Pantry

Price **A**nd **N**u**TR**ition anal**Y**sis of cooking recipes

Studienarbeit

Herbstsemester 2022

Studiengang Informatik
OST – Ostschweizer Fachhochschule

Autoren: Daniel Frick & Zvonimir Serkinic
Betreuerin: Prof. Dr. Mitra Purandare

Abstract

Wenn man in einem Kochbuch oder im Internet ein Rezept betrachtet, hätte man oft gerne weiterführende Informationen darüber, was der Einkauf der Produkte für das Rezept kostet und welche Nährwerte das Rezept enthält.

Genau bei diesem Problem setzt die entwickelte Web-App «Pantry» (**P**rice **A**nd **N**u**TR**ition anal**Y**sis of cooking recipes) an. «Pantry» ermöglicht es dem Benutzer auf einfache Weise ein Rezept mit mehreren Zutaten zu erfassen. Darauf schlägt die Web-App dem Benutzer ein Produkt je Zutat vor. Des Weiteren werden dem Benutzer der Gesamtpreis für den Kauf aller Produkte, der spezifische Preis für sein Rezept, als auch die totalen Nährwerte, welche sein Rezept enthalten, ausgewiesen. Auf Grundlage dieser von «Pantry» gelieferten Informationen kann der Benutzer dann leichter entscheiden, ob er ein Rezept zubereiten möchte.

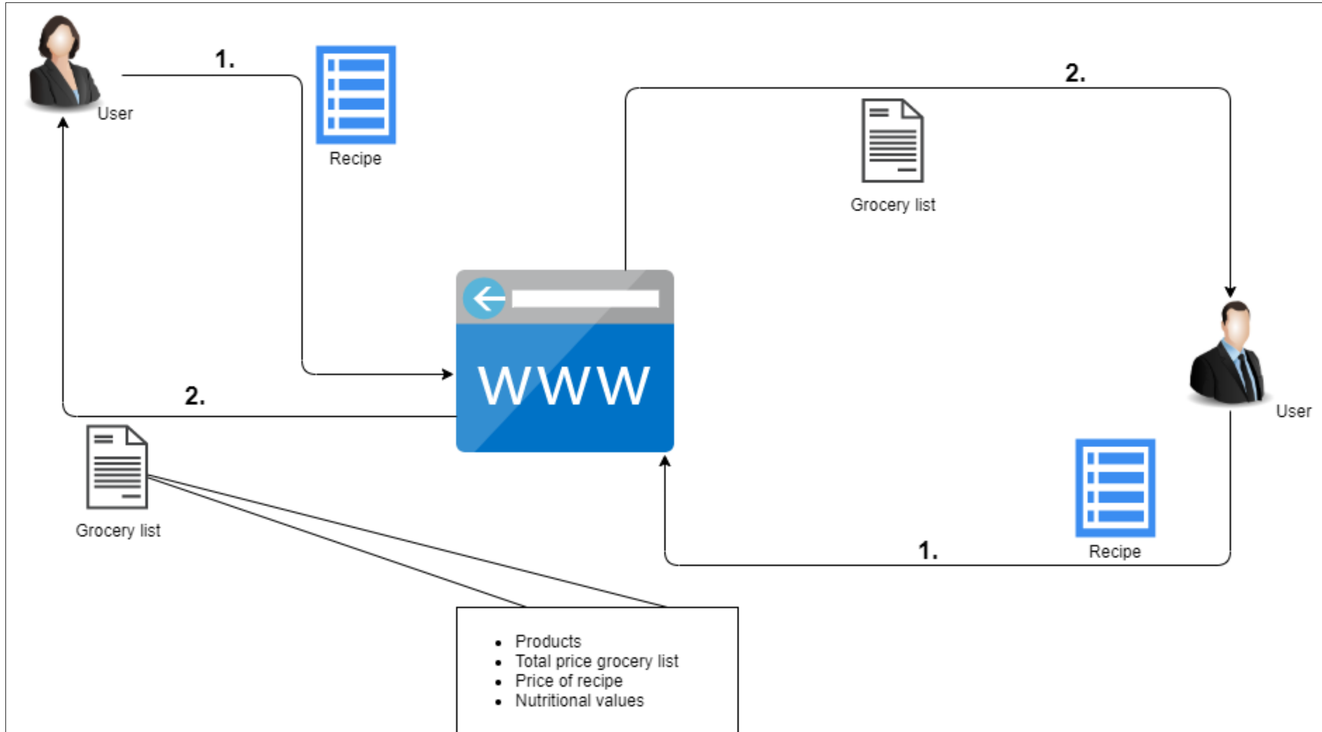
Durch eine zusätzlich aufgebaute Produktdatenbank ist «Pantry» in der Lage Produktvorschläge zu machen. Um diese Datenbank zu befüllen, wurde ein Web-Scraper in Python mit Hilfe der Libraries Scrapy und Playwright entwickelt, welcher die Produkte von der öffentlichen Coop-Webseite sammelt, aufbereitet und in die Datenbank einfügt.
Die Web-App wurde dabei mit React realisiert, welche im Hintergrund eine REST-API verwendet. Die auf Spring Boot basierende REST API greift dann auf die Produktdatenbank zurück, um dem Benutzer Produktvorschläge für seine gewünschten Zutaten machen zu können.

Die entwickelte Web-App «Pantry» erfüllt die gesetzten Ziele, hat aber noch Weiterentwicklungspotenzial. So wäre es in Zukunft sicher sinnvoll den Web-Scraper auf weitere Verkäufer (z.B. Migros) auszuweiten, damit «Pantry» Produkte von verschiedenen Anbietern vorschlagen kann.

Management Summary

Ausgangslage

Bei «Pantry» (**P**rice **A**nd **N**u**TR**ition anal**Y**sis of cooking recipes) handelt es sich um eine Idee für eine App, welche es auf dem Schweizer Markt in dieser Form noch nicht gibt.
Die «Pantry»-App soll es dem Benutzer ermöglichen auf einfache Weise ein Rezept mit mehreren Zutaten zu erfassen. Darauf soll die App dem Benutzer eine Liste von Produktvorschlägen, welche als Einkaufsliste verwendet werden kann, liefern. Pro Produktvorschlag sollen dabei herkömmliche Produktinformationen wie Menge, Preis und Verkäufer des Produktes dargestellt werden. Zusätzlich sollen verfügbare Nährwerte des Produktes, basierend auf der gewünschten Menge für das Rezept, ausgewiesen werden. Als Information über das ganze Rezept soll die App dem Benutzer sowohl den Gesamtpreis für seinen Einkauf, den spezifischen Preis für sein Rezept, als auch die totalen Nährwerte, welche sein Rezept enthält, zur Verfügung stellen.
Die von der «Pantry»-App gelieferten Informationen können den Benutzer einerseits dabei unterstützen zu entscheiden, ob er ein Rezept zubereiten möchte. Andererseits ermöglicht die App dem Benutzer anhand der zusammengestellten Einkaufsliste seinen Einkauf zeiteffizienter zu gestalten.

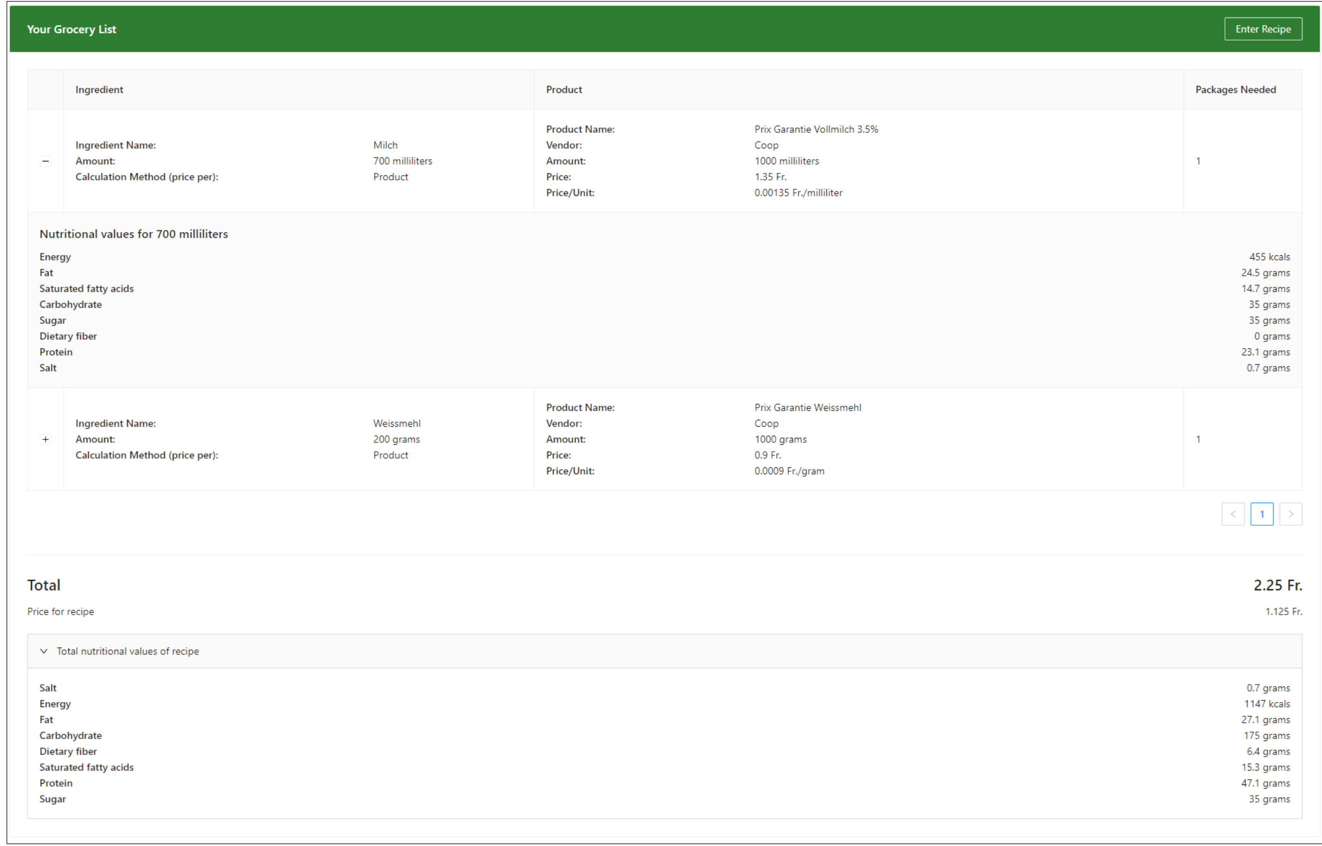


Vorgehen

Anhand einer ersten Analyse wurden die Anforderungen an die «Pantry»-App erhoben. Darauf wurde eine sinnvolle Architektur für die Umsetzung evaluiert. Es war schnell klar, dass auf eine klassische Architektur mit Frontend, Backend und Datenbank gesetzt werden soll. Was zusätzlich als weitere Komponente dazu kam, war ein Web-Scraper. Ein Web-Scraper ist ein Werkzeug, welches es ermöglicht spezifische Daten von öffentlich zugänglichen Webseiten zu extrahieren. Die Entwicklung eines solchen Web-Scraper half dabei die Datenbank von «Pantry» mit Produktdaten zu befüllen. Eine ausführliche Auseinandersetzung mit dem Thema Web-Scraping zu Beginn des Projekts stellte sicher, dass der Web-Scraper schlussendlich mit den am besten geeigneten Tools realisiert wurde.
Während der Implementierung der «Pantry»-App wurden diverse Qualitätsmassnahmen wie eine CI-Pipeline und Code Reviews eingesetzt, um die Softwarequalität möglichst hochzuhalten. Des Weiteren wurde auch ein Testkonzept entworfen und umgesetzt.

Ergebnisse

Entstanden ist die Web-App «Pantry», welche die Funktionalität aus der beschriebenen Ausgangslage komplett abdeckt. Der Web-Scraper konnte in der zur Verfügung stehenden Zeit für Coop realisiert werden, so dass die Datenbank zurzeit mit über 10’000 Produkten von Coop befüllt ist. Das Backend wiederum bietet die Möglichkeit an, ein Rezept zu empfangen und darauf eine Liste mit Produktvorschlägen zurückzuliefern.
Die Web-App selbst ermöglicht auf einfache Weise die Erfassung eines Rezepts und zeigt anschliessend in der potentiellen Einkaufsliste alle gewünschten Informationen an. Die Abbildung unten zeigt exemplarisch eine von «Pantry» vorgeschlagene Einkaufsliste für zwei Zutaten.



Danksagung

An dieser Stelle möchten wir uns herzlich bei Mitra Purandare, die unsere Studienarbeit betreut hat, für die zahlreichen guten Inputs und die angenehme Zusammenarbeit bedanken.
Des Weiteren möchten wir uns auch bei den Korrekturlesern unserer Arbeit, als auch bei den Testpersonen für unsere Usability Tests bedanken.

Inhaltsverzeichnis

Abstract I
Management Summary II
Danksagung IV
Inhaltsverzeichnis V
Abbildungsverzeichnis VIII
Tabellenverzeichnis VIII
1 Einleitung 1
1.1 Ausgangslage 1
1.2 Vision 1
1.3 Grobes Umsetzungskonzept 2
1.4 Ähnliche Produkte / bestehende Lösungsansätze 3
1.5 Rahmenbedingungen 3
2 Analyse 4
2.1 Funktionale Anforderungen 4
2.1.1 User Story «Preisrechner» 4
2.1.2 User Story «Web-Scraping» 4
2.1.3 User Story «Nährwerte-Rechner» 5
2.1.4 User Story «Alternative Produkte» (Erweiterung) 5
2.1.5 User Story «Rezept Sammlung» (Erweiterung) 5
2.2 Nicht-funktionale Anforderungen 5
2.2.1 Usability 6
2.2.2 Performance 6
2.2.3 Supportability 6
3 Architektur und Design 7
3.1 Architektur und Technologien 7
3.2 Datenbank Modell 9
3.3 Mockups 10
4 Qualitätsmassnahmen 11
4.1 Git-Workflow 11
4.2 Linting 12
4.3 Continous Integration 12
4.4 Testkonzept 12
4.4.1 Unit Tests 13
4.4.2 Usability Tests 13

5 Umsetzung **15**
5.1 Frontend 15
5.1.1 Entscheid bezüglich verwendeten Libraries 15
5.1.2 Verzeichnisstruktur 16
5.1.3 Herausforderungen 16
5.2 Backend 17
5.2.1 Entscheid bezüglich verwendeten Libraries, Frameworks und Plugins 17
5.2.2 Verzeichnisstruktur 19
5.2.3 Herausforderungen 19
5.3 Web-Scraper 22
5.3.1 Ansatz zum Scrapen der Coop-Webseite 22
5.3.2 Entscheid bezüglich verwendeten Libraries 23
5.3.3 Spiders 24
5.3.4 Items 25
5.3.5 Pipelines 25
5.3.6 Settings 27
5.3.7 Herausforderungen 27
5.3.8 Scraper in Betrieb nehmen 29
5.3.9 Dauer des Scraping-Prozesses 29
5.4 Deployment 30
5.4.1 Verteiltes System mit Docker 30
5.4.2 Einsatzmöglichkeiten 31
6 Resultate und Fazit **32**
6.1 Zielerreichung 32
6.1.1 Funktionale Anforderungen 32
6.1.2 Nicht-funktionale Anforderungen 33
6.2 Schlussfolgerungen 34
6.3 Weiterentwicklung 35
7 Projektmanagement **36**
7.1 Rollen und Verantwortlichkeiten 36
7.2 Prozesse 36
7.3 Tooling 37
7.4 Meilensteine 37
7.5 Projektplan 38
7.6 Risikomanagement 38
7.6.1 Identifizierte Risiken 38
7.6.2 Risikoevaluation 39

7.6.3 Abschliessende Risikobeurteilung 42
7.7 Zeitrapportierung 42
Glossar 43
Abkürzungsverzeichnis 44
Literaturverzeichnis 45

Abbildungsverzeichnis

Abbildung 1: Architektur und Technologien 7
Abbildung 2: Datenbank Modell 9
Abbildung 3: Verzeichnisstruktur Frontend 16
Abbildung 4: Beispiel einer Entity Klasse 17
Abbildung 5: Testreport in GitLab 18
Abbildung 6: API-Dokumentation 18
Abbildung 7: Verzeichnisstruktur Backend 19
Abbildung 8: Beispiel zur Berechnungsmethode «Cheapest price per product» 20
Abbildung 9: Gesammelte Produktkategorienlinks auf Startseite 22
Abbildung 10: Next-Button auf Produktkategorie-Seite 23
Abbildung 11: Nährwerte auf der Produktdetail-Seite 23
Abbildung 12: Basisdaten auf der Produktdetail-Seite 23
Abbildung 13: ProductItem 25
Abbildung 14: Docker-Container im verteilten System 30
Abbildung 15: Loadbalancer verteilt Anfragen auf Backend-Instanzen 33
Abbildung 16: Unit Test Codeabdeckung vom CalculatorService 34
Abbildung 17: Risikomatrix vom 30.09.2022 39
Abbildung 18: Risikomatrix vom 11.10.2022 40
Abbildung 19: Risikomatrix vom 25.10.2022 40
Abbildung 20: Risikomatrix vom 22.11.2022 41
Abbildung 21: Risikomatrix vom 20.12.2022 41
Abbildung 22: Geleistete Arbeitszeit pro Woche 42

Tabellenverzeichnis

Tabelle 1: Testarten 12
Tabelle 2: Testing Tools 13
Tabelle 3: Geschwindigkeitsvergleich Scraping 24
Tabelle 4: Rollen und Verantwortlichkeiten 36
Tabelle 5: Tooling 37
Tabelle 6: Meilensteine 37
Tabelle 7: Identifizierte Risiken 38

1 Einleitung

Die Einleitung wurde in einer frühen Phase des Projekts geschrieben, als der Titel der Arbeit noch «Price Calculator for Smart Eating App» gelautet hat (siehe Aufgabenstellung). In diesem Kapitel wird weiterhin vom «Price Calculator» und nicht von «Pantry» gesprochen, da entschieden wurde, die Einleitung nicht noch einmal komplett zu überarbeiten.
Die Vision war zu Beginn des Projekts etwas abweichend von dem, was schlussendlich realisiert wurde, dies aufgrund der neuen funktionalen Anforderungen, welche im Verlaufe des Projekts hinzugekommen sind (siehe Kapitel 2.1.3). Durch diese neuen Anforderungen war dann der Titel der Arbeit nicht mehr passend, weshalb eine Namensänderung des Titels der Arbeit hin zu «Pantry» vorgenommen wurde.

