



Ingenium Education

Kolleg für Informatik

DIPLOMARBEIT

**Web-App „coOps lite“
zum Organisieren von Abläufen in FoodCoops**

Verfasst im Schuljahr 2025/26 von

Clemens Kolbitsch

Leonard Ostler

Luis Vázquez

Betreuer:

DI Andreas Schwarzl, BSc

Projektpartner:

IG FoodCoops

Graz, 14. September 2026

Eidesstattliche Erklärung

Ich erkläre ehrenwörtlich, dass ich den vorliegenden Diplomarbeitsteil / die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____

Clemens Kolbitsch

Leonard D. Ostler

Luis A. Vázquez de la Serna

Kurzfassung

FoodCoops werden ehrenamtlich organisiert. Wiederkehrende Aufgaben sollen deshalb möglichst fair unter den Mitgliedern verteilt werden. In der Praxis übernehmen jedoch häufig dieselben wenigen Personen einen großen Teil der Arbeit. Außerdem werden Aufgaben, Abläufe und Zuständigkeiten oft mit unterschiedlichen Werkzeugen verwaltet. Ziel dieser Diplomarbeit ist es, diese Organisation einfacher und übersichtlicher zu gestalten.

Für die IG FoodCoops wurde dafür die mobilfreundliche Webanwendung coOps lite entwickelt. Das Konzept führt Aufgaben und mehrstufige Abläufe in einem gemeinsamen System zusammen. Bei der Planung wurden Rückmeldungen von Mitgliedern verschiedener FoodCoops berücksichtigt.

Begleitend zur Entwicklung werden drei Forschungsthemen behandelt: Der Einfluss der Projektleitung auf den Projekterfolg, die visuelle und interaktive Gestaltung von Aufgabenabläufen sowie die Einbindung automatisierter Schritte.

Im umgesetzten Prototyp können sich Nutzer:innen mit Testkonten anmelden, zwischen Organisationsbereichen wechseln und entsprechend ihren Berechtigungen deren Stammdaten und Typzuordnungen bearbeiten. Fähigkeiten, Einschränkungen und deren Zuschreibungen können eingesehen werden. Anhand von statischen Beispieldaten lassen sich außerdem Aufgaben und Abläufe anzeigen und filtern. Der Bearbeitungsstatus einzelner Arbeitsschritte lässt sich lokal im Browser verändern; eine Speicherung im Backend ist noch nicht umgesetzt.

coOps lite bildet damit eine technische Grundlage für die gemeinsame Organisation von Aufgaben in FoodCoops und kann durch weitere Entwicklung vervollständigt werden.

Abstract

Food coops are organized on a voluntary basis. Recurring tasks should therefore be shared as fairly as possible among members. In practice, however, the same few people often take on a large part of the work. Tasks, workflows and responsibilities are also often managed using several different tools. The aim of this diploma thesis is to make this organization simpler and easier to keep track of.

For this purpose, the mobile-friendly web application coOps lite was developed for IG FoodCoops. The concept brings tasks and multi-step workflows together in one system. Feedback from members of different food coops was included in the planning process.

The development is accompanied by three research topics: The influence of project management on project success, the visual and interactive design of workflows, and the integration of automated steps.

In the realized prototype, users can log in with test accounts, switch between organizational spaces and, depending on their permissions, edit space data and type assignments. Available skills, impairments, and their attributions can be viewed. Tasks and workflows can also be displayed and filtered using static sample data. In addition, the status of individual workflow steps can be changed locally in a browser; saving these changes in the backend has not yet been implemented.

coOps lite provides a basis for the shared organization of tasks in food coops and can be completed with further development.

Inhalt

1.	Einleitung	1
2.	Projekt.....	2
2.1.	Auftraggeber und Projektbeginn	2
2.2.	Stakeholder-Übersicht.....	3
2.2.1.	IG FoodCoops	3
2.2.2.	FoodCoops in Österreich	3
2.2.3.	coOps	4
2.2.4.	Ingenium Education	5
2.2.5.	Diplomarbeitsteam	5
2.3.	Forschungsthema: Der Einfluss der Projektleitung auf den Erfolg von Softwareprojekten	7
2.3.1.	Rolle der Projektleitung und Kriterien für Projekterfolg	8
2.3.2.	Kommunikation	9
2.3.3.	Koordination	11
2.3.4.	Risikomanagement	12
2.3.5.	Fazit zum Forschungsthema	14
2.3.6.	Reflexion der Projektleitung bei coOps lite	14
3.	Beobachtung und Analyse der Anforderungen.....	17
3.1.	Über FoodCoops	17
3.1.1.	Beschreibung	17
3.1.2.	Persönliche Tätigkeit und Erfahrung	18
3.1.3.	Warenbezug in einer FoodCoop	18
3.1.4.	Bislang verwendete Software	18
3.2.	Koordination wiederkehrender Aufgaben	19
3.3.	Automatisierung von EDV-Prozessen	20
3.4.	Vorstudie „FoodCoop 4.0“	21
3.5.	Erfahrungen aus der IG FoodCoops	22
3.6.	coOps: Umfassendes Softwarekonzept für Gemeinschaften	22
3.7.	coOps lite: Funktionaler Prototyp eines Organisationstools für FoodCoops	24
3.7.1.	Zusammenfassung der Themenstellung	24
3.7.2.	Zielsetzung	25
4.	Umsetzungskonzept.....	26
4.1.	User Stories und Einordnung der User-Szenarien.....	26
4.1.1.	Verwendung von User Stories	26
4.1.2.	Perspektive „Mitmachen“	27
4.1.3.	Perspektive „Orga“	27
4.1.4.	Zusammenhang zwischen „Mitmachen“ und „Orga“	28
4.2.	Klickbarer Prototyp	29
4.2.1.	Designkonzept.....	29
4.2.2.	Navigation	30

4.2.3.	Startseite.....	31
4.2.4.	Wechsel des Bereichs	32
4.2.5.	Orga – Können	33
4.2.6.	Mustervorgänge	34
4.2.7.	Aufgabe im Detail.....	37
4.2.8.	Bereichs-Stammdaten verwalten.....	38
4.3.	Einordnung des umgesetzten Umfangs	39
4.4.	Auswahl grundlegender Technologien für die Umsetzung.....	39
5.	Umsetzung des Backend	40
5.1.	Architektur	40
5.2.	Authentifizierung	41
5.2.1.	Anmeldung im Prototyp	41
5.2.2.	Aufbau und Ausstellung des Tokens	42
5.2.3.	Prüfung geschützter Anfragen	43
5.2.4.	Überprüfung mit Postman	43
5.2.5.	Benutzerinformationen und Berechtigungen.....	44
5.2.6.	Erreichter Stand und Grenzen	45
5.3.	Logisches Datenmodell	46
5.3.1.	Soziale Strukturen.....	47
5.3.2.	Authentifizierung und Autorisierung	53
5.3.3.	Fähigkeiten und Einschränkungen.....	57
5.3.4.	Abläufe	59
5.3.5.	Benachrichtigungen	67
5.4.	API-Design	68
5.4.1.	Authentifizierung.....	68
5.4.2.	Grundlegende Daten eines Bereichs	68
5.4.3.	Orga – Verwaltung	69
5.4.4.	Orga – Können	69
5.4.5.	Orga – Struktur.....	70
5.5.	Implementierung des Datenmodells.....	71
5.5.1.	Primärschlüssel.....	71
5.5.2.	Constraints	71
5.5.3.	Datum und Uhrzeit	71
5.5.4.	Entity-Views	72
5.6.	Abfragen und Datenbankentwicklung	74
5.6.1.	Technologieauswahl für Datenbankkommunikation.....	74
5.6.2.	Hierarchie von DDL-Skripten	75
5.6.3.	Abfragesprache und Kennzeichnung von Volatilität	77
5.6.4.	Implementierung der Endpoints.....	79
5.7.	Backend-seitig umgesetzte User Stories	81
5.8.	Testing.....	82

5.8.1.	Zielsetzung und Testumfang	82
5.8.2.	Aufbau der automatisierten Tests	82
5.8.3.	Ausgewählte Testfälle	83
5.8.4.	Grenzen der Überprüfung	85
6.	Umsetzung des Frontend	86
6.1.	Forschungsthema	86
6.2.	Zielsetzung.....	86
6.3.	Auswahl von Technologien für das Frontend	86
6.3.1.	TypeScript	87
6.3.2.	React UI Framework.....	87
6.3.3.	Tailwind CSS.....	88
6.3.4.	Shadcn/ui.....	89
6.3.5.	Drag and Drop - dnd-kit	89
6.3.6.	Formulare und Validierung	89
6.3.7.	Progressive Web App (PWA).....	90
6.3.8.	Toast-Bibliothek.....	91
6.4.	Aufbau des Frontend	92
6.4.1.	Projektstruktur.....	92
6.4.2.	Besonderheiten (Komponenten, Snippets)	98
6.5.	Umgesetzte Frontend-Funktionen und User Stories.....	111
6.5.1.	Verwendete Endpoints anhand ausgewählter Stories	111
6.5.2.	Implementierte bzw. im Frontend verwendete Endpoints	111
6.5.3.	Prototypische User Stories mit Mock-Daten	113
6.5.4.	Authentifizierung.....	115
6.5.5.	Berechtigungsabhängige Benutzeroberfläche.....	121
6.5.6.	Navigation und Menüstruktur.....	122
6.6.	Umsetzung von Usability-Prinzipien.....	126
6.6.1.	Transparente Rückmeldung zum Systemzustand.....	126
6.6.2.	Kontrolle über eigene Aktionen	127
6.6.3.	Konsistente Bedienung.....	127
6.6.4.	Fehler frühzeitig verhindern	128
6.6.5.	Sichtbare Auswahlmöglichkeiten statt Gedächtnisbelastung	128
6.6.6.	Unterschiedliche Wege zur effizienten Nutzung	128
6.6.7.	Übersichtliches und minimalistisches Design	129
6.6.8.	Fehler verständlich erkennen und beheben	129
6.7.	Drag-and-Drop-Interaktion innerhalb eines Ablaufs.....	130
6.8.	Beantwortung der Forschungsfrage	134
6.8.1.	Ziel und Abgrenzung.....	134
6.8.2.	React als Grundlage interaktiver Benutzeroberflächen	134
6.8.3.	Wiederverwendbare UI-Komponenten	135
6.8.4.	Zustandsverwaltung und dynamische Darstellung	136

6.8.5.	Drag-and-Drop als Interaktionskonzept	137
6.8.6.	Flexible und kollaborative Aufgaben-Operationen	140
6.8.7.	Einsatz in FoodCoop-Umgebungen	141
6.8.8.	Fazit	142
7.	Forschungsthema: Einbindung automatisierter Schritte	143
7.1.	Anwendungsfälle für automatische Schritte	143
7.2.	Eingabe benötigter Daten	144
7.3.	Ausführung von benutzerdefiniertem Code	145
8.	Fazit und Ausblick	148
9.	Literaturverzeichnis.....	149
10.	Tabellen- und Abbildungsverzeichnis	153
11.	Abkürzungs- und Symbolverzeichnis	157
12.	Anhang	159
12.1.	User Stories	159
12.2.	Tabellarischer Vergleich von Vorgangstypen	163

1. Einleitung

Eine FoodCoop lebt davon, dass sich ihre Mitglieder Verantwortung teilen. Zur Organisation gehören viele Abläufe: Lieferungen müssen kontrolliert, Rechnungen beglichen und neue Mitglieder eingelernt werden. Dabei hängt die Erledigung einzelner Schritte oft davon ab, dass andere bereits abgeschlossen sind. Fehlt der Überblick über Zuständigkeiten und Fortschritte, entsteht zusätzlicher Abstimmungsaufwand, der häufig bei wenigen engagierten Personen liegt.

Bestehende Software unterstützt FoodCoops vor allem bei der Bestellabwicklung. Für die Organisation anderer Tätigkeiten werden zusätzlich Tabellen, Nachrichten und persönliche Absprachen verwendet. Wenn unklar bleibt, welche Schritte zusammenhängen, wer sie übernimmt und was noch offen ist, können Missverständnisse entstehen und Aufgaben unerledigt bleiben.

Hier setzt coOps lite an. Im Rahmen dieser Diplomarbeit wurde für die IG FoodCoops eine mobilfreundliche Webanwendung konzipiert und teilweise umgesetzt. Sie soll Aufgaben und automatische Schritte als mehrstufige Abläufe abbilden und eine Ergänzung zu bestehenden Softwarelösungen darstellen. Die Anforderungen beruhen auf praktischen Erfahrungen von FoodCoops und auf Rückmeldungen einzelner Mitglieder. Besonders wichtig waren dabei die Erfahrungen von Leonard Ostler, der zum Projektteam gehört und seit vielen Jahren selbst in FoodCoops aktiv ist.

Die Arbeit beschreibt den Weg von den Anforderungen über den Entwurf der Benutzeroberfläche bis zur technischen Umsetzung von Backend und Frontend. Anschließend wird erläutert, wie ausgewählte Funktionen überprüft wurden. Dabei wird zwischen dem geplanten Funktionsumfang und dem tatsächlich erreichten Entwicklungsstand unterschieden.

2. Projekt

2.1. Auftraggeber und Projektbeginn

Der Kontakt zur IG FoodCoops entstand durch Leonard Ostler, der Mitglied des Projektteams und selbst bei den FoodCoops Österreich aktiv ist. Aufgrund seiner bestehenden Verbindung zur Organisation stellte er das geplante Diplomarbeitsprojekt coOps lite intern vor. Nach dieser ersten Vorstellung wurde die IG FoodCoops als externe Auftraggeberin angefragt. Janet Wissuwa übernahm als Vorstandsmitglied und Vertreterin der IG FoodCoops die Rolle der Ansprechpartnerin.

Am 08.09.2025 stellte Projektkoordinator Clemens Kolbitsch das Team und das Vorhaben schriftlich vor. Das Projektteam bestand aus Leonard Ostler, Luis Vázquez und Clemens Kolbitsch und führte das Projekt im Rahmen der Diplomarbeit am Kolleg für Informatik von Ingenium Education durch. Durch den externen Projektauftrag sollte sichergestellt werden, dass die entwickelte Software einen konkreten Praxisbezug besitzt und an den tatsächlichen Bedürfnissen der FoodCoops ausgerichtet wird.

Ein wesentlicher Bestandteil der Zusammenarbeit war die Einbindung der späteren Nutzer:innen. Dazu wurde über das FoodCoops-Vernetzungsforum ein Termin für ein Online-Meeting vereinbart. In diesem Meeting wurde den Teilnehmer:innen das Projekt coOps lite vorgestellt. Anschließend gaben sie Feedback zum Projekt und zu den bisherigen Entwürfen. Die gewonnenen Rückmeldungen bildeten gemeinsam mit der praktischen Erfahrung der IG FoodCoops eine wichtige Grundlage für die Ermittlung der Anforderungen an die Software.

Zur formalen Regelung des Projektauftrags erstellte Projektkoordinator Clemens Kolbitsch einen Softwarevertrag sowie einen Auftragsverarbeitungsvertrag. Der Auftragsverarbeitungsvertrag sollte den Umgang mit personenbezogenen Daten im Rahmen des geplanten Betriebes regeln. Beide Vertragsunterlagen wurden am 21.09.2025 an die IG FoodCoops übermittelt. Die Umsetzung des Projekts wurde unentgeltlich vereinbart. Die Verträge dienten der Festlegung der Zusammenarbeit, der Verantwortlichkeiten und der rechtlichen Rahmenbedingungen.

Ein konkreter Fertigstellungstermin wurde nicht vereinbart. Die Vertragslaufzeit beginnt mit der Unterzeichnung und endet mit der Abnahme der Software.

Janet Wissuwa prüfte und unterzeichnete beide Verträge und übermittelte sie am 22.09.2025 an das Projektteam zurück. Am 01.10.2025 wurden die Unterlagen auch vom Projektteam gegengezeichnet. Mit der beidseitigen Unterzeichnung war der Projektauftrag verbindlich bestätigt. Anschließend wurde die Ausarbeitung des Entwurfs sowie die strukturierte Sammlung von Rückmeldungen aus dem Umfeld der FoodCoops fortgeführt.

2.2. Stakeholder-Übersicht

2.2.1. IG FoodCoops

Die IG FoodCoops – Interessengemeinschaft für FoodCoops versteht sich selbst als „Interessensvertretung von und für die österreichischen FoodCoops“. (foodcoops.at/interessengemeinschaft) Sie ist ein Verein mit Sitz in 1190 Wien. Ihre Organisation erfolgt überwiegend unentgeltlich.

Folgende relevante Interessen wurden identifiziert:

- Bereitstellung von Softwarelösungen, die die Organisation von FoodCoops fördern
- Nutzung von Softwarelösungen, die die Organisation der Interessengemeinschaft selbst erleichtern
- Gewinnung von Personen für Engagement in der Interessengemeinschaft



2.2.2. FoodCoops in Österreich

Die etwa 120 österreichischen FoodCoops sind eigenständige Organisationen, meist mit Rechtsform Verein oder Gesellschaft bürgerlichen Rechts. Ihre Organisation erfolgt in der Regel unentgeltlich durch Mitglieder. Etwa 75 FoodCoops sind Mitglieder der IG FoodCoops.

FoodCoops sind die potenziellen Nutzer:innen von coOps lite. Jedoch verfolgen nicht alle österreichischen FoodCoops die Entwicklung.

Folgende relevante Interessen wurden identifiziert:

- Reduktion ihres eigenen jeweiligen Organisationsaufwandes
- Gleichmäßigere Verteilung ihres Organisationsaufwandes auf ihre Mitglieder
- Erhöhung der Wirksamkeit und Zuverlässigkeit der geleisteten Arbeit
- Gewinnung weiterer Mitglieder bzw. Abschwächung von Mitgliederschwind (manche FoodCoops haben jedoch ihre gewünschte Größe erreicht und verfolgen kein weiteres Wachstum)

Die genauen Anforderungen unterscheiden sich teilweise stark zwischen den einzelnen FoodCoops. Um die inhomogene Gruppe der FoodCoops in das Projekt einzubinden, wurde eine Feedbackgruppe bestehend aus Mitgliedern mehrerer FoodCoops gebildet.

Eine besondere Nähe zum Projekt besteht bei den beiden Grazer FoodCoops KrautKOOPf (8010 Graz-Geidorf, krautkoopf.at) und EggenLend (8020 Graz-Eggenberg, foodcoop.eggenlend.at). Projektteammitglied Leonard Ostler ist selbst in der Organisation dieser beiden FoodCoops involviert. Sie sind außerdem Mitglieder der IG FoodCoops.



2.2.3. coOps

coOps ist eine von Manuel Leitner und Leonard Ostler gestartete Initiative mit Sitz in Graz. Ziel ist die Entwicklung einer neuen, umfassenden Softwarelösung für die Organisation von Gemeinschaften unterschiedlicher Art. Seit 2024 wird an einem Software-Konzept gearbeitet. Ein Umsetzungsprojekt ist in Planung.

Folgende relevante Interessen wurden identifiziert:

- Weitere Ausarbeitung von entworfenen Teilkonzepten
- Erprobung von Teilkonzepten in der Praxis
- Verwendung von in der Diplomarbeit erarbeiteten Konzepten und Werken



2.2.4. Ingenium Education

Die Ingenium Education GmbH (ingenium.co.at) ist ein Bildungsanbieter mit Sitz in Graz und bietet unter anderem das Kolleg für Informatik an. Ingenium Education stellt den schulischen und organisatorischen Rahmen der Diplomarbeit bereit.

Hauptverantwortlicher Betreuer ist DI Andreas Schwarzl, BSc. Er unterrichtete die Mitglieder des Projektteams während der beiden Ausbildungsjahre im Fach Webprogrammierung und Mobile Computing (WMC) und ist dadurch mit ihren fachlichen Kenntnissen und Arbeitsweisen vertraut. Im Rahmen der Betreuung begleitet er den Projektfortschritt und unterstützt das Team bei fachlichen, organisatorischen und softwarearchitektonischen Fragen.

Folgende relevante Interessen wurden identifiziert:

- Einhaltung der formalen und wissenschaftlichen Vorgaben für Diplomarbeiten
- fristgerechte Planung, Durchführung und Dokumentation des Projekts
- klare Abgrenzung und Nachvollziehbarkeit der individuellen Leistungen
- fachlich korrekte und praxisorientierte Umsetzung des Projekts
- erfolgreicher Abschluss der Diplomarbeit durch das Projektteam

Andreas Schwarzl legte das Diplomarbeitsthema am 19. November 2025 im ABA-Portal an und nahm es am 1. Februar 2026 als Betreuer an. Die Annahme durch die Direktion erfolgte am 3. Februar 2026.

2.2.5. Diplomarbeitsteam

Das Diplomarbeitsteam besteht aus Clemens Kolbitsch, Leonard Ostler und Luis Vázquez. Die Entwicklung von coOps lite erfolgt als gemeinsames Projekt. Entsprechend den Vorgaben für Diplomarbeiten besitzt jedes Teammitglied einen klar abgegrenzten und eigenständig dokumentierten Aufgabenbereich.

