicklungszeit hilfreich, kann aber auch bei Bedarf im Release angeschaltet werden.
quarkus.hibernate-orm.sql-load-script	<i>META-INF/sql/insertDefaultUserData.sql</i>	Hier kann eine <i>SQL</i> -Datei hinterlegt werden, welche beim Starten der Anwendung ausgeführt wird. Momentan sind in dieser Datei die Standardnutzer hinterlegt, wie <i>User</i> , <i>Moderator</i> und <i>Admin</i> .

### 5.1.3 MongoDB Konfiguration

*Autor Daniel Graf*

Für die Einstellung der *MongoDB*-Datenbank für alle weiteren Module müssen in der Konfigurationsdatei zwingend wichtige Verbindungsvariablen festgelegt werden, damit der in *Docker* verwaltete Datenbankcontainer genutzt wird.

Tabelle 2: MongoDB Einstellungen in "application.properties"

Konfigurationsvariable	Wert	Beschreibung
quarkus.mongodb.connection-string	<i>mongodb://localhost:27017</i>	Legt die <i>MongoDB</i> Verbindung mit dem in <i>Docker</i> verwalteten Container fest.
quarkus.mongodb.database	<i>root-db</i>	Setzt den Datenbanknamen in <i>MongoDB</i> fest.

## 5.2 Logging

Autor Daniel Graf

Das *Logging* ermöglicht das Erfassen und Speichern über den aktuellen Programmzustand und spielt eine entscheidende Rolle, um Probleme während der Laufzeit zu erkennen, analysieren und zu beheben. Damit die Anwendung während ihrer Ausführung nachvollziehbar und überwachbar ist, werden deshalb in wichtigen Programmfunktionen entsprechende *Log*-Nachrichten ausgeführt.

Damit eine Klasse entsprechende *Log*-Nachrichten in die Konsole ausgeben kann, wird ein *Logger*-Objekt instanziiert. Dieses erhält die aufrufende Klasse als Parameter (siehe Abbildung 16). Nun kann an den passenden Stellen dieses Objekt mit dem entsprechenden *Logging*-Level aufgerufen werden, um so eine *Log*-Nachricht auszugeben. Die Nachricht enthält nach der aktuellen *Logging*-Konfiguration im Projekt zum einen einen Zeitstempel, das *Logging*-Level sowie auch *Package*- & *Thread* Informationen bezüglich der *JVM*, auf dem der *Logging*-Aufruf durchgeführt wurde. So lässt sich eindeutig nachverfolgen, wann und wo der Aufruf stattgefunden hat.

```
public class AllComputerPartsResource {  
    private static final Logger log = Logger.getLogger(AllComputerPartsResource.class);  
}
```

Abbildung 16: Instanziierung eines *Logger*-Objektes

Durch ein solches *Logging* lassen sich auch Performance-Messungen durchführen, denn gerade bei den Listenressourcen kann es je nach Größe der Liste zu aufwändigen und leistungsintensiven Sortieroperationen kommen. In der Abbildung erkennt man einen Auszug aus der Listenressource für die Computerkomponenten. Schlussendlich wird dann hier ein Millisekundenwert für die Dauer der Sortieroperationen ausgegeben (siehe Abbildung 17).

```
/* *****  
 * Filter Start  
 * ***** */  
long startTime = System.nanoTime();  
// PartName filtering  
if (partNameFilter != null && !partNameFilter.isEmpty()) {  
    log.info("Filtering by partName");  
    parts = AllComputerPartsResource.filterByPartName(parts, partNameFilter);  
}  
  
// Type Filtering  
if (typeFilter != null) {  
    log.info("Filtering by part type");  
    parts = filterByType(parts, typeFilter);  
}  
  
// Manufacturer name filtering  
if (manufacturerNameFilter != null && !manufacturerNameFilter.isEmpty()) {  
    log.info("Filtering by manufacturer name");  
    parts = filterByManufacturerName(parts, manufacturerNameFilter);  
}  
  
long endTime = System.nanoTime();  
long finalTime = endTime - startTime;  
double benchmarkTime = (double) finalTime / 1000 / 1000;  
log.info("Filtering done! took " + (benchmarkTime) + "ms");
```

Abbildung 17: Filtern einer Listenressource mit Performance-Messung in Millisekunden