Die Einleitung soll eine grobe Einführung in das Projekt geben. Als Erstes wird dabei die Ausgangslage beschrieben. Darauf folgt das Ziel der Arbeit in Form einer Vision, welche aus der originalen Aufgabenstellung, die sich im Anhang befindet, hergeleitet wurde. Anschliessend wird ein grobes Umsetzungskonzept vorgestellt, durch welches das Ziel der Arbeit erreicht werden soll.
Zum Schluss wird dann noch auf bereits bestehende Lösungsansätze eingegangen und die Rahmenbedingungen für das Projekt definiert.

1.1 Ausgangslage

Der «Price Calculator» kann von einer grünen Wiese aus entwickelt werden. Es gibt keine technologischen Vorgaben für die Umsetzung, als auch keinen Industriepartner auf welchen Rücksicht genommen werden muss.

Im Zentrum steht dabei einerseits eine aufzubauende Produktdatenbank, welche mit Informationen zu den angebotenen Lebensmitteln von Coop und Migros befüllt werden soll. Andererseits die Entwicklung einer REST-API, welche es ermöglicht für Rezepte geeignete Produktvorschläge zu erhalten. Zusätzlich soll eine Benutzeroberfläche zur Verfügung gestellt werden, um die Nutzung der API zu demonstrieren.

Die Produktdatenbank und die entwickelte REST-API können später dann dazu verwendet werden um die Funktionalität in einer (noch nicht existierenden) «Smart Eating App» zu integrieren.

1.2 Vision

Der «Price Calculator» ist eine Web-App, mit welcher der Preis für ein Rezept berechnet werden kann.

Der Benutzer der Web-App muss dabei nichts weiter tun, als die für sein Rezept notwendigen Zutaten zu erfassen. Darauf präsentiert die Web-App dem Benutzer einen Produktvorschlag je Zutat und damit sozusagen eine Einkaufsliste für sein Rezept. Zusätzlich werden dem Benutzer auch zwei Preise

angezeigt, einerseits der Gesamtpreis für den Einkauf der Produktvorschläge und andererseits der Preis, welcher ihn das eingegebene Rezept spezifisch kostet.
Wenn der Benutzer der Web-App anschliessend mit den präsentierten Produktvorschlägen nicht zufrieden ist, hat er zwei Optionen.

1. Er kann die Zutaten in seinem Rezept genauer spezifizieren, zum Beispiel in dem er die Zutat «Mehl» durch «Weissmehl» ersetzt, um einen passenderen Produktvorschlag zu erhalten.
2. Er kann sich direkt alternative Produkte für einen Produktvorschlag anzeigen lassen und eine Alternative anstelle des Produktvorschlags für sein Rezept verwenden.

Diese Flexibilität erlaubt es dem Benutzer eine für ihn optimale Einkaufsliste zusammenzustellen.

Mithilfe des «Price Calculator» wird es für einen Benutzer nicht nur leichter zu entscheiden, ob er ein Rezept zubereiten möchte, die Web-App ermöglicht es dem Benutzer anschliessend auch seinen Einkauf anhand der zusammengestellten Einkaufsliste zeiteffizienter zu gestalten.

1.3 Grobes Umsetzungskonzept

Der «Price Calculator» setzt sich aus vier zentralen Komponenten zusammen: Frontend, Backend, Datenbank und Web-Scraper. Die Rollen der einzelnen Komponenten werden nachfolgend als Einführung kurz beschrieben, genauere Details zu den einzelnen Komponenten sind im Kapitel 3 (Architektur und Design) und Kapitel 4 (Umsetzung) zu finden.

Datenbank
Eine, wie in der Ausgangslage angekündigt, zentrale Komponente ist die Produktdatenbank. Diese Datenbank soll durch den Web-Scraper befüllt und vom Backend für die Rezeptabfragen genutzt werden können.

Web-Scraper
Der Web-Scraper ist dafür zuständig, die Produktinformationen von den öffentlich zugänglichen Webseiten von Coop und Migros zu «scrapen» und die Produktdatenbank damit zu befüllen.

Backend
Das Backend enthält die Business-Logik und wird als REST API umgesetzt. Dabei wird unter anderem ein Endpunkt zur Verfügung gestellt, welcher es erlaubt ein Rezept zu übermitteln und eine potentielle Einkaufsliste inklusive Preise als Antwort zu erhalten.

Frontend
Das Frontend ist vor allem dazu da die Bedienung des Backends möglichst benutzerfreundlich zu machen. Das Frontend kümmert sich einerseits darum, dass die Daten im korrekten Format an die Backend-Endpunkte geschickt werden. Anderseits ist das Frontend auch dafür da, die von den Backend-Endpunkten gelieferten Ergebnisse in einem übersichtlichen Format für den Benutzer aufzubereiten.

1.4 Ähnliche Produkte / bestehende Lösungsansätze

Es gibt zurzeit kein vergleichbares Produkt auf dem Schweizer Markt, welches sich auf Produktvorschläge für ganze Rezepte fokussiert.
Wenn man allerdings nur nach einer Produktdatenbank mit Produkten von verschiedenen Anbietern sucht, wird man relativ schnell fündig. Ein prominentes Beispiel dafür wäre zum Beispiel «toppreise.ch», wo Preisvergleiche für Produkte von verschiedenen Anbietern angeboten werden. Auf der Webseite von «toppreise.ch» gibt es jedoch keinen Lebensmittel-Bereich.
Wenn man einen Preisvergleich für Lebensmittel in der Schweiz sucht, stösst man eigentlich nur auf einen Anbieter «vergleiche.ch/lebensmittel/», wo im Hintergrund eine Produktdatenbank mit Produkten von verschiedenen Anbieter liegen dürfte. Der Fokus liegt aber auch hier nur auf dem Preisvergleich einzelner Produkte und nicht auf der Zusammenstellung von Rezepten.

1.5 Rahmenbedingungen

Das Projekt wird im Rahmen einer Studienarbeit umgesetzt. Da die Studienarbeit mit 8 ETCS angerechnet wird, steht ein Zeitbudget von 240 Stunden pro Student zur Verfügung. Gesamthaft können daher zirka 480 Stunden in das Projekt investiert werden.

2 Analyse

Zuerst wird auf die, im Rahmen der Analyse erarbeiteten, funktionalen Anforderungen eingegangen, welche aus der Aufgabenstellung abgeleitet wurden. Daraufhin folgt eine Auflistung der für das Projekt zentralen nicht-funktionalen Anforderungen.

2.1 Funktionale Anforderungen

Für die Organisation der funktionalen Anforderungen werden User Stories verwendet, dies vor allem aufgrund der einfachen Verständlichkeit. Die einzelnen User Stories orientieren sich am Schema «Als [Kundentyp] [möchte] ich, [damit]» (Rehkopf (2022)).
Des Weiteren wurden diverse Akzeptanzkriterien je User Story definiert, um weitere wichtige Anhaltspunkte bezüglich der Umsetzung der einzelnen User Stories festzuhalten. Dabei gibt es vereinzelt auch optionale Akzeptanzkriterien, diese haben eine tiefere Priorität und werden nur bei genügend Zeit umgesetzt. Dasselbe gilt für User Stories, welche als «Erweiterung» markiert sind.

2.1.1 User Story «Preisrechner»

Als Benutzer möchte ich anhand meines Rezepts verschiedene Produkte für die einzelnen Zutaten vorgeschlagen bekommen. Zusätzlich möchte ich darüber informiert werden, was mich der Einkauf der Produkte kosten wird, als auch wieviel das Rezept an sich kostet.
Dies dient mir als Entscheidungsgrundlage, ob ich die Produkte für mein Rezept beschaffen möchte.

Akzeptanzkriterien:
- Es kann eine beliebige Anzahl an Zutaten für das Rezept erfasst werden
- Für jede Zutat ist ein Suchtext (z.B. Zucker), eine Menge und eine Einheit definierbar
- Für jede Zutat kann man wählen, ob der Produktvorschlag sich am tiefsten Preis des Produktes oder am tiefsten Preis des Produktes pro Menge (z.B. Kilopreis) orientieren soll
- Man erhält das am besten übereinstimmende Produkt je Zutat vorgeschlagen
- Es wird der Gesamtpreis für den Einkauf der vorgeschlagenen Produkte ausgewiesen
- Es wird der spezifische Preis für das Rezept ausgewiesen
- Das Rezept kann auf einfache Weise angepasst und neu berechnet werden (optional)

2.1.2 User Story «Web-Scraping»

Als Administrator möchte ich die Produktdatenbank aktualisieren können, damit in der Web-App mit den aktuellen Produktdaten gearbeitet werden kann.

Akzeptanzkriterien:
- Die Datenbank ist befüllt mit korrekten Daten
- Das Web-Scraping wird umgesetzt für Migros (optional) und Coop

- Das Web-Scraping zum Aktualisieren der Produktdatenbank kann manuell angestossen werden, worauf die Datenbank automatisch befüllt wird (optional)
- Das Web-Scraping kann über einen REST-Endpoint angestossen werden (optional)

2.1.3 User Story «Nährwerte-Rechner»

Als Benutzer möchte ich neben den Preisinformationen zu einem Rezept auch die Nährwerte sehen.

Akzeptanzkriterien:
- Zu den Produkten des Rezepts werden, falls vorhanden, die Nährwerte, auf die gewünschte Menge heruntergerechnet, ausgewiesen
- Produkte bei welchen die Nährwerte fehlen, werden als solche ausgewiesen
- Es werden die aufsummierten Nährwerte des Rezepts ausgewiesen

Diese User Story kam erst nach dem Advisor Meeting vom 15. November hinzu und wurde dabei als wichtiger als die vorher fest eingeplante User Story «Alternative Produkte» eingestuft. Aus diesem Grund wurde die User Story «Alternative Produkte» zu einer Erweiterung umgewandelt und die User Story «Nährwerte-Rechner» an ihrer Stelle fix eingeplant.

2.1.4 User Story «Alternative Produkte» (Erweiterung)

Als Benutzer möchte ich für jedes vorgeschlagene Produkt zusätzlich aus ähnlichen Alternativen auswählen können, um dadurch einen optimalen Produktwarenkorb für mein Rezept zusammenstellen zu können.

Akzeptanzkriterien:
- Man erhält eine Übersicht über Alternativen zum vorgeschlagenen Produkt
- Es kann ein alternatives Produkt ausgewählt werden, welches das vorgeschlagene Produkt im Preisrechner ersetzt

2.1.5 User Story «Rezept Sammlung» (Erweiterung)

Als Benutzer möchte ich bereits erfasste Rezepte speichern können, damit ich diese zu einem späteren Zeitpunkt wieder anschauen und erneut berechnen kann.

2.2 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen (NFR) orientieren sich an FURPS+. Dabei wird nur auf die für das Projekt am relevantesten Bereiche von FURPS+ eingegangen. Für die Priorisierung der NFRs werden folgende drei Priorisierungsstufen verwendet:
Hoch (+), Normal (*) und Niedrig (-)

2.2.1 Usability

NFR1: Gute Bedienbarkeit (-)
Die Web-App ist gut bedienbar. Alle relevanten Informationen werden in der Benutzeroberfläche übersichtlich dargestellt und ein neuer Benutzer findet sich schnell zurecht.

Akzeptanzkriterien (manuelle Prüfung durch Usability Test):
- Ein Benutzer, welcher die Web-App zum ersten Mal benutzt, schafft es ein Rezept von 5 Zutaten innerhalb von 2 Minuten zu erstellen.

2.2.2 Performance

NFR2: Paralleler Betrieb (*)
Eine Spring Boot Applikation kann im normalen Betrieb bis zu 200 Anfragen pro Sekunde abarbeiten. Um die Performance zu erhöhen, können mehrere Instanzen gleichzeitig hochgefahren werden. Dazu wird ein Loadbalancer benötigt.

Akzeptanzkriterien (manuelle Prüfung):
- Anfragen werden von zwei Backend-Instanzen abgearbeitet.

NFR3: Schnelle Anfragen (+)
Die Round-Trip Time für einen Request (Frontend → Backend → Frontend) ist gering.

Akzeptanzkriterien (manuelle Prüfung):
- Nach Eingabe eines Rezepts und dem Drücken des Senden-Buttons wird die generierte Einkaufsliste innerhalb von 2 Sekunden angezeigt.

2.2.3 Supportability

NFR4: Wartbarkeit (*)
Eine gute Testabdeckung durch Unit Tests erlaubt es die Wartbarkeit der Software über die Projektdauer hinaus zu gewährleisten.

Akzeptanzkriterien (automatisierte Prüfung):
- Die Unit Tests bieten im Backend eine Code-Abdeckung von mindestens 80% für die Service-Klassen.
- Alle Tests sind grün.

3 Architektur und Design

Als Erstes wird die gewählte Architektur vorgestellt und die Wahl der Technologien begründet. Anschliessend folgt das geplante Datenbank-Modell und ein Einblick in die ersten Ansätze für das UI in Form von Mockups.

3.1 Architektur und Technologien

Das Diagramm in Abbildung 1 zeigt den Aufbau der gewählten Software-Architektur. Die einzelnen Container (hellblaue Boxen) enthalten dabei einerseits die Deklaration der für die Umsetzung ausgewählten Technologien, andererseits eine kompakte Beschreibung der Verantwortlichkeit innerhalb des Pantry-Systems. Darüber hinaus ist im Diagramm auch die Kommunikation zwischen den einzelnen Containern beschrieben und die dafür jeweils verwendete Technologie deklariert.

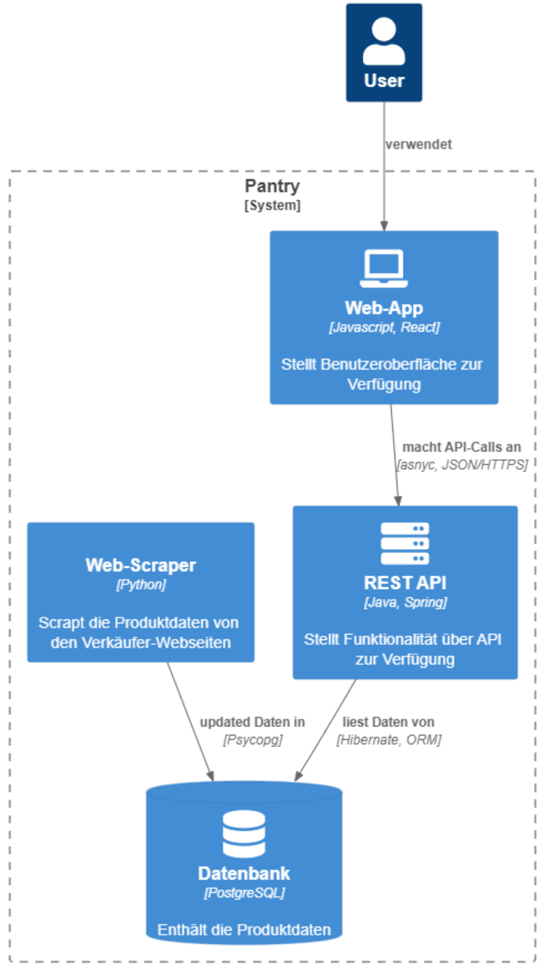


Abbildung 1: Architektur und Technologien

Nachfolgend wird die Wahl der Technologien für die einzelnen Container begründet:

Frontend

Im Frontend hat man sich für eine Web-App anstelle einer nativen App entschieden. Dies aus mehreren Gründen.

