ARC-42
Dokumentation für das Softwareprojekt:

Kantinerado 

Gruppe:	Gruppe 5
Namen:	Thomas Kircher, Marcel Hasselberg, Michael Schick, Niklas Schmidt
Modul:	Software-Engineering 1
Studienjahrgang:	INF2022

1.1 Qualitätsziele

Funktionale Anforderungen:

Optional:	Anforderung	Beschreibung
Nein	Bestellzusammenfassungen	Überblickseiten mit denen: → Hungernder seine Bestellungen sieht → Kantinenmitarbeiter die bestellten Gerichte sieht
Nein	Registrierungsmöglichkeit	Benutzer können sich als Hungernde registrieren und der Admin kann noch Benutzer mit anderen Rollen anlegen.
Nein	Bestellmöglichkeit	Benutzer sollen die Möglichkeit haben basierend auf dem Speiseplan Gerichte zu bestellen.
Nein	Speiseplan	Ein Speiseplan muss angelegt und abgefragt werden können.
Nein	Rollenbasiertes Zugriffssystem	Je nach der Rolle hat man andere Funktionalitäten
Ja	Kalenderintegration für Speisekarten	Einbindung eines Kalenders zur Anzeige von Speisekarten.
Ja	Filterfunktionen für spezielle Ernährungsbedürfnisse	Möglichkeit zur Filterung von Gerichten nach bestimmten Kategorien wie vegetarisch, vegan oder Allergien.
Nein	Rollenbasiertes Zugriffssystem	Implementierung eines Systems, das unterschiedliche Zugriffsrechte je nach Benutzerrolle gewährt (Mitarbeiter, Professoren, Studenten etc.).
Ja	QR-Code-Funktionalität für Buchung und Abholung	Bereitstellung von QR-Codes, um Gerichte zu buchen oder abzuholen.
Nein	Monatliche oder Wöchentliche Übersichten für Nutzer	Anzeige von Übersichten über die eigenen Bestellungen auf monatlicher oder wöchentlicher Basis für die Nutzer.
Nein	Übersicht über alle Bestellungen für Mitarbeiter	Bereitstellung einer zentralen Übersicht für Mitarbeiter, um alle Bestellungen einzusehen.
Ja	Menüvorschläge	Angebot von Menüvorschlägen basierend auf den Präferenzen und dem Feedback der Benutzer.

Optional:	Anforderung	Beschreibung
Nein	Bestellzusammenfassungen	Überblickseiten mit denen: → Hungernder seine Bestellungen sieht → Kantinenmitarbeiter die bestellten Gerichte sieht
Nein	Registrierungsmöglichkeit	Benutzer können sich als Hungernde registrieren und der Admin kann noch Benutzer mit anderen Rollen anlegen.
Nein	Bestellmöglichkeit	Benutzer sollen die Möglichkeit haben basierend auf dem Speiseplan Gerichte zu bestellen.
Nein	Speiseplan	Ein Speiseplan muss angelegt und abgefragt werden können.
Nein	Rollenbasiertes Zugriffssystem	Je nach der Rolle hat man andere Funktionalitäten
Ja	Benachrichtigungen bei neuen Menüplänen	Benachrichtigungen für Benutzer über neue oder aktualisierte Speisepläne.

Nicht funktionale Anforderungen:

Anforderung	Beschreibung	Messwert
Sprachauswahl	Möglichkeit für Benutzer, die Sprache der Benutzeroberfläche auszuwählen.	Mindestens 2 sprachig
Barrierefreie Nutzung mit Screen Readern ermöglichen	Integration von Funktionen, die eine Nutzung des Systems durch Screen Reader ermöglichen.	Die Applikation kann grundsätzlich mit einem Screenreader verwendet werden
Vollständig responsives Design	Gewährleistung, dass die Anwendung auf verschiedenen Geräten und Bildschirmgrößen optimal dargestellt wird.	- Mobile First umgesetzt - Anwendung auf 3 verschieden Geräten getestet
Hohe Kontraste / Farbwahl	Unterstützung für hohe Kontraste und gegebenenfalls die Möglichkeit, Farben anzupassen, um die Zugänglichkeit zu verbessern.	Kontraste sollen für 90% der Menschen ausreichend sein

Anforderung	Beschreibung	Messwert
Keine Lizenzkosten	Gewährleistung, dass die Anwendung keine Komponenten benutzt die kosten.	
Betreibbarkeit in einem Container	Möglichkeit, die Anwendung in einem Container-Umgebung zu betreiben, um die Portabilität und Skalierbarkeit zu verbessern.	Auf mindestens 2 Plattformen getestet
Sicherheit	Implementierung von Sicherheitsmechanismen	- Alle kritischen API Routen abgesichert - Autorisierung/ Authentifizierung - Passwörter verschlüsselt gespeichert -Datenbank nicht von außen erreichbar
Zuverlässigkeit	Sicherstellung, dass die Anwendung robust ist und Fehler darstellt	- Verfügbarkeit von min. 95% - Fehler werden ausgegeben
Dokumentation und Wartbarkeit im Backend	Bereitstellung einer ausführlichen Dokumentation und gut strukturierter Codebasis, um die Wartbarkeit des Backends zu gewährleisten.	- Stukturierter Code - Dokumentation - Änderungen werden mit CI/CD Pipeline in die Produktion geladen

Stakeholder

Rolle	Kontakt	Erwartungshaltung
Product Owner	*Markus Völk*	*funktionsfähige Software, strukturiertes Vorgehen*
Developer	*Marcel Hasselberg*	*Gute Wartbarkeit/Strukturierung*
Hungernde	-	*Reibungslose/Intuitive Essensbestellung*
Kantinenmitarbeiter	-	Automatisierte Reibungslose Essenbestellung

Randbedingungen

Organisatorisch:

- Zeitraum von 2 Monaten zur Entwicklung
- Zeitliche Einschränkungen des Entwicklerteams

Technisch:

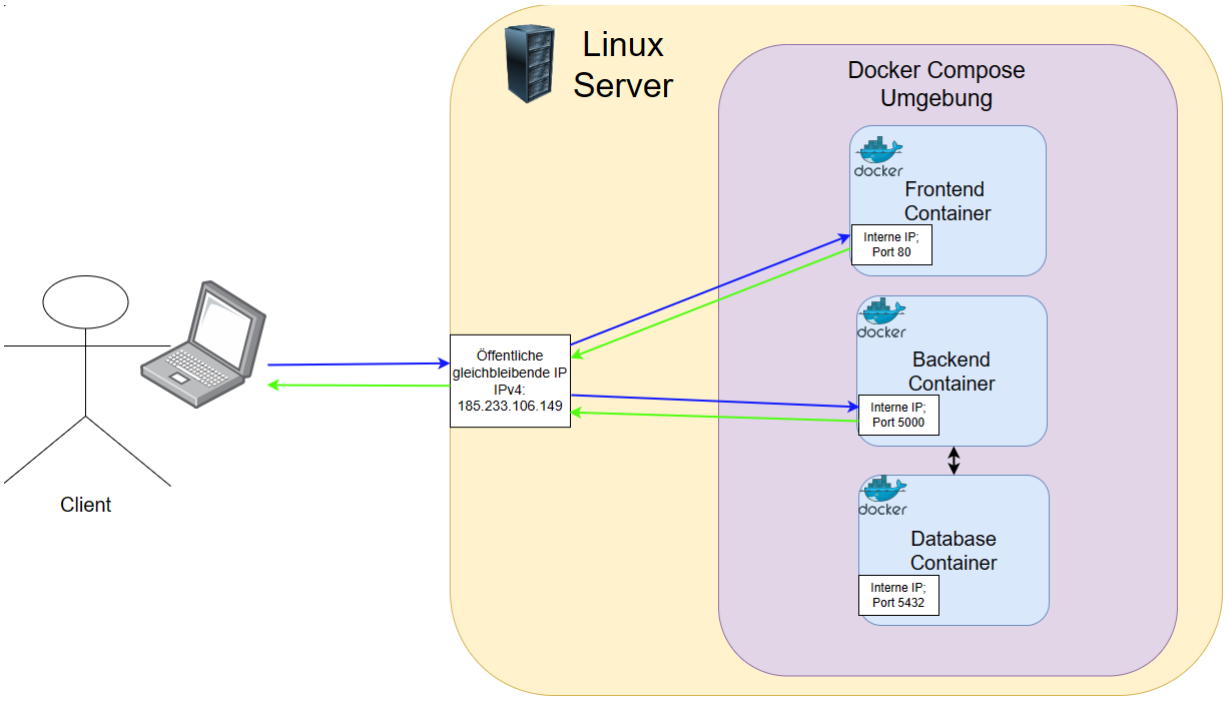
- WebApp (Desktop und Smartphone)
- Bestellung bis Donnerstag 18:00 Uhr möglich
- Öffentlicher Zugriff möglich

Kontextabgrenzung

Fachlicher Kontext

Die Software soll keine Möglichkeit zur Zahlung der Bestellungen haben.

Technischer Kontext



Die gesamte Anwendung läuft auf einem Linux Server, der eine öffentliche IP-Adresse hat. Auf diesem läuft dann ein Docker Compose Netzwerk, in welchem sich die drei Container befinden. Hierbei werden nur der Frontend und der Backend Container von außen erreichbar gemacht, indem ihre Ports an die öffentliche IP-Adresse weitergeleitet werden. Im Backend Container befinden sich die API mit welcher auf die Datenbank zugegriffen

wird. Deshalb ist es nicht nötig den Datenbank Container nach außen sichtbar zu machen, denn innerhalb des Docker Compose Netzwerks können alle Container untereinander kommunizieren. So ist es möglich das über die API im Backend Container Daten von der Datenbank geholt und geändert werden können.
Der Client holt sich also vom Frontend Container die benötigten Ressourcen wie die Benutzeroberfläche. Wenn der Client dann Daten von der Datenbank anfordert, werden die Datenbankinhalte, über die API die im Backend Container definiert ist bereitgestellt.

Beispiel anhand des logins:

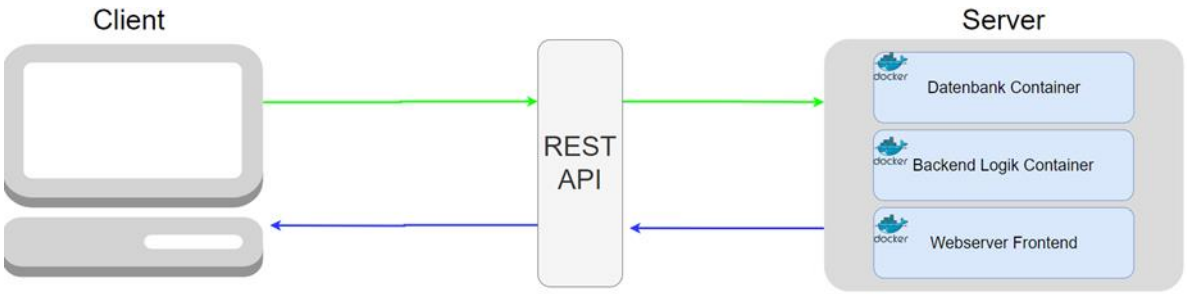
1. Client-Anfrage: Der Benutzer ruft die Login-Seite auf (IP/login)
2. Der Frontend Container empfängt die Anfrage und gibt dem Client die Login Oberfläche zurück
3. Eingabe von Anmeldeinformationen: Der Benutzer gibt seine Anmeldeinformationen ein und sendet das Formular ab.
4. POST-Anfrage an Backend: Der Browser des Clients sendet eine POST-Anfrage mit den Anmeldeinformationen an die API im Backend-Server.
5. Backend-Verarbeitung: Der Backend-Server überprüft die Anmeldeinformationen und führt die Authentifizierungslogik durch, indem er auf die Datenbank zugreift und prüft, ob der Nutzer mit diesem Passwort existiert.
6. Der Backend-Server sendet eine Antwort an den Client. Diese beinhaltet bei erfolgreichem Login einen JWT-Token mit welchem der Client bei folgenden Anfragen erkannt wird und sich so auf andere Routen autoriesen kann.