Im aktuellen Entwicklungsstand der Anwendung werden für die meisten Methoden kurze *Log*-Nachrichten auf dem Info-Level ausgegeben, um dem Betreiber der Anwendung stets relevante Informationen über die Ausführung des Programms zu liefern. So sieht man in der folgenden Abbildung, dass für eine Anfrage der Listenressource sowie deren *Query*-Parameter auch entsprechende Info-Nachrichten ausgegeben werden (siehe Abbildung 18).

```
2023-07-24 18:22:32,931 INFO [de.hso.swe.pro.Com.bou.rs.AllComputerPartsResource] (executor-thread-1) Getting all parts  
2023-07-24 18:22:32,932 INFO [de.hso.swe.pro.Com.bou.rs.AllComputerPartsResource] (executor-thread-1) Filtering by partName  
2023-07-24 18:22:32,933 INFO [de.hso.swe.pro.Com.bou.rs.AllComputerPartsResource] (executor-thread-1) Filtering by part type  
2023-07-24 18:22:32,934 INFO [de.hso.swe.pro.Com.bou.rs.AllComputerPartsResource] (executor-thread-1) Filtering by manufacturer name  
2023-07-24 18:22:32,935 INFO [de.hso.swe.pro.Com.bou.rs.AllComputerPartsResource] (executor-thread-1) Filtering done! took 3.0507ms
```

Abbildung 18: Konsolenausgabe bei der Abfrage einer Listenressource

## 6 Anwenden der PC Part API

Autor Daniel Graf

Im folgenden Kapitel werden die Aspekte der Anwendung während der Nutzung & Anwendung der API geklärt. So werden nachfolgend die implementierten Maßnahmen bezüglich der Nutzerfreundlichkeit erläutert.

### 6.1 Usability & Schemata

Autor Jannis Welkener

Zur einfachen Nutzung der API von unabhängigen Entwicklern werden Schemata für alle *DTOs* angeboten. Diese sind über *OpenAPI* und *SwaggerUI* einsehbar und besitzen genaue Beschreibungen und Beispiele für korrekte Eingaben. Dies geschieht über Annotationen auf Klassen- und Variablenebene (siehe Abbildung 19).

Die Eingaben werden im Rahmen der *Input-Validation* zusätzlich überprüft.

```
@Schema(description = "Schema for adding a new RAM-module to the database")
public class RamDTO extends PcPartDTO{

    @Schema(description = "The amount of RAM-sticks of this module (e.g. 2-4)", required = true, example = "2")
    @NotNull
    @Min(value = 1) @Max(value = 8)
    public int amountOfSticks;

    @Schema(description = "The storage-capacity per stick in GB (e.g. 2"x)", required = true, example = "16")
    @NotNull
    @Min(value = 2) @Max(value = 128)
    public int sizePerStickInGB;

    @Schema(description = "The RAM-type (e.g. 'DDR3' / 'DDR4')", required = true, example = "DDR4")
    @NotNull
    @Size(min = 2, max = 6)
    public String ramType;

    @Schema(description = "The total storage-capacity in GB (amountOfSticks * sizePerStickInGB)", required = true, example = "32")
    @NotNull
    @Min(value = 2)
    public int totalSizeInGB;

    @Schema(description = "The storage-speed of the RAM-module in MHz (e.g. 3600 MHz)", required = true, example = "3600")
    @NotNull
    @Positive
    public int ramSpeedInMHz;

    @Schema(description = "Defines whether the RAM-Module has RGB-lights for aesthetics", required = true, example = "true")
    @NotNull
    public boolean hasRGB;
}
```

Abbildung 19: *Data Transfer Object* für die Computerkomponente: RAM

### 6.2 Input Validation

Autor Daniel Graf

Ein zentraler Aspekt bezüglich der nutzerfreundlichkeit als auch Sicherheit der Anwendung ist die *Input-Validierung*. Diese zielt darauf hinaus, an sogenannten *Trust Boundaries* die übergebenen Daten zu überprüfen und zu validieren.

*Trust Boundaries* definieren Grenzen, an denen vertrauliche Daten von einem Endpunkt zum anderen gelangen. Ein solcher Übergang ist zum Beispiel, wenn ein Nutzer Ressourcen von der API mittels *Input-DTO's* abfragt. Die dort übergebenen Daten werden über entsprechende Schemas und *Data Transfer Objects* Konvertierungen in der Anwendung validiert und für die Weiterverarbeitung bestätigt.