1. Da gemäss der Anforderungsspezifikation keine Hardware nahe Funktionalität, wie z.B. Sensoren, benötigt wird.
2. Die Web-App ist damit auch auf Desktop-Computern ohne Mehraufwand nutzbar.
3. Die grössere Erfahrung innerhalb des Teams in der Entwicklung von Web-Apps gegenüber nativer Apps.
4. Die gemäss Betreuerin geringere Priorität des Frontends gegenüber dem Backend und der Datenbank.

Nach dem Grundsatzentscheid für eine Web-App musste man sich noch für ein Javascript-Framework entscheiden, mit welcher man die Web-App umsetzen möchte. Hier drängten sich die drei populären Frameworks Angular, React und Vue.js auf. Da alle Frameworks für die Frontend-Bedürfnisse geeignet sind, wurde die Entscheidung vor allem auf die bereits gemachten Erfahrungen der Teammitglieder mit den verschiedenen Frameworks abgestützt um dann möglichst gut vorwärtszukommen.
Da leider nicht beide Teammitglieder im gleichen Framework am meisten Erfahrung haben, sondern Daniel in Angular und Zvonimir in React, musste zwischen diesen beiden Frameworks eine Entscheidung getroffen werden. Hier gab am Schluss den Ausschlag, dass sich im Rahmen der Studienarbeit vor allem Zvonimir um die Entwicklung des Frontends kümmern wird. Aus diesem Grund wurden seine bereits gemachten Erfahrungen höher gewichtet und React verwendet.

Backend

Im Backend war nach dem ersten Advisor Meeting bereits klar, dass es eine REST API geben soll. Hier standen generell zwei populäre Ansätze im Fokus, nämlich das Spring Framework und ASP.NET Core, welche sich beide für die Umsetzung des Backend geeignet hätten.
Auch in diesem Fall war der Erfahrungslevel der Teammitglieder wieder unterschiedlich verteilt. Ausschlaggebend war dann wiederum, wie im Frontend, die grössere Erfahrung von Zvonimir mit dem Spring Framework gegenüber ASP.NET Core, da er auch in der Backend Entwicklung den Lead übernehmen wird, während sich Daniel vorwiegend auf den Web-Scraper fokussiert.

Datenbank

Bei der Datenbank bevorzugten beide Teammitglieder eine relationale Datenbank. Dabei fiel die Wahl auf PostgreSQL, da beide Teammitglieder während dem Studium bereits gute Erfahrungen mit dieser kostenlosen Datenbank sammeln konnten.

Web-Scraper

Eine Recherche im Internet hat ergeben, dass Python für das Web-Scraping die Programmiersprache der Wahl ist. Da die Teammitglieder zudem im Web-Scraping Bereich über keinerlei Erfahrungen verfügen, gab es keinen Grund nicht auf Python zu setzen.

3.2 Datenbank Modell

Das gewählte Datenbank Modell, welches in Abbildung 2 dargestellt wird, besteht aus vier Tabellen. Diese werden nachfolgend kurz beschrieben.

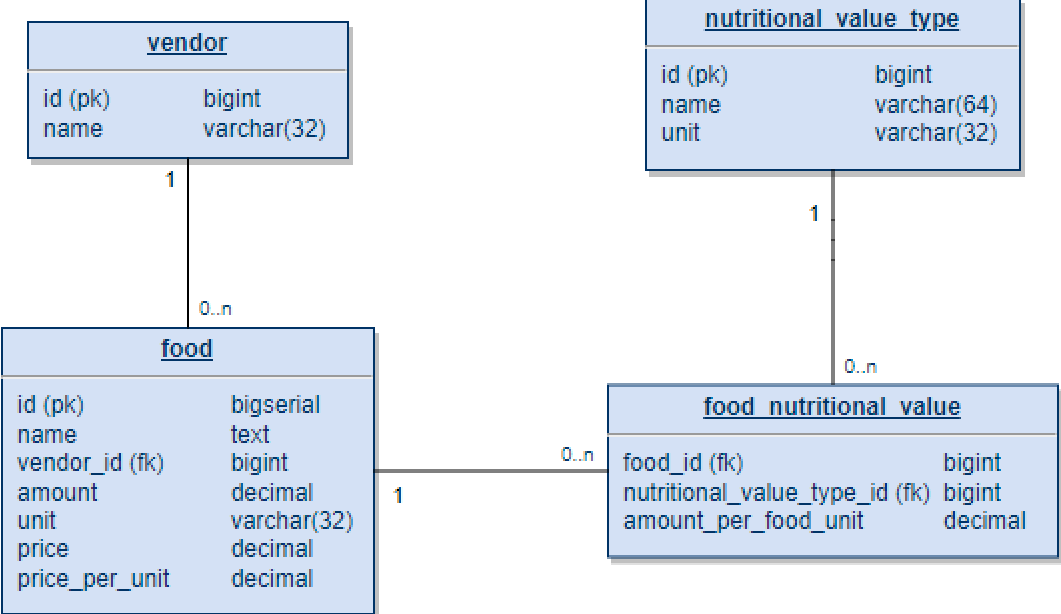


Abbildung 2: Datenbank Modell

Vendor:
Die «vendor»-Tabelle enthält die einzelnen Verkäufer (z.B. Migros und Coop) dessen Produktdaten in der Datenbank vorhanden sind. Diese Tabelle wird im Voraus befüllt.

Food:
Die «food»-Tabelle enthält die Basisinformationen zu den einzelnen Produkten. Diese Tabelle wird durch den Web-Scraper befüllt, welcher die Produktinformationen von den Webseiten der Verkäufer sammelt, aufbereitet und schliesslich in die Tabelle einfügt.

Nutritional Value Type:
Die «nutritional_value_type»-Tabelle enthält die unterschiedlichen Nährwerttypen (z.B. Fett, Protein), welche unterstützt werden. Diese Tabelle wird im Voraus befüllt.

Food Nutritional Value:
Die «food_nutritional_value»-Tabelle enthält die Nährwertinformationen zu einem Produkt. Die einzelnen Nährwerte sind in der «amount_per_food_unit»-Spalte dabei jeweils auf die Basiseinheit des Produktes, also «pro Gramm» oder «pro Milliliter», heruntergerechnet. Diese Tabelle wird, wie die «food»-Tabelle, durch den Web-Scraper befüllt.

3.3 Mockups

Für die Umsetzung der Web-App wurden zu Beginn des Projekts erste Mockups erstellt. Im Zentrum stand dabei, dass die Benutzeroberfläche möglichst simpel und einfach zu bedienen sein soll. Dem Benutzer sollte es beispielsweise leichtfallen, neue Zutaten für sein Rezept zu erfassen. Die erstellten Mockups sind im Anhang zu finden.

Die Mockups beziehen sich auf die ursprüngliche Idee des «Price Calculator», da sie zu Beginn des Projekts erarbeitet wurden. Aus diesem Grund sind in den Mockups keine Nährwertinformationen zu finden.

4 Qualitätsmassnahmen

In diesem Kapitel werden die angewendeten Qualitätsmassnahmen vorgestellt, welche dabei helfen sollen am Ende ein qualitatives Endprodukt abzuliefern. Als Erstes wird auf den gewählten Git-Workflow eingegangen. Danach folgt eine Aufzählung der verwendeten Linters und eine Beschreibung der für das Projekt aufgebauten CI-Pipeline. Abschliessend wird dann noch auf das Testkonzept eingegangen.

4.1 Git-Workflow

Der gesamte Quellcode wird in GitLab in einem einzelnen Repository verwaltet. Dabei werden zwei geschützte Branches «main» und «develop» eingesetzt, wodurch kein direktes Pushen auf die Branches möglich ist. Um auf diese Branches Code zu pushen, wird mit Merge-Requests gearbeitet. Dies hat einerseits den Vorteil, dass durch die gegenseitige Kontrolle die Code-Qualität gesteigert werden kann und andererseits, dass beide Teammitglieder einen guten Überblick über den Stand des Projekts haben.
Die Entwicklung neuer Funktionalität findet dabei über Feature-Branches statt. Der Workflow sieht folgendermassen aus:

1. Lokal den «develop»-Branch pullen.
2. Ausgehend vom «develop»-Branch einen neuen Feature-Branch erzeugen.
   a. Als Name für den neuen Branch ist generell die Nummer des Tasks im YouTrack zu verwenden. Beispiel: *pc-30*.
3. Änderungen lokal vornehmen um Ziele des Tasks zu erfüllen, dabei einen oder mehrere Commits vornehmen.
   a. Die Commit-Nachrichten sollen folgendes Format haben: *pc-30: comment*
      i. Zuerst kommt die Nummer aus YouTrack für die Zuordnung des Commits zum Task
      ii. Dann folgt ein Kommentar in englischer Sprache, welcher maximal eine Zeile lang sein soll
4. Den lokalen Feature-Branch mit den Änderungen pushen und einen Merge-Request erstellen.
   a. Wenn der Reviewer mit den Änderungen zufrieden ist, kann er den Feature-Branch direkt in den «develop»-Branch mergen.
   b. Wenn der Reviewer mit den Änderungen nicht zufrieden ist, dann schreibt er einen Kommentar mit seinen Verbesserungsvorschlägen an den Ersteller des Merge-Requests. Dieser kann dann die Verbesserungen anbringen und erneut zur Kontrolle freigeben. Dies geht dann solange hin und her bis Fall (a.) eintritt.

Der «main»-Branch ist für die produktive Umgebung gedacht. In diesen wird vom «develop»-Branch aus gemerged, sobald neue Funktionalität in einem stabilen Zustand vorliegt.

4.2 Linting

Im Frontend wird für das Linting auf ESLint und Prettier gesetzt. Im Backend wiederum wird Checkstyle zur statischen Codeanalyse eingesetzt.

4.3 Continous Integration

Im GitLab ist eine CI Pipeline eingerichtet, welche für Merge-Requests aktiviert ist. Dies bedeutet, dass bei jedem neuen Merge-Request die Pipeline durchlaufen wird. Dasselbe passiert auch, wenn neue Commits zu einem bestehenden Merge-Request hinzukommen.

Die Pipeline überprüft dabei mehrere Dinge:
- Im Frontend wird geprüft, ob der Code keine ESLint Fehler aufweist und ob der Code in einem von Prettier akzeptierten Zustand vorliegt
- Im Backend wird geprüft, ob der Code die in Checkstyle definierten Regeln einhält. Zudem wird kontrolliert, ob alle Unit Tests erfolgreich durchlaufen.

Falls eine dieser Überprüfungen fehlschlägt, schlägt die Pipeline fehl. Dann ist es nicht möglich den Merge-Request zu mergen bis die Probleme im Code beseitigt sind.

Diese Massnahmen erzwingen, dass alle Entwickler sich an einen Code-Style halten und verhindern somit Konflikte. Die Unit Tests, welche alle erfolgreich durchlaufen müssen, sind zwar keine Garantie, dass neuer Code vollumfänglich korrekt funktioniert, jedoch ein notwendiger Schritt um die Codequalität hochzuhalten.

4.4 Testkonzept

Das Ziel des Testkonzepts ist es die wichtigsten Bereiche der Software abzudecken und dadurch den Qualitätsstandard zu sichern. Dabei werden zwei verschiedene Arten von Tests eingesetzt, welche der nachfolgenden Tabelle entnommen werden können.

Testart	Wann?	Wer?	Wie?
Unit Tests	Beim Erstellen des Merge Requests	GitLab CI Pipeline	Automatisch
Usability Tests	Nach Fertigstellung des Prototypen (M4) und des Endproduktes (M5)	Teammitglied mit Drittperson	Manuell

Tabelle 1: Testarten

4.4.1 Unit Tests

Die Unit Tests beschränken sich im Rahmen der Studienarbeit auf das Backend. Dies aus folgenden Gründen:

- Der grösste Teil der Logik wird im Backend verbaut.
- Das Backend bzw. die REST-Schnittstelle hat im Rahmen des Projekts einen höheren Stellenwert als das Frontend.

Im Backend liegt der Fokus für die Unit Tests auf den Service-Klassen, welche die Business Logik enthalten. Hier wurde gemäss NFR4 (Wartbarkeit) eine Code-Abdeckung von 80% als Ziel gesetzt.

Um die Unit Tests im Backend aufzugleisen, werden die Tools aus der nachfolgenden Tabelle eingesetzt.

Tool	Verwendung
JUnit5	Unit Testing
Mockito	Mocking

Tabelle 2: Testing Tools

JUnit5 wurde ausgewählt, da das Framework sehr weit verbreitet ist und damit bei allfälligen Problemen viel Material zur Verfügung steht.

4.4.2 Usability Tests

Durchführungen
Es sind zwei Usability Tests geplant. Der erste Usability Test soll durchgeführt werden, sobald der Prototyp entwickelt ist, dies vor allem um erstes Feedback bezüglich dem NFR1 (Gute Bedienbarkeit) zu erhalten. Der zweite Usability Test ist dann zum Ende des Projekts geplant, wenn das Endprodukt fertig entwickelt ist. Auch hier steht das NFR1 noch einmal im Fokus, zusätzlich aber auch das NFR3 (Schnelle Abfragen), welches eine hohe Priorität hat.

Ablauf
Beide Teammitglieder führen die geplanten Usability Tests jeweils mit einer Drittperson durch. Für den Usability Test wurde dabei ein Testprotokoll entwickelt, welches die Testperson anweist, was sie zu tun hat.
Während des Usability Tests ist das Teammitglied als Moderator dabei. Das Ziel ist es nun, dass die Testperson möglichst selbstständig, ohne die Hilfe des Moderators, dem Testprotokoll folgt und dabei seine Eindrücke mitteilt. Die Aufgabe des Moderators ist es das Feedback aufzunehmen und für allfällige Fragen jederzeit zur Verfügung zu stehen.
Nach Abschluss der Usability Tests wird das Feedback der Testpersonen jeweils zusammengetragen und allfällige Verbesserungsmöglichkeiten in die Entwicklung miteinbezogen.

Resultat

Der erste Usability Test wurde schlussendlich nicht durchgeführt. Dies liegt daran, dass sich im Verlauf des Projekts herausgestellt hat, dass der Fokus weniger auf das UI gelegt und die Zeit besser in die Entwicklung des Web-Scrapers und des Backends investiert werden soll. Das NFR1 (Gute Bedienbarkeit) wurde deshalb im Projektverlauf auch von der Priorität «Normal» auf «Niedrig» zurückgestuft.

Der zweite Usability Test konnte in Sprint 11 durchgeführt werden. Die daraus gewonnenen Erkenntnisse flossen so weit möglich noch in das Endprodukt ein. Die Protokolle der durchgeführten Usability Tests sind im Anhang zu finden.

5 Umsetzung

Das folgende Kapitel beschreibt die Umsetzung der einzelnen Komponenten des Projekts. Zuerst wird auf Frontend und Backend eingegangen. Anschliessend gibt es Informationen zur Umsetzung des Web-Scrapers und am Schluss geht es noch um das Deployment bzw. die Auslieferung der Applikation.

5.1 Frontend

Wie in Kapitel 3.1 beschrieben, wurde React als Library für das Frontend gewählt. Bei der Umsetzung des Frontends wurden jedoch noch weitere Libraries eingesetzt, deren Wahl nachfolgend begründet wird. Anschliessend wird auf die Verzeichnisstruktur, sowie auf verschiedene Herausforderungen während der Entwicklung eingegangen.

5.1.1 Entscheid bezüglich verwendeten Libraries

Nachfolgend werden die Entscheidungen bezüglich den verwendeten Libraries begründet. Zuerst wird die Wahl von Ant Design als Komponenten-Library begründet. Anschliessend wird erklärt, welche Lösung für das Routing bzw. die Navigation innerhalb der Web-App eingesetzt wird.

5.1.1.1 Komponenten-Library

Für React gibt es viele verschiedene und gute Komponenten-Libraries. Einige dieser Libraries wurden auf die Eignung für das Frontend geprüft. Die Wahl wurde dabei frühzeitig auf die folgenden vier Libraries eingeschränkt:

- Ant Design
- Material UI
- React-Bootstrap
- Semantic UI

Jede dieser Libraries hätte eingesetzt werden können. Material UI zum Beispiel zeichnet sich durch äusserst schöne Designs aus. Es enthält viele gute Komponenten, welche für dieses Projekt geeignet gewesen wären. Dasselbe gilt für React-Bootstrap. Bei beiden dieser Libraries geht es in erster Linie um Design und daher sind die Funktionen der einzelnen Komponenten einfach gehalten. Wohl auch deshalb bieten diese beiden Libraries keine optimale Option für die Umsetzung des komplexen Formulars für die Rezepterfassung an. Aus diesem Grund kamen nur noch Ant Design und Semantic UI React in Frage. Hier fiel die Entscheidung aufgrund bereits gemachter Erfahrungen in früheren Projekten mit Ant Design zu Gunsten dieser Library aus.

5.1.1.2 React Router

Da die Web-App aus mehreren Seiten besteht, benötigt es auch ein Routing. Hierfür genügte die Verwendung der Standardlösung React Router, welche auch vom React-Team empfohlen wird.