Clemens Kolbitsch

Projektleitung und Hauptansprechpartner

Hauptverantwortung: Projektkoordination, Dokumentation, Testing sowie Login und Authentifizierung.

Clemens Kolbitsch koordiniert Termine, Anforderungen, Meilensteine und den Projektfortschritt. Als Hauptansprechpartner übernimmt er die Kommunikation mit der Betreuung und weiteren Stakeholder:innen. Zu seinen Aufgaben gehören außerdem die Projektdokumentation, das Risiko- und Issue-Tracking sowie die Überprüfung der Akzeptanzkriterien.



Während der Entwicklung wurde sein Verantwortungsbereich um das systematische Testing der Software erweitert. Darüber hinaus trägt er die technische Verantwortung für die Login- und Authentifizierungsfunktionen, insbesondere für die Schnittstellenspezifikation, das Testing und die Abnahme der Umsetzung.

Technische Verantwortungsbereiche

Die technische Entwicklung von coOps lite wird zwischen Leonard Ostler und Luis Vázquez aufgeteilt. Leonard Ostler verantwortet vorwiegend Backend und Architektur, während Luis Vázquez den Schwerpunkt Frontend und UX/UI übernimmt.

Leonard Ostler

Product Owner und Lead Developer

Hauptverantwortung: Produktanforderungen, Softwarearchitektur und Backend.

Leonard Ostler priorisiert die Anforderungen und User-Stories. Er ist für die Softwarearchitektur sowie zentrale Komponenten des Backend, der Datenbank und der API verantwortlich. Außerdem führt er Code-Reviews durch und übernimmt die technische Abstimmung mit den Stakeholder:innen.



Luis Vázquez

Frontend und UX/UI

Hauptverantwortung: UX/UI-Konzept, Frontend-Architektur und Frontend-Implementierung.

Luis Vázquez ist für die Konzeption und Umsetzung der Benutzeroberfläche von coOps lite verantwortlich. Sein Aufgabenbereich umfasst die Entwicklung von Mockups und Benutzerflüssen sowie die technische Umsetzung responsiver und wiederverwendbarer Frontend-Komponenten mit React und TypeScript. Dazu gehören unter anderem Navigation, Kontextwechsel, Filter- und Detailansichten.



Darüber hinaus übernimmt er die visuelle Gestaltung der Anwendung und der verschiedenen Benutzeroberflächen. Bei der Umsetzung berücksichtigt er insbesondere Benutzerfreundlichkeit, mobile Nutzung, Barrierefreiheit, Übersichtlichkeit und Performance sowie die Anbindung der Benutzeroberfläche an die vom Backend bereitgestellten Daten.

2.3. Forschungsthema: Der Einfluss der Projektleitung auf den Erfolg von Softwareprojekten

Softwareprojekte erfordern neben technischer Entwicklung die Abstimmung von Anforderungen, Zuständigkeiten und verfügbaren Ressourcen. Die Projektleitung klärt mit den Beteiligten, welche Anforderungen umgesetzt werden, wer die Aufgaben übernimmt und welche Entscheidungen noch ausstehen. Dieser Teil der Diplomarbeit untersucht folgende Forschungsfrage:



Wie beeinflusst die Rolle und das Handeln

der Projektleitung den Erfolg von

Softwareprojekten in den Bereichen

Kommunikation, Koordination und Risikomanagement?



Ziel ist es, anhand wissenschaftlicher Beiträge und ausgewählter Fachliteratur herauszuarbeiten, welche Aufgaben und Verhaltensweisen der Projektleitung die Zielerreichung unterstützen können.

Die Untersuchung ist literaturgestützt. Wissenschaftliche Studien zur Teamarbeit werden mit grundlegender Projektmanagementliteratur und praxisbezogenen Darstellungen verbunden. Der Scrum Guide dient zur Einordnung agiler Verantwortlichkeiten. Die Dokumentation von Atlassian erläutert die praktische Anwendung von Workflows und Sprintplanung. Die Literatursauswahl ist thematisch begründet und stellt keine systematische Literaturübersicht dar.

Im abschließenden Reflexionsabschnitt werden die theoretischen Ansätze auf ausgewählte Erfahrungen aus coOps lite bezogen. Grundlage sind projektbezogene Chatnachrichten, Feedbacknotizen, schriftliche Rückmeldungen der Entwickler und die eigene Einschätzung des Verfassers. Da dieser zugleich die Projektleitung übernahm, werden dokumentierte Ereignisse und persönliche Bewertungen voneinander unterschieden.

2.3.1. Rolle der Projektleitung und Kriterien für Projekterfolg

Aufgaben und Verantwortlichkeiten

Zu den Aufgaben der Projektleitung gehören die Klärung der Ziele, die Planung und Nachverfolgung der Arbeit sowie die Abstimmung mit den Beteiligten. Whitten nennt außerdem die Festlegung von Rollen, die Bearbeitung wichtiger Risiken und Probleme, die Einbindung des Kunden und die Unterstützung der Teammitglieder. (Whitten, 1999, S. 16)

Die Projektleitung muss nicht jede fachliche Entscheidung selbst treffen. Sie muss jedoch klären, wer darüber entscheidet und welche Aufgaben bis dahin nicht weiterbearbeitet werden können. Im Arbeitsablauf muss nachvollziehbar sein, wer Anforderungen priorisiert, technische Fragen entscheidet und Ergebnisse überprüft.

Die konkrete Verteilung dieser Aufgaben hängt vom Vorgehensmodell ab. Scrum definiert die Verantwortlichkeiten von Product Owner, Scrum Master und Developers. Eine zusätzliche Projektleiterrolle ist innerhalb des Scrum Teams nicht vorgesehen. Das Team managt sich selbst und entscheidet intern über die Organisation seiner Arbeit. Deshalb dürfen klassische Projektleitungsaufgaben nicht ohne Weiteres mit den Aufgaben des Scrum Masters gleichgesetzt werden. (Schwaber & Sutherland, 2020, S. 5-7)

Kriterien für Projekterfolg

Um den Erfolg eines Projekts beurteilen zu können, müssen die vereinbarten Ergebnisse, Termine und verfügbaren Mittel bekannt sein. Zeit, Kosten und Leistungsumfang bilden dabei miteinander verbundene Zielgrößen. Wird der Funktionsumfang erweitert, kann dies zusätzliche Arbeitszeit oder weitere Ressourcen erfordern. Watt berücksichtigt neben diesen Größen auch Qualität und Risiken als relevante Rahmenbedingungen der Projektarbeit. (Watt, 2014, Kap. 2)

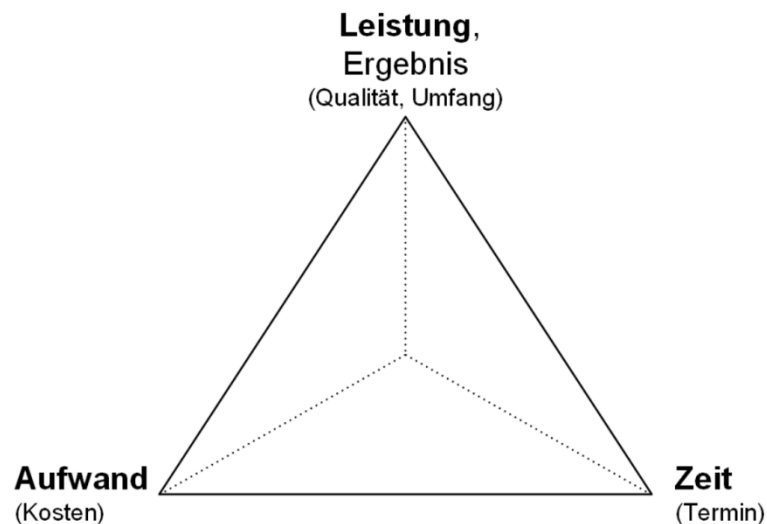


Abbildung 1: Das magische Dreieck des Projektmanagements (Peterjohann, 2026)

Das Dreieck (Abbildung 1) veranschaulicht die Abhängigkeiten zwischen den Zielgrößen, bildet den Erfolg eines Softwareprojekts aber nicht vollständig ab. Eine Anwendung kann beispielsweise innerhalb des vorgesehenen Zeitrahmens entstehen und dennoch die tatsächlichen Anforderungen der Nutzer:innen verfehlen. Neben Terminen und Aufwand ist deshalb zu prüfen, ob das Ergebnis die vereinbarten Funktionen und Qualitätsanforderungen erfüllt. (Watt, 2014, Kap. 2)

Bei einem zusätzlichen Funktionswunsch muss geklärt werden, welchen Nutzen er hat und welcher Aufwand dadurch entsteht. Reichen Zeit und Ressourcen nicht aus, muss der Umfang angepasst oder eine andere Aufgabe zurückgestellt werden.

2.3.2. Kommunikation

Informationsbedarf und gemeinsames Verständnis

Kommunikationsplanung legt fest, welche Informationen die Beteiligten benötigen, wann diese verfügbar sein müssen und wie sie übermittelt werden. Eine vollständige Sammlung von Nachrichten hilft nur begrenzt, wenn wichtige Entscheidungen darin schwer auffindbar sind oder nicht deutlich wird, wer darauf reagieren soll. (Watt, 2014, Kap. 15)

Dabei lassen sich synchrone und asynchrone Kommunikationsformen unterscheiden. Besprechungen ermöglichen unmittelbare Rückfragen. Schriftliche Mitteilungen können dagegen zeitversetzt bearbeitet und später erneut eingesehen werden. Bei der Auswahl der Kommunikationsmittel sind auch die Erfahrung des Teams und ein möglicher Schulungsbedarf zu berücksichtigen. (Watt, 2014, Kap. 15)

Bei unklaren Anforderungen kann ein Gespräch offene Fragen klären. Anschließend sollten die Entscheidung, die Zuständigkeit und noch offene Punkte schriftlich festgehalten werden. Aus einer Nachricht sollte außerdem hervorgehen, ob eine Rückmeldung oder eine Handlung erwartet wird.

Kommunikation als Bestandteil der Teamarbeit

Hoegl und Gemuenden betrachten Kommunikation als eine von sechs Dimensionen der Qualität von Teamarbeit (Abbildung 2). Weitere Dimensionen sind Koordination, ausgewogene Beiträge der Teammitglieder, gegenseitige Unterstützung, gemeinsamer Einsatz und Zusammenhalt. Ihre Untersuchung zeigt Zusammenhänge zwischen der Qualität der Teamarbeit und der Teamleistung. Kommunikation wird dabei als Bestandteil des gemeinsamen Arbeitens betrachtet und nicht als isolierte Tätigkeit einer Führungsperson. (Hoegl & Gemuenden, 2001, S. 435)

Sechs Dimensionen der Teamwork Quality

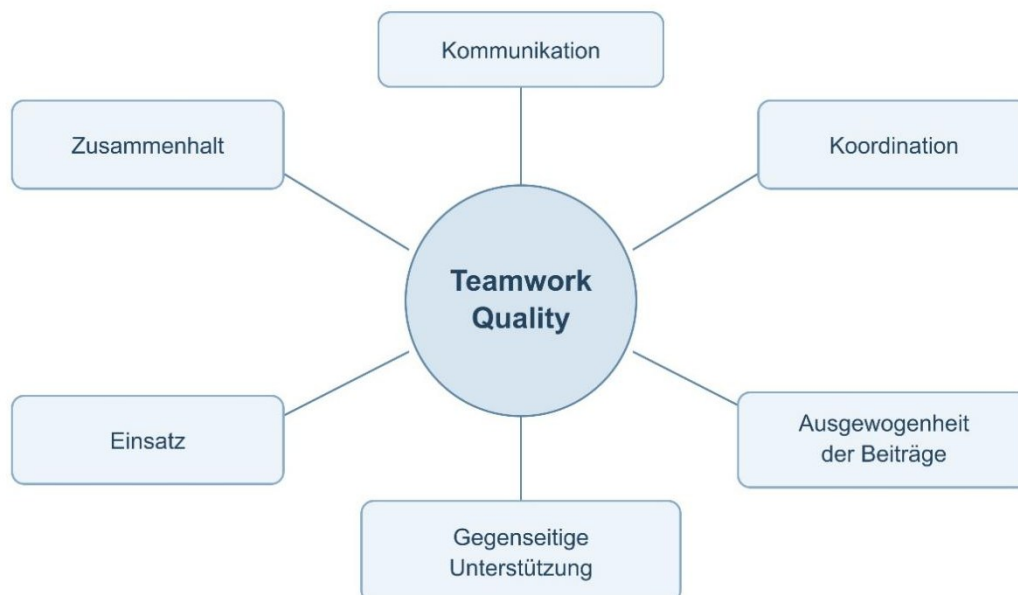


Abbildung 2: Sechs Dimensionen der Teamwork Quality nach Hoegl und Gemuenden (Eigene Darstellung)

Eine neuere Untersuchung von Verwijs und Russo befasst sich mit der Wirksamkeit von Scrum Teams. Ihr Modell umfasst Reaktionsfähigkeit, die Berücksichtigung der Stakeholder-Bedürfnisse, kontinuierliche Verbesserung, Teamautonomie und Unterstützung durch das Management. Besonders hervorgehoben wird das Zusammenspiel zwischen Stakeholder-Orientierung und der Fähigkeit, auf neue Bedürfnisse zu reagieren. (Verwijs & Russo, 2023)

Bei der Bearbeitung von Rückmeldungen sollte geklärt werden, ob eine bestehende Anforderung unklar ist, ein Fehler vorliegt oder eine zusätzliche Funktion gewünscht wird. Davon hängt ab, welche Entscheidung erforderlich ist.

Gesprächsführung und Entscheidungen

Bei unterschiedlichen Auffassungen muss zunächst geklärt werden, worin die Uneinigkeit besteht. Beteiligte können dieselben Begriffe verwenden und trotzdem unterschiedliche Ergebnisse erwarten. Rückfragen und kurze Zusammenfassungen helfen, solche Unterschiede sichtbar zu machen.

Whitten zählt die Unterstützung bei der Problemlösung zu den Aufgaben der Projektleitung (Whitten, 1999, S. 16). Bei einer Besprechung zu einer offenen Frage sollte deshalb am Ende feststehen, ob sie geklärt ist oder welche Information für eine Entscheidung noch fehlt.

2.3.3. Koordination

Aufgaben und Abhängigkeiten

Koordination stimmt einzelne Tätigkeiten aufeinander ab. Ausgangspunkt ist der vorgesehene Projektumfang: Welche Ergebnisse sollen entstehen und woran wird ihre Erfüllung erkannt? Daraus werden Tätigkeiten abgeleitet, zeitlich geordnet und hinsichtlich ihrer benötigten Ressourcen eingeschätzt. Abhängigkeiten bestimmen dabei, welche Vorarbeiten für nachfolgende Aufgaben erforderlich sind. (Watt, 2014, Kap. 9–10)

In Softwareprojekten entstehen solche Abhängigkeiten beispielsweise zwischen Frontend und Backend. Beide Bereiche können getrennte Aufgaben bearbeiten, müssen sich aber über die ausgetauschten Daten und verfügbaren Schnittstellen verständigen. Vor der Umsetzung einer gemeinsamen Funktion müssen Frontend und Backend beispielsweise klären, welche Daten übertragen werden und wie die benötigte Schnittstelle verwendet wird. Weitere Details können im Verlauf der Entwicklung ergänzt werden.

Watt beschreibt hierfür eine schrittweise Verfeinerung der Planung. Unmittelbar bevorstehende Tätigkeiten werden genauer geplant, während spätere Aufgaben zunächst gröber beschrieben bleiben. Neue Erkenntnisse fließen in die weitere Planung ein. (Watt, 2014, Kap. 10)

Die Projektleitung muss daher neben dem Bearbeitungsstand auch überprüfen, ob die ursprünglichen Annahmen noch zutreffen. Ist eine benötigte Schnittstelle nicht verfügbar oder erweist sich eine Vorarbeit als aufwendiger, müssen davon betroffene Aufgaben neu abgestimmt werden.

Werkzeuge und Arbeitsregeln

Jira ist eine Software von Atlassian zur Planung und Nachverfolgung von Aufgaben in Projekten. Aufgaben werden als einzelne Vorgänge erfasst und durchlaufen einen festgelegten Arbeitsablauf, den sogenannten Workflow. Dieser besteht aus Status und Übergängen: Ein Status kennzeichnet den Bearbeitungsstand einer Aufgabe, beispielsweise „To Do“, „In Progress“ oder „Done“. Übergänge legen

fest, zwischen welchen Status gewechselt werden kann. So lässt sich der vereinbarte Arbeitsablauf im Werkzeug abbilden. (Atlassian a)

Die technische Konfiguration legt jedoch nicht von selbst fest, welche Qualität ein Ergebnis haben muss. Das Team muss zusätzlich vereinbaren, wann eine Aufgabe begonnen werden kann, wann sie geprüft wird und unter welchen Bedingungen sie als abgeschlossen gilt.

Aufgabenbeschreibungen sollten das erwartete Ergebnis und wesentliche Voraussetzungen erkennen lassen. Werden mehrere Werkzeuge verwendet, muss außerdem nachvollziehbar bleiben, wo Anforderungen, Entwürfe und Entscheidungen abgelegt sind.

Sprintplanung und verfügbare Kapazität

Sprintplanung verbindet ein gemeinsames Ziel mit einem ausgewählten Arbeitsumfang. West beschreibt die Festlegung des Sprintziels, die Auswahl geeigneter Backlog-Einträge und die Berücksichtigung der verfügbaren Teamkapazität. Aufwandsschätzungen beruhen auf dem zum Planungszeitpunkt verfügbaren Wissen. Sie beschreiben, was das Team voraussichtlich bearbeiten kann. (West)

Bei der Auswahl der Sprintaufgaben müssen vorhersehbare Abwesenheiten und andere Verpflichtungen berücksichtigt werden. Ebenso ist zu prüfen, ob benötigte Vorarbeiten rechtzeitig vorliegen.

2.3.4. Risikomanagement

Risiken erkennen und behandeln

Risikomanagement beschäftigt sich mit Unsicherheiten, die sich auf die Projektziele auswirken können. Boehm unterscheidet zwischen Risikobewertung und Risikosteuerung. Zur Bewertung gehören Identifikation, Analyse und Priorisierung. Die Steuerung umfasst Maßnahmenplanung, Risikobearbeitung und Überwachung. (Abbildung 3) (Boehm, 1991, S. 34)

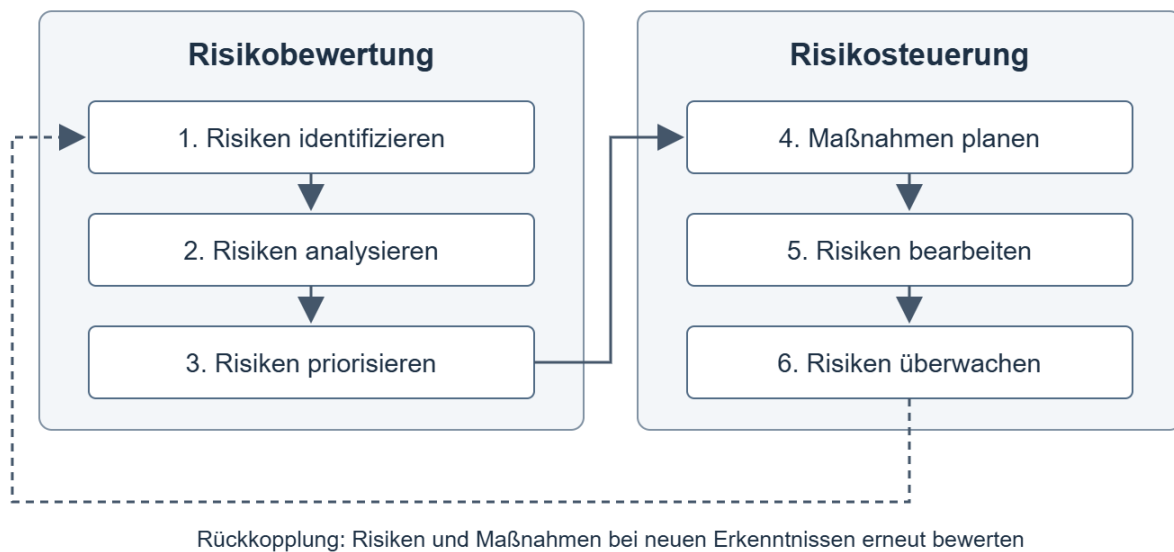


Abbildung 3: Schritte des Software-Risikomanagements (Eigene vereinfachte Darstellung in Anlehnung an Boehm (1991, S. 34, Abb. 2); Rückkopplung ergänzt)

Eine praktische Form der Dokumentation ist das Risikoregister. Darin werden Risiken, ihre Bewertung, Zuständigkeiten und vorgesehene Maßnahmen festgehalten. Die Übersicht wird während des Projekts überprüft und angepasst, damit neue Erkenntnisse berücksichtigt werden können. (Atlassian b)

Auch eine einfache Tabelle kann als Risikoregister dienen. Sie sollte das Risiko, seine Eintrittswahrscheinlichkeit, mögliche Auswirkungen, die Priorität, eine verantwortliche Person und die vereinbarte Maßnahme enthalten. Ein Überprüfungstermin macht deutlich, wann die Einschätzung erneut betrachtet werden soll.