So wird beispielsweise sichergestellt, das übergebene ID's immer eine Länge von 24 haben und die Zeichen die einer 128-bit *UUID* entsprechen. Dazu wird ein *Regular Expression* Validierung durchgeführt (siehe Abbildung 20).

```
@Schema(description = "Fan Id of which to update the computer configuration with", example = "")
@Nullable
@Pattern(regexp = "[0-9a-fA-F]{24}")
public String fanId = null;
```

Abbildung 20: *Input Validation* für eine ID innerhalb einer *DTO*

Um die nutzerfreundlichkeit weiter zu erhöhen, sind diese Input-Restriktionen in den *DTO*'s der Anwendung dokumentiert und spezifiziert. Damit lassen sich diese Eingaberestriktionen auch in der graphischen Oberfläche von SwaggerUI anzeigen.

### 6.3 Antwortnachrichten

Autor Daniel Graf

Im Rahmen der Nutzerfreundlichkeit sind angemessene Rückgabewerte für die Anwendung wichtig und erleichtern das Auslesen der Daten sowie auch die Fehlerbehandlung.

Im Falle, dass das Eingabeschema nicht korrekt, teilweise oder gar nicht ausgefüllt ist, wird dank der Input-Validation eine entsprechende Fehlermeldung zurückgegeben, welche klar spezifiziert, welche Parameter fehlerhaft ausgefüllt wurden.

Ansonsten werden die Daten für den Pfad entsprechend zurückgegeben. Dazu wird bei einigen Pfaden auch eine Rückgabe nach dem *HATEOAS* Format zurückgegeben. Diese enthält dann Informationen bezüglich der angeforderten Ressource, als auch relevante Links zu weiteren Ressourcenbeziehungen. In der Abbildung sieht man beispielsweise die Antwort auf den Pfad „*GET /computers/private*“ mit den entsprechenden Daten der Computerkonfiguration, als auch einen Link zu den dazugehörigen Kommentaren & Reviews. Das Schema für das *HATEOAS*-Format ist für die Anwendung in einem *shared* Ordner definiert, so dass alle Module die gleichen Rahmenanforderungen für dieses Format haben.

```
{
  "data": {
    "commentAmount": 7,
    "id": "64b72cb540bad31dc9457f76",
    "isPublic": true,
    "pcName": "\\AwesomeNewComputerName\\",
    "pcOwner": "admin",
    "userRating": 4
  },
  "links": {
    "related": "http://localhost:8080/computers/public/64b72cb540bad31dc9457f76",
    "self": "http://localhost:8080/computers/private/64b72cb540bad31dc9457f76"
  },
  "relationships": {
    "comments": {
      "related": "http://localhost:8080/computers/64b72cb540bad31dc9457f76/comments"
    }
  }
}
```

Abbildung 21: Erhaltene Antwort auf den Pfad *"/computers/private"* mit *GET* im *HATEOAS* Format

## 6.4 Filtering & Queries für Listenressourcen

Autor Daniel Graf

Da die Anwendung manche Ressourcen als Liste anbietet, ist es vorteilhaft, für diese Listenressourcen eine Filterung anzubieten. Diese Filterung lässt sich per *Query*-Parameter anfragen. In der folgenden Abbildung sieht man, dass man die Listenressource mit relevanten Filtermöglichkeiten erhalten kann. Auch kann man die Liste mittels *Page* & *PageSize* in ihrer Größe anpassen (siehe Abbildung 22).

The screenshot displays the API interface for the endpoint `GET /pc-parts/all-parts` with the description "Retrieve all computer components". Below the endpoint information, there is a "Parameters" section with a "Try it out" button. The parameters are listed in a table with two columns: "Name" and "Description".

Name	Description
hasImageFilter boolean (query)	--
manufacturerNameFilter string (query)	manufacturerNameFilter
page integer(\$int32) (query)	page
pageSize integer(\$int32) (query)	pageSize
partNameFilter string (query)	partNameFilter
typeFilter string (query)	Available values : CPU, RAM, GPU, PSU, MOTHERBOARD, FAN, CASE --

Abbildung 22: Listenressource für Computerkomponenten mit Filteroptionen



## 7 Programmresistenz & Fehlertoleranz


Autor Daniel Graf

Abschließend werden die Maßnahmen zur Erhöhung der Fehler- & Störungsresistenz der Anwendung erläutert.