Lösungsstrategie

Deployment:

1. Verbindung auf den Server mittels ssh
2. Aktuellen Stand mit git pull ziehen
3. Umgebungsvariablen in .env Datei setzen
4. Die Docker Compose Umgebung neu starten

Architektur- /Entscheidungen:



Clientseite:

Auf dem Client werden folgende Komponenten betrieben:

- Die Weboberfläche, die dem Benutzer die Interaktion mit dem System ermöglicht.
- Die Frontend-Logik, die die Benutzerinteraktion steuert. Dazu gehört die Verarbeitung von Aktionen wie das Zusammenfassen von Gerichten im Warenkorb zu einer Bestellung und deren Übermittlung als JSON an den Server. Ebenso wird die Verwaltung von Benutzereinstellungen und die Anforderung von Bestellhistorien hier abgewickelt.

Die Entscheidung, die Weboberfläche und die Frontend-Logik auf dem Client zu betreiben, bietet den Benutzern eine reaktionsschnelle und benutzerfreundliche Benutzererfahrung.

Serverseite:

Die Serverkomponenten sind in Docker-Containern organisiert, um eine einfache Austauschbarkeit und Bereitstellung zu gewährleisten. Dies ermöglicht die Ausführung der Container auf verschiedenen Arten von Servern. Außerdem erleichtert es dem Entwicklerteam die agile Entwicklung der Anwendung.

Kommunikation zwischen Containern: Die genaue Kommunikationsstrategie zwischen den Containern wird noch definiert. Eine potenzielle Technologie zur Verwaltung des Container-Netzwerks und zur Kontrolle der Container ist Docker Compose. Hiermit können Container orchestriert und verwaltet werden.

Serveranwendungen: Auf dem Server laufen folgende Anwendungen in separaten Containern:

- Datenbank-Container: Enthält die PostgreSQL-Datenbank und ihre Daten.
- Backend-Logik-Container: Hier befindet sich die gesamte Backend-Logik, die für die Verarbeitung von Anfragen vom Client und die Bereitstellung von Daten zuständig ist.
- Webserver-Frontend-Container: Ein Container, der einen Webserver ausführt auf welchem Angular gehostet wird. Dies ist typischerweise ein Nginx, Node.js oder Apache-Server. Der Webbrowser des Clients erstellt beim Zugriff dann eine HTTP-Anfrage an diesen Webserver, um die Ressource (HTML-Datei, CSS-Datei, JavaScript-Datei usw.) anzufordern, die für die angezeigte Webseite benötigt wird. Der Webserver soll dann dem Client die angeforderten Ressourcen bereitstellen. Der Webbrowser des Clients empfängt die HTTP-Antwort des Servers und rendert den Inhalt entsprechend.

REST-API:

Die Kommunikation zwischen Client und Server erfolgt über eine REST-API.

Die REST-API wurde gewählt, da sie eine einfache, plattformunabhängige und skalierbare Möglichkeit bietet, Daten zwischen Client und Server auszutauschen. Durch die Verwendung von standardisierten HTTP-Methoden. Die REST-API ermöglicht es dem Client, mit dem Server zu kommunizieren, Anfragen zu senden und Daten zu empfangen.

Zum Austausch von Daten wie Bestellungen oder Zusammenfassung soll das Datenaustauschformat JSON verwendet werden. Durch die Verwendung von JSON als Datenaustauschformat wird eine effiziente Übertragung von Informationen ermöglicht, was zu einer verbesserten Leistung und Interoperabilität führt.

Datenbank (MariaDB / PostgreSQL / MySQL):

Alle drei Datenbanken sind Industrie Standard und werden da auch benutzt. Im gesamten sind alle drei Relationale Datenbanken von der Funktionalität sehr ähnlich. Kurz gesagt mit jeder dieser drei Datenbanksystemen könnte unser Projekt gut umgesetzt werden. Wir haben uns jedoch für PostgreSQL entscheiden, da es schon Vorerfahrung im Team gibt, es gute Open-Source Lizenzen und eine Community hat. Falls das Projekt mal weiterentwickelt wird bietet PostgreSQL noch viele weitere Komplexe Funktionalitäten.

Backend (Flask, Django):

Beide Frameworks bringen ihre Vor- und Nachteile. Es ist jedoch nicht klar, ob die Vorteile von Django mit diesem Projekt vollkommen ausgenutzt werden können. Die Anfängerfreundliche Entwicklung mit Flask ist allerdings ein sehr großer Vorteil, da nicht

alle Mitglieder des Teams Vorerfahrungen mit Python haben.
Somit ist unser Framework für das Backend Flask.

FrontEnd (React.js, Vue.js, Angular):

Letztendlich hat sich die Auswahl durch die bisherigen Erfahrungen des Entwicklers für die Kombination aus Angular und Flask entschieden. Dies bietet eine leistungsstarke Lösung für die Entwicklung moderner Webanwendungen. Durch die Integration dieser beiden Frameworks können Entwickler eine klar strukturierte, skalierbare und performante Anwendung entwickeln. Fortführend wird auch eine klare Trennung zwischen Front- und Backend, durch eine Anbindung Angular durch eine Restful API von Flask, sichergestellt. Dahingehend werden Skalierbarkeit und Wartbarkeit der Anwendung weiter unterstützt.

Bausteinsicht

Whitebox Gesamtsystem

Enthaltene Bausteine
Frontend, Backend, Datenbank

Begründung
Die Aufteilung einer Webanwendung in Frontend, Backend und Datenbank ermöglicht eine klarere Strukturierung, erleichtert die Skalierbarkeit und fördert die Sicherheit. Jede Komponente kann unabhängig voneinander entwickelt, getestet und optimiert werden, was zu einer verbesserten Wartbarkeit und Leistung führt.

Name	Verantwortung
Frontend	Kommunikation mit Benutzer (Anzeige/Annahme von Daten)
Backend	API: Schnittstelle zwischen Frontend und Datenbank/Validierung der Daten
Datenbank	Speichern von Daten

Ebene 2

Whitebox Frontend

Unser Frontend besteht aus Angular Komponenten, Beispiele sind:

- Calendar.component (wird verwendet, um Zeitspannen auszuwählen z.B für die home Komponente)
- Dish.component (wird verwendet, um dishes anzulegen)
- Header.component (wird dauerhaft am Anfang der Webseite angezeigt. Beinhaltet die Einstellungen und das Menü)

- Home.component (bildet den Startbildschirm unserer Anwendung. Sie zeigt Speisepläne in einem auswählbaren Zeitraum an. Außerdem können Bestellungen erstellt und bearbeitet werden)
- Login.component (ermöglicht Usern sich mit ihren E-Mails und Passwörtern anzumelden)
- Create-meal-plan.component (Seite auf welcher Speisepläne für die übernächste Woche erstellt werden können)
- …