Umfang, Lernbedarf und technische Unsicherheit

Boehm nennt unter anderem unrealistische Termine und Budgets, ungeeignete Funktionen sowie zusätzliche, nicht notwendige Funktionalität als Risiken von Softwareprojekten. Diese Punkte betreffen sowohl die Auswahl der Anforderungen als auch die Einschätzung des erforderlichen Aufwands. (Boehm, 1991, S. 35)

Wenn die Erfahrung mit einer technischen Aufgabe fehlt, ist ihre Aufwandsschätzung besonders unsicher. Eine zeitlich begrenzte Erprobung kann zeigen, welche Schwierigkeiten auftreten und ob die geplante Lösung umsetzbar ist. Ebenso ist zu prüfen, ob zusätzliche Funktionen für das vereinbarte Ziel erforderlich sind oder zunächst zurückgestellt werden können.

Risikomanagement ist damit keine einmalige Prüfung zu Projektbeginn. Ändern sich Anforderungen, verfügbare Kapazitäten oder technische Annahmen, müssen auch die geplanten Maßnahmen überprüft

werden. Bei jeder Überprüfung ist deshalb zu entscheiden, ob die bisherige Maßnahme ausreicht oder angepasst werden muss.

2.3.5. Fazit zum Forschungsthema

Die Projektleitung kann den Erfolg von Softwareprojekten unterstützen, indem sie offene Entscheidungen klärt, voneinander abhängige Aufgaben abstimmt und auf erkennbare Risiken reagiert. Im Bereich der Kommunikation muss deutlich werden, welche Information oder Entscheidung benötigt wird. Bei der Koordination geht es darum, Zuständigkeiten und technische Voraussetzungen rechtzeitig abzustimmen. Im Risikomanagement müssen Unsicherheiten bewertet und Maßnahmen nachverfolgt werden.

Der Beitrag der Projektleitung hängt damit nicht allein von der Planung zu Projektbeginn ab. Entscheidend ist auch, ob neue Anforderungen, Verzögerungen und veränderte Kapazitäten während der Entwicklung aufgegriffen werden. Welchen Anteil eine einzelne Person am Projekterfolg hat, lässt sich aus den herangezogenen Untersuchungen zur Teamarbeit jedoch nicht bestimmen. (Hoegl & Gemuenden, 2001; Verwijs & Russo, 2023)

2.3.6. Reflexion der Projektleitung bei coOps lite

Kommunikation und Eingrenzung der Anforderungen

Bei coOps lite übernahm ich die Projektleitung. Eine meiner wichtigsten Aufgaben war, mit Leonard zu klären, welche seiner Ideen wir umsetzen wollten, und diese als Anforderungen festzuhalten. Im Einzelchat vom 12.09.2025 bat ich Leonard, sich zunächst auf zentrale Funktionalitäten zu konzentrieren und daraus eine Grundlage für Prototyp und erste nutzbare Version zu schaffen. In seiner schriftlichen Rückmeldung bewertete er die Klärung der User Stories als hilfreich. (Kolbitsch & Ostler, 2025–2026, Nachricht vom 12.09.2025; Ostler, 2026)

Rückblickend war dabei nicht allein die Häufigkeit unserer Gespräche entscheidend. Wichtig war, dass Vorstellungen konkretisiert und schriftlich festgehalten wurden. Die Abstimmung blieb jedoch anspruchsvoll, weil neue Ideen zugleich neue Fragen zum Umfang aufwarfen. Luis hätte sich eine stärkere gemeinsame Auswahl und Begrenzung der Funktionen gewünscht. (Vázquez, 2026)

Bei der bereits erwähnten Präsentation vor der FoodCoops-Feedbackgruppe am 12. Oktober 2025 wurde zehn Teilnehmenden ein klickbarer, mit Figma erstellter Prototyp vorgestellt. Die anschließenden Rückmeldungen zeigten, dass der Zweck und die Bedienung der geplanten Anwendung noch Fragen

aufwarfen. Die Notizen nennen eine komplizierte Konfiguration, kleine Schrift und die notwendige Abgrenzung gegenüber der bestehenden Software “Foodsoft”. Es war besonders wichtig, dass coOps lite als Ergänzung und nicht als Ersatz für Foodsoft verstanden wurde. Das Feedback unterstützte die Diskussion über eine einfachere Gestaltung; eine daraus entstandene Softwareänderung ist nicht belegt. (Projektteam coOps lite, 2025, S. 1-2)

Jira, Sprints und technische Abstimmung

Mit der Einrichtung von Jira wollte ich eine strukturierte Arbeitsumgebung schaffen. Ich passte Felder und Workflow an und erläuterte dem Team die Nutzung. Da Jira für mich neu war, benötigten Einrichtung und Einarbeitung zusätzliche Zeit. Ein von Leonard eingerichteter gemeinsamer Dropbox-Ordner wurde für die Ablage von Dokumenten genutzt; Figma diente zur Erstellung des klickbaren Entwurfs. (Projektteam coOps lite, 2025–2026, Nachrichten vom 07.08., 07.09. und 08.12.2025)

Luis beschrieb Jira als hilfreich, um offene und erledigte Aufgaben sowie ihre Reihenfolge zu erkennen. Leonard hob die interne Terminkoordination und die Abstimmung mit dem Betreuer hervor. (Ostler, 2026; Vázquez, 2026)

Unsere Projektkoordination orientierte sich an Scrum, wurde jedoch an die Rahmenbedingungen des Schulprojekts angepasst. Diese vereinfachte Arbeitsweise wird hier als „Scrum light“ bezeichnet.

Die beiden durchgeführten Sprints empfand ich als besonders intensive und produktive Phase. Diese Einschätzung beruht auf meiner Wahrnehmung, nicht auf einer vergleichenden Messung. Aufgrund schulischer Verpflichtungen und des erforderlichen Lernaufwands konnten wir weitere Sprints nicht in derselben Form fortsetzen. Im Rückblick hätte die Planung stärker berücksichtigen müssen, wie viel gemeinsame Arbeitszeit über einen längeren Zeitraum tatsächlich verfügbar war.

Trotz der Aufgabenplanung blieben technische Abhängigkeiten unzureichend geklärt. Leonard berichtete, dass grundlegende Backend-Arbeiten aufwendiger waren als erwartet, während das Frontend ohne ausreichend genaue Vorgaben weiterentwickelt wurde. Er nannte ein früheres API-Design und klarere Branch-Regeln als Verbesserungsmöglichkeiten. (Ostler, 2026)

Umfang, Lernaufwand und Zielerreichung

Eine Begrenzung des MVP-Umfangs wurde am 19.11.2025 im Gruppenchat festgehalten. Leonard veröffentlichte einen gemeinsam mit mir erarbeiteten Vorschlag, der unter anderem die Ausführung von durch Anwender:innen definiertem Programmcode zunächst ausschloss und weitere Funktionen zurückstellte. Diese Entscheidung betraf die Planung; sie belegt nicht die vollständige Umsetzung des vorgesehenen MVP. (Projektteam coOps lite, 2025–2026, Nachricht vom 19.11.2025)

Der Lernaufwand blieb dennoch erheblich. Obwohl wir auf Kenntnisse aus dem Unterricht zurückgreifen konnten, ging deren Anwendung im Projekt deutlich über die bisherigen Übungsaufgaben hinaus. Den zusätzlichen Einarbeitungsaufwand hatte ich bei der Planung nicht ausreichend berücksichtigt. Hinzu kamen organisatorische Wartezeiten, etwa bei der Freischaltung von Figma-Bildungskonten. (Kolbitsch & Vázquez, 2025–2026, Nachrichten vom 02.09. und 19.09.2025; Projektteam coOps lite, 2025–2026, Nachricht vom 12.11.2025)

Ein durchgängig geführtes Risikoregister lag nicht vor. In einer gemeinsamen Risikoübersicht hätten wir offene technische Fragen, den erforderlichen Einarbeitungsaufwand und externe Abhängigkeiten festhalten können. Das hätte uns eine Grundlage gegeben, um Aufgaben früher einzugrenzen und zeitliche Reserven gezielter einzuplanen.

Der angestrebte MVP wurde innerhalb der Diplomarbeit nicht vollständig umgesetzt. Ein konkreter Abnahmetermin ist im Softwarevertrag nicht festgelegt (IG FoodCoops & Projektteam coOps lite, 2025, Abschn. 5). Da coOps lite über die Diplomarbeit hinaus weiterentwickelt wird, bildet der dokumentierte Entwicklungsstand ein Zwischenergebnis. Rückblickend hätte eine frühere Eingrenzung des Funktionsumfangs und Abstimmung der technischen Schnittstellen dennoch dazu beitragen können, mehr zusammenhängende Funktionen fertigzustellen.

3. Beobachtung und Analyse der Anforderungen

Dem Diplomarbeitsprojekt gehen jahrelange Beobachtungen, Entwicklung früherer Softwarelösungen und die Durchführung einer Vorstudie voraus, an denen Teammitglied Leonard Ostler selbst maßgeblich beteiligt war. Diese Vorgeschichte wird im Folgenden zusammengefasst und die betreffenden Stakeholder dabei näher vorgestellt.

3.1. Über FoodCoops

3.1.1. Beschreibung

Eine FoodCoop (food cooperative, zu Deutsch: Lebensmittelkooperative) ist eine Gruppe von Privatpersonen, die gemeinsam Produkte von ausgewählten Bezugsquellen wie Bauernhöfen, Produzent:innen und Händlern beziehen. (Abbildung 4)

In Österreich gibt es im Jahr 2026 rund 120 FoodCoops. Die meisten davon haben die Rechtsform Verein, sind rein ehrenamtlich selbstorganisiert und erheben keinen Preisaufschlag. Ihre Mitgliederzahl variiert zwischen etwa 10 und 200. Die ersten FoodCoops in Österreich wurden 2007 gegründet.

Die IG FoodCoops, die sich als Interessengemeinschaft der FoodCoops in Österreich versteht, definiert FoodCoops als „nicht gewinnorientierte, selbstverwaltete Gemeinschaften, die eine Infrastruktur zur Besorgung und Verteilung von fairen und nachhaltigen Produkten (hauptsächlich Lebensmittel) bieten. Alle Mitglieder einer FoodCoop müssen das Recht zur Mitbestimmung haben.“ (IG FoodCoops, 2017)

FoodCoops und ähnliche Gruppen gibt es weltweit in verschiedenen Formen. Sie bezeichnen sich mitunter auch als Food Hubs oder Einkaufsteams. Das Open Food Network (openfoodnetwork.org) wurde 2012 in Australien als Plattform für Food Hubs gegründet und ist inzwischen auch in weiteren Ländern aktiv. In Österreich haben sich jedoch andere Softwares durchgesetzt.



Abbildung 4: Illustration des FoodCoop-Prinzips (Grafik von gafinen.com unter CC-BY-NC-SA Lizenz)

3.1.2. Persönliche Tätigkeit und Erfahrung

Leonard Ostler ist seit 2017 Mitglied der Grazer Foodcoop KrautKOOPf sowie seit 2020 in der IG FoodCoops aktiv. In dieser Zeit hat er Erfahrung in sämtlichen Organisationsbereichen wie Bestellorganisation, Lieferantenbetreuung, Finanzabwicklung, Öffentlichkeitsarbeit, Lagereinrichtung, EDV, Softwareentwicklung, Aufgabenkoordination, Mitgliederverwaltung und Moderation von gemeinschaftlichen Entscheidungsprozessen gesammelt. Seit 2022 leistet er unterstützende Arbeit in der FoodCoop EggenLend.

2023 hat er außerdem an dem Projekt FoodCoop 4.0 – Vorstudie (foodcoop40.org) maßgeblich mitgewirkt.

3.1.3. Warenbezug in einer FoodCoop

Es gibt verschiedene mögliche Abläufe für den Warenbezug in einer FoodCoop. Zu den üblichen zählen:

- **Sammelbestellung:** Die Mitglieder der FoodCoop erhalten eine Liste von Artikeln, die bei einer bestimmten Lieferant:in bestellbar sind. Sie können bis zu einem festgelegten Bestellschluss eintragen, welche Artikel und welche Menge jeweils sie erhalten möchten. Anschließend wird die Gesamtbestellung an die Lieferant:in geschickt und von dieser idR an den Abholraum der FoodCoop geliefert. Dort wird die Lieferung von Mitgliedern der FoodCoop kontrolliert. Danach können die anderen Mitglieder ihren Anteil abholen. Abschließend erfolgen die interne Abrechnung (Belastung der Guthabenkonto der Mitglieder) sowie das Begleichen der Rechnung der Lieferant:in.
- **Lagerartikel:** Ware wird von der FoodCoop im Voraus bestellt oder von Lieferant:innen auf Kommission geliefert. Mitglieder können ohne Vorbestellung flexible Mengen entnehmen und eintragen. Die Abrechnung erfolgt nachträglich.
- **Übergreifende Sammelbestellung:** Eine FoodCoop bestellt gemeinsam mit weiteren FoodCoops oder anderen Bestellpartnern, um Transportaufwand einzusparen oder Mindestbestellmengen zu erfüllen.

Im Detail unterscheiden sich die Abläufe häufig je nach FoodCoop und auch je nach Lieferant:in: Beispielsweise ob bei Lieferungen lediglich die Gesamtmenge je Artikel kontrolliert wird und jedes Mitglied seinen Anteil später selbst herausnimmt, oder die Ware im Zuge der Kontrolle auch für die Mitglieder aufgeteilt wird („Einkistln“).

3.1.4. Bislang verwendete Software

Die FoodCoop 4.0 Vorstudie hat festgestellt: „Bei österreichischen FoodCoops sind die beiden Open-Source-Softwares Foodsoft und FoodCoopShop am weitesten verbreitet. Beide sind webbasierte

Softwares, in der eine auf einem Server gehostete Installation via Webbrowser bedient werden kann.“ (Arhar, Seebacher, Ostler, & Pabst, 2023, S. 12)

Bei diesen Softwares steht der Ablauf der Sammelbestellung im Fokus, daher werden sie auch als Bestellsoftwares bezeichnet. Dieser Ablauf ist fest einprogrammiert und lässt sich nicht dynamisch anpassen, wenn z. B. bei einer bestimmten Lieferant:in ein zusätzlicher Schritt erforderlich ist. Außerdem lassen sich keine Zuständigkeiten für bestimmte Schritte (z. B. welche Mitglieder diese Woche die Lieferungen kontrollieren) einstellen.

Die Foodsoft verfügt über ein rudimentäres Feature für Aufgabenkoordination. Es beschränkt sich darauf, dass einzelne Aufgaben angelegt, übernommen bzw. zugewiesen und als erledigt markiert werden können. Außerdem können wiederkehrende Aufgaben angelegt werden, für die in einem festgelegten Rhythmus automatisch Aufgaben angelegt werden. Jener Rhythmus (Anzahl von Tagen) kann jedoch nur global in einer Foodsoft-Instanz, also für alle Aufgaben in einer FoodCoop, eingestellt werden. Außerdem können keine Unteraufgaben angelegt, Verknüpfungen mit Bestellungen hergestellt, erforderliche Fähigkeiten für eine Aufgabe definiert oder Aufgaben automatisch an geeignete Mitglieder zugewiesen werden. Die Zuteilung der Aufgaben muss also entweder über Eigeninitiative (selbst übernehmen) oder manuelle Koordination erfolgen. (IG FoodCoops, 2024)

3.2. Koordination wiederkehrender Aufgaben

Eine zentrale Herausforderung in FoodCoops besteht in der fairen Verteilung des durch die Mitglieder unentgeltlich zu leistenden Arbeitsaufwandes. Viele FoodCoops berichten über Probleme bei der Verfolgung des Ziels der „Aufgabenfairteilung“.

Insbesondere bei regelmäßigen Aufgaben, bei denen angestrebt wird, dass sich alle Mitglieder der FoodCoop oder einer Arbeitsgruppe abwechseln, kommt es vor, dass sich für einzelne Termine niemand einträgt, die Zuständigkeit nicht geklärt ist und die Aufgabe unerledigt bleibt.

Die Foodkoop KrautKOOPf konnte mit dem eigens programmierten Tool *Task-Rotation* (github.com/twothreenine/task-rotation) erreichen, Verantwortlichkeiten für Termine automatisiert an Mitglieder zuzuweisen. Es werden bevorzugt Mitglieder ausgewählt, die bislang die geringste Zahl an Aufgaben im Verhältnis zu ihrer Mitgliedschaftsdauer erledigt haben. Um zu vermeiden, dass Mitglieder, die in jenem Verhältnis aufzuholen haben, direkt mehrfach hintereinander zugewiesen werden, kann global ein Zuweisungs-Abstandsfaktor (in Relation zur Anzahl der aktiven Mitglieder) eingestellt werden. Indem für einen Termin stets mindestens ein Mitglied ausgewählt wird, das schon bei mehreren Terminen dabei war, wird sichergestellt, dass notwendige Erfahrung vorhanden ist. Neue Mitglieder werden zusammen mit erfahrenen eingeteilt und so von diesen eingelernt. Die Zuverlässigkeit der Erledigung und die Verteilung des Arbeitsaufwands konnten durch die Verwendung dieses Tools

erheblich verbessert werden. Problematisch ist, dass Task-Rotation auf Online-Tabellen basiert, die nicht mobilfreundlich sind.

3.3. Automatisierung von EDV-Prozessen

Eine weitere große Herausforderung in FoodCoops stellt die Automatisierung arbeitsintensiver EDV-Prozesse dar. Beispiele dafür sind:

- Aktualisierung von Artikeldaten in der Bestellsoftware
- Eröffnung von Sammelbestellungen
in Foodsoft nur händisch; in FoodCoopShop können direkt Wiederholungsregeln eingestellt werden (FoodCoopShop – Features)
- Übermittlung von Bestellergebnissen an einen Lieferanten-Webshop
Foodsoft und FoodCoopShop bieten nur Übermittlung per E-Mail an
- Onboarding neuer Mitglieder – u. a. Accounts in mehreren Tools anlegen
- Offboarding von Mitgliedern, die ihre Mitgliedschaft beenden möchten

Bei der Aktualisierung von Artikeldaten ist zu berücksichtigen, dass die von den Lieferant:innen übermittelten Preislisten individuelle Formate haben. Da FoodCoops für sie idR nur einen von vielen Vertriebskanälen darstellen, müssen sie ein Format verwenden, das für ihre jeweiligen Kanäle (beispielsweise auch ausgedruckt in Papierform) passt, um die Preisliste nicht mehrfach aufbereiten zu müssen. Standardisierte Formate setzen sich bislang nicht durch. Ähnlich ist es bei Webshop-Systemen. Die händische Synchronisation in die FoodCoop-Bestellsoftware, teilweise wöchentlich mit über 100 Artikeln, bedeutet für die Bestellorganisator:in oder ggf. die Lieferant:in selbst einen großen Aufwand.

Mit dem eigens programmierten Tool *Lieferscraping* (github.com/twothreenine/krautkoopf_lieferscraping) konnte in den FoodCoops EggenLend und KrautKOOPf der regelmäßige, ehrenamtliche Aufwand erheblich reduziert werden. Außerdem wurden die Korrektheit und Zuverlässigkeit der Daten durch Reduktion von menschlichen Fehlerquellen (Tippfehler, Rechenfehler, Übersehen von Änderungen) verbessert. Dadurch wird Verwirrung vermieden und Korrekturaufwand erspart. Lieferscraping wird bislang ausschließlich auf einem lokalen PC betrieben und eignet sich in dieser Form nicht für einen Betrieb auf dem Server der IG FoodCoops. Während sich zuvor mehrere Mitglieder um die händische Abwicklung verschiedener EDV-Prozesse kümmerten, fielen diese durch die Umstellung auf Lieferscraping-Skripte alle Leonard Ostler zu. Die teilautomatische Ausführung diverser Prozesse durch ein einzelnes Mitglied steht dem Ziel der fairen Arbeitsverteilung entgegen.

3.4. Vorstudie „FoodCoop 4.0“

Um den Bedarf an Softwarelösungen zur Aufgabenkoordination und Automatisierung sowie zur Integration verschiedener Tools in eine gemeinsame Benutzeroberfläche genauer zu untersuchen und Lösungsansätze weiter zu erforschen, wurde 2023 eine Vorstudie von der FoodCoop EggenLend in Kooperation mit der FH Joanneum durchgeführt.