### 7.1 Fault Tolerance

Autor Daniel Graf

Um die Anwendung in seiner Funktion weiter abzusichern und fehlertoleranter zu machen, wird für die angebotenen REST-Endpunkte & deren interne Methoden eine *Fault Tolerance* implementiert. Diese werden per Annotation an die Methoden angefügt] (siehe Abbildung 23) [`@FaultTolerance`].



```

@POST
@Authenticated
@Path("/{id}/visibility")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
@Operation(summary = "Set to 'public' or 'private' to change if other users can see the computer configuration")
@APIResponse(responseCode = "400", description = "Invalid ID")
@Timeout(value = 2500)
@CircuitBreaker(requestVolumeThreshold = 10)
http://localhost:8080/computers/private/{id}/visibility
public Response setPCtoPublic(@Context SecurityContext securityContext,
    @PathParam("id") @Valid @Pattern(regexp = "[0-9a-fA-F]{24}") String id,
    @Valid VisibilityInputDTO visibility) {

```

Abbildung 23: REST-Endpoint Methode mit Fault Tolerance Annotationen

Die Implementierung von *Fault Tolerance* mit beispielsweise *Retry* und *Timeout* einer *REST-API* spielen gerade im realen Betrieb eine große Rolle, um die Anwendung gegenüber Fehler und Störungen resistenter zu machen.

Denn gerade in realen Betriebsumgebungen können Störungen und Netzwerkprobleme, Ressourcenengpässe oder anderweitige Systemstörungen dazu führen, dass das Programm in seiner Funktion gehindert wird. Eine Kombination aus angemessenen *Fault Tolerance* Annotationen soll da entgegenwirken und die Verfügbarkeit der Anwendung erhöhen.

#### 7.1.1 Retry

Mit der *Retry*-Annotation wird festgelegt, wie oft eine Methode ihre Funktion im Falle eines Fehlers erneut durchführt. Somit können vorübergehende Fehler bewältigt werden, indem die Methode oder Funktion wiederholt wird, um zum gewünschten Ergebnis zu kommen. Ein typisches Szenario für diese Annotation ist, wenn die Methode mit einer externen Ressource interagiert.

Dies könnte zum Beispiel eine Datenbank sein oder das Abfragen von Informationen aus einem anderen Modul. Durch Netzwerk- oder andere Probleme könnte es hier nämlich dazu kommen, dass die abgefragten Daten nicht verfügbar sind oder zu lange brauchen. In einem solchen Fall, wird dann versucht die Anfrage erneut zu senden, um so die Chance auf einen erfolgreichen Methodenabschluss zu erhöhen.



### 7.1.2 Timeout:

Mit der *Timeout*-Annotation lässt sich festlegen, dass nach einer bestimmten Zeit die aufgerufene Funktion einen *Timeout*-Fehler auswirft. Dies ist grundsätzlich für die meisten Pfade sinnvoll, da so im Falle einer Serverüberlastung die Pfade einen *Timeout*-Fehler erzeugen, um so wieder Ressourcen im Server zu befreien.

Grundsätzlich funktioniert die *Timeout* Annotation so, dass wenn nach der festgelegten Zeit die Methode immer noch nicht abgeschlossen ist, ein *Timeout*-Fehler erzeugt wird. Dies ist zum Beispiel der Fall, wenn externe Ressourcen aus anderen Modulen zu lange auf sich warten lassen oder anderweitig blockiert sind.

## 8 Unit- & Integration Testing

*Autor Daniel Graf*

Damit die Anwendung nicht nur im Rahmen seiner angebotenen Funktionen nutzbar ist, sondern sich auch korrekt verhält, ist das Einbinden von Testing-Elementen im Rahmen der Entwicklung unvermeidlich. In diesem Kapitel werden die Konzepte und Ziele der Testarten näher erläutert.