Whitebox Datenbank

Unsere Datenbank besteht aus folgenden Tabellen:

- users
- orders
- mealPlan
- dishes
- allergies
- user_allergy_association
- dish_allergy_association

Whitebox Backend

Unser Backend basiert auf Angular. Die Hauptkomponente ist die Datei app.py. In dieser sind alle API-Routen implementiert. Außerdem wird dort die Verbindung zur Datenbank aufgebaut. Wenn hier eine API-Route aufgerufen wird und Interaktion mit der Datenbank gefordert ist, werden Funktionen in den Datenbank-Repositories aufgerufen. Diese Datenbank-Repositories sind Klassen welche Funktionen bereitstellen, um mit der Datenbank zu interagieren. Diese befinden sich im Ordner „Web-App\Backend\DB_Repositories\“. Hier befindet sich außerdem noch die Datei models.py in welcher das Datenbankmodell festgelegt wurde.

Beispiele sind:
Die Routen create_meal_plan und meal_plan, welche Funktionen im mealPlanRepository aufrufen.

Ebene 3

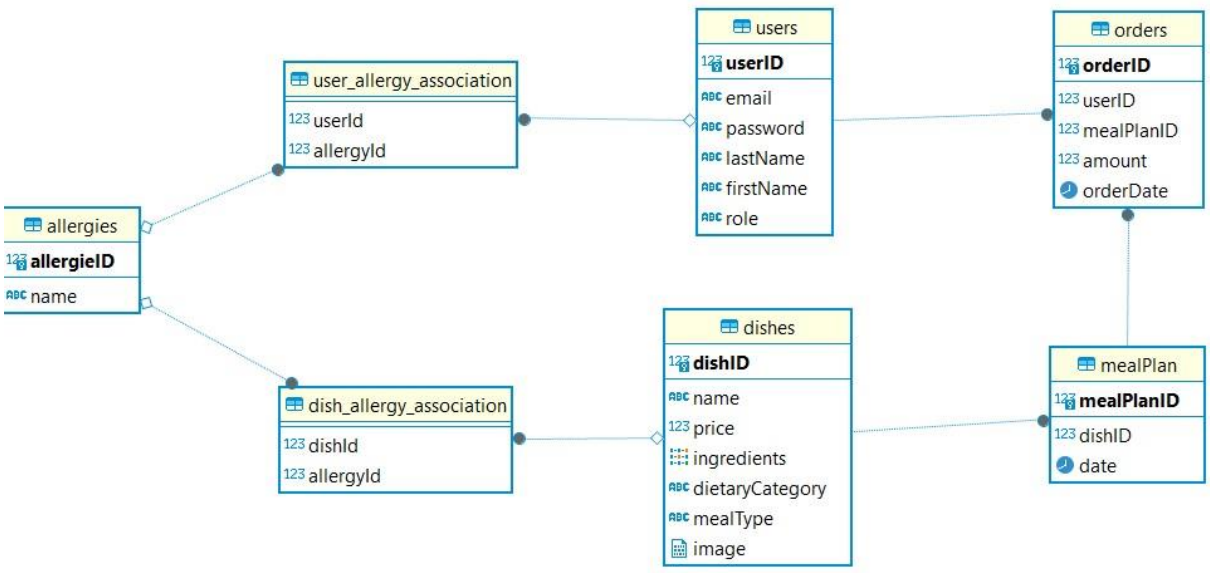
Whitebox Frontend.create-meal-plan.component

Das Erstellen von Speiseplänen wird gestaltet durch das Ziehen von Gerichten in Felder, welche die Tage der Woche darstellen. Dafür werden alle Gerichte nach ihren mealType über einen API-Call geladen. Außerdem werden die Speisepläne überprüft. Regeln dafür sind, keine Suppe am Samstag, jeden Tag ein vegetarisches Gericht und in jedem Tag muss ein Gericht vorhanden sein. Dies wird durch die Funktion checkLists gemacht. Die Funktion createDishPlan erzeugt aus den vom Client zugewiesen Gerichten eine JSON welche über einen API Call an das Backend gesendet wird, welches den Speiseplan dann in die Datenbank schreibt.

Whitebox Frontend.home.component

In dieser Komponente sieht man den Speiseplan, sofern man nicht eingeloggt ist. Wenn man eingeloggt ist, sieht man den Speiseplan und kann hier direkt bestellen. Der Speiseplan wird über die API-Route /meal_plan/<Anfangsdatum>/<Enddatum> geholt.
Falls man in der Woche schon eine Bestellung abgegeben hat, wird diese angezeigt. So hat man dann die Möglichkeit seine Bestellung zu ändern. Falls man noch keine Bestellung angelegt hat, kann man eben eine neue machen.
Das ganze geschieht über den API Call getOrdersByUser aus der Datenbank geladen. Danach können durch Usereingaben die Bestellungen aktualisiert werden und geändert an das Backend zurückgeschickt werden. Außerdem wird der Preis der Bestellungen angezeigt und bei Änderung aktualisiert, dafür wird die Funktion getOrderPrice verwendet.