5.1.2 Verzeichnisstruktur

Als Ausgangslage für die Verzeichnisstruktur im Frontend konnte die vom Create-React-App erzeugte Struktur verwendet werden.
Die Web-App besteht aus zwei Ansichten (Views) und mehreren kleineren Komponenten. «App.jsx» enthält die Routing-Logik für das Wechseln zwischen den Ansichten.
Im Verzeichnis «styling» gibt es eine CSS-Datei «colors.css», welche alle verwendeten Farben enthält. Dazu gibt es eine weitere CSS-Datei «columnRepsonsiveProperties.cs», welche Responsive-Eigenschaften enthält. Als «Utils» gibt es zwei simple Funktionen. Eine Funktion für das korrekte Anzeigen von Einheiten (Beispiel: 1 gram / 2 grams) und eine weitere Funktion für die Überprüfung ob Nährwerte für ein Rezept vorhanden sind.
Alles was mit Anfragen an das Backend zu tun hat liegt im «api» Verzeichnis.
Besonders ist noch die «Caddyfile»-Datei. Diese enthält die Konfiguration für einen Caddy-Webserver, welcher im Docker-Container verwendet wird. Mehr Informationen dazu befinden sich im Kapitel 5.4.
Ansonsten ist die Struktur ziemlich einfach gehalten und weicht nicht vom Standard ab.

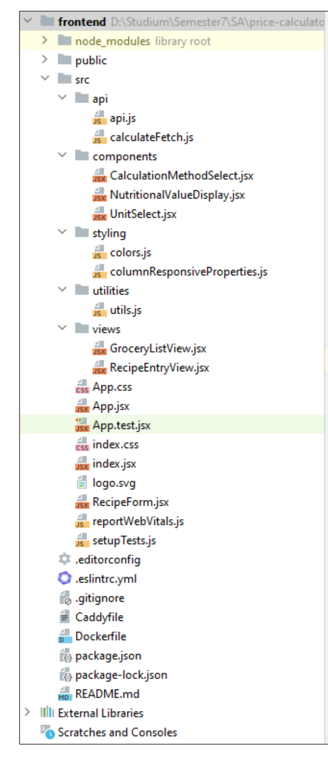


Abbildung 3: Verzeichnisstruktur Frontend

5.1.3 Herausforderungen

Da im Frontend mit bereits bekannten Libraries gearbeitet wurde, kam es während der Umsetzung zu keinen grösseren Problemen. Trotzdem gab es zwei Knackpunkte.

5.1.3.1 Styling

Beim ersten Prototyp wurde kein eigenes Styling eingesetzt, sondern nur das von Ant Design vorgegebene Styling verwendet. Dieses Styling war äusserst schlicht und nicht sehr ansprechend. Hier merkte man, dass eine Komponenten-Library wie zum Beispiel Material UI eine gute Sache gewesen wäre. Da das Frontend gesamthaft weniger im Vordergrund der Arbeit stand, wurde einfach mit eigenen Farben gearbeitet, um das Design attraktiver zu machen.

5.1.3.2 Neue User Story «Nährwerte-Rechner»

Zu Beginn war das UI der vorgeschlagenen Einkaufsliste hauptsächlich auf die Darstellung von Zutaten und die dazu vorgeschlagenen Produkte ausgerichtet. Mit der nachträglichen Berücksichtigung (vgl. Kapitel 2.1.3) der Nährwerte musste das UI nochmals komplett überarbeitet werden um diese zusätzlichen Informationen auch noch sauber darstellen zu können.
Eine der grösseren Herausforderungen war dabei der Umgang mit Sonderfällen. Beispielsweise wenn man nur Produkte hat, welche über keine Nährwertangaben verfügen oder nur Zutaten, zu denen keine Produkte gefunden werden konnten. Nach ausgiebigem Testen konnten aber auch diese Sonderfälle alle korrekt behandelt werden.

5.2 Backend

Dieses Kapitel behandelt das auf Spring Boot basierende Backend. Da das Spring Framework mit Spring Boot nicht alle benötigten Funktionalitäten des Backends abdeckt, werden weitere Libraries, Frameworks benötigt. Die wichtigsten zusätzlich eingesetzten Libraries und Frameworks als auch Plugins werden nachfolgend beschrieben sowie der Grund für ihre Verwendung erläutert. Danach wird auf die Verzeichnisstruktur eingegangen und zum Schluss auf verschiedene Herausforderungen während der Entwicklung.

5.2.1 Entscheid bezüglich verwendeten Libraries, Frameworks und Plugins

Spring Boot liefert bereits eine sehr gute Grundlage für eine REST-API, trotzdem werden noch einige weitere Libraries und Frameworks benötigt, um eine vollständige und auch nützliche Applikation bauen zu können. Diese Libraries und Frameworks als auch zusätzlich verwendete Plugins werden nachfolgend vorgestellt.

5.2.1.1 Datenbank

Da für die Datenbank PostgreSQL verwendet wird, bietet sich Spring Data JPA mit Hibernate als darunterliegenden OR-Mapper an. Dies ist eine Kombination, welche sich in früheren Projekten der Teammitglieder bereits bewährt hat. Auf der Grundlage, dass Spring Data JPA mit Hibernate alles was benötigt wird, abdeckt, wurde auf die Evaluation anderer Möglichkeiten verzichtet.
Die Spring Data JPA mit Hibernate bietet dabei eine einfache Möglichkeit um Daten aus der Datenbank zu laden und zu manipulieren. So können die Entitäten der Datenbank ganz einfach als Klassen beschrieben und verwendet werden (siehe Abbildung 4). Zudem können Repositories verwendet werden, welche die SQL-Abfragen für die Datenbank selbstständig generieren.

```java
@Entity
public class Food {

    4 usages
    @Id
    @Column
    private Long id;

    4 usages
    @Column
    private String name;

    4 usages
    @ManyToOne
    @JoinColumn(name = "vendor_id")
    private Vendor vendor;

    4 usages
    @Column
    private BigDecimal amount;

    4 usages
    @Column
    @Enumerated(EnumType.STRING)
    private SupportedUnit unit;

    4 usages
    @Column
    private BigDecimal price;

    4 usages
    @Column
    private BigDecimal pricePerUnit;

    6 usages
    @OneToMany(mappedBy = "food")
    @JsonManagedReference
    List<FoodNutritionalValue> foodNutritionalValueList;
```

Abbildung 4: Beispiel einer Entity Klasse

5.2.1.2 Unit Testing

Für das Unit Testing des Backends wurde JUnit5 gewählt. Auch hier war die Entscheidung einfach, da dieses Framework alles was benötigt wird abdeckt.
Zum Mocken von Repositories ist das Mockito Framework im Einsatz. Die Entscheidung für das Mockito Framework erfolgte aufgrund der einfachen Integrität in JUnit5 und aufgrund des breiten Einsatzes in der Industrie.

Für das Generieren der Testreports wird das Jacoco-Plugin in Kombination mit dem Maven-Surefire-Report-Plugin verwendet. Diese Plugins wurden gewählt, weil sie am besten geeignet sind für die Integration in GitLab. Jacoco erlaubt die Definition von einer benötigten Testabdeckung in Prozent und eine Visualisierung der Testabdeckung im Code. Das Maven-Surefire-Report-Plugin wiederum kann aus den generierten Reports von Jacoco, einen Testreport generieren welcher visuell im GitLab dargestellt werden kann. Somit ist nach jedem Commit eine saubere Übersicht der Tests gegeben.

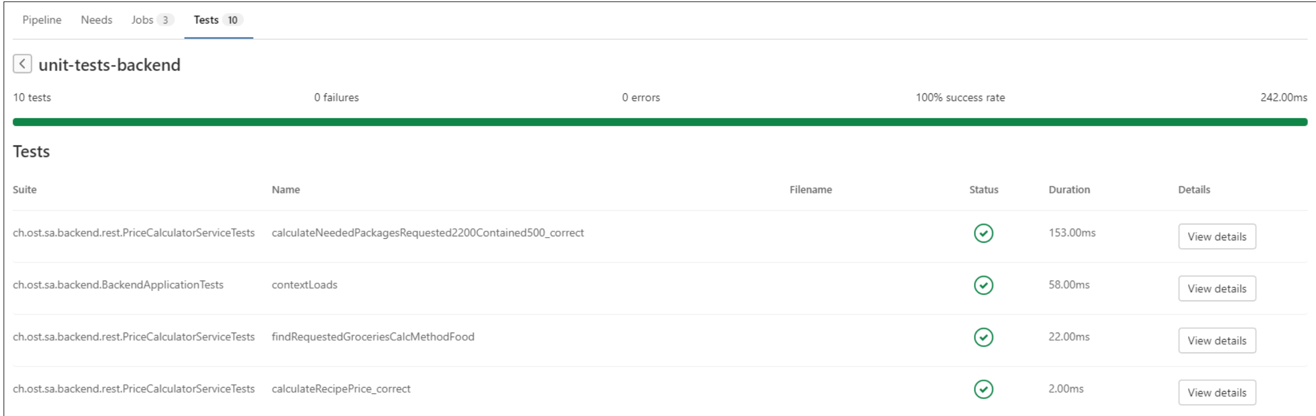


Abbildung 5: Testreport in GitLab

5.2.1.3 API-Dokumentation

Zur Generierung der API-Dokumentation ist springdoc-openapi im Einsatz. Diese Library wurde gewählt, da sich dadurch mit wenig Aufwand eine saubere API-Dokumentation generieren lässt.

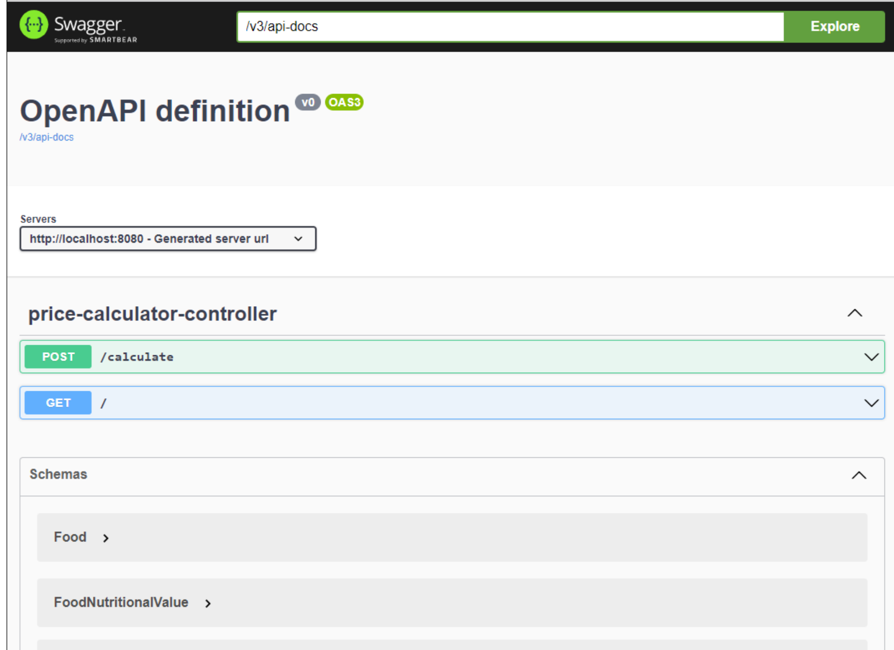


Abbildung 6: API-Dokumentation

Ausserdem erlaubt es OpenAPI die Endpoints des Backends auf einfache Weise zu testen. Dies war bei der Umsetzung von enormem Nutzen.

5.2.1.4 Dokumentation für Entwickler

Auch für Entwickler kann eine Dokumentation mit JavaDocs generiert werden. Dies ist für Java-Applikationen der Standard. Diese Dokumentation ermöglicht neuen Entwicklern die schnelle Einarbeitung in die bestehende Codebasis und hat deshalb einen hohen Nutzen.

5.2.2 Verzeichnisstruktur

Beim Backend ist eine einfache Verzeichnisstruktur genügend. In der Standard Javastruktur ist der gesamte Code abgelegt. Im Zentrum des Backends stehen der «CalculatorController» und ein dazugehöriger «CalculatorService». Für die Datenbank werden, wie bereits in Kapitel 5.2.1.1 beschrieben, Repositories verwendet.

Das Verzeichnis «config» enthält die für Checkstyle benötigte Konfiguration. Im Verzeichnis «database» wiederum sind alle Skripte für das Aufsetzen der Datenbank vorhanden.

Die Applikation ist im «application.yml» konfiguriert. Es gibt eine Konfiguration für die lokale Entwicklung als auch eine für die produktive Umgebung. Dabei erweitert bzw. überschreibt die «produktive» Konfiguration die Grundkonfiguration.
Auf das Trennen der beiden Konfigurationen in einzelne Dateien wurde bewusst verzichtet, da beide Dateien noch sehr klein sind und es somit übersichtlicher ist dies in einer Datei zu behalten.

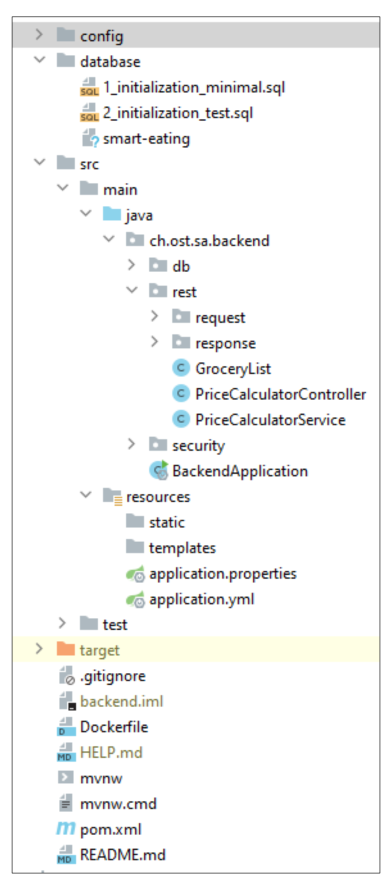


Abbildung 7: Verzeichnisstruktur Backend

5.2.3 Herausforderungen

Bei der Umsetzung des Backends konnte ziemlich schnell eine gute Grundlage geschaffen werden. Dies geschah ohne grössere Herausforderungen, so konnten bereits früh Produkte für Zutaten vorgeschlagen werden. Als diese Grundlage geschaffen war, kamen jedoch die Herausforderungen.

5.2.3.1 Anwendung der Berechnungsmethoden

Eine der grössten Herausforderungen war die einfache und korrekte Anwendung der beiden Berechnungsmethoden auf die potenziellen Produkte:

- «Cheapest price per unit» soll das Produkt mit dem tiefsten Preis auf die Einheit heruntergerechnet finden.
- «Cheapest price per product» soll das günstigste Produkt finden.

Berechnungsmethode: Cheapest price per unit

In einer ersten Version wurden zuerst alle Produkte, welche den gesuchten Zutatennamen enthalten von der Datenbank zurückgeliefert. Dabei wurden darauf alle Produkte durchgegangen und von jedem Produkt der Preis pro Einheit berechnet. Diesen Preis erhält man, indem man die Menge durch den Preis teilt. Anschliessend wurden alle Preise verglichen und das günstigste Produkt kam in die Einkaufsliste.
Das Hauptproblem dieser Lösung war es, dass sie sehr ineffizient ist. Es wäre besser, wenn die Datenbank selber den Preis pro Einheit genormt auf die Menge (z.B. auf 1 Gramm) enthält. Genau dies konnte dann im Web-Scraper so umgesetzt werden, wodurch der Code im Backend stark vereinfacht werden konnte. So war es bei der Berechnungsmethode «Cheapest price per unit» nun möglich den Preis pro Einheit in der Datenbank direkt für Vergleiche zwischen den potenziellen Matches zu verwenden, wodurch es sehr einfach wurde das optimale Produkt zu finden.

Berechnungsmethode: Cheapest price per product

Die zweite Berechnungsmethode, bei welcher nach dem günstigsten Produkt gesucht wird, erschien zu Beginn einfach in der Umsetzung. Es sah so aus, als könnte man bei dieser Methode einfach aus allen potenziellen Matches das günstigste Produkt, welches mindestens die gewünschte Menge enthält, zurückgeben. Dabei wurde jedoch vergessen zu berücksichtigen, dass ein Produkt evtl. billiger ist als ein anderes Einzelprodukt, wenn man mehrere Packungen davon kauft. Dies kann mithilfe der folgenden Abbildung und der anschliessenden Erklärung besser nachvollzogen werden.