Mittels einer Fokusgruppe und einer Online-Umfrage mit FoodCoop-Mitgliedern sowie Befragungen von Lieferant:innen und Interessent:innen wurde das Potenzial nutzerfreundlicher Softwarelösungen herausgearbeitet und Anforderungen gesammelt. Automatisches Einlesen von Produktlisten, verbessertes Aufgabenverteilungsmanagement und Visualisierung von Abläufen gehörten dabei zu den als am wichtigsten bewerteten Anforderungen. Dabei wurde u. a. die Möglichkeit der Einbindung von Lieferant:innen in Abläufen gewünscht. Als nichtfunktionale Anforderungen wurden eine möglichst leicht verständliche und somit barrierearme Benutzeroberfläche, Tauglichkeit für Handy-Bildschirme, Push-Benachrichtigungen und Privatsphäre-Einstellungen genannt. (Arhar, Seebacher, Ostler, & Pabst, 2023, S. 29-31)

Im Rahmen der Vorstudie wurde ein Konzept für zusammengesetzte Abläufe entwickelt, in einem klickbaren Prototyp visualisiert (Abbildung 5) und der Fokusgruppe aus FoodCoop-Mitgliedern vorgestellt.

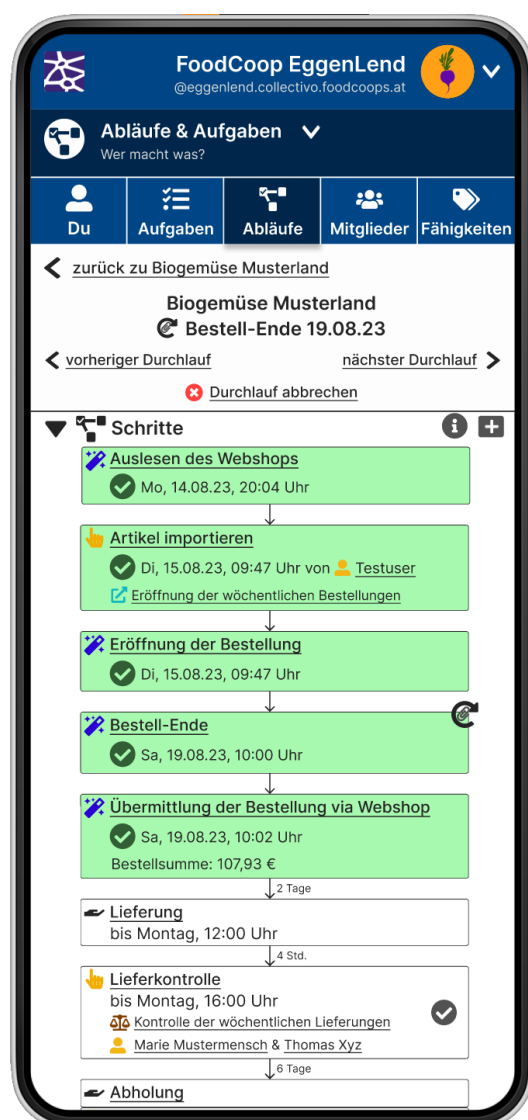


Abbildung 5:
Durchlauf einer beispielhaften Sammelbestellung
(Klickbarer Prototyp aus FoodCoop 4.0)

3.5. Erfahrungen aus der IG FoodCoops

Eine wesentliche Leistung der IG FoodCoops besteht darin, ihren Mitgliedern mehrere Softwares auf ihrem eigens betriebenen Cloud-Server zur Verfügung zu stellen. Dazu gehören unter anderem:

- Instanzen der Bestellsoftware Foodsoft für einzelne FoodCoops sowie für die IG
- Instanzen der Forensoftware Discourse für einzelne FoodCoops sowie für das Vernetzungsforum
- WordPress für Webauftritte der einzelnen FoodCoops sowie foodcoops.at
- Nextcloud

Die IG betreibt die Webseite foodcoops.at, die eine Übersicht über alle FoodCoops in Österreich bietet (auch denen, die keine IG-Mitglieder sind). Obwohl die Daten der Mitglieder-FoodCoops in deren Foodsoft-Instanzen ebenfalls auf dem IG-Server liegen, wird für die Auflistung der FoodCoops auf foodcoops.at eine separate Liste mit Daten der FoodCoops in der IG-Foodsoft gepflegt und Ansprechpartner:innen der FoodCoops regelmäßig kontaktiert.

Zentrales Vereinsorgan der IG FoodCoops ist das Koordinationsteam. Daneben ist vorgesehen, dass sich jeweils eine Arbeitsgruppe um Öffentlichkeitsarbeit, Webseite, EDV-Infrastruktur, Finanzen und die Moderation des Vernetzungsforums kümmert. Tatsächlich gestaltet es sich jedoch schwierig, Mitglieder der FoodCoops zusätzlich für ehrenamtliches Engagement in der IG zu gewinnen. Insbesondere wenn spezielle Fähigkeiten wie Grafikdesign zur Erstellung von Infomaterial für Öffentlichkeitsarbeit oder IT-Kenntnisse für das Betreiben des IG-Servers gefragt sind, ist es schwierig geeignete Personen zu finden. Dies ist darauf zurückzuführen, dass Aufrufe im Vernetzungsforum übersehen werden oder aktive Mitglieder mit ihrem ehrenamtlichen Arbeitsaufwand bereits zu stark ausgelastet sind, um die Aufrufe in ihrer eigenen FoodCoop weiterzuleiten oder geeignete Personen zu vermitteln. Es wäre daher hilfreich, wenn Arbeitsgruppen der IG FoodCoops Einsicht hätten, welche Mitglieder aus einzelnen FoodCoops bestimmte Fähigkeiten haben, um sie direkt anfragen zu können.

3.6. coOps: Umfassendes Softwarekonzept für Gemeinschaften

Ab 2024 arbeitete Leonard Ostler gemeinsam mit Manuel Leitner, einem Softwareentwickler und Vereinskollegen aus KrautKOOPf und dem Freiraum-Verein Spektral (8020 Graz), ein weiterführendes Konzept für eine neue Software für Gemeinschaften aus.

Basierend auf ihren persönlichen, langjährigen Erfahrungen in der Teilnahme und gemeinschaftlichen Mitorganisation in verschiedenen Vereinen und Initiativen sowie den Beobachtungen der FoodCoop 4.0 Vorstudie haben sie detailliert analysiert, welche Funktionalitäten benötigt werden, um sich gemeinschaftlich besser organisieren zu können. Dabei haben sie erkannt, dass sich Anforderungen

aus FoodCoops, Freiräumen und anderen Gruppen wie Wohnprojekten, Gemeinschaftsgärten, genossenschaftlichen Läden oder Vereinen generell stark überschneiden: Sammelbestellungen gibt es etwa nicht nur in FoodCoops, sondern auch z. B. in Wohnprojekten. Ablauf- und Aufgabenkoordination sowie Automatisierung von EDV-Prozessen sind in diversen gemeinschaftlich organisierten Gruppen relevant. Besonderes Augenmerk haben sie auf die Abbildung von Gruppenstrukturen, -rollen und -beziehungen gelegt. Weiters sind die Verwaltung von Geräten, Räumen, Vorräten und Finanzen sowie die Planung von Treffen und Veranstaltungen vorgesehen. Auch digitale Kommunikation soll in das System integriert werden.

Mit coOps soll ein neues, umfassendes, solides System mit optimal verknüpfbaren Elementen (Abläufe, Ressourcen usw.) geschaffen werden, das alle essenziellen Grundbausteine bietet, die in Gemeinschaften benötigt werden. Zusätzlich soll die Anbindung externer, spezialisierter Tools möglich sein. Die Interaktion zwischen auf verschiedenen Servern betriebenen coOps-Instanzen soll ermöglicht werden. coOps soll nicht nur die Organisation innerhalb einer Gemeinschaft, sondern auch Kooperationen zwischen Gruppen abdecken. Für FoodCoops, Produzent:innen und weiteren Akteuren soll ein dezentrales Netzwerk für kooperativen Handel entstehen.

Im Rahmen von coOps wurden u. a. folgende Teilkonzepte entworfen:

- Personen, Haushalte, Produzent:innen, FoodCoops und andere Gemeinschaften werden als gemeinsamer Entitätstyp „Soziale Entität“ behandelt. Diese können auch verschachtelt werden, z. B. eine Arbeitsgruppe als Soziale Sub-Entität in einer Gemeinschaft.
- Mitgliedschaften, Lieferantenbeziehungen oder Eigentumsanteile werden als Beziehungen zwischen Sozialen Entitäten betrachtet. Ähnlich werden Rollen wie Vorstand oder Kassier:in abgebildet.
- An Beziehungen können Rollen geknüpft sein: Z. B. wer ist seitens der FoodCoop Kontaktperson für einen bestimmten Lieferanten, und wer ist Kontaktperson seitens des Lieferanten?
- Musterabläufe werden als „Operation Patterns“ bezeichnet. Diese können beliebig verschachtelt werden, d. h. einzelne Schritte als (Sub-) Operation Patterns enthalten. Schritte können in Abhängigkeit zueinander gestellt werden. Neben manuellen Schritten können auch automatisierte Hintergrundoperationen eingebunden werden.
- Durchläufe und einmalige Abläufe sind „Operations“. Diese können ebenfalls verschachtelt sein und in Abhängigkeit zueinander stehen.
- Musterabläufe können Auftrittsregeln („Occurrence Rules“) haben, die Durchläufe automatisch erstellen. Statt „Wiederholungsregel“ wurde dieser Begriff gewählt, da sie bereits vor dem ersten Durchlauf konfiguriert werden können.

- Abläufe können Eingabefelder enthalten. Ergebnisse, Mitteilungen und Rückfragen können direkt im Ablauf eingegeben werden. Eingaben (inkl. hochgeladene Dateien) können in Hintergrundoperationen automatisiert verarbeitet werden, wahlweise mit benutzerdefiniertem Code oder mittels vordefinierter Funktionen, Schleifen und Fallunterscheidungen („intuitive Programmierumgebung“ ohne Eingabe von Code).

Manuel Leitner und Leonard Ostler möchten ihr Konzept künftig auf der Webseite coOps-app.org präsentieren.

3.7. coOps lite: Funktionaler Prototyp eines Organisationstools für FoodCoops

Da die Bereitstellung von coOps als einsatzbereite Software wegen des enormen geplanten Funktionsumfangs und der hohen nichtfunktionalen Ansprüche erst in mehreren Jahren zu erwarten ist, besteht Bedarf an einer Übergangslösung für FoodCoops. Dabei bietet es sich an, im Rahmen von coOps entworfene Teilkonzepte in vereinfachter Form einzusetzen. Die dabei gewonnenen Erkenntnisse kommen wiederum der Entwicklung von coOps zugute. Unter Einverständnis von Manuel Leitner wurde die in dieser Diplomarbeit zu entwickelnde Software „coOps lite“ genannt. Da sie sowohl als provisorische Lösung für den Produktiveinsatz als auch zur Erforschung von Konzepten dienen soll, wird sie als funktionaler Prototyp angesehen.

3.7.1. Zusammenfassung der Themenstellung

Basierend auf langjährigen persönlichen Erfahrungen eines Teammitglieds in FoodCoops sowie durch Austausch mit Mitgliedern anderer FoodCoops und Durchführung einer Vorstudie wurden im Vorfeld dieser Arbeit folgende Problemfelder identifiziert:

1. Die Abläufe, z. B. aus welchen Schritten eine Sammelbestellung besteht, unterscheiden sich je nach FoodCoop und häufig auch je nach Lieferant:in. Da die bestehenden Bestellsoftwares keine dynamische Abbildung von individuellen Abläufen erlauben, wird der reale Ablauf allenfalls in Textform dokumentiert. Dieser Umstand hat einen hohen manuellen Koordinationsaufwand und Unklarheit über Fortschritte zur Folge.
2. Aufgabenverteilung findet in FoodCoops entweder auf Basis von Eigeninitiative oder durch ehrenamtliche Koordinationsarbeit statt. Dies führt häufig zu einer unfairen Arbeitsverteilung zwischen Mitgliedern. Außerdem kommt es vor, dass wichtige Aufgaben unerledigt bleiben, für die sich niemand zuständig fühlt.
3. Das manuelle Aktualisieren von Artikeldaten in der Bestellsoftware ist zeitaufwändig und fehleranfällig. Lieferant:innen verwenden individuelle Preislistenformate und standardisierte

Schnittstellen setzen sich bislang nicht durch. In manchen FoodCoops führen Mitglieder daher eigens programmierte Skripte auf lokalen Geräten aus. Es fehlt eine Umgebung, in der solche Automatisierungen anderen Mitgliedern zur Verfügung gestellt werden können.

3.7.2. Zielsetzung

Ziel dieser Arbeit ist die Konzeption und prototypische Umsetzung einer modernen Web-App als zentrales Organisationstool für FoodCoops. coOps lite soll ihnen ermöglichen, sich komfortabler und zuverlässiger zu organisieren, den manuellen Aufwand zu senken sowie Aufgaben fairer unter den Mitgliedern zu verteilen.

Es soll eine neue Softwarelösung entwickelt werden, die in Ergänzung zu den bestehenden Softwares betrieben werden soll. Damit sich möglichst viele Mitglieder einer FoodCoop an den Abläufen beteiligen können, ist die Wichtigkeit einer intuitiven, mobilfreundlichen und plattformunabhängigen Benutzeroberfläche besonders hervorzuheben.

coOps lite soll die dynamische Abbildung individueller Abläufe unterstützen. Manuelle und automatisierte Schritte sollen integriert und in Abhängigkeit zueinander gesetzt werden können. Abläufe sollen sich nach definierbaren Regeln automatisch wiederholen können. Die Zuweisung von Aufgaben an Mitglieder soll optional automatisch erfolgen, unter Berücksichtigung von Eignung und Fairness.

4. Umsetzungskonzept

Bevor mit der Programmierung begonnen wurde, erfolgte zunächst die Ausarbeitung eines Umsetzungskonzepts bestehend aus User Stories und einem klickbaren Prototyp mithilfe von Figma (figma.com).

4.1. User Stories und Einordnung der User-Szenarien

4.1.1. Verwendung von User Stories

Die Anforderungen wurden bewusst nicht in einem Pflichtenheft festgelegt, da der Funktionsumfang während der Entwicklung flexibel bleiben sollte. Den wesentlichen fachlichen Input lieferte Leonard Ostler auf Grundlage seiner langjährigen Erfahrung in FoodCoops. Auf dieser Grundlage erstellte er die User Stories und beschrieb die aus seiner Sicht benötigten Funktionen. Clemens Kolbitsch hinterfragte die Anforderungen, präziserte Formulierungen und passte einzelne User Stories gemeinsam mit Leonard Ostler an. Rückmeldungen der IG FoodCoops und der Feedbackgruppe dienten als ergänzende Perspektive.

Die User Stories orientieren sich an folgendem Aufbau:

Als [Rolle] möchte ich [Funktion], um [Nutzen].

Must have User zwischen verschiedenen Organisationen (IG FoodCoops, meinen FoodCoop(s)) hin- und her wechseln können, ohne mich neu einloggen zu müssen.

Abbildung 6: Ausschnitt aus der User-Story-Datei zum Organisationswechsel (US 74)

In dieser Sammlung wurde der Nutzen nicht bei jeder User Story ausdrücklich ergänzt. Er ergibt sich jedoch meist aus dem fachlichen Zusammenhang. Zusätzlich wurden die User Stories Rollen, Menüpunkten und Prioritäten zugeordnet. Als Rollen treten Nutzer:innen, Koordinator:innen und Administrator:innen auf. Die Priorisierung orientiert sich an den Kategorien *Must have*, *Should have*, *Nice to have* und *Not to have* (Abgrenzung). Darüber hinaus wurde festgehalten, ob eine Funktion im Prototyp dargestellt oder bereits umgesetzt wurde. Eine Liste der *Must have* und *Should have* User Stories befindet sich im Anhang 12.1.

Die User Stories beschreiben einen wesentlich größeren Funktionsumfang als im Rahmen der Diplomarbeit tatsächlich umgesetzt wurde. Dazu gehören beispielsweise automatische Aufgabenzuteilungen, konfigurierbare Benachrichtigungen, wiederkehrende Abläufe und automatisierte Arbeitsschritte. Diese Funktionen bleiben teilweise späteren Entwicklungsphasen vorbehalten. Für die Einordnung des Projektergebnisses ist daher zwischen geplanten Anforderungen, Funktionen des klickbaren Prototyps und technisch umgesetzten Funktionen zu unterscheiden.

Die Anforderungen wurden nach den Nutzungssituationen „Mitmachen“ und „Orga“ gegliedert. Diese Einordnung unterscheidet die Mitarbeit an Aufgaben von der Organisation und Verwaltung.

4.1.2. Perspektive „Mitmachen“

Der Bereich „Mitmachen“ richtet sich an Mitglieder, die sich über anstehende Tätigkeiten informieren, Aufgaben übernehmen oder bereits zugewiesene Aufgaben bearbeiten möchten. Im Vordergrund steht nicht die Konfiguration organisatorischer Strukturen, sondern die Beantwortung konkreter Fragen aus dem Alltag:

- Welche Aufgaben sind für mich relevant?
- Wofür bin ich verantwortlich?
- Welche Tätigkeiten sind noch offen?
- Wann ist eine Aufgabe fällig?
- Welche Arbeitsschritte wurden bereits erledigt?
- Was kann ich als Nächstes tun?



Abbildung 7:
Menüpunkt „Mitmachen“

Die zugehörigen User Stories betreffen insbesondere das Anzeigen, Suchen und Filtern von Aufgaben und Abläufen sowie das Bearbeiten einzelner Arbeitsschritte. Die konkrete Gestaltung und technische Umsetzung dieser Funktionen werden im Frontend-Kapitel beschrieben.

Weitere User Stories betreffen Benachrichtigungen über Aufgaben, Zuweisungen und andere relevante Ereignisse. Diese Funktionen sind bisher hauptsächlich im klickbaren Prototyp dargestellt.

4.1.3. Perspektive „Orga“

Der Bereich „Orga“ richtet sich an Personen, die innerhalb eines ausgewählten Bereichs organisatorische Verantwortung übernehmen. Dazu können Administrator:innen einer FoodCoop oder Koordinator:innen einzelner Arbeitsgruppen gehören. Während „Mitmachen“ die Durchführung von Aufgaben unterstützt, dient „Orga“ ihrer Vorbereitung, Strukturierung und Verwaltung.

Ein wichtiger Anwendungsfall ist die Verwaltung von Organisationsbereichen, die in coOps lite als Bereiche (Spaces) abgebildet werden. Ein Bereich kann beispielsweise eine FoodCoop, eine Arbeitsgruppe oder ein anderer organisatorischer Teilbereich sein. Die Verschachtelung solcher Bereiche ermöglicht es, die tatsächliche Struktur einer FoodCoop abzubilden.

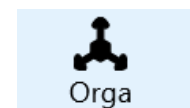


Abbildung 8:
Menüpunkt „Orga“

Berechtigte Personen können vorhandene Bereiche umbenennen und ihnen Typen Sozialer Entitäten zuordnen. So kann beispielsweise unterschieden werden, ob ein Bereich eine FoodCoop, eine Arbeitsgruppe oder eine andere Organisationsform repräsentiert.

Eine weitere Gruppe von User Stories betrifft Fähigkeiten und Einschränkungen. Fähigkeiten beschreiben, welche Voraussetzungen eine Person oder eine andere Soziale Entität für bestimmte Aufgaben erfüllt. Beispiele sind der Besitz eines Schlüssels oder Kenntnisse über eine bestimmte Tätigkeit. Im Orga-Bereich können vorhandene Fähigkeiten und ihre Zuordnungen eingesehen werden. Das Hinzufügen einer Zuordnung über die Oberfläche ist noch nicht umgesetzt.

Auch Aufgaben und Abläufe besitzen eine eigene organisatorische Ansicht. Koordinator:innen sollen nachvollziehen können, welche konkreten Aufgaben und welche wiederverwendbaren Ablaufmuster innerhalb eines Bereichs vorhanden sind. Die bestehende Oberfläche stellt diese bereits dar und bietet teilweise Such- und Filtermöglichkeiten. Weiterführende Funktionen wie das vollständige Erstellen, Duplizieren oder Bearbeiten wiederkehrender Ablaufmuster sind im Anforderungskatalog vorgesehen, aber noch nicht durchgehend umgesetzt.

4.1.4. Zusammenhang zwischen „Mitmachen“ und „Orga“

„Mitmachen“ und „Orga“ bieten unterschiedliche Zugänge zu denselben fachlichen Daten. Während Mitglieder in „Mitmachen“ ihre Aufgaben einsehen und bearbeiten, stehen in „Orga“ Funktionen zur Verwaltung zur Verfügung. Welche Funktionen angezeigt werden, hängt vom aktiven Bereich und den jeweiligen Berechtigungen ab. Eine Person kann daher innerhalb einer Arbeitsgruppe organisatorische Rechte besitzen und in einer anderen Gruppe ausschließlich die Mitmachen-Funktionen verwenden.