Whitebox Datenbank

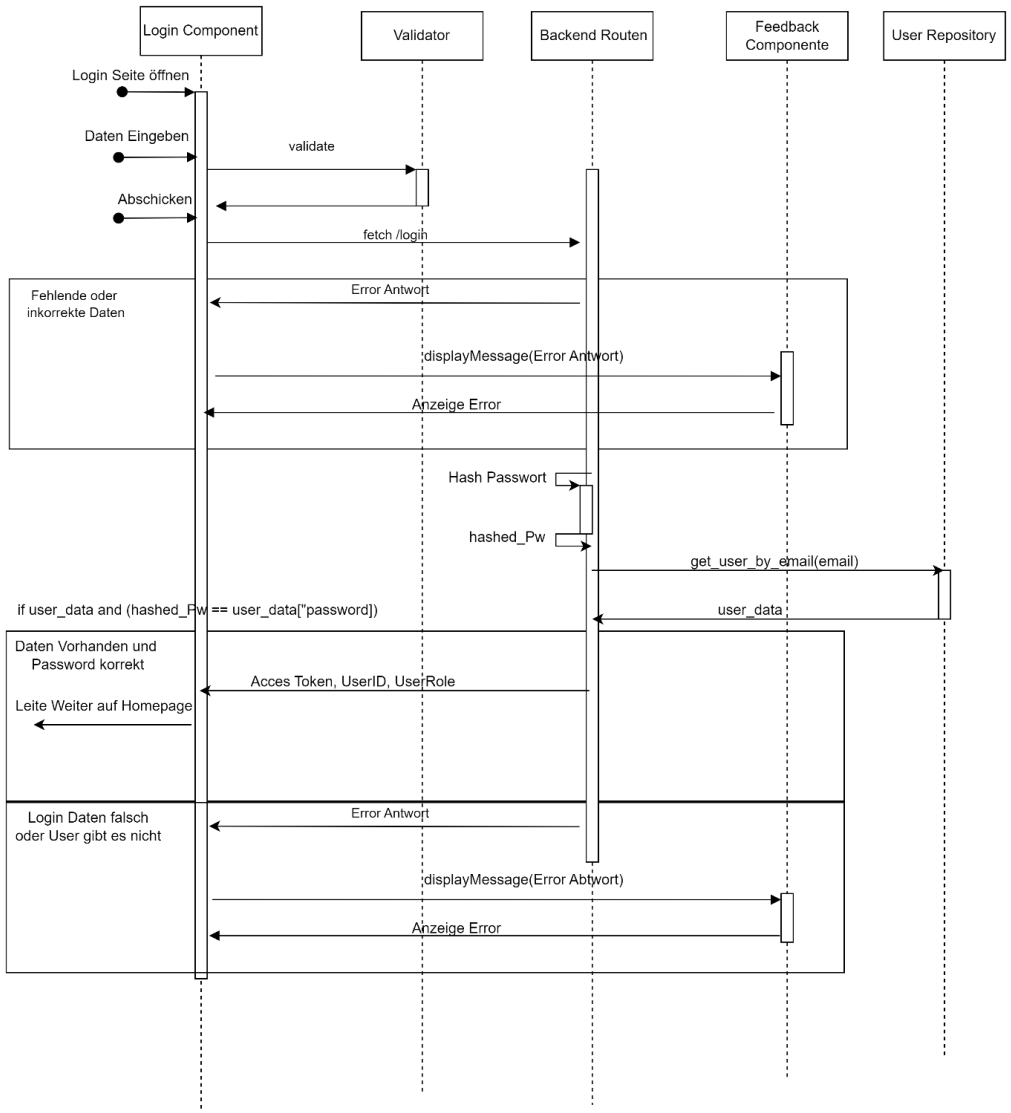


Laufzeitsicht

Im Folgenden sind 2 wichtige Beispielszenerien, die so in der Applikation ablaufen.

Dieses Sequenzdiagramm beschreibt den Ablauf des Logins, sofern alles richtig gemacht wurde und kein Fehler auftritt. Zuerst muss der Benutzer seine Login Daten eingeben. Die Validator-Komponente sorgt dafür das nur logische Daten eingegeben werden können wie z. B. E-Mails. Es kann das Formular auch erst abgeschickt werden, nachdem alles den Validator Regeln entspricht. Nach dem Abschicken liest das Backend dann die Daten aus. Zuerst wird im Backend das übergebene Passwort gehashed, da die Passwörter in der Datenbank aus Sicherheitsgründen gehashed abgespeichert sind. Dies geschieht mit SHA 56- Anschließend wird geschaut, ob es den Benutzer gibt und ob das übergebene Passwort mit dem für den Nutzer abgespeicherten Password übereinstimmt. Wenn ja wird der Access Token, Benutzer Rolle und die Benutzer ID zurückgegeben.

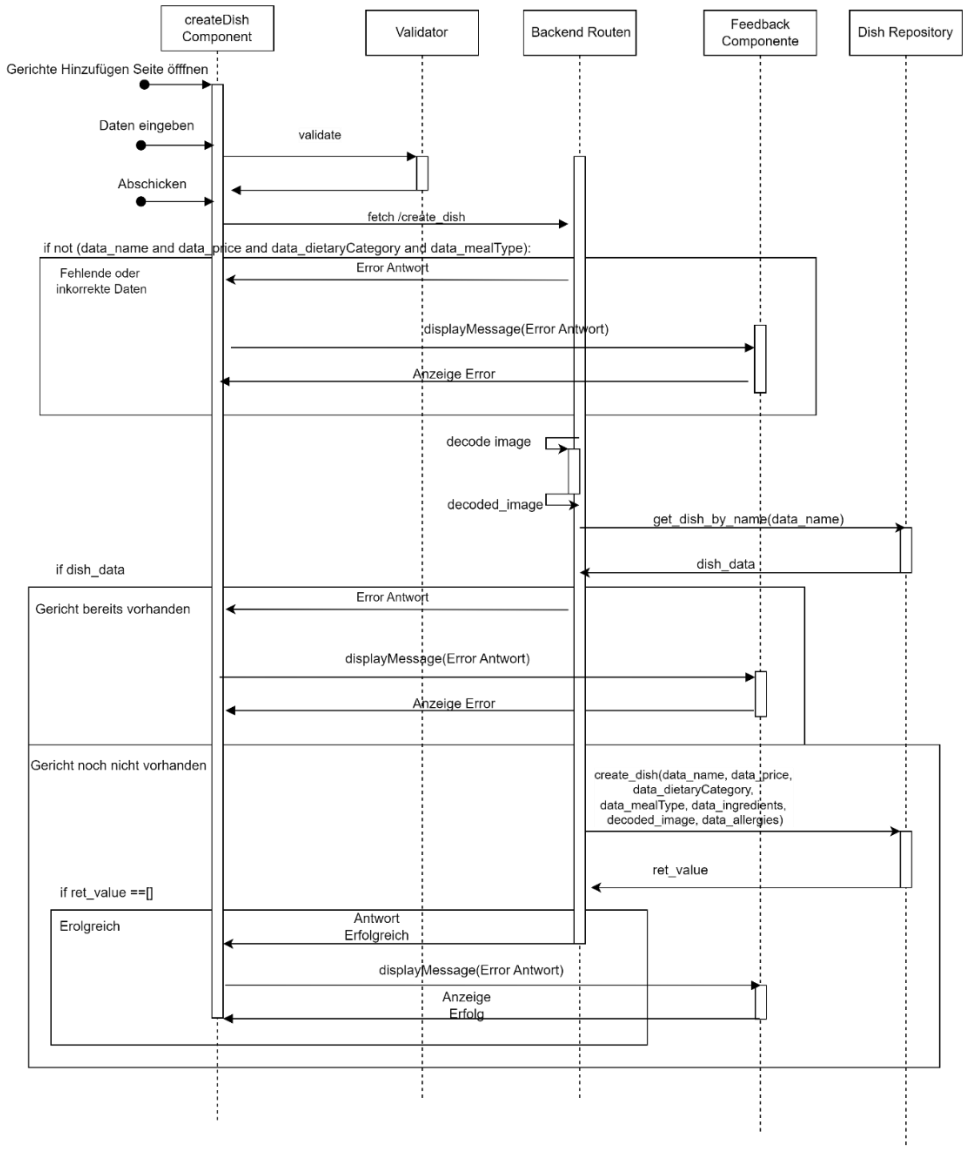
Hierbei können noch folgende Fehlerauftreten. Der JSON entspricht nicht den Vorgaben. Der User existiert nicht oder das Passwort stimmt nicht.