### 8.1.1 Testarten

*Autor Daniel Graf*

Es wird im Rahmen des Testing zwischen zwei hauptsächlichen Testarten unterschieden, welche im Zusammenspiel einzelne und konkrete Funktionen, als auch das System als Ganzes auf Korrektheit bezüglich der Anforderungskriterien überprüfen.

Um Testklassen in Quarkus erstellen zu können ist grundsätzlich die `@QuarkusTest` Annotation von Bedeutung. Mit dieser lässt sich eine Klasse für das Testing markieren. Um einen API-Endpoint zu testen, wird die `@TestHTTPEndpoint` Annotation genutzt, mit der gezielt spezifische Endpunkte getestet werden können (siehe Abbildung 24) [`@QuarkusTest`].

```
@QuarkusTest
@TestHTTPEndpoint(PrivateComputersResource.class)
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
public class PrivateComputersTest {
    private static final Logger log = Logger.getLogger(PrivateComputersTest.class);
    private final Jsonb jsonb = JsonbBuilder.create();

    @Test
    @Order(1)
    public void shouldNotCreatePrivateComputerWhenAnonymous() {
        String newConfigurationName = "MyPc1";
        Response res = given()
            .contentType(ContentType.JSON)
            .body(newConfigurationName).put("/create-new-pc");
        assertEquals(Status.UNAUTHORIZED.getStatusCode(), res.getStatusCode());
    }
}
```

Abbildung 24: Testklasse für private Computerkonfigurationen

### 8.1.2 Unit Tests

*Autor Daniel Graf*

Unit Tests konzentrieren sich auf das Testen von einzelnen und spezifischen Funktionalitäten. Sie befinden sich deshalb meistens auf der Ebene der Klasse oder Methode und überprüfen somit, ob konkrete Funktionen im Programmcode sich richtig verhalten. So wird mit ihnen beispielsweise überprüft, ob eine Methode einen korrekten Rückgabewert zurückgibt oder getestet, ob in bestimmten Fällen auch eine Exception ausgelöst wird.

### 8.1.3 Integration Testing:

*Autor Daniel Graf*

Das Integration Testing zielt darauf hinaus, das gesamte System in seinen Modulen zu überprüfen. So soll hier eine Kommunikation und Funktion zwischen den verschiedenen Modulen getestet werden. Zum Beispiel wird die Kommunikation zwischen dem "PC-Konfigurationen" und dem "PC-Komponente" Modul getestet. So kann im Laufe der Entwicklung sichergestellt werden, dass diese Module nicht nur für sich alleinstehend korrekt funktionieren, sondern auch in Kombination reibungslos zusammenarbeiten.

### 8.1.4 Testing – Musskriterien

*Autor Daniel Graf*

Ein zentraler Aspekt des Testing, ist die Sicherstellung der im Proposal besprochenen Programmanforderungen. Daher sollten die Musskriterien bei der Erstellung der Testfälle dringend beachtet werden. Die Musskriterien (abgekürzt) lauten wie folgt:

- Registrierung auf der Plattform
- Abfragen von Pc-Komponenten
- Speichern einer PC-Konfiguration
- PC-Konfiguration öffentlich oder privat setzen
- PC-Konfigurationen abfragen
- Neue Teile einreichen

### 8.1.5 Testing – Wunschkriterien

*Autor Daniel Graf*

Auch die Wunschkriterien spielen im Testing eine große Rolle, weshalb für diese Kriterien auch Testfälle erstellt werden. So kann sichergestellt werden, dass auch diese Programmfunktionen nach dem Proposal korrekt implementiert und bereitgestellt werden. Diese lauten (abgekürzt) wie folgt:

- Bewertungen einer Komponente einsehen
- Bewertungen einer PC-Konfiguration einsehen
- Filter & Query Funktion für Listenressourcen

### 8.1.6 Testing Sicherheitsfunktionen

*Autor Daniel Graf*

Da die Anwendung eine Rollenbasierte Zugriffskontrolle implementiert, ist diese im Rahmen des *Testing* auch zu überprüfen. So muss überprüft werden, dass bestimmte Pfade nur von den richtigen Nutzerrollen sinngemäß ausgeführt werden dürfen.