Produktmenge	Preis		100	250	500	1000	1500	2000	2500
1000	1.85		1.85	1.85	1.85	1.85	3.7	3.7	5.55
1000	2.95		2.95	2.95	2.95	2.95	5.9	5.9	8.85
1000	1.95		1.95	1.95	1.95	1.95	3.9	3.9	5.85
500	1.75		1.75	1.75	1.75	3.5	5.25	7	8.75
500	1.6		1.6	1.6	1.6	3.2	4.8	6.4	8
2500	4.5		4.5	4.5	4.5	4.5	4.5	4.5	4.5
1000	2.95		2.95	2.95	2.95	2.95	5.9	5.9	8.85

Abbildung 8: Beispiel zur Berechnungsmethode «Cheapest price per product»

Für das Aufzeigen des Problems wurden sieben verschiedene Mehle von der Coop-Webseite verwendet, welche unterschiedliche Produktmengen und Preise haben. Eine Zeile repräsentiert dabei jeweils ein Produkt. Auf der rechten Seite sieht man in der Kopfzeile verschiedene Mengen in Gramm, welche man von den verschiedenen Produkten kaufen möchte.
Die berechneten Zellen zu den einzelnen Mengen enthalten nun jeweils den Preis für die minimale Anzahl an Packungen, welche benötigt werden, um die gewünschte Menge zu erreichen. So sieht man beispielsweise, dass wenn man 2500 Gramm vom ersten Produkt kaufen möchte, muss man dafür 5.55 Franken ausgeben, weil man drei Packungen benötigt um die 2500 Gramm abzudecken (3 * 1.85 = 5.55).
Die grün markierten Zellen stellen jeweils den günstigen Preis für die gewünschte Menge dar. Man kann von der Abbildung ablesen, dass das günstigste Produkt je nach gewünschter Menge variiert.
Erst durch die Berücksichtigung dieses Sachverhalts konnte die Logik so implementiert werden, dass bei der Berechnungsmethode «Cheapest price per product» immer das für die gewünschte Menge optimale Produkt und die davon benötigte Anzahl Packungen vorgeschlagen wird.

5.2.3.2 Falsche Einheiten

Eine weitere Herausforderung war der Umgang mit den Einheiten. Konkret stellte sich die Frage, wie sollte die Applikation handeln, wenn man zum Beispiel nach Mehl in Stücken statt Gramm sucht. Zu Beginn wurden falsche Ergebnisse angezeigt, bzw. wurden die Masseinheiten gar nicht berücksichtigt. Das Ganze wurde schlussendlich so gelöst, dass Anfragen ohne Einheiten und Mengenangaben direkt verworfen werden. In der Einkaufsliste wird dann einfach vermerkt, dass kein Produkt zur angefragten Zutat gefunden werden konnte.

5.2.3.3 Berechnung der totalen Nährwerte

Die Einbindung der Nährwerte kam erst später im Verlauf des Projekts als Anforderung hinzu (vgl. Kapitel 2.1.3). Der Code war davor nur auf den Preis, die Mengen und die Einheiten ausgerichtet. Durch die von Anfang an klar definierte Struktur war es nicht sehr schwierig die neue Anforderung einzubringen. Die grösste Herausforderung lag darin die Summe der Nährwerte je Nährwertkategorie für das gesamte Rezept korrekt zu berechnen und so zu strukturieren, dass das Frontend gut damit umgehen kann. Dies liess sich schlussendlich lösen, indem alle Nährwerte einer Einkaufsliste in ihre jeweilige Nährwertkategorie gruppiert und dann deren Summen gebildet wurden.

5.3 Web-Scraper

Aufgrund der fehlenden Vorkenntnissen im Bereich Web-Scraping und der zeitlichen Rahmenbedingungen innerhalb der Studienarbeit, konnte nur das Scraping der Coop-Webseite umgesetzt werden.
Nachfolgend wird als Erstes der gewählte Scraping-Ansatz anhand der Coop-Webseite beschrieben. Anschliessend folgt eine Begründung des Entscheids, warum für das Web-Scraping am Ende auf «Scrapy & Playwright» gesetzt wurde. Danach werden die für den Web-Scraper erstellten Spider, Item und Pipelines vorgestellt. Es folgt eine Beschreibung der verwendeten Settings. Dann wird auf verschiedene Herausforderungen während der Entwicklung eingegangen. Zum Abschluss wird dann darauf hingewiesen, wie der Scraper in Betrieb genommen werden kann und wie lange das Scrapen aller Produkte von der Coop-Webseite in etwa dauert.

5.3.1 Ansatz zum Scrapen der Coop-Webseite

Es gibt verschiedene Möglichkeiten, wie man Daten aus einer Webseite extrahieren kann. Eine Möglichkeit besteht darin, dass man von einer Startseite aus einfach alle Links zusammensucht und zu allen hin navigiert, welche innerhalb derselben Domain (z.B. coop.ch) liegen und dies rekursiv macht, bis man durch alle Webseiten durch ist.
Diese Vorgehensweise wäre für das Extrahieren der Produktdaten von der Coop-Webseite allerdings sehr ineffizient, da schlussendlich nur die Lebensmittel von Interesse sind. Aus diesem Grund wurde geprüft, ob das Scraping auch einfacher und effizienter lösbar wäre und tatsächlich ist dies möglich. Nachfolgend wird der angewendete Scraping-Prozess anhand der Coop-Webseite beschrieben ohne auf technische Details einzugehen.

Der Scraping-Prozess wird von der Startseite der Coop-Webseite aus gestartet. Als Erstes werden auf der Startseite die Links auf die verschiedenen Produktkategorien gesammelt. Dabei wird einerseits der Link auf «Traiteur & Torten» ausgeschlossen, da dort keine sinnvollen Produktdaten drin sind und andererseits auch der Link auf die «Spezielle Ernährung», da diese Produktdaten in den anderen Kategorien bereits vorhanden sind.

Abbildung 9: Gesammelte Produktkategorienlinks auf Startseite

Als Nächstes werden alle gesammelten Produktkategorienlinks annavigiert, worauf jeweils eine neue Seite geöffnet wird, welche alle Produkte einer Kategorie enthält. Ein Beispiel einer Produktkategorie-Seite für die «Früchte & Gemüse» Kategorie ist hier zu finden.

Auf den Produktkategorie-Seiten sind die Produkte nun in mehrere Sub-Pages aufgeteilt und es werden jeweils standardmässig 30 Produkte pro Sub-Page angezeigt. Da die in dieser Ansicht pro Produkt dargestellten Informationen nicht ausreichen (die Nährwerte fehlen), müssen die einzelnen

Produktdetail-Seiten annavigiert werden. Aus diesem Grund gilt es alle Produktlinks auf der Sub-Page zu sammeln und diesen Links zu den Produktdetail-Seiten zu folgen. Dies wird dann rekursiv so lange über den Next- Button am Ende der Sub-Page gemacht, bis alle Produktlinks der Kategorie gesammelt und annavigiert wurden.

Abbildung 10: Next-Button auf Produktkategorie-Seite

Der letzte Schritt ist dann das eigentliche Scrapen der interessanten Daten von den Produktdetail-Seiten.

Abbildung 12: Basisdaten auf der Produktdetail-Seite

Abbildung 11: Nährwerte auf der Produktdetail-Seite

5.3.2 Entscheid bezüglich verwendeten Libraries

Für das Web-Scraping in Python gibt es einige Libraries, die einem dabei behilflich sein können. Die ersten Gehversuche im Web-Scraping wurden dabei nach einer ersten Recherche mit den Libraries «Selenium» und «BeautifulSoup» gemacht. Es stellte sich dabei schnell heraus, dass das Scrapen einer so umfangreichen und komplexen Webseite wie der von Coop mit nur mit diesen Libraries schwierig werden dürfte, so fehlte beispielsweise auch eine Möglichkeit zur einfachen Parallelisierung der Requests.

Aus diesem Grund wurde recherchiert, ob es nicht ein Framework gibt, welches das Web-Scraping erleichtert. Diese Recherche führte dann ziemlich schnell zu «Scrapy». Diese Library wird in sehr vielen Tutorials für das Scrapen von Webseiten verwendet und wurde deshalb zur ersten Anlaufstelle, welche sich dann auch bewährt hat. «Scrapy» allein reichte allerdings noch nicht, da «Scrapy» nur auf statische HTML-Seiten ausgerichtet ist, wo der Server jeweils direkt das ganze HTML zurückliefert.
Dies ist bei der Coop-Webseite jedoch nicht der Fall, da es sich um eine dynamische Webseite handelt, bei welcher Teile des HTML erst nachträglich über Javascript nachgeladen werden. Hier wurde also eine zusätzliche Library benötigt, welche es ermöglicht zu warten, bis ein bestimmter HTML-Tag nachgeladen ist. Erst dann kann man nämlich auf den HTML-Code zugreifen und die Daten von Interesse (Links, Produktdaten) aus dem HTML extrahieren.

Hier wurden bei der Analyse mehrere potentielle Libraries identifiziert, welche mit «Scrapy» kombiniert werden können. Dabei handelt es sich um das zu Beginn bereits ausprobierte «Selenium», sowie «Playwright», «Splash» und «Puppeteer». Folgender Artikel bietet einen guten Überblick über die vier

Libraries. Der Artikel erleichterte die Entscheidung für eine der Libraries sehr. So ist «Scrapy & Playwright» aktuell die «most up to date», am einfachsten einzusetzende und wahrscheinlich auch mächtigste Library für das Scrapen von dynamischen Webseiten (ScrapeOps (2022)).

Man entschied sich dazu die beiden Varianten «Scrapy & Playwright» und «Scrapy & Selenium» zusätzlich auch noch selber bezüglich Scraping-Geschwindigkeit zu vergleichen. Dies war einfach möglich, da sich die Libraries sehr ähnlich in «Scrapy» integrieren lassen.
Für den Vergleich wurde gemessen, wie lange es dauert, ausgehend von einer Produktkategorie-Seite, alle Produkte dieser Kategorie zu scrapen (vgl. Kapitel 5.3.1). Das Ergebnis in der Tabelle unten, sprach schlussendlich klar für die Verwendung von «Scrapy & Playwright».

Produktkategorie	Scrapy & Playwright	Scrapy & Selenium
Früchte & Gemüse (545 Produkte)	8min	20min
Milchprodukte & Eier (1295 Produkte)	18min	52min

Tabelle 3: Geschwindigkeitsvergleich Scraping

Neben «Scrapy & Playwright» wurden noch zwei weitere Libraries zur Realisierung des Web-Scrapers benötigt. Einerseits wird «BeautifulSoup» für das Extrahieren der Produktinformationen von den Produktdetail-Seiten eingesetzt (mehr dazu in Kapitel 5.3.7.1) und andererseits wird «Psycopg» verwendet um die frisch gescrapten Produktdaten in die PostgreSQL-Datenbank zu übertragen.

5.3.3 Spiders

Die Spider sind in «Scrapy» der Ausgangspunkt eines Scraping-Prozesses. In einem Spider wird definiert von welcher Webseite aus der Scraping-Prozess gestartet wird, welchen Links gefolgt und schlussendlich welche Daten von den besuchten Seiten extrahiert werden sollen (Scrapy (2022a)).
Der finale Spider zum Scrapen der Coop-Webseite wurde inkrementell in Form von mehreren Spidern erarbeitet. Diese Spider sind dabei ebenfalls im Endprodukt enthalten, da sie nützlich sind um einzelne Aspekte des finalen Spiders einfacher weiterzuentwickeln und testen zu können. Im Folgenden werden die Spider in der Reihenfolge, in welcher sie entwickelt wurden, kurz beschrieben.

5.3.3.1 CoopScrapSingleProductSpider

Dieser Spider fokussiert sich komplett auf das Extrahieren der interessanten Produktdaten von einer Produktdetail-Seite. Erst waren dies nur der Produktname, die Menge, Einheit und der Preis des Produktes, später kamen dann noch die Nährwerte hinzu.

5.3.3.2 CoopCollectProductLinksSpider

Bei diesem Spider geht es darum, ausgehend von einer Produktkategorie-Seite alle Produktlinks zu sammeln. Dabei wird über den Next-Button jeweils so lange weiter gemacht bis alle Sub-Pages durchnavigiert und die Produktlinks gesammelt sind.

5.3.3.3 CoopScrapCategoryProductsSpider

Bei diesem Spider handelt es sich um eine Kombination der ersten beiden Spider. Ausgehend von einer Produktkategorie-Seite werden alle Produktlinks gesammelt und die entsprechenden Produktdetail-Seiten annavigiert. Dort können anschliessend die interessanten Produktdaten extrahiert werden. Dieser Spider ermöglicht es also die Produktdaten aller Produkte einer Kategorie zu scrapen.

5.3.3.4 CoopCollectCategoryLinksSpider

Dieser Spider geht von der Coop-Startseite aus und macht nichts weiter als die interessanten Produktkategorienlinks zu sammeln. Die Links auf die Kategorien «Traiteur & Torten» sowie «Spezielle Ernährung» werden dabei ausgeschlossen.

5.3.3.5 CoopScrapAllProductsSpider

Der finale Spider kombiniert nun noch die Ansätze aus dem CoopCollectCategoryLinksSpider und dem CoopScrapCategoryProductsSpider und ermöglicht es damit die Produktdaten aller Produkte von allen interessanten Produktkategorien zu scrapen.

5.3.4 Items

Das Hauptziel vom Scraping ist es strukturierte Daten aus einer unstrukturierten Quelle, in diesem Fall aus den Coop-Webseiten, zu erhalten. Um dies zu erreichen, können in «Scrapy» die in den Spidern extrahierten Daten in Items abgelegt und anschliessend weiterverarbeitet werden (Scrapy (2022b)).
Diese Möglichkeit wird vom entwickelten Web-Scraper genutzt. So wird ein ProductItem eingesetzt, welches alle interessanten Informationen zu einem Produkt kapselt (siehe Abbildung 7). Dabei werden die im CoopScrapAllProductsSpider von einer Produktdetail-Seite extrahierten Daten in ein solches ProductItem abgelegt, bevor das Item an die Pipelines zur Weiterverarbeitung weitergereicht wird.

```
class ProductItem(scrapy.Item):
    name = scrapy.Field()
    price = scrapy.Field()
    amount = scrapy.Field()
    unit = scrapy.Field()
    price_per_unit = scrapy.Field()
    nutritional_values_per = scrapy.Field()
    energy_kcal = scrapy.Field()
    fat = scrapy.Field()
    saturated_fatty_acids = scrapy.Field()
    carbohydrate = scrapy.Field()
    sugar = scrapy.Field()
    dietary_fiber = scrapy.Field()
    protein = scrapy.Field()
    salt = scrapy.Field()
```

Abbildung 13: ProductItem

5.3.5 Pipelines

Nachdem ein Item von einem Spider gescrapt wurde, kann es zusätzlich eine Reihe von Pipelines durchlaufen, welche sequentiell abgearbeitet werden. Bei einer Pipeline handelt es sich dabei um eine einfache Methode, welche ein gescraptes Item als Parameter erhält. Innerhalb der Methode kann das Item dann modifiziert, gedroppt oder auf andere Weise weiterverarbeitet werden (Scrapy (2022c)).
Für die Aufbereitung der vom CoopScrapAllProductsSpider gelieferten Items bis hin zum Speichern der Produktdaten in der Datenbank kommen bei unserem Web-Scraper vier solche Pipelines zum Einsatz. Diese werden der Reihe nach, in welcher sie aufgerufen werden, kurz beschrieben und es wird auf ihren jeweiligen Funktionszweck eingegangen.

5.3.5.1 TransformCoopBaseInfoPipeline

Diese Pipeline kümmert sich um die Aufbereitung der Basisinformationen eines Produktes. Dazu gehören die Felder «price», «amount», «unit» und «price_per_unit» des ProductItem. In der Pipeline wird unter anderem die «unit», ausgehend von Kilogramm, Milligramm zu Gramm und von Zentiliter, Deziliter, Liter zu Milliliter, konvertiert und der «amount» entsprechend umgerechnet.
Des Weiteren wird auch noch das Feld «price_per_unit» anhand der Felder «price» und «amount» berechnet und im ProductItem gesetzt. Dieses zusätzlich berechnete «price_per_unit» Feld kann dann später in die Datenbank abgelegt werden, was im Backend den Preisvergleich zwischen einzelnen Produkten massgeblich erleichtert. Im Falle einer ungültigen «unit» wird ein gescraptes ProductItem gedroppt.

5.3.5.2 AvoidDuplicatesPipeline

Diese Pipeline kümmert sich darum, dass kein Produkt mehr als einmal in die Datenbank eingefügt wird. Diese Pipeline ist notwendig, da es auf der Coop-Webseite Produkte gibt, welche in mehreren Kategorien vorhanden sind. Die Pipeline unterhält dabei eine Liste aller ProductItems, welche die Pipeline bereits durchlaufen haben. Wenn ein neues ProductItem in die Pipeline kommt, wird geprüft, ob bereits ein ProductItem in der Liste vorhanden ist mit den gleichen Werten in den Feldern «name», «price», «amount» und «unit». Falls dies der Fall ist, dann wird das ProductItem gedroppt, ansonsten wird das neue ProductItem in die Liste aufgenommen.

5.3.5.3 TransformCoopNutritionInfoPipeline

Diese Pipeline kümmert sich um die Aufbereitung der Nährwerte eines Produktes. Hier wird zuerst geprüft, ob überhaupt Nährwertinformationen vorliegen, anhand des «nutritional_values_per» Feldes im ProductItem. Falls dies der Fall ist, dann werden, wo möglich, alle gescrapten Nährwerte aus dem ProductItem auf «pro Gramm» oder «pro Milliliter» Basis umgerechnet. Auch diese Umrechnungen erfolgen aus dem einfachen Grund, dass dies im Backend das Rechnen mit den Nährwerten wesentlich vereinfacht.