Das folgende Sequenzdiagramm zeigt, wie ein Gericht angelegt wird. Zuerst wird durch den Validator wieder sichergestellt, dass die Eingaben nur aus zugelassenen Daten bestehen und die Pflichtfelder befüllt sind. Hierbei wird auch geschaut, dass nur PNGs hochgeladen werden die kleiner als 10MB sind. Nach dem Absenden wird das Gericht angelegt, sofern es noch keins mit dem gleichen Namen gibt.

Folgende Fehler können auftreten:

- Das übergebene JSON entspricht nicht dem gültigen Format.
- Gericht mit dem gleichen Namen existiert bereits.
- Allergien für ein Gericht wurden angegeben die, die Datenbank nicht kennt


createDish Component
Validator
Backend Routen
Feedback Componente
Dish Repository
Gerichte Hinzufügen Seite öfffnen
Daten eingeben
validate
Abschicken
fetch /create_dish
if not (data_name and data_price and data_dietaryCategory and data_mealType):
Fehlende oder inkorrekte Daten
Error Antwort
displayMessage(Error Antwort)
Anzeige Error
decode image
decoded_image
get_dish_by_name(data_name)
dish_data
if dish_data
Gericht bereits vorhanden
Error Antwort
displayMessage(Error Antwort)
Anzeige Error
Gericht noch nicht vorhanden
create_dish(data_name, data_price, data_dietaryCategory, data_mealType, data_ingredients, decoded_image, data_allergies)
ret_value
if ret_value ==[]
Erolgreich
Antwort Erfolgreich
displayMessage(Error Antwort)
Anzeige Erfolg

Querschnittliche Konzepte

JWT

Der JSON Web Token (JWT) ist ein genormter Access-Token, welcher häufig für Authentifizierungsaufgaben verwendet wird. So auch in dieser Applikation. Hier wird bei erfolgreichem Login ein JWT an den Client zurückgesendet. Dieser enthält dann die ID des Benutzers.
Ein JWT besteht aus drei Teilen:

- Header: Dieser enthält Metadaten über den Token, wie den Typ des Tokens und den verwendeten Algorithmus zur Signierung.
- Payload: Enthält die Nutzdaten des Tokens in unserem Fall die Benutzer ID.
- Signatur: Wird aus dem Header und dem Payload mit einem geheimen Schlüssel erstellt. Diese dient zur Überprüfung der Integrität des Tokens.

Dieser Token wird dann später genutzt, um die Zugriffe auf die API-Routen zu berechtigen (mehr dazu im nächsten Punkt). Hierbei prüft der Server die Gültigkeit des Tokens (in diesem Fall 1h), indem er die Signatur mit dem geheimen Schlüssel entschlüsselt und die im Payload enthaltenen Infos auswertet. Der Schlüssel ist dem Server bekannt, da dieser ja beim Login den Token schon erstellt hat.

Zustandslosigkeit durch JWT: Da in dieser Applikation eine REST API verwendet wird, welche zustandslos ist, ist es von Vorteil das JWT's auch zustandslos sind. So muss der Server keine Informationen über die Session des Clients speichern, was z.B. bei Cookies der Fall wäre. JSON Web Tokens hingegen senden alle Informationen, die zur Authentifizierung und Autorisierung benötigt werden mit dem Token mit. Das bringt uns den Vorteil das wir im Backend Server keine Zustandsinformationen der Clients speichern müssen, sondern die Clients nur anhand der ID die im Token gespeichert ist, identifizieren und so ihre Informationen aus der Datenbank holen können.

Erstellung des Tokens:

```
access_token = create_access_token(identity=user_data["userID"],
expires_delta=timedelta(hours=1))
```

API-Routen Absicherung

Wie bereits im obigen Text erwähnt, werden unsere API-Routen abgesichert. Dies betrifft alle Routen die kritische Informationen herausgeben oder Informationen abändern. Hierzu haben wir einen Python Decorator geschrieben der unter folgendem Pfad eingesehen werden kann: „Web-App\Backend\decorators.py". Dieser muss einfach in die API-Routen

eingebunden werden, was wie folgt geschieht:

```python
@app.route('/all_users')
@jwt_required()
@permission_check(user_repo)
def all_users():
    data = user_repo.get_all_users()
    if data:
        return jsonify(data)
    return jsonify({api_message_descriptor:  f"{get_api_messages.ERROR.value}Benutzer nicht gefunden"}), 404
```