4.2. Klickbarer Prototyp

Mithilfe von Figma wurde ein umfangreicher klickbarer Prototyp erstellt. Abbildung 9 zeigt eine Übersicht des Umfangs. Eine Präsentationsansicht ist unter a29.at/7HG8Nd abrufbar.

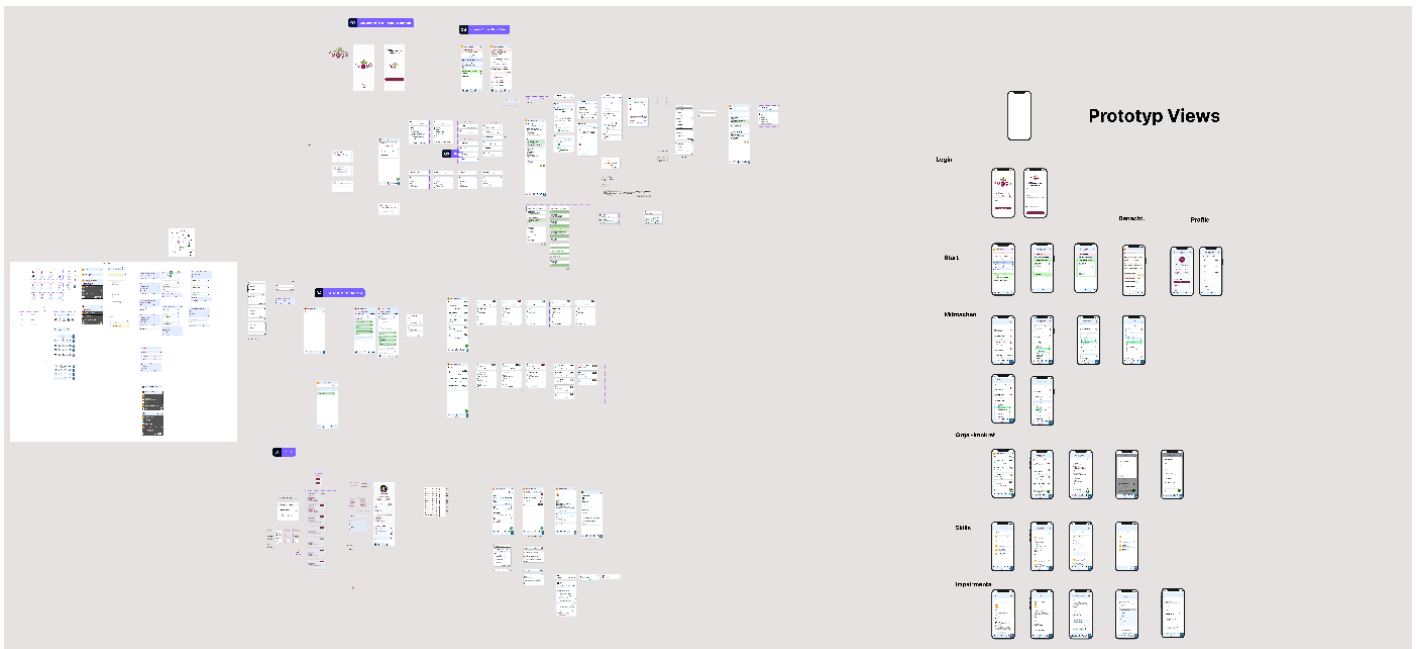


Abbildung 9: Gesamtübersicht der in Figma erstellten UI-Entwürfe und Prototypansichten

4.2.1. Designkonzept

Der Prototyp wurde von dem in der FoodCoop 4.0 Vorstudie (Kapitel 3.4) entworfenen Prototyp inspiriert. Manche Symbole und UI-Elemente wurden davon übernommen.

Im Team wurde ein Logo für coOps lite (Abbildung 10) entworfen. Unter Einverständnis von Manuel Leitner und der IG FoodCoops wurde es sowohl an das coOps-Logo als auch das Logo und die Farben der IG FoodCoops angelehnt.

Das IG-Dunkelrot wurde in die Farbpalette für UI-Elemente übernommen. Außerdem wurde hellblau als primäre Farbe für Menü-Hintergründe hinzugefügt. (Abbildung 11)

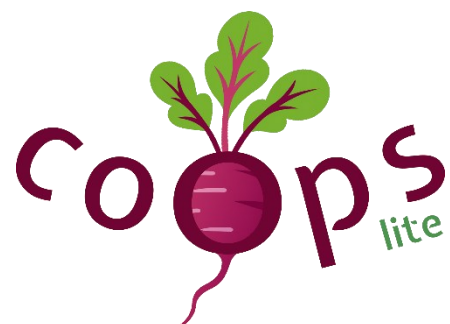


Abbildung 10: coOps lite Logo

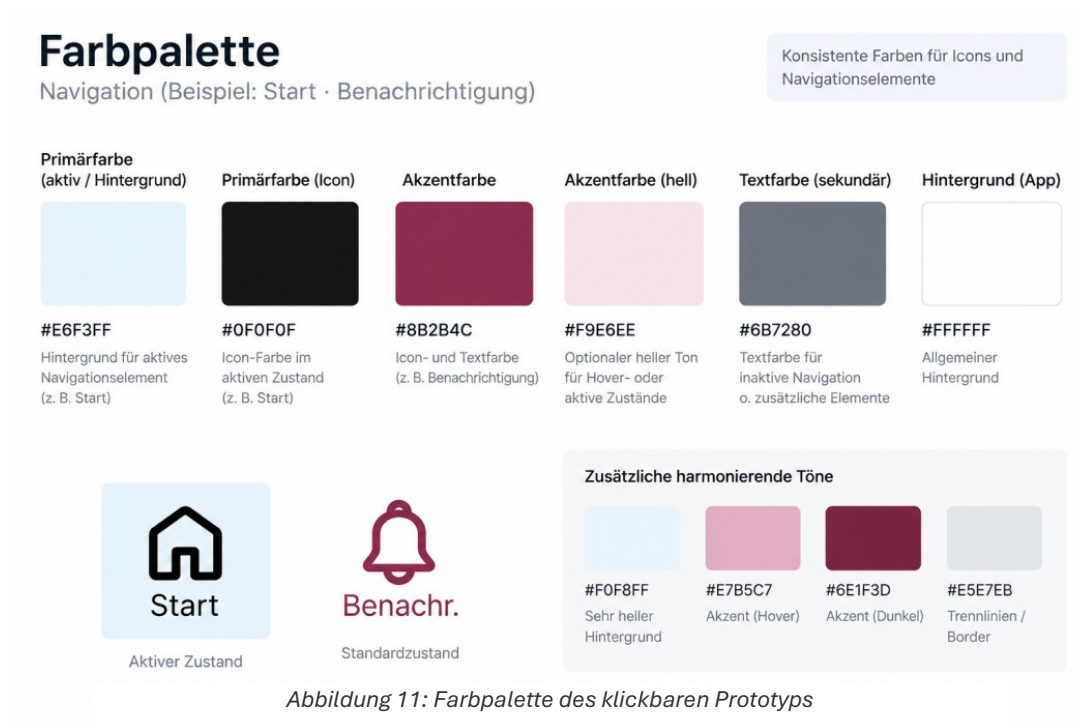


Abbildung 11: Farbpalette des klickbaren Prototyps

4.2.2. Navigation

Zur Navigation innerhalb eines Bereichs wird am unteren Bildschirmrand ein Hauptmenü eingeblendet (Abbildung 12). Hier schlägt sich die Unterteilung in „Mitmachen“ und „Orga“ nieder.



Abbildung 12: Hauptmenü im klickbaren Prototyp

Submenüs wurden für Mitmachen (Abbildung 13) und Orga (Abbildung 14) gestaltet.

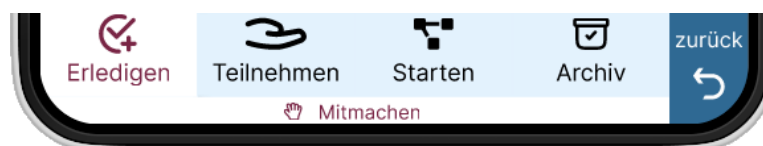


Abbildung 13: Mitmachen-Submenü im klickbaren Prototyp

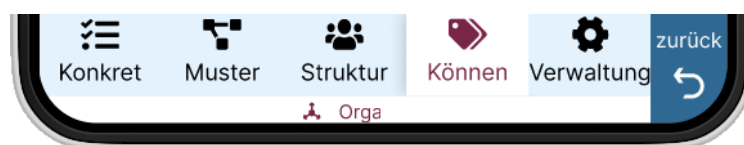


Abbildung 14: Orga-Submenü im klickbaren Prototyp

4.2.3. Startseite

User Story 7

„Als User möchte ich eine Gesamtübersicht über Ankündigungen, Benachrichtigungen und für mich zu erledigende Aufgaben aus allen meinen Organisationen/Spaces sehen können.“

Umsetzung

Start bietet einen Überblick über für die eingeloggte Benutzer:in relevante Informationen:

- Persönliche Benachrichtigungen
- Aufgaben, die sie aktuell erledigen kann
- Aktivitäten und Veranstaltungen, an denen sie teilnehmen könnte

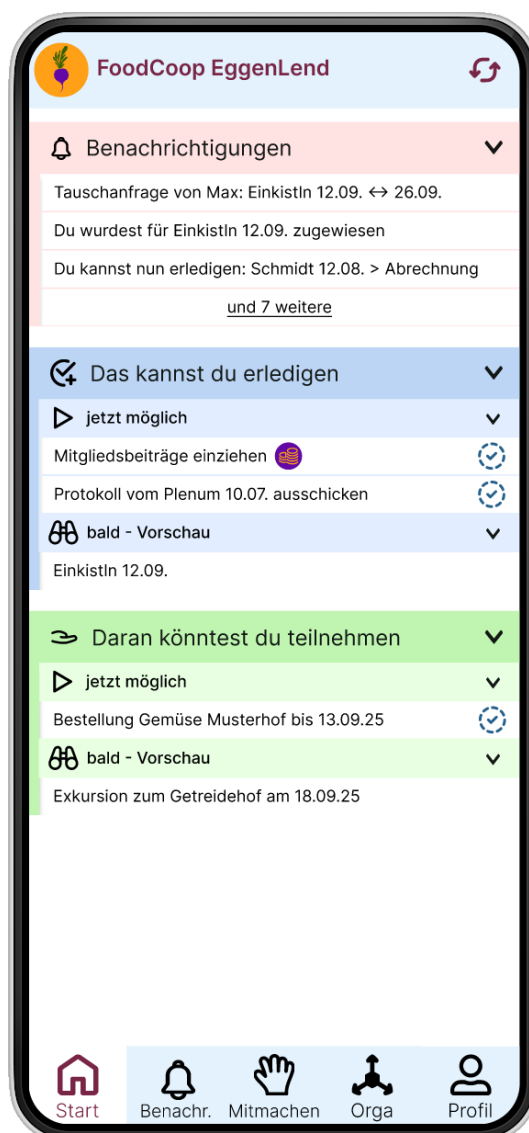


Abbildung 15: Startseite im klickbaren Prototyp

4.2.4. Wechsel des Bereichs

User Story 74

„Als User möchte ich zwischen verschiedenen Organisationen (IG FoodCoops, meinen FoodCoop(s)) hin- und her wechseln können, ohne mich neu einloggen zu müssen.“

Umsetzung

Ein Klick auf die Bereichsanzeige im oberen Menü öffnet die verfügbaren Bereiche. Im Figma-Entwurf ist „FoodCoop EggenLend“ inklusive der Arbeitsgruppe „Finanzteam“ dargestellt. Darunter stehen weitere Organisationen und der persönliche Bereich zur Auswahl. Die angeführten Beziehungen, etwa „Mitglied“ oder „Koordinator:in“, ordnen die Person dem jeweiligen Bereich zu.

Nach der Auswahl eines Eintrags schließt sich das Menü und die Bereichsanzeige übernimmt den neuen Bereich. Die Benutzer:in bleibt dabei angemeldet.



Abbildung 16: Bereichsauswahl im klickbaren Prototyp

Berechtigungen im gewählten Bereich

Dieselbe Benutzer:in kann deshalb in verschiedenen Bereichen unterschiedliche Berechtigungen besitzen. Die Aufnahme zeigt die Bereichsauswahl; ein Berechtigungsunterschied zwischen zwei Bereichen ist darin nicht abgebildet.

4.2.5. Orga – Können

User Story 24

„Als Koordinator:in möchte ich nachschauen können, welche Mitglieder eine bestimmte Fähigkeit haben.“

Umsetzung

Im Figma-Prototyp werden Fähigkeiten und Einschränkungen in separaten Reitern aufgelistet. Als Oberbegriff für das Menü wurde „Können“ gewählt. In der Liste der Fähigkeiten weisen Avatare bzw. eine Zahl auf die Mitglieder hin, die über eine bestimmte Fähigkeit verfügen. Bei ausgeklappten Fähigkeiten werden die Namen der Personen angezeigt. (Abbildung 17)

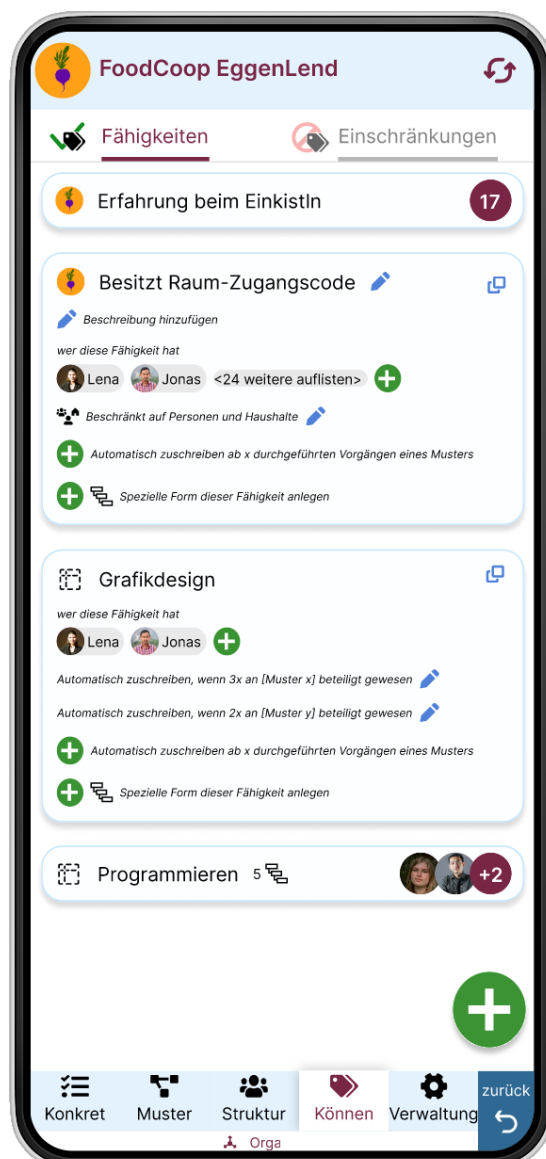


Abbildung 17: Fähigkeiten-Übersicht unter „Orga – Können“ im klickbaren Prototyp

4.2.6. Mustervorgänge

Im Bereich „Orga – Muster“ können Mustervorgänge verwaltet werden. Es kann eingestellt werden, aus welchen Schritten ein Ablauf besteht, wann er wiederkehrt und wie Aufgaben zugeteilt werden sollen. Die folgenden Ansichten zeigen diese Funktionen am Beispiel „Einkistln“.

User Story 31: Schritte eines Musterablaufs

„Als Koordinator:in möchte ich abbilden können, dass eine Musteraufgabe beziehungsweise ein Musterablauf aus mehreren Schritten besteht (mit Drag-and-Drop).“

Das Muster „Einkistln“ besteht aus mehreren Schritten. Zunächst werden die Uhrzeit und das Ausdrucken der Listen vereinbart. Die weiteren Schritte sind jeweils voneinander abhängig. Die Verbindungslinien zeigen die Gliederung des Ablaufs. (Abbildung 18)

Der grün hinterlegte Abschnitt referenziert eine Liste von Schritten aus anderen Abläufen. Um die Schritte zu bearbeiten, kann unterhalb des Ablaufs in einen Bearbeitungsmodus gewechselt werden. So werden unbeabsichtigte Änderungen per Drag-and-Drop vermieden.

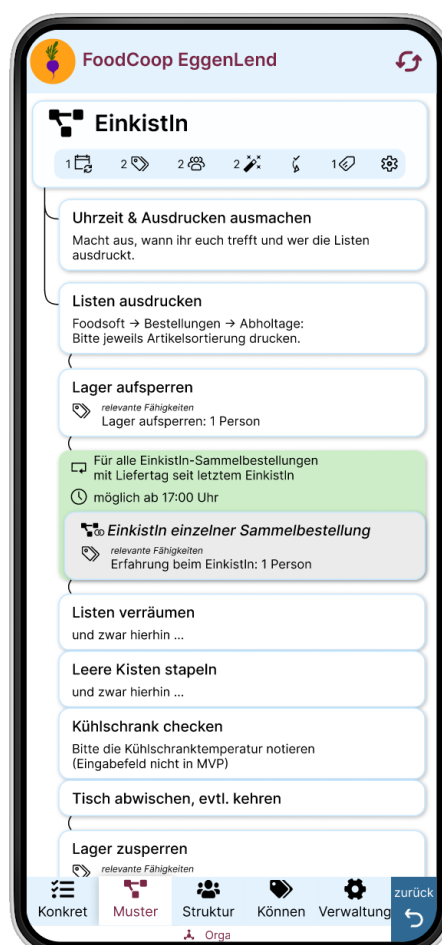


Abbildung 18: Schritte des Musters „Einkistln“ im klickbaren Prototyp

User Story 33: Wann sich ein Ablauf wiederholt

„Als Koordinator:in möchte ich konfigurieren können, wann eine Musteraufgabe beziehungsweise ein Musterablauf auftritt, zum Beispiel jeden zweitletzten Freitag im Monat.“

Für eine Auftrittsregel kann eine Zeitperiode wie Tag, Woche, Monat oder Jahr ausgewählt werden. Je nach Art der Zeitperiode kann ein Wochentag, ein Tag des Monats usw. ausgewählt werden. Es kann eingestellt werden, bis zu welcher Uhrzeit die Aufgabe erledigt werden soll. Die Auftrittsregel kann sich entweder auf den gesamten Ablauf oder einen bestimmten Schritt beziehen.

Außerdem kann konfiguriert werden, wie lang im Voraus die einzelnen Durchläufe angelegt werden und wann damit begonnen werden soll. Ein Ende kann nach einer bestimmten Anzahl von Wiederholungen oder zu einem Datum festgelegt werden. Über das Plus-Symbol lässt sich eine weitere Auftrittsregel ergänzen. (

Abbildung 19)

The screenshot shows a mobile app interface for 'FoodCoop EggenLend'. The main section is titled 'EinkistIn'. Below the title, there are several icons representing different task types. The primary configuration is for a 'Wöchentliches EinkistIn' (Weekly Shopping List). The settings include:

- Alle** (All) with a counter set to **1** and a unit of **Wochen** (Weeks).
- am** (on) set to **Montag** (Monday).
- fällig bis** (valid until) set to **19:30** Uhr.
- bezogen auf Schritt** (related to step) set to **EinkistIn (gesamt)** (Shopping List (total)).
- Anlegen** (Create) with a counter set to **2** and a unit of **Monate** (Months).
- im Voraus** (in advance).
- beginnend** (starting) set to **sofort** (immediately).
- beenden nach** (end after) with a toggle switch and a counter set to **1** **Mal** (times).
- beenden mit** (end with) with a toggle switch and a date field set to **TT.MM.JJJJ**.

At the bottom, there is a green plus icon and the text **weitere Auftrittsregel** (further occurrence rule).

Abbildung 19: Auftrittsregel im klickbaren Prototyp

User Story 36: Automatische Zuweisung

„Als Koordinator:in möchte ich konfigurieren können, dass Iterationen einer Musteraufgabe beziehungsweise eines Musterablaufs (also die konkreten Aufgaben oder Termine) automatisch an geeignete Mitgliederpersonen zugeteilt werden (nach Fähigkeiten und Fairness).“

Die automatische Zuweisung kann für einen Musterablauf aktiviert und deaktiviert werden. Es kann sich entweder um eine fixe Zuweisung oder um eine Zuweisungsanfrage, die das Mitglied annehmen oder ablehnen kann, handeln. Erforderliche Fähigkeiten, die im Ablauf angegeben sind, werden automatisch berücksichtigt.

Es kann eingestellt werden, wie lange im Voraus die Zuweisung erfolgen und ob das zugewiesene Mitglied eine Benachrichtigung erhalten soll.