5.3.5.4 DatabasePipeline

Die Aufgabe dieser Pipeline ist es die in den letzten drei Pipelines aufbereiteten ProductItems in die Datenbank zu übertragen. Da die Produktdaten von Coop in der Datenbank nur geupdated werden sollen, wenn der ganze Scraping-Prozess erfolgreich durchgelaufen ist, werden während dem Scraping-Prozess alle gescrapten Items nur in ein Array aufgenommen.
Erst am Schluss, wenn der CoopScrapAllProductsSpider geschlossen wird und es keine Probleme gab, wird über «Psycopg» eine Verbindung zur Datenbank aufgebaut. Nun werden als Erstes alle Produkte von Coop aus der Datenbank gelöscht. Anschliessend werden aus dem Array, welches alle gescrapten und aufbereiteten ProductItems enthält, die entsprechenden Insert-Queries generiert und die Datenbank mit den frischen Produktdaten von Coop befüllt.

5.3.6 Settings

Die «Scrapy»-Settings erlauben das Verhalten der «Scrapy»-Komponente zu beeinflussen (Scrapy (2022d)). Es gibt eine grosse Anzahl an Möglichkeiten, welche glücklicherweise nicht benötigt wurden. Dies liegt vor allem daran, dass die Coop-Webseite bis jetzt keine spezifischen Massnahmen gegen Web-Scraper einzusetzen scheint (z.B. Captchas oder IP Blocking).
An dieser Stelle wird nur kurz auf zwei interessante Settings eingegangen, welche in «settings.py» explizit gesetzt wurden.

5.3.6.1 PLAYWRIGHT_DEFAULT_NAVIGATION_TIMEOUT

Das gesetzte Timeout wird bei den «Playwright»-Requests berücksichtigt (GitHub (2022)). Dieses Setting ist auf 180 Sekunden erhöht worden, weil mit dem Standard-Timeout von 30 Sekunden zu viele Timeout-Fehler passiert sind.

5.3.6.2 PLAYWRIGHT_ABORT_REQUEST

Nimmt eine Predicate-Funktion entgegen, welche einen Request erhält und entscheidet, ob der Request abgebrochen werden soll (GitHub (2022)). Dieses Setting wird genutzt, um Requests, welche Bilder abrufen, abzubrechen. Da keine Bilder benötigt werden, kann dadurch einerseits der Scraping-Prozess beschleunigt und andererseits die Coop-Webseite weniger belastet werden.

5.3.7 Herausforderungen

In diesem Kapitel wird auf die grössten Herausforderungen bei der Entwicklung des Web-Scrapers eingegangen. Viele der Themen erklären Entscheidungen, welche in Bezug auf den Source-Code des Web-Scrapers getroffen wurden.

5.3.7.1 Einsatz von BeautifulSoup zum Scrapen der Produktdaten

Für das Extrahieren von Informationen aus HTML-Sourcen bietet «Scrapy» seine eigenen Selektoren. Dabei werden sowohl XPath als auch CSS-Selektoren unterstützt (Scrapy (2022e)).
Im entwickelten Web-Scraper wird wo möglich auf die von «Scrapy» zur Verfügung gestellten Selektoren zurückgegriffen. Allerdings gab es an einem Ort Probleme. Das Extrahieren des Produktnamens von den Produktdetail-Seiten, wollte mit den Selektoren von «Scrapy» nicht gelingen. Aus diesem Grund musste eine alternative Lösung gesucht werden, worauf man rasch auf die populäre «BeautifulSoup» Library gestossen ist. Mit der Hilfe dieser Library war es möglich den Produktnamen zu extrahieren. Als Folge davon wurde entschieden alle Produktinformationen aus den Produktdetail-Seiten mit «BeautifulSoup» zu extrahieren.
Einerseits musste das HTML der Produktdetail-Seiten sowieso für «BeautifulSoup» aufbereitet werden, egal ob nur etwas oder mehrere Dinge über die Library extrahiert werden. Andererseits ist das saubere Extrahieren der gewünschten Produktdaten mit «BeautifulSoup» doch um einiges einfacher als mit den Selektoren von «Scrapy», da bei «BeautifulSoup» einige zusätzliche Features zur Verfügung stehen.

5.3.7.2 Verwendung von WSL

Das für die Integration von «Playwright» in «Scrapy» zu verwendende Package «scrapy-playwright» funktioniert im nativen Windows nicht (GitHub (2022)). Dies war am Anfang ein Problem, da beide Teammitglieder vorwiegend mit Windows Rechner unterwegs sind. Nach ersten Versuchen mit einer Ubuntu VM zur Entwicklung des Web-Scrapers, wurde relativ rasch der Schritt zu WSL auf Windows gemacht. Dies war die richtige Entscheidung, so wurde die Entwicklung dadurch angenehmer. Zudem lief auch der Scraping-Prozess schneller, da dadurch alle Ressourcen des Windows Rechners genutzt werden konnten.

5.3.7.3 Notwendigkeit des «dont_filter»-Attributs

Bei der Entwicklung des CoopScrapCategoryProductsSpider gab es das Problem, dass beim Sammeln der Produktdaten, ausgehend von einer Produktkategorie-Seite, am Ende immer einige Produkte fehlten. Erst durch das Setzen des «dont_filter»-Attributs auf «true» in allen «Playwright»-Requests wurden alle Produkte korrekt gescrapt. Warum dies ohne das Setzen von «dont_filter=True» auf allen Requests nicht korrekt funktioniert, konnte nicht herausgefunden werden.
Das «dont_filter»-Attribut ist im Normalfall nicht gesetzt, was zur Folge hat, dass identische URL-Aufrufe nur einmal erfolgen und danach rausgefiltert werden. Dies ist oft notwendig um Crawling-Loops zu verhindern (Scrapy (2022f)).
Beim Scrapen der Coop-Webseite kann das «dont_filter»-Attribut aber auf «true» gesetzt werden, da der gewählte Scraping-Ansatz (siehe Kapitel 5.3.1) keine solche Crawling-Loops produzieren kann. Es ist zwar möglich, dass eine URL von einem Produkt mehrmals aufgerufen und gescrapt wird z.B. wenn ein Produkt in zwei verschiedenen Produktkategorien enthalten ist. Durch die AvoidDuplicatesPipeline wird eine mehrmalige Eintragung eines Produktes in die Datenbank anschliessend aber verhindert.

5.3.7.4 Notwendigkeit von einem Kontext pro annavigierte Seite

Ein weiteres Problem trat auf beim Versuch alle Produktdaten der Kategorie «Vorräte» mithilfe des CoopScrapCategoryProductsSpider zu scrapen. Dabei produzierte der Spider nach einiger Zeit jeweils eine «JavaScript heap out of memory» Exception, worauf der Scraping-Prozess jeweils nicht mehr weitermachen konnte.
Es zeigte sich ziemlich schnell, dass andere Personen bereits mit demselben Problem konfrontiert waren. Auf GitHub gibt es zur «playwright» Library ein geschlossenes Issue, welches geholfen hat das Problem in den Griff zu bekommen.
Die Lösung ist es im Spider für jede annavigierte Seite einen neuen Kontext zu verwenden. Damit dies möglich ist, muss in allen «Playwright»-Requests neben dem neuen Kontext jeweils auch noch die Page inkludiert werden. Dies aus dem Grund, dass es nur über die Page möglich ist, auf den Kontext zuzugreifen und diesen wieder zu schliessen. Dabei muss sowohl der Kontext als auch die Page nach jedem Abruf einer Seite manuell geschlossen werden, damit die Ressourcen wieder freigegeben werden.
Die Verwendung von einem Kontext je annavigierte Seite führte anschliessend dazu, dass der CoopScrapCategoryProductsSpider und später der CoopScrapAllProductsSpider ohne Memory-Exception durchliefen.

5.3.7.5 Notwendigkeit einer eigenen Retry-Implementierung

Es ist leider jederzeit möglich, dass ein Request für das Abholen einer Seite fehlschlägt. So kam es während dem Scrapen selten aber immer wieder einmal zu Timeout- als auch zu HTTP500-Fehlern. Dies ist bei der Nutzung von «Scrapy» alleine weniger ein Problem, da eine RetryMiddleware existiert, welche automatisch die fehlgeschlagenen Requests erneut versucht (Scrapy (2022g)).
Da bei unserem Web-Scraper die Requests allerdings über «Playwright» laufen, funktionierte diese RetryMiddleware nicht. Diese fehlenden Retries waren ein grosses Problem, da es dadurch möglich war, dass wenn einer der Requests beim Durchiterieren über die Sub-Pages einer Produktkategorie fehlschlug, darauf alle restlichen Produkte dieser Kategorie nicht mehr gescrapt wurden.
Um diesem Problem zu begegnen, musste eine eigene Retry-Funktionalität für fehlgeschlagene Requests aufgebaut werden. Dies konnte dann so umgesetzt werden, dass ein fehlgeschlagener Request jeweils dreimal neu probiert wird. Falls der Request dann immer noch nicht erfolgreich ist, wird für das weitere Vorgehen zwischen zwingend erforderlichen Requests und solchen die nicht unbedingt erforderlich sind unterschieden.
Während bei zwingend erforderlichen Requests nach drei Fehlversuchen der Scraping-Prozess komplett abgebrochen wird, wird bei nicht unbedingt erforderlichen Requests (Aufruf der Produktdetail-Seiten) nichts gemacht, die Produktdaten der entsprechenden Produktdetail-Seite damit einfach nicht gescrapt. Dieser Ansatz scheint am sinnvollsten um möglichst alle Produktdaten zu scrapen und dabei möglichst selten den Scraping-Prozess abbrechen zu müssen, welcher mit dem CoopScrapAllProductsSpider doch sehr lange dauert.

5.3.8 Scraper in Betrieb nehmen

Da es Probleme gab beim Integrieren des Web-Scrapers in das Docker-Setup und am Ende die Zeit dafür nicht mehr ausreichte, ist es zurzeit nur möglich den Web-Scraper lokal laufen zu lassen. Eine ausführliche Beschreibung, wie der Scraper auf einer lokalen Maschine in Betrieb genommen werden kann, ist im «readme.md» im Unterverzeichnis «scraper» auf dem Git-Repository zu finden.

5.3.9 Dauer des Scraping-Prozesses

Ein kompletter Run mit dem CoopScrapAllProductsSpider dauert auf dem für das Scraping verwendeten Windows-Computer zwischen 2 Stunden und 30 Minuten und 2 Stunden und 40 Minuten. Dabei werden etwas mehr als 10'000 Produkte von der Coop-Webseite gescrapt und in der Produktdatenbank aktualisiert.
Die Dauer ist natürlich abhängig von der Internetgeschwindigkeit und der Leistungsfähigkeit des Computers. Die Zeitangaben sind daher nur als Referenz zu verstehen.

5.4 Deployment

Im Verlaufe der Umsetzung hat sich « Pantry» stark weiterentwickelt, dabei wurde die Skalierbarkeit der Applikation zu einem wichtigen Thema. In diesem Kapitel wird beschrieben, welche Tools für die Verbesserung der Skalierbarkeit eingesetzt wurden. Dabei wird auch darauf eingegangen, wie die einzelnen Komponenten zusammenarbeiten.

5.4.1 Verteiltes System mit Docker

In Bezug auf das NFR2 (Paralleler Betrieb) wurde festgestellt, dass eine Spring Boot Applikation maximal 200 Anfragen pro Sekunde verarbeiten kann. Dies war für die Erfüllung des NFRs jedoch nicht genug, weshalb die Applikation zu einem verteilten System ausgebaut werden musste.
Das aufgebaute verteilte System besteht dabei aus fünf Docker-Containern. Es gibt einen Container für den Webserver, welcher das Frontend ausliefert. Zwei Container mit Spring Boot Instanzen, welche auf die Datenbank zugreifen, die sich wiederum in einem eigenen Container befindet. Als auch ein Container für den Loadbalancer.

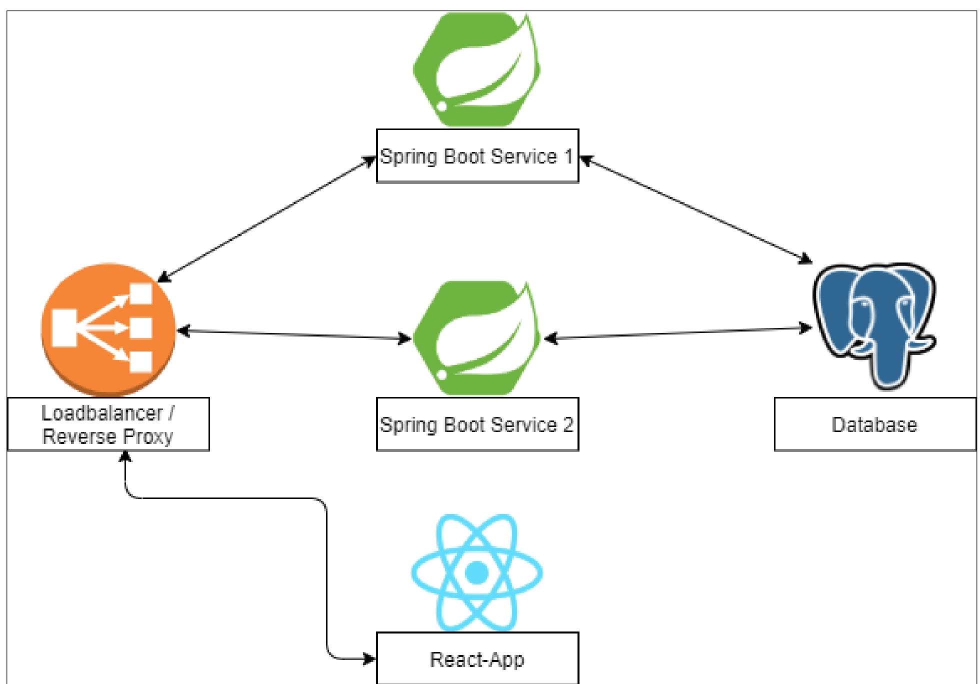


Abbildung 14: Docker-Container im verteilten System

5.4.1.1 Loadbalancer

Nginx wird gleichzeitig als Loadbalancer und Reverse Proxy eingesetzt. Alle Anfragen von Benutzern werden vom Loadbalancer an eine für die Anfrage geeignete Komponente weitergeleitet. Dabei sorgt der Loadbalancer auch dafür, dass Anfragen für das Backend gleichmässig auf die beiden Spring Boot Instanzen verteilt werden. Der Benutzer kommuniziert dabei nur mit dem Loadbalancer und merkt nichts davon, ob seine Anfrage von der ersten oder zweiten Backend-Instanz abgearbeitet wird.

Wenn ein Benutzer im Browser ganz normal die URL aufruft, dann wird eine Anfrage an den Loadbalancer geschickt. Das Frontend ist aber nach wie vor auch direkt ansprechbar und zwar über Port 80.

5.4.1.2 Backend

Beide Backend-Instanzen sind so aufgesetzt, dass sie vom Loadbalancer aus erreichbar sind. Die Backend-Instanzen greifen dabei beide auf die Datenbank zu. Dies ist kein Problem, da die Zugriffe auf die Datenbank nur lesend sind.

5.4.1.3 Frontend

Der Frontend Container besteht aus der Web-App selbst und einem Caddy-Webserver, welcher die Web-App ausliefert. Die Anfragen vom Frontend aus, welche für das Backend gedacht sind, werden dabei mit «/api» ergänzt. Mit dieser Ergänzung weiss der Loadbalancer, wie er mit diesen Anfragen umzugehen hat.

5.4.1.4 Datenbank

Die Datenbank kann in zwei verschiedenen Modi aufgesetzt werden. Zum einen gibt es den Testmodus, in welchem die Datenbank mit nur 30 Testprodukten befüllt und gestartet wird. Zum anderen gibt es den produktiven Modus bei welchem ein PostgreSQL-Backup in Form eines Dumps, welches sich im Repository befinden muss, in der Datenbank wiederhergestellt wird.

5.4.2 Einsatzmöglichkeiten

Die Applikation kann überall wo Docker läuft als verteiltes System aufgesetzt und gestartet werden. Somit ist bei zukünftigen Weiterentwicklungen von «Pantry» die Infrastruktur leicht erweiterbar.

6 Resultate und Fazit

Zu Beginn dieses Kapitels wird auf die Zielerreichung eingegangen, diese bezieht sich dabei auf die Erfüllung der in der Analyse definierten funktionalen und nicht-funktionalen Anforderungen. Danach folgen unsere Schlussfolgerungen bezüglich der Resultate und abschliessend werden diverse Möglichkeiten zur Weiterentwicklung vorgestellt.

6.1 Zielerreichung

Das Hauptziel der Studienarbeit ist es, die in der Analyse definierten Anforderungen zu erfüllen. Anbei wird sowohl auf den Erfüllungsgrad der funktionalen als auch der nicht-funktionalen Anforderungen eingegangen.

6.1.1 Funktionale Anforderungen

An dieser Stelle wird auf die einzelnen User Stories aus der Analyse eingegangen und ihr Erfüllungsgrad aufgezeigt.