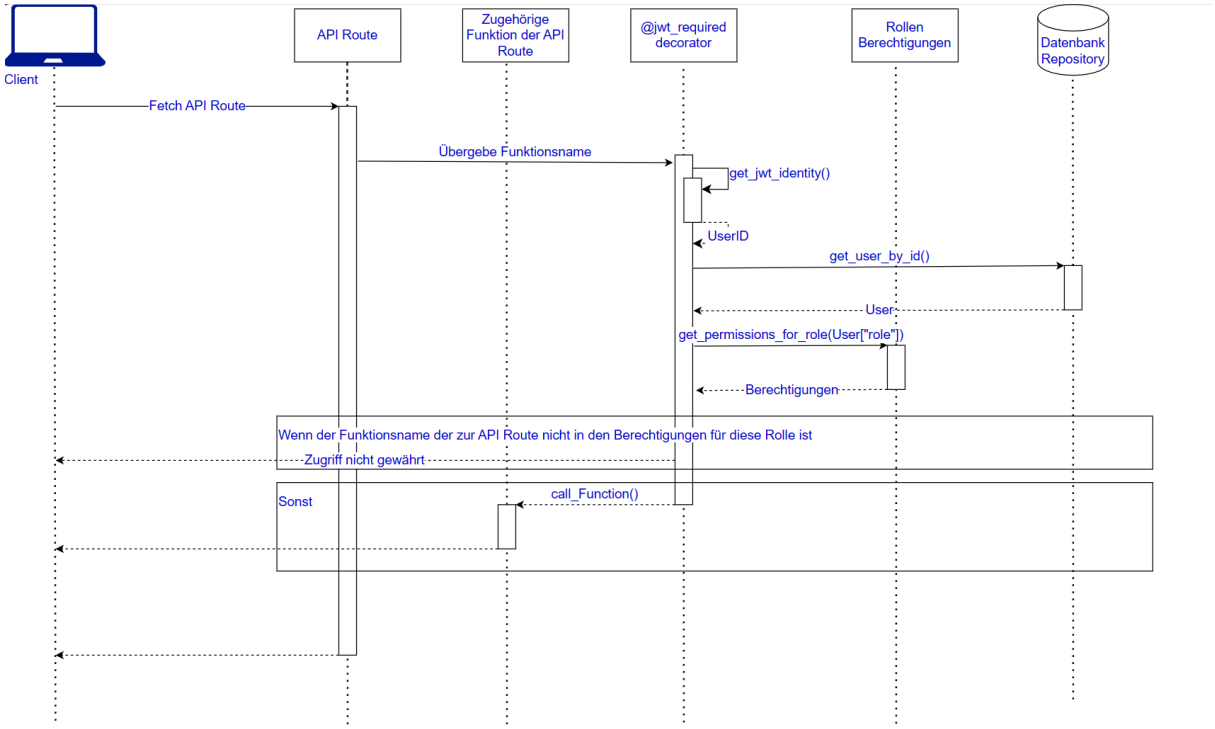
In der Datei „Web-App\Backend\role_permissions.py“ werden dann die Berechtigungen abgespeichert, die eine Rolle hat. Das sieht wie folgt aus:

```
permissions_data = """
admin:hello,all_users,user_by_id,user_by_email,dish_by_id,create_dish,allergy_by_userid,create_user_as_admin,dish_by_name,create_dish,create_order,orders
hungernde:hello,dish_by_id,allergy_by_userid,dish_by_names,create_order,orders_by_user,dish_by_mealType,meal_plan,set_user_allergies
kantinenmitarbeiter:dish_by_id,create_dish,dish_by_name,create_dish,orders_by_user,dish_by_mealType,meal_plan,create_meal_plan,orders_sorted_by_dish
"""
```

Hier werden lediglich die Namen der Funktion die zu einer API-Route gehört hineingeschrieben und auf dieser Basis wird die Autorisierung durchgeführt. In dem Ausschnitt der Berechtigungen kann man sehen das nur die Rolle admin berechtigt ist auf die API all_users zuzugreifen. Wenn nun ein Client die Route all_users aufruft wird erst der Python Decorator @jwt_required aufgerufen. Hier passiert dann folgendes:

1. Die UserID des Benutzers wird aus dem JSON Web Token geholt
2. Der Benutzer mit dieser UserID wird aus der Datenbank geladen
3. Es wird geschaut was für Berechtigungen der geladene Benutzer hat, was anhand seiner Rolle geschieht
4. Wenn in dem Berechtigungsstring der Rolle, der zur API gehörige Funktionsname enthalten ist hat man Zugriff auf die API.

Dies sieht dann im genauen wie folgt aus:



Verschiedene Umgebungen

Um verschiedene Umgebungen effizient zu handhaben, setzen wir in dieser Anwendung auf die Verwendung von Docker Compose auf einem Linux-Server. Um jedoch eine reibungslose lokale Testumgebung zu gewährleisten, ohne dass Codeänderungen erforderlich sind, verwenden wir .env-Dateien. Diese Dateien enthalten umgebungsspezifische Konfigurationswerte, darunter:

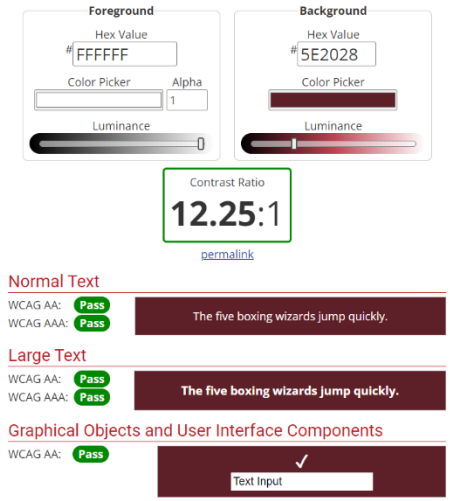
- JWT_SECRET_KEY: Das ist der geheime Schlüssel, welcher für die JSON Web Tokens genutzt wird
- POSTGRES_PW: Das Passwort für die PostgreSQL Datenbank
- ANGULAR_ENVIRONMENT: Diese Variable gibt an, ob der Code auf dem Server oder lokal ausgeführt wird. Dies ist entscheidend, da im Frontend API-Anfragen durchgeführt werden und sich die URI des Backend-Servers der die API's beinhaltet, je nach Umgebung ändert. In der lokalen Testumgebung ist die URI beispielsweise "localhost:5000", während sie auf dem Server "IP-Adresse der Applikation:5000" ist. Diese Anpassung erfolgt automatisch im Code, je nachdem, ob dieser Variable der Wert "development" oder "production" zugewiesen wird.

Die Verwendung von Docker Compose ermöglicht es uns, unsere Anwendungsumgebung einfach zu konfigurieren und zu verwalten, während die .env-Dateien uns ermöglichen, reibungslos zwischen verschiedenen Umgebungen zu wechseln, ohne dass wir den Code manuell ändern müssen.