Da es vorkommen kann, dass Mitglieder Termine tauschen oder versäumen, kann es Mitglieder geben, die etwas „aufzuholen“ haben. Da es aber stressig sein kann, mehrmals direkt hintereinander dran zu sein, ist eine Gewichtung zwischen Aufholen und Rotieren sinnvoll. Dies kann mit einem Regler eingestellt werden. (Abbildung 20)

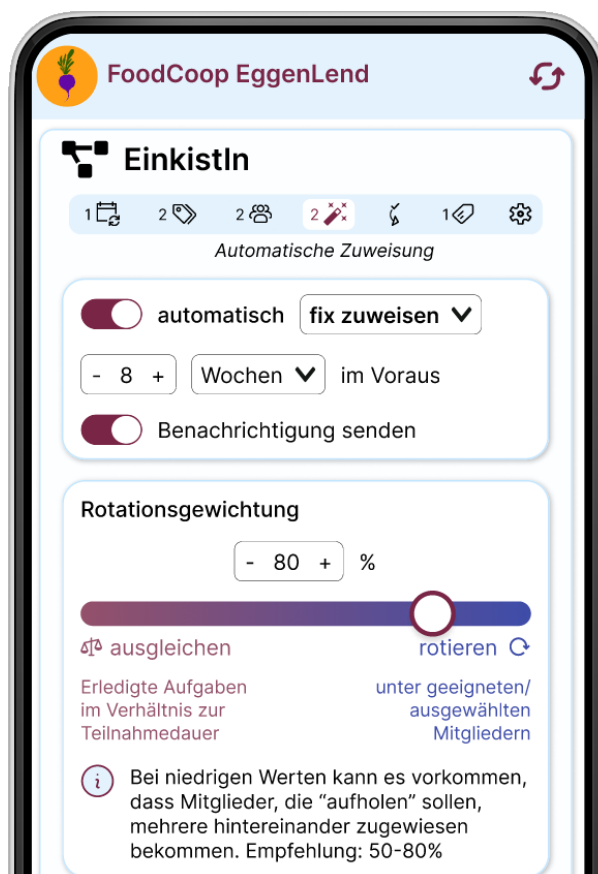


Abbildung 20: Automatische Zuweisung mit Rotationsgewichtung im klickbaren Prototyp

4.2.7. Aufgabe im Detail

User Story 112

„Als Koordinator:in möchte ich die Details einer Aufgabe einsehen können, insbesondere die zugewiesenen Benutzer, Labels und erforderlichen Fähigkeiten, damit ich den Ablauf und die Zuständigkeiten vollständig nachvollziehen kann“.

Die Benutzeroberfläche zeigt die wichtigsten Eigenschaften einer Aufgabe übersichtlich an. Beim Klicken auf die Aufgabe öffnet sich eine Detailansicht. Dort werden zugewiesene Personen, Labels, Anzahl der Schritte und erforderliche Fähigkeiten angezeigt. Dadurch können die Eigenschaften der Aufgabe eingesehen werden, ohne die aktuelle Seite verlassen zu müssen. (Abbildung 21)

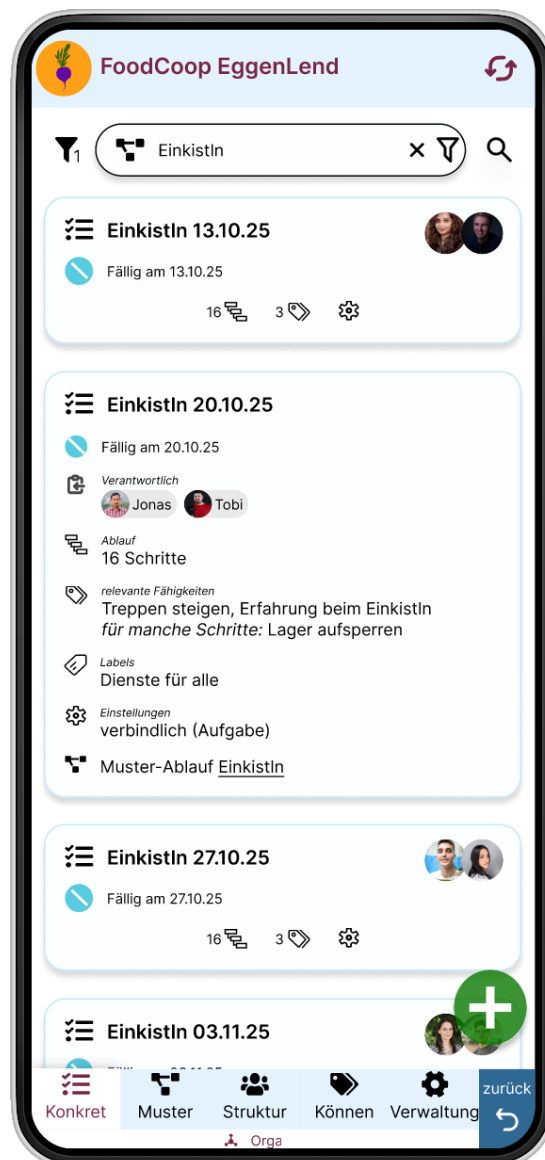


Abbildung 21: Orga – Konkret im klickbaren Prototyp

4.2.8. Bereichs-Stammdaten verwalten

User Story 15

„Als berechtigte Person möchte ich die Stammdaten eines Space einsehen und bearbeiten können“.

Im Orga-Menü unter „Verwaltung“ werden unter anderem Name, Abkürzung, Avatar und übergeordneter Bereich angezeigt. (Abbildung 22) Beim Klicken auf ein Stift-Symbol wird das Feld in den Bearbeitungsmodus versetzt. Diese Möglichkeit steht jedoch nur für Benutzer:innen mit der entsprechenden Berechtigung zur Verfügung. Andere können die Daten nur ansehen und die Bearbeitungsfunktion beziehungsweise der zugehörige Button wird nicht angezeigt. Außerdem können nur Berechtigte den Bereich auflösen.

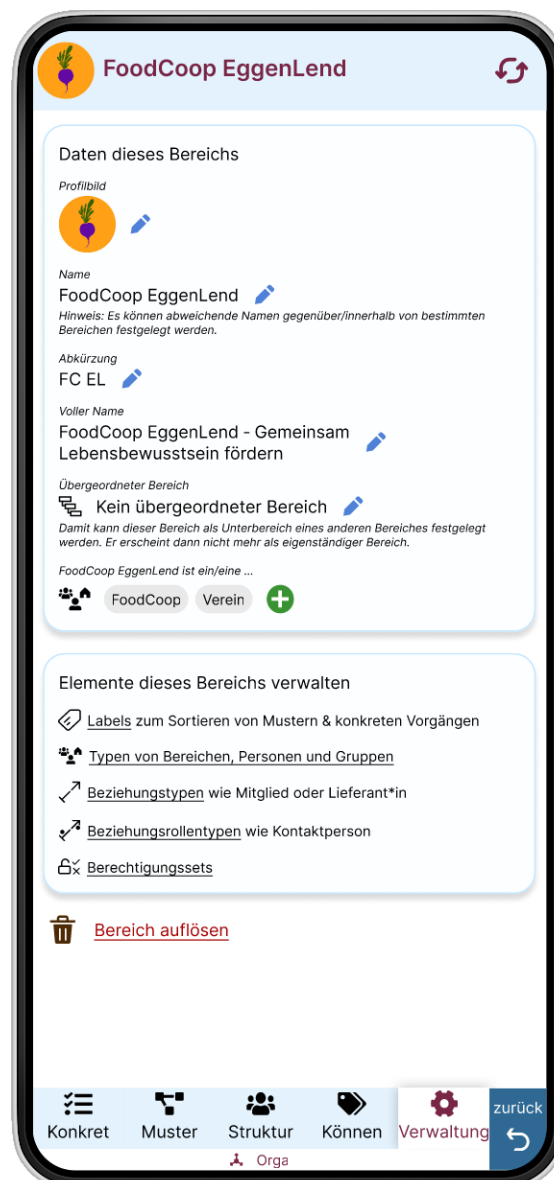


Abbildung 22: Orga – Verwaltung im klickbaren Prototyp

4.3. Einordnung des umgesetzten Umfangs

Im Rahmen der Diplomarbeit wurde ein abgegrenzter Teil der User Stories umgesetzt. Dazu gehören der Wechsel zwischen Bereichen, das Verwalten der Stammdaten eines Bereichs und die Anzeige von Fähigkeiten.

Für Aufgaben und Abläufe wurden Übersichten, Such- und Filtermöglichkeiten entwickelt. Arbeitsschritte können dargestellt, neu angeordnet und in ihrem Bearbeitungsstatus verändert werden. Die Aufgaben- und Ablaufansichten arbeiten mit Beispieldaten; eine dauerhafte Speicherung der dort vorgenommenen Aufgabenänderungen ist noch nicht umgesetzt. Weitere Anforderungen sind bisher im Figma-Prototyp dargestellt oder als Konzept vorgesehen.

4.4. Auswahl grundlegender Technologien für die Umsetzung

Die Auswahl der Technologien für coOps lite erfolgte schrittweise im Verlauf der Projektplanung. Im ersten Konzept war React bereits für das Frontend vorgesehen. Für das Backend wurden Java und TypeScript als Möglichkeiten in Betracht gezogen, während bei der Datenbank PostgreSQL und ein NoSQL-Ansatz zur Diskussion standen. Im weiteren Verlauf wurde zunächst ein PERN-Stack mit PostgreSQL, Express, React und Node.js vorgesehen. Auch NestJS und darauf basierende Projektvorlagen wurden besprochen. Der zusätzliche Einarbeitungsaufwand in Node.js und insbesondere NestJS wurde aber als zu hoch eingeschätzt. Diese Überlegungen sind in den Sitzungsprotokollen des vierten Quartals 2025 dokumentiert.

Anfang 2026 wurde der Wechsel zu Java und Spring Boot für das Backend beschlossen. Ausschlaggebend dafür waren die vorhandenen Kenntnisse des Projektteams aus dem Unterricht. Dort hatten die Teammitglieder bereits mit Java, Spring Boot, React und Vite sowie mit SQL gearbeitet. Mit dem Wechsel wollten wir den zusätzlichen Einarbeitungsaufwand begrenzen und auf vorhandene Unterrichtskenntnisse zurückgreifen. React mit TypeScript blieb als Grundlage des Frontends erhalten. PostgreSQL wurde als relationales Datenbanksystem eingesetzt.

Für die Versionsverwaltung wurde das GitHub-Repository github.com/coOps-lite/coOps-lite verwendet. Die Projektbasis ist in `backend/pom.xml`, `frontend/package.json` und `SecurityConfig.java` nachvollziehbar.

5. Umsetzung des Backend

5.1. Architektur

Das Backend und das Frontend von coOps lite liegen gemeinsam in einem GitHub-Repository, sind jedoch in den Verzeichnissen *backend* und *frontend* voneinander getrennt. Der Backend-Anwendungscode liegt unter *backend/src/main/java/at/foodcoops* und ist nach technischen Verantwortlichkeiten in mehrere Pakete gegliedert. Konfigurationen und SQL-Skripte befinden sich unter *backend/src/main/resources*, während die Tests getrennt unter *backend/src/test* abgelegt sind.

Das Paket *controller* bildet die Schnittstelle zum Frontend. Die Controller nehmen Anfragen entgegen, lesen übermittelte Parameter und Daten aus und rufen die zuständigen Services auf. Anschließend liefern sie das Ergebnis mit einem passenden HTTP-Statuscode zurück. Die Aufteilung orientiert sich an den jeweiligen Funktionsbereichen. Beispielsweise behandelt der *AuthController* Anfragen zur Authentifizierung, der *ManageSpaceController* die Verwaltung von Organisationsbereichen und der *AbilitiesController* Funktionen zu Fähigkeiten und Einschränkungen.

Im Paket *service* befindet sich die Verarbeitung hinter diesen Schnittstellen. Die Services koordinieren Datenbankaufrufe und bereiten deren Ergebnisse für die Rückgabe auf. Die Berechtigungsprüfung und fachliche Verarbeitung ist wesentlich in PostgreSQL-Funktionen und gespeicherten Prozeduren umgesetzt. Services wie *ManageSpaceService* greifen dafür über *Jdbc* auf die Datenbank zu.

Die Entitätstypen sind im Paket *model* abgebildet. Die zugehörigen Klassen beschreiben mithilfe von JPA die Zuordnung zu Datenbanktabellen und deren Beziehungen. Ergänzend enthält das Paket *repository* Schnittstellen auf Basis von Spring Data JPA für den Zugriff auf diese Entitätstypen. Im Backend bestehen damit zwei Zugriffswege nebeneinander: der Zugriff über JPA-Repositories und der direkte Aufruf von SQL-Funktionen beziehungsweise Prozeduren.

Für den Datenaustausch mit dem Frontend werden eigene Anfrage- und Antwortstrukturen verwendet. Diese liegen im Paket *responseToClient* und sind teilweise als Java-Records umgesetzt. Sie legen fest, welche Angaben eine Schnittstelle entgegennimmt und welche Informationen zurückgegeben werden. Dadurch können Antworten gezielt für eine Ansicht zusammengestellt werden, beispielsweise aus den Stammdaten eines Organisationsbereichs und seinen Typzuordnungen.

Übergreifende Einstellungen sind im Paket *config* zusammengefasst. Die dort enthaltene *SecurityConfig* konfiguriert den Zugriffsschutz mit Spring Security und die Prüfung von JSON Web Tokens. Ergänzende Klassen zur Fehlerbehandlung und Hilfsfunktionen befinden sich in den Paketen

exception und *util*. Die Initialisierung der Datenbank wird unter anderem durch *DbInit* gesteuert. Dabei werden SQL-Skripte in festgelegter Reihenfolge geladen, Entity Views erzeugt sowie Referenz- und Entwicklungsdaten initialisiert.

Ein typischer Anfrageablauf lässt sich an der Bearbeitung eines Organisationsbereichs nachvollziehen: Der Controller nimmt die Anfrage entgegen und übergibt sie zusammen mit der Benutzerkennung an den Service. Dieser ruft die zuständige Datenbankprozedur auf. Das Ergebnis wird anschließend in eine Antwortstruktur übertragen und vom Controller an das Frontend zurückgegeben. Die Aufteilung folgt der Schichtenarchitektur. Dabei werden die Verarbeitung von HTTP-Anfragen, die Service-Verarbeitung und der Datenzugriff voneinander getrennt.

5.2. Authentifizierung

In coOps lite können Mitglieder unterschiedlichen FoodCoops und Arbeitsgruppen angehören. Welche Informationen sie einsehen und welche Änderungen sie vornehmen dürfen, hängt vom Organisationsbereich und ihren Berechtigungen ab. Das Backend muss deshalb jede geschützte Anfrage zunächst einer Benutzerkennung zuordnen können. Erst danach lässt sich prüfen, ob die angeforderte Aktion zulässig ist.

Dabei sind Authentifizierung und Autorisierung zu unterscheiden. Die Authentifizierung betrifft den Nachweis der Identität, die Autorisierung die Zugriffsrechte. Ein angemeldetes Mitglied darf beispielsweise nicht automatisch die Stammdaten jeder FoodCoop bearbeiten. Im Backend überprüft Spring Security das übermittelte Token. Fachliche Berechtigungen werden zusätzlich anhand der Datenbank geprüft. Die Verwaltung des Anmeldestatus in React wird im Frontend-Kapitel beschrieben.

5.2.1. Anmeldung im Prototyp

Für den späteren Produktivbetrieb ist ausschließlich eine Anmeldung über Single Sign-on (SSO) in Verbindung mit bereits bestehender FoodCoop-Software vorgesehen. Eine eigene Anmeldung mit Benutzername und Passwort wurde deshalb nicht umgesetzt. Bis zur Anbindung des externen Anmeldedienstes wird ein Mock-Login verwendet, mit dem sich die weiteren Funktionen anhand von Testbenutzern entwickeln und überprüfen lassen.

Das Frontend übermittelt die Kennung eines Testbenutzers an *POST /api/auth/mock-login*. Der *AuthController* nimmt die Anfrage entgegen und ruft den *AuthService* auf. Dieser erzeugt ein Token und lädt die zugehörigen Benutzerinformationen. Das Frontend erhält das Token und die Benutzerinformationen und setzt damit den Anmeldestatus.

Der Mock-Login ersetzt jedoch keinen Identitätsnachweis: Es wird kein Passwort geprüft. Die übermittelte Benutzerkennung bestimmt, für wen das Token ausgestellt wird. Der Ablauf ist daher eine Entwicklungslösung.

Für die Anbindung eines externen Anmeldedienstes ist *POST /api/auth/sso/consume* vorbereitet. Die Verarbeitung des externen Tokens ist hier noch nicht umgesetzt; der Endpunkt antwortet mit *501 Not Implemented*.

Endpunkt	Aufgabe	Stand
<i>POST /api/auth/mock-login</i>	Token und Benutzerinformationen ausgeben	Implementiert
<i>GET /api/auth/user-info</i>	Benutzerinformationen mit geprüftem Token laden	Implementiert und geschützt
<i>POST /api/auth/sso/consume</i>	Externes SSO-Token verarbeiten	Vorbereitet

Tabelle 1: Endpunkte der Authentifizierung

5.2.2. Aufbau und Ausstellung des Tokens

Zur Zuordnung weiterer Anfragen verwendet coOps lite JSON Web Tokens (JWT). Ein JWT überträgt Angaben, sogenannte Claims, zwischen Systemen. Die verwendete signierte Form besteht aus drei durch Punkte getrennten Teilen: Header, Nutzdaten und Signatur. Der Header beschreibt unter anderem das verwendete Verfahren, die Nutzdaten enthalten die Claims. Die Signatur ermöglicht es, Veränderungen zu erkennen. Sie verschlüsselt den Inhalt nicht; die Angaben sind grundsätzlich lesbar. (Jones, Bradley, & Sakimura, 2015, Abschnitte 3 und 5)

Die Methode *AuthService.generateToken()* erstellt das Token mithilfe der Bibliothek JJWT. Sie trägt den Aussteller (*iss*), die Benutzerkennung als Zeichenfolge (*sub*), den Ausstellungszeitpunkt (*iat*) und den Ablaufzeitpunkt (*exp*) ein. Die Bedeutung dieser Claims ist im technischen Standard RFC 7519 für JSON Web Tokens festgelegt. Nach Ablauf darf das Token nicht mehr für einen Zugriff akzeptiert werden. (Jones, Bradley, & Sakimura, 2015, Abschnitt 4.1)

Die Gültigkeitsdauer wird über *jwt.expiration* konfiguriert. Beim Initialisieren prüft *AuthService*, ob sie größer als null ist. Aussteller und geheimer Signaturschlüssel stammen ebenfalls aus der Konfiguration. Zur Prüfung ist in *SecurityConfig* das Verfahren HS256 eingestellt, das einen gemeinsamen geheimen Schlüssel verwendet.

Das Token enthält keine vollständige Liste der Organisationsbereiche und Berechtigungen. Diese Informationen werden getrennt aus der Datenbank geladen. Beim Wechsel zwischen

Organisationsbereichen bleibt die Benutzerkennung unverändert. Ob die Person im ausgewählten Bereich eine Aktion ausführen darf, wird unabhängig davon geprüft.

5.2.3. Prüfung geschützter Anfragen

Nach der Anmeldung übermittelt das Frontend das Token im HTTP-Header in der Form *Authorization: Bearer <Token>*. Bei einem Bearer-Token genügt dessen Besitz für die Verwendung. Deshalb muss es vor unbefugtem Zugriff geschützt und bei der Übertragung über das Netzwerk durch HTTPS abgesichert werden. (Jones & Hardt, 2012, Abschnitte 2.1 und 5)

Spring Security prüft eingehende Anfragen, bevor der zuständige Controller aufgerufen wird. *SecurityFilterChain* legt fest, welche Endpunkte öffentlich erreichbar sind. Dazu gehören die Login-Endpunkte und ausgewählte Schnittstellen der API-Dokumentation. Die übrigen Anfragen benötigen eine Authentifizierung.

Der konfigurierte *JwtDecoder* prüft die Signatur, den Aussteller und die zeitliche Gültigkeit des Tokens. Ein Konverter liest anschließend die Benutzerkennung aus *sub* und wandelt sie in einen *Long*-Wert um. Ist dies nicht möglich, wird die Authentifizierung abgelehnt. (Spring Security)

Der Controller erhält die geprüfte Benutzerkennung über *@AuthenticationPrincipal*. Fehlt das Token oder ist es ungültig, wird eine geschützte Anfrage im regulären Sicherheitsablauf mit *401 Unauthorized* abgelehnt. Die Konfiguration ist zustandslos (*SessionCreationPolicy.STATELESS*): Das Backend speichert den Anmeldestatus nicht in einer HTTP-Sitzung. Jede geschützte Anfrage muss ein gültiges Token vorweisen.