6.1.1.1 User Story «Preisrechner»

Bei dieser User Story konnten alle erforderlichen Akzeptanzkriterien erfüllt werden. Eine Einschränkung gibt es beim Akzeptanzkriterium *«Man erhält das am besten übereinstimmende Produkt je Zutat vorgeschlagen»*. Das dafür notwendige «Pattern Matching» wurde nur auf simple Weise implementiert. So werden zurzeit über eine LIKE-Suchanfrage, mit dem gesuchten Zutatennamen als Argument, alle potentiellen Produkte in der Datenbank abgefragt. Dies ist noch keine optimale Lösung, es fehlte jedoch die Zeit um ein fortschrittlicheres «Pattern Matching» zu implementieren.
Das optionale Akzeptanzkriterium *«Das Rezept kann auf einfache Weise angepasst und neu berechnet werden»* konnte aus Zeitgründen ebenfalls nicht umgesetzt werden.

6.1.1.2 User Story «Web-Scraper»

Bei dieser User Story konnten alle erforderlichen Akzeptanzkriterien erfüllt werden. So konnte der Web-Scraper für Coop umgesetzt werden und die Datenbank der «Pantry» App ist mit korrekten Daten befüllt (mehr als 10'000 Coop Produkte).
Von den optionalen Akzeptanzkriterien konnte eines teilweise erfüllt werden. So kann das Web-Scraping durch Befolgen des im Readme des Web-Scrapers beschriebenen Vorgehens manuell angestossen werden, wodurch sich die Produktdaten von Coop in einer «Pantry»-Datenbank automatisch aktualisieren lassen. Dies funktioniert aktuell allerdings nur lokal und konnte nicht mehr ins Deployment integriert werden, wodurch die Datenbank direkt auf dem Server aktualisiert werden sollte. Als Folge davon konnte auch das optionale Akzeptanzkriterium *«Das Web-Scraping kann über einen REST-Endpoint angestossen werden»* nicht erfüllt werden.

Das letzte optionale Akzeptanzkriterium wäre die Realisierung des Web-Scrapers für Migros gewesen, auch dies war aus zeitlichen Gründen nicht möglich.

6.1.1.3 User Story «Nährwerte-Rechner»

Bei dieser User Story, welche erst im Verlaufe des Projekts hinzugekommen ist, konnten alle Akzeptanzkriterien erfüllt werden. Die Nährwerte werden zu den einzelnen Produkten, welche für das Rezept vorgeschlagen werden, ausgewiesen. Bei Produkten ohne Nährwertinformationen wird man darauf hingewiesen und schlussendlich werden einem auch die aufsummierten Nährwerte des Rezeptes ausgewiesen.

6.1.1.4 User Stories «Alternative Produkte» und «Rezept Sammlung»

Diese als Erweiterung markierten User Stories konnten im Zeitrahmen des Projekts nicht mehr angegangen werden.

6.1.2 Nicht-funktionale Anforderungen

An dieser Stelle wird auf die einzelnen nicht-funktionalen Anforderungen aus der Analyse eingegangen und ihr Erfüllungsgrad aufgezeigt.

6.1.2.1 NFR1: Gute Bedienbarkeit

Es wurden zwei Usability Tests, einer je Teammitglied, gegen Ende des Projekts durchgeführt. Das Akzeptanzkriterium zur Erfassung eines Rezepts mit 5 Zutaten innerhalb von 2 Minuten wurde dabei in beiden durchgeführten Usability Tests erfüllt.

6.1.2.2 NFR2: Paralleler Betrieb

Die Applikation kann, wie im Kapitel 5.4 beschrieben, als verteiltes System eingesetzt werden. Wenn dies der Fall ist, werden Anfragen vom Loadbalancer auf die beiden Backend-Instanzen gleichmässig verteilt. Durch diese Verteilung sind mehr als 200 parallele Anfragen möglich. Bei Bedarf nach noch mehr Performance wäre es zudem möglich auf einfache Weise weitere Backend-Instanzen hinzuzufügen.

Das Akzeptanzkriterium, dass die Anfragen von zwei Backend-Instanzen abgearbeitet werden ist erfüllt, was der Abbildung unten entnommen werden kann.

```
pantry-backend2       | /calculate called
pantry-loadbalancer   |             - - [22/Dec/2022:12:38:42 +0000] "POST /api/calculate HTTP/1.1" 200 2984
/537.36" "-"
pantry-backend1       | /calculate called
pantry-loadbalancer   |             - - [22/Dec/2022:12:38:56 +0000] "POST /api/calculate HTTP/1.1" 200 2984
/537.36" "-"
```

Abbildung 15: Loadbalancer verteilt Anfragen auf Backend-Instanzen

6.1.2.3 NFR3: Schnelle Anfragen

Während der Entwicklung wurde «Pantry» immer wieder auf dem Server manuell getestet und es sind nie langsame Anfragen aufgetreten. Die als Akzeptanzkriterium definierte zwei Sekunden Marke für die Anzeige der Einkaufsliste wurde nie überschritten, womit das Kriterium erfüllt ist.

6.1.2.4 NFR4: Wartbarkeit

Die zum Ziel gesetzte Unit Test Codeabdeckung von 80% für den Service im Backend wurde erreicht, wie der Abbildung unten entnommen werden kann. Dabei sind alle Tests grün, womit beide Akzeptanzkriterien erfüllt sind.

Class	Class, %	Method, %	Line, %
CalculatorService	100% (2/2)	100% (8/8)	91.4% (64/70)

Abbildung 16: Unit Test Codeabdeckung vom CalculatorService

6.2 Schlussfolgerungen

Der Umbau zu einem verteilten System mit Docker-Containern und dessen Aufsetzung haben neben der Entwicklung des Web-Scrapers mit Abstand am meisten Zeit in Anspruch genommen. Der grosse Zeitaufwand, welcher in diese beiden Dinge gesteckt wurde, führte leider dazu, dass diverse andere Features aus Zeitgründen am Ende nicht mehr realisiert werden konnten.

Ein weiterer Punkt, welcher das Projekt stark beeinflusst hat, waren die Advisor Meetings in welchen immer wieder viele neue spannende Ideen aufgekommen sind. Dies führte zu Änderungen bei den Anforderungen und deren Priorisierung, was schlussendlich auch einen Einfluss auf die Planung des Projekts hatte. So entstand in einem Advisor Meeting zum Beispiel die neue User Story «Nährwerte-Rechner», welche mit hoher Priorität umgesetzt werden sollte. Der Aufwand für die Umsetzung dieser User Story musste anschliessend an anderer Stelle kompensiert werden.

Da in den Advisor Meetings immer wieder viele weitere interessante Ideen aufkamen für welche die Zeit der Studienarbeit niemals ausreichen würde, fasste man schlussendlich den Entschluss, dass es ein eigenes Kapitel für die Weiterentwicklungsmöglichkeiten geben soll. In diesem Kapitel sollen dabei sowohl Dinge, welche aus Zeitgründen nicht mehr umgesetzt werden konnten, als auch zusätzlich aufgekommene Ideen, aufgelistet werden.

6.3 Weiterentwicklung

Es gibt diverse Weiterentwicklungsmöglichkeiten, welche in Betracht gezogen werden können. Im Folgenden werden diese beschrieben.

6.3.1.1 Verbessertes «Pattern Matching»

Das «Pattern Matching» ist in «Pantry» noch nicht zufriedenstellend gelöst (Details zur Umsetzung befinden sich in Kapitel 6.1.1.1). Um «Pantry» aufzuwerten, ist es zwingend notwendig das «Pattern Matching» zu verbessern. Eine potenzielle Möglichkeit wäre es in PostgreSQL die Erweiterung «pg_trgm» einzusetzen, welche es ermöglicht Wörter bzw. Strings nach Ähnlichkeit zu sortieren.

6.3.1.2 User Story «Alternative Produkte»

Diese im Verlauf des Projekts zu einer Erweiterung zurückgestufte User Story würde die «Pantry» Web-App noch nützlicher machen. So könnte zwischen mehreren Produktvorschlägen für eine Zutat, die für einen am besten Passende ausgesucht und die Einkaufsliste damit an die eigenen Bedürfnisse angepasst werden.

6.3.1.3 User Story «Rezept Sammlung»

Diese User Story, welche ebenfalls als Erweiterung eingeplant war, würde die Speicherung von Rezepten ermöglichen, so dass diese zu einem späteren Zeitpunkt erneut berechnet werden können. Damit dies überhaupt realisiert werden kann, wäre unter anderen eine Benutzerverwaltung inklusive Login notwendig.

6.3.1.4 Scrapen der Produkte von weiteren Verkäufern

Es wäre wünschenswert den Web-Scraper auf weitere Verkäufer (Migros, Lidl, Aldi) auszuweiten, so dass «Pantry» die Produkte von verschiedenen Verkäufern für seine Produktvorschläge berücksichtigen kann. Die Auswahlmöglichkeit zwischen allen gängigen Verkäufern der Schweiz würde den Nutzen von «Pantry» wesentlich steigern.

6.3.1.5 Möglichkeit mehrere Rezepte auf einmal an einen Backend Endpoint zu übergeben

Das Ziel wäre es den Benutzern die Möglichkeit zu geben mehrere Rezepte auf einmal an einen Backend-Endpoint zu senden. Eine solche Erweiterung könnte aufgrund der Codebasis mit überschaubarem Aufwand implementiert werden.

6.3.1.6 Kubernetes für automatische Skalierung

Nach dem Umbau zu einem verteilten System ist in einem Advisor Meeting die automatische Skalierung der Applikation zum Thema geworden. Da keiner der Teammitglieder Erfahrung mit diesem Bereich hat, wurden in einem ersten Schritt die Möglichkeiten evaluiert. Das Resultat der Evaluation war dann die Feststellung, dass die Applikation mit einem Umbau zu Kubernetes automatisch skalierbar gemacht werden könnte.

7 Projektmanagement

7.1 Rollen und Verantwortlichkeiten

Teammitglied	Rolle	Verantwortlichkeiten
Daniel Frick	Software-Entwickler	- Web-Scraping - Dokumentation - Projektmanagement - Issue- und Timetracking
Zvonimir Serkinic	Software-Entwickler	- Front- und Backend - Datenbank - Infrastruktur (GitLab, Server) - Kommunikation mit Betreuerin

Tabelle 4: Rollen und Verantwortlichkeiten

Wichtige Zusatzinformation
Die Verantwortlichkeiten beschreiben nicht wer die damit verbundenen Aufgaben ausführt. Es geht lediglich darum, welche Person in welchem Aufgabengebiet den Lead übernimmt.

7.2 Prozesse

Zur Planung und Durchführung des Projekts wird für die Grobplanung auf RUP und für die Feinplanung auf Scrum gesetzt.
Für die Grobplanung nach RUP wird ein grober Projektplan erstellt. Dieser Projektplan wird während der Projektdauer fortlaufend aktualisiert und falls notwendig angepasst. Mehr Informationen zum Projektplan finden sich in Kapitel 7.5.
Für die Feinplanung nach Scrum wird ein Agiles-Board im YouTrack verwendet. Das Board bietet einerseits einen guten Überblick über noch ausstehende Arbeiten eines Sprints und andererseits erlaubt es eine komfortable Verwaltung der User Stories und Tasks. Das Board kann hier eingesehen werden.
Als Sprintdauer in Scrum wird eine Woche verwendet. Dies erlaubt eine maximale Flexibilität um auf unerwartete Probleme oder Änderungswünsche zu reagieren. Das Sprint Planning für den nächsten Sprint wird jeweils direkt nach dem Advisor Meeting gemacht.

7.3 Tooling

Tool	Verwendung
Teams, WhatsApp	Teamkommunikation
YouTrack	Issue-Management und Zeiterfassung
MS Office Produkte	Dokumentation
draw.io & PlantUML	Diagramme erstellen
GitLab	Code-Verwaltung, CI
JetBrains IntelliJ IDEA	Entwicklung Backend (Spring Boot)
JetBrains Webstorm	Entwicklung Frontend (React)
JetBrains PyCharm	Entwicklung Web-Scraper
PostgreSQL	Datenbank
pgAdmin	GUI Tool für PostgreSQL
Docker	Auslieferung Software

Tabelle 5: Tooling

7.4 Meilensteine

Meilenstein	Termin
M1: Aufgabenstellung	04.10.2022
M2: Anforderungen & Analyse	11.10.2022
M3: Architektur & Design	11.10.2022
M4: Prototyp	08.11.2022
M5: Endprodukt	13.12.2022
M6: Abgabe	23.12.2022

Tabelle 6: Meilensteine

7.5 Projektplan

Der grobe Projektplan wird mit Excel erstellt. Im Anhang befinden sich sowohl der initial erstellte Projektplan, als auch der fortlaufend aktualisierte Projektplan.

7.6 Risikomanagement

Die Risikoanalyse soll eine Übersicht über mögliche Risiken des Projekts geben. Die Liste ist nicht abschliessend und kann im Verlaufe des Projekts bei Notwendigkeit angepasst, als auch um weitere Risiken ergänzt werden.

7.6.1 Identifizierte Risiken

Nr.	Beschreibung	Massnahmen
R1	Die Entwicklung des Web-Scrapers dauert länger als geplant.	- Vorzeitige Vertiefung im Thema - Fokus auf einen der Verkäufer (Coop) - Unterstützung durch Betreuerin
R2	Die Umsetzung des komplexen «Pattern Matching» bei der Produktsuche nimmt mehr Zeit in Anspruch als geplant.	- Vorzeitige Auseinandersetzung mit Regex - Abklärung der Möglichkeiten in PostgreSQL - Unterstützung durch Betreuerin
R3	Das Aufsetzen von CI und das Deployment auf dem Server dauern länger als geplant.	- Einrichtung von CI frühzeitig angehen - Server früh bestellen und Deployment früh genug angehen
R4	Die Entwicklung des responsiven Designs im Frontend dauert länger als geplant.	- Wissen aus absolvierten Webmodulen auffrischen - Bereits bekannte Libraries verwenden
R5	Die Web-App lässt sich mit der geplanten Architektur nicht umsetzen.	- Bekannte Technologien verwenden - Erstellung eines End-to-End Prototypen
R6	Daten (Dokumentation, Code) könnten verloren gehen.	- Backups des OneDrive-Ordners - Gesamter Code in Repository auf GitLab verwalten

Tabelle 7: Identifizierte Risiken

7.6.2 Risikoevaluation

Die identifizierten Risiken werden alle zwei Wochen bezüglich Wahrscheinlichkeit und Schweregrad neu beurteilt. Die grafische Risikomatrix basiert auf Thesos (2019).

7.6.2.1 Risiko Bewertung vom 30.9.2022

- **R1:** Das Web-Scraping ist aufgrund der mangelnden Erfahrung des Teams auf diesem Gebiet sehr wahrscheinlich die grösste Herausforderung dieses Projekts. Dementsprechend hat dieses Risiko die grösste Wahrscheinlichkeit einzutreten.
- **R2:** Auch beim «Pattern Matching» fehlt es an Erfahrung, weshalb die Wahrscheinlichkeit eines Eintretens höher liegt als bei den weiteren Risiken.
- **R3:** Im Bereich CI/CD fehlt es vor allem im Zusammenhang mit GitLab an Erfahrung, das Gleiche gilt für weitere DevOps Tätigkeiten, so dass die Wahrscheinlichkeit eines Eintretens als relativ hoch eingeschätzt werden muss.
- **R4:** Da beide Entwickler bereits Erfahrungen mit responsiven Designs im Webbereich haben, wird die Eintrittswahrscheinlichkeit dieses Risiko als eher gering eingeschätzt.
- **R5:** Für die Architektur werden bewährte Technologien eingesetzt, mit welchen einer der Teammitglieder (Zvonimir) täglich arbeitet. Die Wahrscheinlichkeit, dass die Architektur später noch angepasst werden muss, ist daher als gering einzuschätzen
- **R6:** Der Verlust von Daten ist immer ein Thema, dies kann jedoch durch die geplanten Massnahmen abgefedert werden, so dass ab der kommenden Risikobewertung die Wahrscheinlichkeit eines Datenverlusts auf ein Minimum reduziert werden kann.