Diese können in Quarkus mit einer *TestSecurity* Annotation getestet werden. Hier wird festgelegt, wie der Nutzernamen lautet und welche Rolle dieser Nutzer besitzt. Hier kann dann über die *HTTP*-Antwort überprüft werden, ob auf die Ressource zugegriffen werden darf, oder nicht. Bei einem erlaubten Zugriff sollte der Statuscode einem Wert von 200 (OK) entsprechen. Ein beispielhafter Test wird in der folgenden Abbildung dargestellt (Abbildung 25).

```
@Test
@Order(4)
@TestSecurity(user = "admin", roles = { "admin" })
void shouldAccessAdminWhenAdminAuthenticated() {

    given()
    .get("/users/admin-commands/get-users")
    .then()
    .statusCode(Status.OK.getStatusCode());
}
```

Abbildung 25: Testing - Zugriffskontrolle

## 9 Zusammenfassung und Fazit

*Autor Daniel Graf*

Abschließend werden in diesem Kapitel die Entwicklungsergebnisse der *PC Part API* zusammengefasst und diskutiert.

### 9.1 Zusammenfassung

*Autor Daniel Graf*

Zusammenfassend lässt sich sagen, dass die Entwicklungsschritte einer nutzerfreundlichen Computerkomponenten API erfolgreich gelungen sind. Mithilfe eines angemessenen *Domain-Driven-Design* Modellierungsansatzes konnte ein skalierbares und wartbares System- & Modulschema entworfen werden, welches mithilfe von passenden Datenbank-Container ihre Daten in angemessener Form speichert und verwaltet.

Dank der in Quarkus bereitgestellten Funktionalitäten und Erweiterungen konnten Qualitätssicherungsverfahren wie *Testing & Logging* erfolgreich konfiguriert und umgesetzt werden.

### 9.2 Fazit

*Autor Daniel Graf*

Die entwickelte API zur Verwaltung von Computerteilen bietet eine solide Grundlage für den Umgang mit Computerteilen, Computerkonfigurationen und Bewertungen. Die Verwendung von bewährten Designansätzen wie dem Domain-Driven-Design hat die Struktur als auch Wartbarkeit der Anwendung deutlich verbessert. Auch die Entscheidung, MongoDB für die Persistierung der Computerteile hat sich nach einer passenden Konfiguration als vorteilhaft bewährt.

Die Nutzerverwaltung wurde hinsichtlich des Proposal angemessen implementiert und bietet im finalen Produkt die Grundlage für eine Rollenbasierte Zugriffskontrolle für die Ressourcen.

Weitere Entwicklungen bzw. Features können in der Zukunft dank des grundlegenden Aufbaues der hier entwickelten Software unkompliziert eingefügt werden.

Abschließend lässt sich sagen, dass dank der umgesetzten Maßnahmen und Technologien eine gut strukturierte und benutzerfreundliche API entwickelt wurde, welche den Anforderungen des Proposal gerecht ist.

## 10 Referenzen

[@FaultTolerance] Quarkus Fault Tolerance Dokumentation, <https://quarkus.io/guides/smallrye-fault-tolerance>, zuletzt zugegriffen am 24.07.2023

[@Quarkus] Quarkus Guide, <https://quarkus.io/get-started/>, zuletzt zugegriffen am 24.07.2023

[@QuarkusTest] Quarkus Testing Guide, <https://quarkus.io/guides/getting-started-testing>, zuletzt zugegriffen am 24.07.2023

## Eidesstattliche Erklärung

Hiermit erkläre ich/ erklären wir an Eides statt, dass ich / wir die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt habe / haben. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche einzeln kenntlich gemacht. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bissendorf, 23.07.23

Ort, Datum

Osnabrück, 23.07.23

Ort, Datum

J. Wimmer

Unterschrift

D. Gölz

Unterschrift

## Urheberrechtliche Einwilligungserklärung

Hiermit erkläre ich/ Hiermit erklären wir, dass ich/wir damit einverstanden bin/sind, dass meine/ unsere Arbeit zum Zwecke des Plagiatsschutzes bei der Fa. Ephorus BV bis zu 5 Jahren in einer Datenbank für die Hochschule Osnabrück archiviert werden kann. Diese Einwilligung kann jederzeit widerrufen werden.

Bissendorf, 23.07.23

Ort, Datum

Osnabrück, 23.07.23

Ort, Datum

J. Wimmer

Unterschrift

D. Gölz

Unterschrift