5.2.4. Überprüfung mit Postman

Postman ist ein Werkzeug, mit dem HTTP-Anfragen direkt an das Backend gesendet und dessen Antworten überprüft werden können. Damit lässt sich der Zugriffsschutz unabhängig von der Benutzeroberfläche testen.

Bei Aufruf von *GET /api/auth/user-info* ohne Token lehnt das Backend die Anfrage mit *401 Unauthorized* ab (Abbildung 23). Mit gültigem Token liefert es *200 OK* und die Benutzerinformationen zurück. Der Tokenwert ist in der Aufnahme verdeckt. (Abbildung 24)

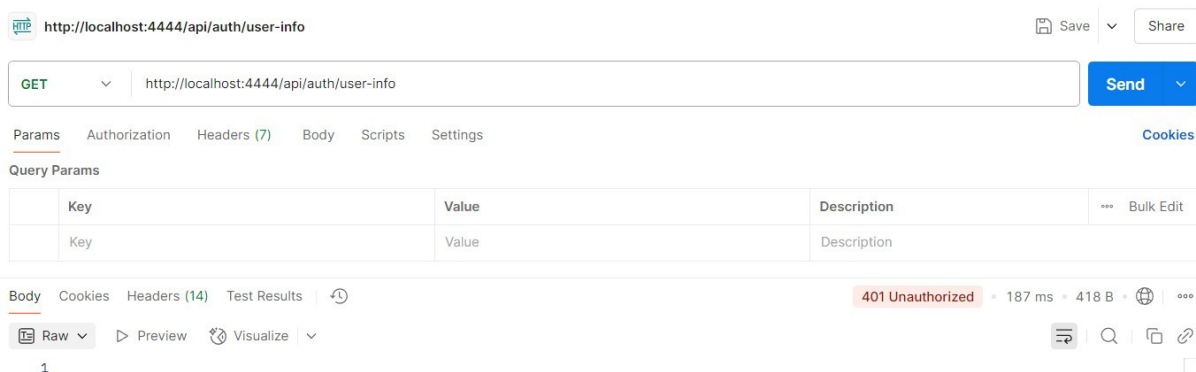


Abbildung 23: Abfrage der Benutzerinformationen ohne Token (Screenshot aus Postman)

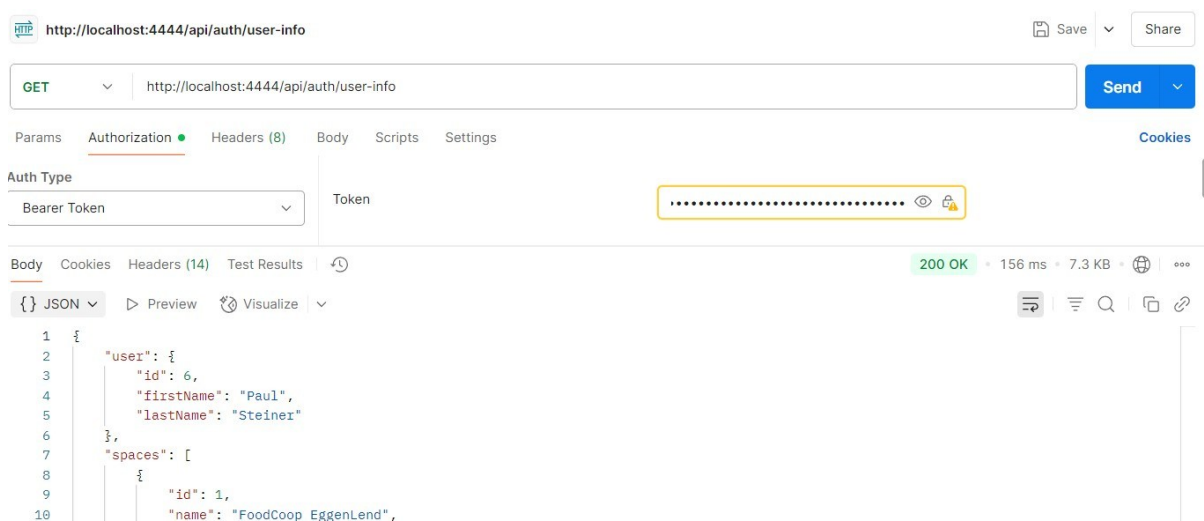


Abbildung 24: Abfrage der Benutzerinformationen mit gültigem Token

5.2.5. Benutzerinformationen und Berechtigungen

Der *AuthService* lädt die Benutzerinformationen über die SQL-Funktion *get_full_user_login_info*. Das Ergebnis umfasst Personendaten, Organisationsbereiche, Soziale Entitäten, Beziehungen und Berechtigungen. Beim erneuten Laden der Anwendung ruft das Frontend mit dem gespeicherten Token *GET /api/auth/user-info* auf und stellt aus der Antwort den Anmeldestatus wieder her.

Die Berechtigungen steuern auch die sichtbaren Menüs und Bearbeitungsmöglichkeiten. Das Ausblenden einer Schaltfläche ersetzt jedoch keine serverseitige Zugriffskontrolle. Eine Anfrage könnte unabhängig von der Oberfläche gesendet werden.

Beim Bearbeiten eines Organisationsbereichs übergibt der *ManageSpaceController* deshalb die geprüfte Benutzerkennung an den Service. Die Datenbankprozedur *edit_manage_space_settings* prüft unter anderem die Berechtigung *edit_social_entity*. Fehlt diese, liefert sie *forbidden*, das

der Controller in *403 Forbidden* übersetzt. Ein gültiges Token erlaubt somit nicht automatisch die gewünschte Änderung.

5.2.6. Erreichter Stand und Grenzen

Im Prototyp sind die Ausstellung signierter Tokens, deren Prüfung bei geschützten Anfragen und das Laden benutzerbezogener Informationen umgesetzt. Es werden unter anderem abgelaufene Tokens, fremde Signaturschlüssel, abweichende Aussteller und Anfragen ohne Token behandelt. Die Postman-Aufnahmen zeigen ergänzend den manuellen Aufruf eines geschützten Endpunkts.

Der Mock-Login ist öffentlich erreichbar und stellt Tokens ohne Passwortprüfung oder externen Identitätsnachweis aus. Vor einem produktiven Betrieb müsste dieser Zugang entfernt oder auf eine abgeschottete Entwicklungsumgebung beschränkt und die vorgesehene SSO-Anmeldung fertiggestellt werden.

Das Frontend speichert das Token derzeit im Local Storage. Beim Abmelden wird es dort gelöscht und der lokale Anmeldestatus zurückgesetzt. Das Backend erhält dabei keinen Auftrag, das ausgestellte Token zu widerrufen. Eine zuvor angefertigte Kopie könnte deshalb bis zum Ablauf weiterhin verwendet werden. Eine automatische Erneuerung über Refresh-Tokens ist im beschriebenen Ablauf nicht vorgesehen.

5.3.1. Soziale Strukturen

Eine wichtige Grundlage für gemeinschaftliches Organisieren ist die digitale Abbildung der Gruppenstruktur. In FoodCoops bedeutet das konkret unter anderem:

- Wer ist derzeit Mitglied?
- Wessen Mitgliedschaft ist derzeit pausiert?
- Welche Arbeitsgruppen gibt es und wer ist darin jeweils aktiv?
- Mit welchen Lieferant:innen wird kooperiert?
- Gibt es weitere Kooperationen, etwa mit anderen FoodCoops oder der IG FoodCoops?
- Wer ist jeweils Kontaktperson gegenüber bestimmten Kooperationspartner:innen?

Ziel ist einerseits, Mitgliedern einen Überblick über die Gemeinschaft zu verschaffen. Andererseits handelt es sich um eine wichtige Datenbasis für die Anknüpfung von Abläufen und Aufgaben.

Soziale Entitäten und Bereiche

Aus coOps stammt das Konzept natürliche Personen, Haushalte, Produzent:innen, FoodCoops und andere Gemeinschaften allesamt als gemeinsamer Entitätstyp „Soziale Entitäten“ (SocialEntity, kurz SE) zu behandeln.

Die Verschachtelung Sozialer Entitäten wurde in coOps lite über das Attribut *parent* umgesetzt. Wenn gesetzt, ist die SE eine Child-Entität der Parent-Entität und existiert nur im Kontext jener Parent-Entität, beispielsweise als Arbeitsgruppe oder Projektteam innerhalb einer Organisation. Bei einer eigenständigen Sozialen Entität hat *parent* den Wert *null*.

Als primäre Selbstbezeichnung einer Sozialen Entität wird das Attribut *name* eingeführt, weiters die optionalen Attribute *abbreviation* und *fullName*. Die „FoodCoop EggenLend“ wird intern beispielsweise häufig als „FC EL“ abgekürzt, während der volle Vereinsname „FoodCoop EggenLend – Gemeinsam Lebensbewusstsein fördern“ lautet.

Jede Soziale Entität bildet einen eigenen Bereich mit den ihr zugeordneten Abläufen und anderen Elementen. In diesem Kontext wird eine Soziale Entität als Bereich (Space) bezeichnet.

Typen Sozialer Entitäten

Zur Einordnung Sozialer Entitäten wird der Entitätstyp „Typ Sozialer Entität“ (SocialEntityType, kurz SET) eingeführt. Er gibt die Art, die Rechtsform oder den Gegenstand einer Sozialen Entität an. Beispiele: Natürliche Person, Haushalt, FoodCoop, Arbeitsgruppe, Produzent:in, Verein, Gesellschaft bürgerlichen Rechts.

Um Typen als Subtypen anderer Typen (beispielsweise Arbeitsgruppe und Projektteam als verschiedene Formen eines Teams) klassifizieren zu können, wird eine Verschachtelung über das Attribut *derivedFrom* ermöglicht.

Es wird eine Liste vordefinierter Typen bereitgestellt. Zusätzlich können Soziale Entitäten eigene („private“) Typen definieren, mit denen sie ihre jeweiligen Child-SEs einordnen können. Dazu kann das Attribut *onlyWithinSocialEntity* gesetzt werden. Wenn *null*, handelt es sich um einen generischen Typ, der in allen Bereichen zur Verfügung steht.

Um die Bezeichnung in der Benutzeroberfläche korrekt im Singular und Plural wiedergeben zu können, werden sowohl *singularTitle* als auch *pluralTitle* als Pflichtattribute definiert. Zusätzlich können optional *singularAbbreviation* und *pluralAbbreviation* gesetzt werden, um Typen platzsparend mit gebräuchlichen Abkürzungen repräsentieren zu können, z. B. FC/FCs für FoodCoop(s).

Weiters kann in *description* eine Beschreibung gespeichert werden, die den Begriff erläutern und bei der Einordnung Sozialer Entitäten hilfreich sein kann.

Typ-Zugehörigkeiten

Eine Soziale Entität ist möglicherweise mehreren Typen zuzuordnen. FoodCoop EggenLend ist beispielsweise sowohl eine FoodCoop als auch ein Verein. Zwischen SocialEntity und SocialEntityType besteht somit eine Many-to-Many-Beziehung. Um zusätzliche Angaben zu einer Zuordnung speichern zu können (beispielsweise die ZVR-Nummer des Vereins), wird das Beziehungsattribut *note* eingeführt und die Beziehung in eine eigene Entität namens SocialEntityTypeAttribution (SETA) aufgelöst.

Beziehungen zwischen Sozialen Entitäten

Die Betrachtung von Mitgliedschaften und Lieferantenbeziehungen als Beziehungen zwischen konkreten Sozialen Entitäten geht ebenfalls auf coOps zurück. Rollen wie Vorstand oder Kassier:in sind systematisch verwandt: „Paul ist Mitglied von FoodCoop EggenLend“, „Paul ist Kassier von FoodCoop EggenLend“. Eine Trennung zwischen Beziehung und Rolle ist nicht immer möglich. In coOps lite werden sie daher gemeinsam als Beziehungen (SocialEntityRelation, kurz SER) aufgefasst.

Um die Richtung einer Beziehung (z. B. *wer* ist Mitglied *bei wem*) abzubilden, wurden die Fremdschlüsselattribute *originSocialEntity* und *targetSocialEntity* definiert. Da eine Soziale Entität (als *origin*) mehrere Beziehungen einer bestimmten Sozialen Entität (als *target*) haben kann – beispielsweise Paul als Mitglied und Kassier:in von FoodCoop EggenLend – wird die gleiche Kombination von *origin* und *target* in mehreren SER-Datensätzen bewusst zugelassen.

Eine Beziehung hat stets einen Anfangszeitpunkt (Attribut *startsAt*). Sie kann außerdem einen Endzeitpunkt haben (*endsAt*). Zur Unterscheidung vorläufiger (geplant bis, aber verlängerbar) und

bestätigter (z. B. beendete Mitgliedschaft) Endzeitpunkte wurde das Boolean-Attribut *endConfirmed* eingeführt.

Details zu einer Beziehung (z. B. Mitgliedsnummer) können durch das Attribut *note* gespeichert werden.

Die Art einer Beziehung (z. B. Mitgliedschaft) ist bei vielen Beziehungen gleich. Anstatt diese Bezeichnung in jeder Beziehung einzeln zu speichern, wurde das auf einen separaten Entitätstyp verweisende Fremdschlüsselattribut *relationType* angelegt.

Beziehungstypen

Typen von Beziehungen zwischen Sozialen Entitäten (*SocialEntityType*, kurz SERT) sind beispielsweise: Mitglied, Ordentliches Mitglied, Lieferant:in, Kassier:in, Bewohner:in.

Dieser Entitätstyp wurde nach dem gleichen Schema wie *SocialEntityType* umgesetzt: Beziehungstypen können voneinander abgeleitet werden (z. B. Ordentliches Mitglied als Subtyp von Mitglied) und entweder generisch oder nur innerhalb eines bestimmten Bereichs existieren (so könnte eine bestimmte FoodCoop „Zahlendes Mitglied“ als spezielle Mitgliedskategorie einführen). Ebenso können Bezeichnung und Abkürzung im Singular und Plural angegeben und durch eine Beschreibung ergänzt werden.

Manche Beziehungstypen machen nur zwischen bestimmten Typen Sozialer Entitäten Sinn. Beispielsweise können nur natürliche Personen Bewohner:innen eines Haushalts sein. Würden beim Anlegen von Beziehungen unpassende Soziale Entitäten (beispielsweise eine FoodCoop als Bewohner:in) auswählbar sein, könnte das bei Nutzer:innen Verwirrung auslösen und zur Speicherung realitätsverletzender Daten führen. Zur Vorbeugung dessen werden die *LimitedToOriginsOfType* und *LimitedToTargetsOfType* hergestellt. So kann sowohl der Ursprung einer Beziehung („wer ist ...“) als auch das Ziel von Beziehungen („... bei wem“) auf Soziale Entitäten bestimmter Typen (inkl. deren Subtypen) beschränkt werden. Existieren in der Beziehung keine Einträge für einen bestimmten Beziehungstyp, gilt für diesen keine Beschränkung.

Beziehungsrollen

In einer FoodCoop ist es üblich, dass unterschiedliche Mitglieder für einzelne Lieferant:innen die Rolle der Kontaktperson seitens der FoodCoop einnehmen. Außerdem verwenden manche FoodCoops ein Buddy-System, bei dem neue Mitglieder jeweils einen „Buddy“ zugewiesen bekommen, der ihnen beim Einstieg in die FoodCoop hilft. Zwischen IG FoodCoops und einzelnen FoodCoops gibt es Ansprechpartner:innen auf beiden Seiten. Mitunter gibt es für die Entrichtung des Mitgliedsbeitrags eine andere Ansprechpartner:in als für die Vernetzung in der Interessengemeinschaft.

Rollen wie Kontaktperson oder Ansprechpartner:in beziehen sich also auf bestimmte Beziehungen und liegen dabei auf einer der beiden Beziehungsseiten, in manchen Fällen sind sie auch beidseitig. Deshalb wurde der Entitätstyp Beziehungsrolle (SocialEntityRelationRole, kurz SERR) genannt.

Das Fremdschlüsselattribut *concerningRelation* stellt die Verknüpfung zur Beziehung her sowie *assignedSocialEntity* die Zuweisung einer Sozialen Entität. Letzteres darf *null* sein, um unbesetzte Beziehungsrollen abbilden zu können. Wenn eine Beziehungsrolle doppelt besetzt wird (z.B. zwei Kontaktpersonen, die sich abwechseln oder gemeinsam auftreten), werden zwei Beziehungsrollen gespeichert.

Die Seitenzugehörigkeit wird über die beiden Boolean-Attribute *onOriginSide* und *onTargetSide* realisiert. *origin* bzw. *target* bezieht sich dabei auf die Richtung der Beziehung. Sind beide *true*, handelt es sich um eine beidseitige Beziehungsrolle. Es ergibt keinen Sinn, wenn beide *false* sind.

Details zu einer Beziehungsrolle können durch das Attribut *note* gespeichert werden.

Eine Beziehungsrolle hat, wie auch Beziehungen, stets einen Anfangszeitpunkt (*startsAt*) und optional einen Endzeitpunkt (*endsAt*). Die Unterscheidung vorläufiger und bestätigter Endzeitpunkte mittels Boolean-Attribut *endConfirmed* wurde übernommen.

Da die Art einer Beziehungsrolle (z. B. Kontaktperson) bei vielen Beziehungsrollen gleich ist, macht es auch hier Sinn die Bezeichnung dieser Art auf einen separaten Entitätstyp auszulagern, der über das Fremdschlüsselattribut *ofType* referenziert wird.

Beziehungsrollentypen

Der Entitätstyp Beziehungsrollentyp (SocialEntityRelationRoleType, kurz SERRT) wurde nach dem gleichen Schema wie SET und SERT umgesetzt: Voneinander ableitbar und entweder generisch oder nur innerhalb eines bestimmten Bereichs. Bezeichnung und Abkürzung können ebenfalls im Singular und Plural angegeben und durch eine Beschreibung ergänzt werden.

Für manche Beziehungsrollentypen kommen nur Sozialen Entitäten bestimmten Typs in Frage. Beispielsweise sollte eine Kontaktperson stets eine natürliche Person sein. Dafür wurde die Many-to-Many-Beziehung *LimitedTowardsTypes* eingeführt. Existieren darin keine Einträge für einen bestimmten Beziehungsrollentyp, gilt für diesen keine Beschränkung.

Abweichende Namen von Sozialen Entitäten in Bereichen

Welche Bezeichnung (Name) für eine Soziale Entität passend ist, hängt von dem Kontext des Bereichs ab: In einer kleinen FoodCoop reichen üblicherweise Vornamen oder Spitznamen; eine Anzeige des vollen Namens wäre unnötig viel Information. Wenn ein Mitglied mit dem gleichen Vornamen beitrifft,

werden die beiden Mitglieder häufig mit dem ersten Buchstaben des Nachnamens oder über unterschiedliche Spitznamen unterschieden. Im Kontext der IG FoodCoops sind volle Namen wichtig, hier wäre es sogar realistisch, dass Kombinationen aus Vor- und Nachnamen mehrfach vorkommen. Im Kontext einer Arbeitsgruppe in der IG FoodCoops genügen wiederum Vornamen.

Die Eigenbezeichnung der Sozialen Entität eignet sich nicht unbedingt. Beispiel: Lena und Miriam nennen ihren Haushalt intern „Unser Nest“. Als Haushalt haben sie eine gemeinsame Mitgliedschaft bei KrautKOOPf. Gegenüber KrautKOOPf nennen sie ihn aber „Lena & Miri“, da eine Auflistung des Mitglieds als „Unser Nest“ wenig aufschlussreich wäre. Benutzer:innen die freie Wahl eines verfügbaren Benutzernamens zu überlassen und jenen als eindeutige Bezeichnung zu verwenden, wäre somit problematisch.

Zunächst wurde in Erwägung gezogen, den abweichenden Namen als optionales Attribut in *SocialEntityRelation* unterzubringen. Da eine Soziale Entität jedoch in mehreren Beziehungen zur selben Sozialen Entität stehen kann, könnten unterschiedliche Namen z. B. für *Paul als Mitglied* und *Paul als Kassier* gespeichert werden. Wenn dargestellt werden soll, dass Paul einer bestimmten Aufgabe im Gruppenbereich zugewiesen ist, ist jedoch nicht unbedingt klar, welche Beziehung hierfür relevant ist.