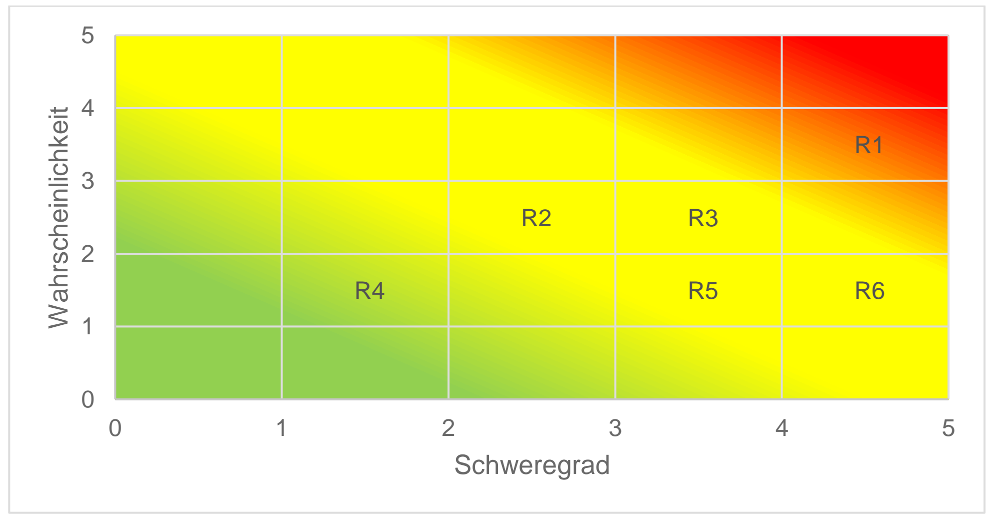


Abbildung 17: Risikomatrix vom 30.09.2022

7.6.2.2 Risiko Bewertung vom 11.10.2022

Anmerkungen zu den Verschiebungen:

- **R6:** Git-Repository und Backup für OneDrive-Ordner eingerichtet

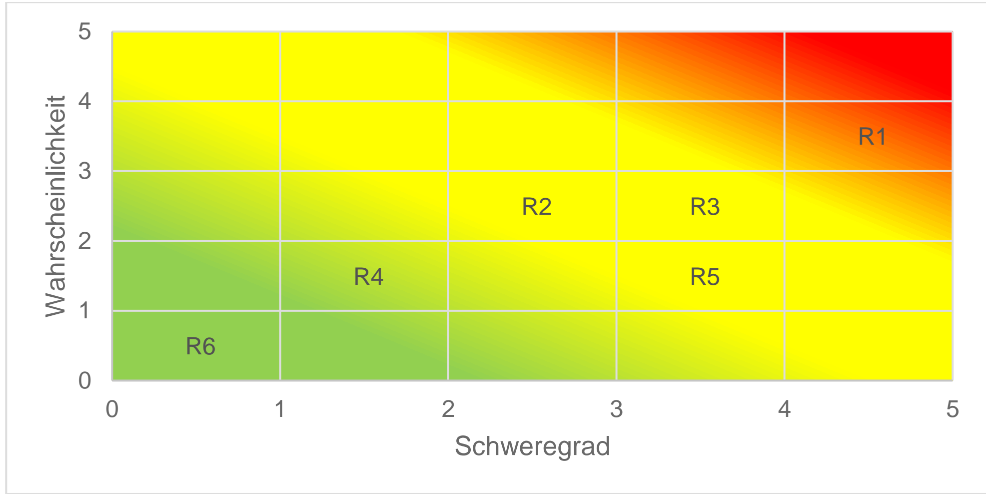


Abbildung 18: Risikomatrix vom 11.10.2022

7.6.2.3 Risiko Bewertung vom 25.10.2022

Anmerkungen zu Verschiebungen:

- **R1:** Der erste Ansatz mit «BeautifulSoup4» & «Selenium» gibt eine gewisse Sicherheit, dass das Minimum Requirement der User Story «Web-Scraping», die Datenbank mit sinnvollen Daten abzufüllen, erfüllt werden kann.
- **R3:** Die CI-Pipeline wurde mittlerweile eingerichtet, damit sinkt die Wahrscheinlichkeit eines Eintretens. Das Deployment ist aber weiterhin offen, was mit einem Restrisiko behaftet ist.
- **R4:** Mit der Entscheidung für Ant-Design als Design Framework für das Frontend ist die Überzeugung da, dass sich das Design sinnvoll umsetzen lässt.

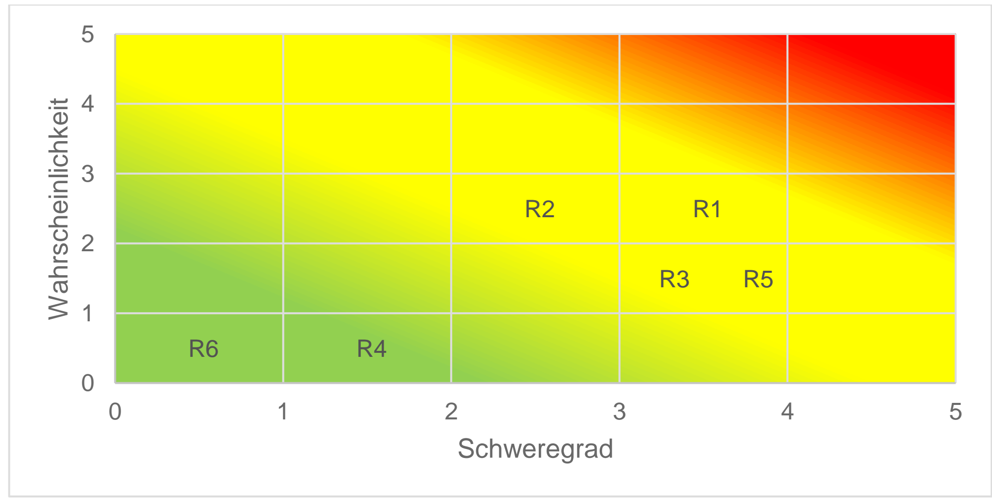


Abbildung 19: Risikomatrix vom 25.10.2022

7.6.2.4 Risiko Bewertung vom 08.11.2022

Keine Verschiebungen

7.6.2.5 Risiko Bewertung vom 22.11.2022

R1: Das Web-Scraping der gewünschten Produktdaten von der Coop-Webseite funktioniert. Es fehlt nur noch die Übertragung der Produktdaten in die Datenbank. Das Risiko, dass das Web-Scraping der Coop-Webseite nicht in der zur Verfügung stehend Zeit gelingt, konnte praktisch eliminiert werden.
R5: Die geplante Architektur funktioniert wie erwartet. Das Risiko, dass die Architektur noch einmal angepasst werden muss, besteht nicht mehr.

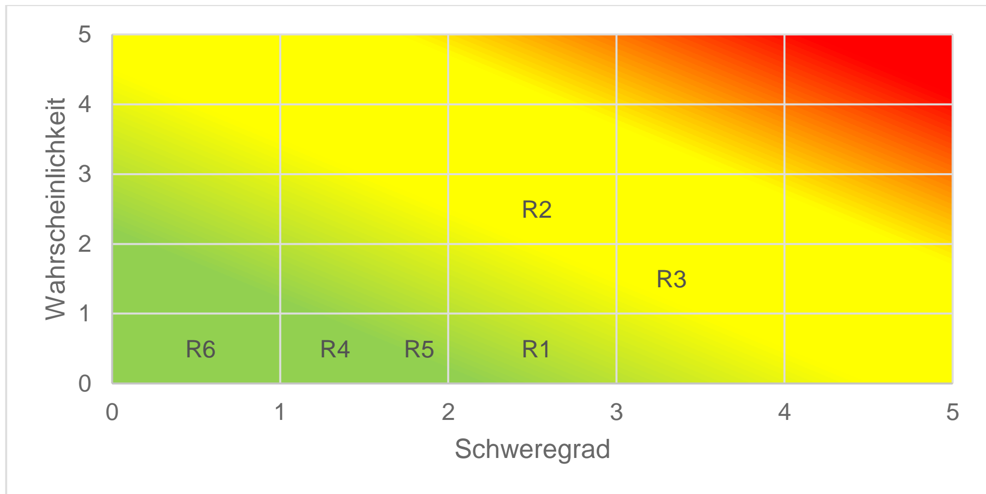


Abbildung 20: Risikomatrix vom 22.11.2022

7.6.2.6 Risiko Bewertung vom 06.12.2022

Keine Verschiebungen

7.6.2.7 Risiko Bewertung vom 20.12.2022

R3: Das Deployment der Web-App «Pantry» auf dem Server konnte umgesetzt werden. Dadurch sinkt der Schweregrad des Risikos. Allerdings konnte der Web-Scraper aus zeitlichen Gründen nicht mehr in das Deployment integriert werden.

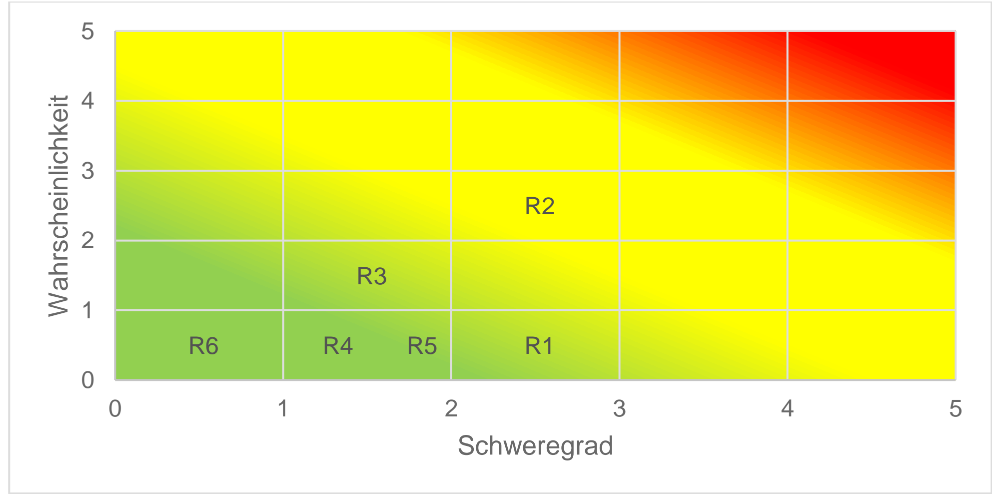


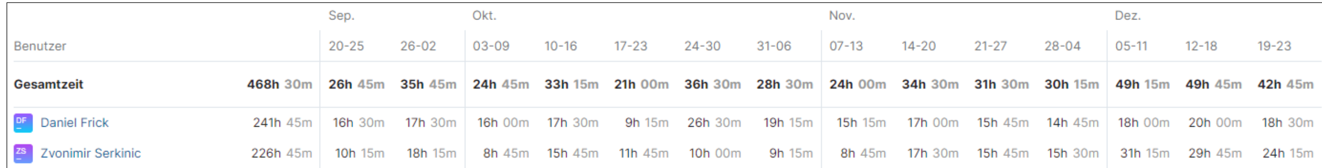
Abbildung 21: Risikomatrix vom 20.12.2022

7.6.3 Abschliessende Risikobeurteilung

Die Risiken R1, R3, R4, R5 und R6 konnten im Verlauf des Projekts erfolgreich eliminiert werden. Anders sieht es bei den Risiken R2 und R3 aus.
Bei R2 ist das Risiko eingetreten, weil früh klar wurde, dass die Umsetzung eines komplexen «Pattern Matching» viel Zeit in Anspruch nehmen wird. Der Fokus lag darum vorerst auf anderen Dingen um eine lauffähige Web-App abliefern zu können. Gegen Ende des Projekts musste dann festgestellt werden, dass die Zeit für die Implementierung eines komplexeren «Pattern Matching» nicht mehr ausreicht.
Bei R3 konnte einerseits das Risiko zwar bezüglich Schweregrad minimiert werden, da es gelungen ist, die Web-App auf dem Server zu deployen und die Datenbank auf dem Server über ein Datenbank-Backup mit den gescrapten Coop-Produkten auszustatten. Andererseits ist es in der zur Verfügung stehenden Zeit nicht gelungen den Web-Scraper auch noch in das Deployment zu integrieren, weshalb das Risiko nicht als vollständig eliminiert betrachtet werden kann.

7.7 Zeitrapportierung

Die Zeitrapportierung erfolgt im YouTrack auf den einzelnen Tasks. Es wird dabei auf eine Genauigkeit von 15 Minuten rapportiert. Die Abbildung unten zeigt die geleistete Arbeitszeit der beiden Teammitglieder pro Woche.



		Sep.		Okt.					Nov.				Dez.		
Benutzer		20-25	26-02	03-09	10-16	17-23	24-30	31-06	07-13	14-20	21-27	28-04	05-11	12-18	19-23
Gesamtzeit	**468h** 30m	**26h** 45m	**35h** 45m	**24h** 45m	**33h** 15m	**21h** 00m	**36h** 30m	**28h** 30m	**24h** 00m	**34h** 30m	**31h** 30m	**30h** 15m	**49h** 15m	**49h** 45m	**42h** 45m
DF Daniel Frick	241h 45m	16h 30m	17h 30m	16h 00m	17h 30m	9h 15m	26h 30m	19h 15m	15h 15m	17h 00m	15h 45m	14h 45m	18h 00m	20h 00m	18h 30m
ZS Zvonimir Serkinic	226h 45m	10h 15m	18h 15m	8h 45m	15h 45m	11h 45m	10h 00m	9h 15m	8h 45m	17h 30m	15h 45m	15h 30m	31h 15m	29h 45m	24h 15m

Abbildung 22: Geleistete Arbeitszeit pro Woche

Im YouTrack stehen weitere Berichte zur Verfügung, welche auch mit einem Gastaccount zugänglich sind.

Folgende Berichte stehen zur Verfügung:
- Zeit pro Woche Gesamt (Eine Gesamtübersicht über die Arbeitszeit jedes einzelnen Teammitglieds (vgl. Abbildung von oben))
- Zeit pro Woche nach Arbeitstyp (Detaillierte Übersicht über die Arbeitszeit gruppiert nach Arbeitstyp)
- Sprint X (Auswertung pro Sprint, hier sieht man an welchen Tickets jedes Teammitglied während des Sprints gearbeitet hat und welcher Zeitaufwand dafür aufgewendet wurde)

Die Berichte sind hier zugänglich.

Glossar

Caddy ist ein Webserver.

Docker ist eine Software, um Anwendungen mittels Containervirtualisierung bereitzustellen.

FURPS ist ein Akronym, welches folgende klassischen Softwarequalitätsmerkmale beschreibt: Functionality, Usability, Reliability, Performance und Supportability.

Git ist ein verteiltes Versionskontrollsystem zur Verwaltung von Quellcode.

GitLab ist eine webbasierte DevOps-Plattform, die neben der Verwaltung von Git-Repositories weitere Funktionalitäten wie CI/CD Pipelines anbietet.

Loadbalancer ist ein Baustein, welcher den hereinkommenden Datenverkehr automatisch auf mehrere Ziele verteilt.

Nginx ist ein Webserver, der auch als Reverse Proxy und Loadbalancer verwendet werden kann.

React ist ein Javascript Framework für das Erstellen von User Interfaces.

REST API beschreibt eine Schnittstelle, welche dem REST Architekturstil entspricht.

pgAdmin ist ein frei zugängliches, webbasiertes GUI Tool zur Administration von PostgreSQL Datenbanken.

PostgresSQL ist ein frei zugängliches objekt-relationales Datenbankmanagementsystem.

Spring Framework ist ein sehr umfangreiches Application Framework für die Java Plattform.

Web-Scraping bezeichnet den Vorgang öffentlich zugängliche HTML-Seiten abzurufen und daraus bestimmte Daten zu extrahieren.

Windows Subsystem for Linux ist ein Feature von Windows, welches es Entwicklern erlaubt eine Linux-Umgebung zu verwenden, ohne eine zusätzliche virtuelle Maschine zu benötigen.

XPath ist eine Abfragesprache, um Teile eines XML-Dokumentes abzufragen oder zu transformieren.

YouTrack ist ein Issue Tracking System, welches unter anderem die Arbeit mit agilen Boards ermöglicht.

Abkürzungsverzeichnis

API	Application programming interface
CI	Continous Integration
CD	Continous Delivery
CSS	Cascading Style Sheets
GUI	Graphical User Interface
HTML	HyperText Markup Language
NFR	Non-functional requirements
REST	Representational state transfer
RUP	Rational Unified Process
URL	Uniform Resource Locator
WSL	Windows Subsystem for Linux

Literaturverzeichnis

GitHub, 2022, scrapy-playwright: Playwright integration for Scrapy von der Webseite, https://github.com/scrapy-plugins/scrapy-playwright, 09.12.2022.

Rehkopf, Max, 2022, User Storys mit Beispielen und Vorlage von der Webseite, https://www.atlassian.com/de/agile/project-management/user-stories, 28.09.2022.

ScrapeOps, 2022, Scrapy Javascript Rendering: The 4 Best Scrapy Libraries to Scrape JS Heavy Websites von der Webseite, https://scrapeops.io/python-scrapy-playbook/scrapy-javascript-rendering-guide/, 08.12.2022.

Scrapy, 2022a, Spiders von der Webseite, https://docs.scrapy.org/en/latest/topics/spiders.html, 03.12.2022.

Scrapy, 2022b, Items von der Webseite, https://docs.scrapy.org/en/latest/topics/items.html, 03.12.2022.

Scrapy, 2022c, Item Pipeline von der Webseite, https://docs.scrapy.org/en/latest/topics/item-pipeline.html, 03.12.2022.

Scrapy, 2002d, Settings von der Webseite, https://docs.scrapy.org/en/latest/topics/settings.html, 10.12.2022.

Scrapy, 2022e, Selectors von der Webseite, https://docs.scrapy.org/en/latest/topics/selectors.html, 04.12.2022.

Scrapy, 2002f, Requests and Responses von der Webseite, https://docs.scrapy.org/en/latest/topics/request-response.html, 09.12.2022.

Scrapy, 2022g, Downloader Middleware von der Webseite, https://docs.scrapy.org/en/latest/topics/downloader-middleware.html, 10.12.2022.

Thehos, Andreas, 2019, Risikomatrix in Excel von der Webseite, https://thehosblog.com/2019/02/22/risikomatrix-in-excel/, 25.09.2022.