Angular-Services

Angular Services sind in Angular-Anwendungen eine Art von Singleton-Klassen, die einen bestimmten Zweck erfüllen, der an verschiedenen Teilen der Anwendung benötigt wird. Der Hauptzweck von Angular Services besteht darin, eine wiederverwendbare und einheitliche Schnittstelle bereitzustellen, die von verschiedenen Teilen der Anwendung genutzt werden kann. Dadurch wird der Code leichter wartbar, erweiterbar und testbar. Außerdem fördern Services das Prinzip der „Separation of Concerns“, da sie die Geschäftslogik von der Benutzeroberfläche trennen. Ein weiterer wichtiger Aspekt ist, dass Services in Angular als Singleton-Instanzen behandelt werden, was bedeutet, dass während des Lebenszyklus der Anwendung nur eine einzige Instanz eines bestimmten Services existiert. Dadurch wird Ressourcenverbrauch minimiert und konsistente Datenverwaltung gewährleistet. Zusammenfassend werden Angular Services verwendet, um Funktionalitäten zu kapseln, Daten zu verwalten, Code zu organisieren und die Wiederverwendbarkeit sowie Testbarkeit von Angular-Anwendungen zu verbessern. In diesem Projekt werden sie hauptsächlich verwendet, um Daten zu speichern, die auf der gesamten Anwendungen benötigt werden, wie z.B. die UserID oder die Rolle. Außerdem wird der Großteil der API-Anfragen an das Backend in Services gemacht.

Qualitätsanforderungen



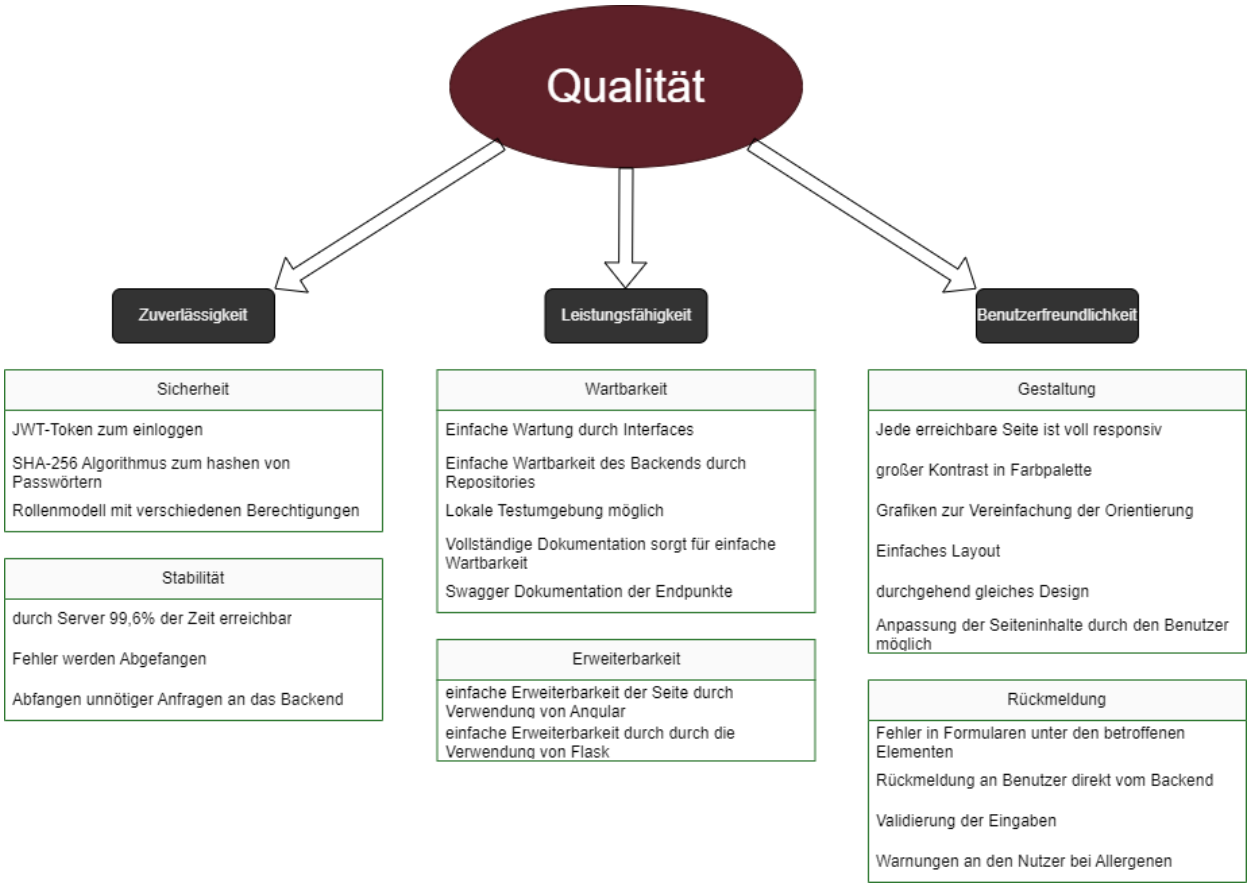
Benutzerfreundlichkeit: Die App soll eine einfache und schnelle Navigation enthalten. Des Weiteren soll die App einheitlich in Stil und Farbgebung sein.

Zuverlässigkeit: Die App soll eine Verfügbarkeit von mindestens 99 % während der Geschäftszeiten aufweisen, um Ausfallzeiten zu minimieren.

Geschwindigkeit: Die App soll Bestellungen innerhalb von maximal 30 Sekunden verarbeiten, um eine reibungslose Erfahrung zu gewährleisten.

Sicherheit: Die App soll verschiedene Nutzer Verwalten können und fortführend durch ein Rollensystem in verschiedene Bereiche aufgegliedert sein. Auch Perosnenbezogene Daten sollen sicher gespeichert sein.

Qualitätsbaum



Risiken und technische Schulden

Es wurde eine Evaluation der verschiedenen Risiken in während den Sprints erstellt, diese ist unter Dokumentation/Risiko_management.xlsx zu finden.

Technische Schulden:

- Fehlende Implementierung der Mehrsprachigkeit
- Darkmode
- Naming Convention

Glossar

Begriff	Definition
Hungernder	*Der Hungernde ist in dieser Dokumentation, sowie in der App als Nutzer der App zu sehen, welcher außerhalb der Kantine ist und Essen bestellen will. Dieser hat keine Zugriffe auf interne Prozesse/Seiten.*