Als Konsequenz wurde versucht, dieses Problem mit einem Entitätstyp „Beziehungsbasis“ (*SocialEntityRelationBasis*, kurz *SERB*) zu lösen. Die Kombination von *socialEntityA* und *socialEntityB* musste eindeutig sein und auch in umgekehrter Form nicht vorkommen. Dazu konnten *deviatingNameOfSocialEntityA* und *deviatingNameOfSocialEntityB* gespeichert werden. Eine *SocialEntityRelation* referenzierte dann nicht mehr direkt *originSocialEntity* und *targetSocialEntity*, sondern lediglich eine *relationBasis* und gab als *direction* entweder *A_TO_B*, *B_TO_A* oder *MUTUAL* an (Enum). Bei *SocialEntityRelationRole* wurden die Boolean-Attribute *onOriginSide* und *onTargetSide* durch *onSideA* und *onSideB* ersetzt. Dieser Ansatz birgt jedoch unter anderem den Nachteil, dass selbst zum Finden der ausgehenden Beziehungen einer Sozialen Entität in der Tabelle *social_entity_relation_bases* die beiden Spalten *social_entity_a* und *social_entity_b* durchsucht und diese Beziehungsbasen referenzierende Beziehungen anschließend auf die Richtung hin untersucht werden mussten. Um derartige Komplexität zu vermeiden, wurde das Konzept der Beziehungsbasis verworfen.

Um dennoch abweichende Namen speichern zu können, wurde schließlich der Entitätstyp *DeviatingNameBetweenSocialEntities* eingeführt. Die Fremdschlüsselattribute *ofSocialEntity* und *towardsSocialEntity* bilden einen zusammengesetzten Primärschlüssel und stellen somit sicher, dass für eine Kombination höchstens ein abweichender Name existiert. *deviatingName* ist der

Name, mit dem die mittels *ofSocialEntity* verknüpfte Soziale Entität gegenüber der mittels *towardsSocialEntity* verknüpften Sozialen Entität auftritt.

Einhaltung von Konsistenz

Das entworfene logische Modell enthält mehrere zirkuläre Referenzstrukturen zwischen Entitätstypen. Es ist zu untersuchen, wo Inkonsistenzen auftreten könnten.

Mehrere Entitätstypen verwenden reflexive Beziehungen (Selbstreferenzen) mittels *parent* bzw. *derivedFrom*. Dadurch sollen baumartige Strukturen ermöglicht werden. Technisch wären dadurch jedoch auch zirkuläre Bezüge (A referenziert sich selbst als *parent*; A und B referenzieren sich gegenseitig als *parent*; längere Schleifen) möglich, die keine Abbildung als verschachtelte Liste zulassen würden. Zur Einhaltung der Konsistenz ist Azyklizität sicherzustellen. In der Graphentheorie sind die anzustrebenden Strukturen als *gerichtete azyklische Graphen* (directed acyclic graph, DAG) bekannt. Da ein Element nicht mehrere Elemente als *parent* bzw. *derivedFrom* haben kann, ist genauer von *Bäumen* zu sprechen. Zumal mehrere eigenständige Elemente (wo *parent* bzw. *derivedFrom null*) und somit disjunkte Bäume existieren dürfen, handelt es sich insgesamt um einen *Wald*.

Beim Entitätstyp *SocialEntityRelation* handelt es sich um eine aufgelöste, reflexive Many-to-Many-Beziehung von *SocialEntity*. Es handelt sich um einen gerichteten Graph von *origin* zu *target*. Beziehungsgeflechte müssen im API und in der Benutzeroberfläche nicht als verschachtelte Listen darstellbar sein. Das österreichische Vereinsrecht lässt die Mitgliedschaft von juristischen Personen, so auch anderen Vereinen, zu. Gegenseitige Mitgliedschaften sind grundsätzlich erlaubt und zwischen miteinander in Kooperation stehenden Vereinen mitunter üblich. Unternehmen dürfen grundsätzlich wechselseitig beteiligt oder durch Ringbeteiligungen verflochten sein, wobei gesetzliche Einschränkungen gelten. (Jovic, 2019, S. 38-39) Weiters können sich Produzent:innen oder Händler:innen gegenseitig beliefern. Zyklen sind bei Beziehungen zwischen Sozialen Entitäten also grundsätzlich zuzulassen. Die Abbildung einer Selbstbeziehung (*origin = target*) erscheint nicht sinnvoll, aber auch nicht problematisch.

Es bleibt zu untersuchen, inwiefern weitere zirkuläre Referenzen einzuschränken sind:

Dürfen Sozialen Entitäten private Typen (SETs) anderer Bereiche zugewiesen werden, die somit in ihrem Bereich nicht zur Verfügung stehen? Private Typen (SET, SERT und SERRT) einer SE sollen grundsätzlich auch in ihren untergeordneten Bereichen (*children* der SE) verfügbar sein und rekursiv an mögliche weitere Ebenen untergeordneter Bereiche weitergegeben werden. An externe Soziale Entitäten könnten Typzuweisungen (SETAs) nur vergeben werden, wenn Fremdzuweisung erlaubt ist.

Dürfen Beziehungstypen (SERTs) für Beziehungen zwischen Sozialen Entitäten verwendet werden, die nicht in beiden Bereichen zur Verfügung stehen? Ein praktisches Beispiel dafür wäre, dass eine FoodCoop den privaten Beziehungstyp „Zahlendes Mitglied“ erstellt und ihn bestimmten Mitgliedern zuweist. Dies ist also grundsätzlich zuzulassen.

Ob Soziale Entitäten Beziehungsrollen seitens einer Sozialen Entität, mit der sie selbst nicht in Beziehung stehen, einnehmen dürfen, bleibt zu untersuchen.

5.3.2. Authentifizierung und Autorisierung

Es erschien nicht sinnvoll, Anmeldedaten direkt an den Entitätstyp Soziale Entität zu knüpfen. Eine Anmeldung macht nur für natürliche Personen und möglicherweise Bots Sinn. Bei FoodCoops oder Gruppen wären die Authentifizierung betreffende Attribute obsolet. Es ist zwar dienlich, Interaktionen *als* verschiedene Soziale Entitäten durchführen zu können, da eine als Gruppe gemeinsam getroffene Entscheidung (beispielsweise eine Aufgabe anzulegen) nicht einer einzelnen natürlichen Person zuordenbar ist. Es wäre jedoch nicht benutzerfreundlich, dafür eine Abmeldung und erneute Anmeldung unter der betreffenden Sozialen Entität zu erzwingen. Stattdessen ist es erstrebenswert, betreffenden Benutzer:innen das Recht zu verleihen, weitere Soziale Entitäten repräsentieren zu dürfen.

Benutzer:innen

Für die Abbildungen von Zugängen für Benutzer:innen wurde deshalb der Entitätstyp *User* eingeführt. Über das Fremdschlüsselattribut *personalSocialEntity* enthält er eine zwingende One-to-One-Beziehung zu der persönlichen Sozialen Entität (natürliche Person) der Benutzer:in. Außerdem wird zumindest ein Vorname und eine E-Mail-Adresse vorausgesetzt. Weiters kann der Zeitpunkt der Erstellung und der letzten Anmeldung bzw. Aktivität gespeichert werden. Zur Wahrung von Datenschutzrechten ehemaliger Benutzer:innen wird das Attribut *anonymizedAt* vorgesehen, das bei aktiven Usern *null* ist.

Für die Authentifizierung kann ein *passwordHash* gespeichert werden. Es sind jedoch auch Single-Sign-On-Verfahren über bestehende Softwares wie Foodsoft vorgesehen.

Berechtigungssets

Da es in FoodCoops und deren Arbeitsgruppen sehr unterschiedliche Organisationsmodelle gibt und die Einbindung externer Kooperationspartner:innen wie Lieferant:innen zusätzliche Vorsicht bei der Autorisierung erfordert, wurde ein feingliedriges Berechtigungssystem angestrebt. So kann beispielsweise unterschieden werden, ob eine Benutzer:in in einem Bereich auch sämtliche Aufgaben ansehen darf, für die sie nicht zuständig ist (für Mitglieder in einer FoodCoop mit flacher Hierarchie

sinnvoll, bei hierarchischer organisierten Gruppen oder für externe Kooperationspartner:innen jedoch nicht).

Jede Berechtigung für jede Benutzer:in in einem Bereich einzeln vergeben zu müssen würde jedoch einen enormen Verwaltungsaufwand bedeuten. Deshalb wurde der Entitätstyp *Berechtigungsset* (PermissionSet, kurz PS) eingeführt, der mit einem *title* benannt und durch eine *note* erläutert werden kann. Damit nicht jede FoodCoop, Arbeitsgruppe usw. zunächst eigene Berechtigungssets definieren muss, können sie generisch sein und für alle Bereiche zur Verfügung stehen. Zur Markierung „privater“ Berechtigungssets von einzelnen Bereichen steht das Fremdschlüsselattribut *onlyWithinSocialEntity* zur Verfügung, wie bei generischen SETs, SERTs und SERRTs.

Die einzelnen Berechtigungen (permissions) wurden als Boolean-Attribute von PermissionSet abgebildet.

Zuschreibung von Berechtigungssets

Zunächst wurde in Erwägung gezogen, Berechtigungssets direkt einzelnen Benutzer:innen (User) zuzuteilen. Dies würde es jedoch nicht ermöglichen, Aktionen als Gruppe auszuführen, da nicht bekannt wäre, ob die Gruppe diese Aktion ausführen darf. Deshalb wurde eine Zuschreibung von Berechtigungssets an Soziale Entitäten gewählt. Hierfür wurde der Entitätstyp *PermissionSetAttribution* (PSA) eingeführt, der die direkte Zuschreibung eines Berechtigungssets (*ofPermissionSet*) an eine Soziale Entität (*forSocialEntity*) innerhalb eines Bereichs (*withinSocialEntity*) abbildet. Zusätzlich kann eine Notiz (*note*) gespeichert werden.

Dies ermöglicht auch die Zuteilung mehrerer Berechtigungssets an dieselbe Soziale Entität bezogen auf einen Bereich. Wenn eine Berechtigung über mindestens ein zugeteiltes Berechtigungsset gegeben ist (also *true* ist), ist sie als erfüllt zu betrachten.

Da es noch immer recht aufwändig wäre, z. B. jedem Mitglied einer FoodCoop ein Berechtigungsset zuzuschreiben, ist es sinnvoll, neben der direkten Attribuierung weitere Arten der Zuschreibung zu ermöglichen. Dafür dient der Entitätstyp *PermissionSetAttributionByRelationType* (PSABRT). Er stellt die Zuteilung eines Berechtigungssets (*ofPermissionSet*) innerhalb eines Bereichs (*withinSocialEntity*) an alle Soziale Entitäten, die in einer Beziehung bestimmten Typs (*withSocialEntityRelationType*) mit einer bestimmten Sozialen Entität (*relatingSocialEntity*) stehen. Ob es sich dabei um eine eingehende oder ausgehende Beziehung handeln muss, wird durch das Boolean-Attribut *relatingSocialEntityOnTargetSide* definiert. Auch hier kann eine *note* gespeichert werden.

Um auch eine indirekte Attribuierung für z. B. alle, die als Kontaktperson für eine Soziale Entität auftreten, zu ermöglichen, wurde der Entitätstyp *PermissionSetAttributionByRelationRoleType*

(PSABRRT) geplant. Es bestehen jedoch Zweifel, ob diese Attribuierungsart wirklich nützlich und sinnvoll wäre bzw. welche Fremdschlüsselattribute dabei zur Verfügung stehen sollten: Beispielsweise alle Kontaktpersonen (SERRT) gegenüber einer Lieferant:in (SERT), gegenüber einer Produzent:in (SET) oder gegenüber einem von mehreren Typen oder allen dieser Typen. Daher wurde dieser Entitätstyp zwar eingeplant, aber nicht vollends umgesetzt.

Basiszugang

Es gibt bestimmte Aktionen, die allen Benutzer:innen erlaubt sind, die grundsätzlich Zugang zu einem Bereich haben – beispielsweise, dass sie jenen Bereich sehen können sowie die ihnen zugewiesenen Aufgaben abrufen und auch als erledigt markieren können. Es wäre nicht sinnvoll, dafür in PermissionSet Berechtigungsattribute zu definieren, die stets den Wert *true* hätten. Der Basiszugang wird stattdessen dadurch abgebildet, dass es mindestens ein Berechtigungsset attribuiert ist. Dabei muss kein einziges (spezielles) Berechtigungsattribut den Wert *true* haben.

Vererbung von Berechtigungssets zur Vertretung einer Sozialen Entität

Um die Repräsentation Sozialer Entitäten durch andere Soziale Entitäten umzusetzen – beispielsweise dass ein Vorstandsmitglied einer FoodCoop jene FoodCoop gegenüber der IG FoodCoops vertreten darf – wurde beim Entitätstyp Berechtigungsset die spezielle Berechtigung *inheritPermissions* (Boolean-Attribut) ergänzt. Durch die Attribuierung eines Berechtigungssets, bei dem *inheritPermissions* den Wert *true* hat, wird der Sozialen Entität, der es zugeschrieben wird, die Berechtigung verliehen, die *withinSocialEntity* zu repräsentieren, mit all jenen Berechtigungen, die jener *withinSocialEntity* wiederum gegenüber anderen Sozialen Entitäten zugeschrieben sind.

Berechtigungen bei indirekten Beziehungen

Indirekte Beziehungen kommen in der Praxis häufig vor. So wird eine Bewohner:in eines Haushalts, der Mitglied bei einer FoodCoop ist, informell ebenfalls als Mitglied betrachtet und eine Beteiligung an Aufgaben in der FoodCoop wird erwünscht. Eine weitere Ebene liegt vor, wenn die FoodCoop Mitglied bei der IG FoodCoops ist (oder mit ihr in Kooperation steht) und einzelne natürliche Personen in der IG Aufgaben übernehmen können sollen. Der dafür notwendige Zugang könnte mit den bisher definierten Zuschreibungsarten nur mit großem manuellem Aufwand hergestellt werden: In einer FoodCoop müsste für jeden Haushalt, der Mitglied ist, eine PSABRT angelegt werden, das den Mitgliedern jenen Haushalts (wobei *Bewohner:in* ein Subtyp von *Mitglied* sein müsste) Berechtigungen (bzw. einen Basiszugriff) in der FoodCoop erteilt. Weiters müsste in der FoodCoop eine weitere PSABRT für jeden Haushalt angelegt werden, die deren Mitgliedern Zugriff auf die IG FoodCoops erlaubt – und die FoodCoop müsste die Berechtigung haben, Berechtigungen in der IG zu vergeben. Die Vererbung von Berechtigungen wäre hierfür nicht genügend, da indirekte Mitglieder damit nur den Haushalt bzw. die

FoodCoop vertreten dürften und sich nicht als einzelne Person z. B. für eine Aufgabe eintragen könnten, was für die Klarheit in der Aufgabenzuteilung jedoch wichtig ist.

Die Ausbreitung von Berechtigungen soll jedoch nicht für sämtliche in Beziehung stehende Soziale Entitäten erfolgen. So sollen Lieferant:innen einer FoodCoop, die zwecks Einbindung in Bestellabläufe gewisse Berechtigungen im Bereich jener FoodCoop haben, nicht unbedingt einen Zugang zur IG erhalten.

Um Berechtigungen bei indirekten Beziehungen zu realisieren, wurden verschiedene Modelle in Erwägung gezogen:

1. SERs erhalten ein zusätzliches Attribut, das über die Verbreitung von Berechtigungen entscheidet. Es bliebe zu klären: In welche Beziehungsrichtung werden Berechtigungen verbreitet? Müssen beide Beziehungspartner:innen der Verbreitung bzw. dem Erhalt zustimmen? Wird differenziert, welche Berechtigungen / für welche Bereiche verbreitet werden?
2. SERTs erhalten ein Attribut, das über die Verbreitung von Berechtigungen entscheidet. Damit könnte global definiert werden, dass Verbreitung von Berechtigungen etwa bei Mitgliedschaften stattfindet, bei Lieferantenbeziehungen jedoch nicht. Dies wäre eine wenig flexible Lösung. Für mehr Flexibilität könnte dies mit Variante 1 kombiniert werden: SERTs könnten eine Standardeinstellung bieten, die jedoch je einzelner SER überschrieben werden könnte.
3. Die Verbreitung von Berechtigungen wird als Berechtigung (Boolean) in PermissionSet abgebildet: Eine simple Lösung, die jedoch eine Differenzbarkeit, welche Berechtigungen / für welche Bereiche verbreitet würden, ausschließen würde.
4. Die Verbreitung von Berechtigungen wird in der Zuschreibung von Berechtigungen abgebildet. Da es dafür mit PSA, PSABRT und PSABRRT drei verschiedene Entitätstypen gibt und die Verbreitung in allen drei abgebildet werden müsste, erscheint diese Variante wenig elegant.
5. Es wird ein eigener Entitätstyp eingeführt, um die Verbreitung von Berechtigungen abzubilden.

Um die prototypische Umsetzung von coOps lite voranzutreiben, wurde diese Entscheidung zurückgestellt.

Serverweite Berechtigungen und deren Zuschreibung

Es gibt eine Reihe von Interaktionen, die nicht innerhalb eines bestimmten Bereichs stattfinden:

- Erstellung eigenständiger Sozialer Entitäten
- Erstellung von Benutzerzugängen (Users)

- Erstellung und Verwaltung generischer Elemente (generische SETs, SERTs, SERRTs, PermissionSets; weitere in folgenden Kapiteln eingeführt)
- Moderation und Auflösung auf dem Server betriebener Sozialer Entitäten und Benutzerkonten
- Zuschreibung und Entzug von Berechtigungen für genannte Interaktionen

Hierfür wurde der Entitätstyp *ServerwidePermissionSet* (SPS) eingeführt. Wie bei *PermissionSet* kann ein *title* sowie eine *note* vergeben werden und die Berechtigungen werden als Boolean-Attribute abgebildet.

Es wird erwartet, dass serverweite Berechtigungen nur an relativ wenige Soziale Entitäten vergeben werden sollen, beispielsweise an FoodCoops (zum Anlegen von Mitgliedern) und einzelnen Server-Moderator:innen und -Verwalter:innen. Daher wurde zunächst lediglich der Entitätstyp *ServerwidePermissionSetAttribution* (SPSA) für die direkte Zuschreibung von Sets serverweiter Berechtigungen an einzelne Soziale Entitäten eingeplant. Eine spätere Ergänzung z. B. von *ServerwidePermissionSetAttributionByRelationType* (z. B. für alle FoodCoops, die Mitglied bei der IG sind) ist bei Bedarf möglich.

5.3.3. Fähigkeiten und Einschränkungen

Bei Abläufen bzw. einzelnen Schritten kann es beispielsweise erforderlich sein, dass die durchführende Person über Folgendes verfügt:

- Kenntnisse in Grafikdesign
- Erfahrung beim Einkistln („sich auskennen“)
- Zeichnungsberechtigung fürs Bankkonto der FoodCoop
- Besitz eines physischen Schlüssels
- Besitz eines Smartphones
- Fähigkeit Treppen zu steigen
- Fähigkeit schwer zu heben

Es kann sich also um körperliche Fähigkeiten, Kenntnisse, Skills, Erfahrung, Zugangsberechtigungen und Besitz von Gegenständen handeln. Zur Vermeidung unnötiger Komplexität wurden sie im Datenmodell als Entitätstyp „Fähigkeit“ (*Skill*) zusammengefasst.

Es gibt jedoch Fähigkeiten, über die die meisten Personen verfügen und bei denen der Nichtbesitz eine Ausnahme darstellt – beispielsweise Treppen steigen, sehen und hören zu können. Wenn der Besitz solcher Fähigkeiten bei jeder Person, die über sie verfügt, angelegt werden müsste, würde das einen hohen Verwaltungsaufwand auslösen und möglicherweise bei einigen Personen versäumt werden. Außerdem würde die Einblendung solcher Fähigkeiten im Profil jeden Mitglieds, das über sie verfügt, die

Anzeige vieler Informationen bedeuten. Dies könnte dazu führen, dass sowohl der Besitz besonderer Fähigkeiten als auch der Nichtbesitz gewöhnlicher Fähigkeiten in der Benutzeroberfläche leichter übersehen wird. Aus diesen Gründen wurde die Unterscheidung von Fähigkeiten und Einschränkungen (Entitätstyp *Impairment*) vorgenommen: Einschränkungen können etwa (körperliche oder geistige) Behinderungen, Sperren oder der Nichtbesitz gewöhnlicher Gegenstände (z. B. Smartphone) sein.

Zur Bezeichnung haben *Skill* und *Impairment* jeweils das verpflichtende Attribut *title*. Mittels *note* kann eine Erläuterung angefügt werden.

Um den Verwaltungsaufwand je FoodCoop bzw. Sozialer Entität zu verringern, wurde auch hier das Anlegen generischer Elemente, die in jedem Bereich zur Verfügung stehen, geplant. Dafür wurden *Skill* und *Impairment* mit dem Fremdschlüsselattribut *onlyWithinSocialEntity* eingeführt, das nur bei „privaten“ Elementen einzelner Bereiche gesetzt wird. Außerdem wird dadurch ermöglicht, dass der Besitz allgemeiner Fähigkeiten bzw. die Betroffenheit von allgemeinen Einschränkungen nicht in jedem Bereich erneut erfolgen muss. Das ist nützlich, wenn jemand einer weiteren FoodCoop beitrifft / zu dieser wechselt. Weiters können seitens der IG FoodCoops Mitglieder kooperierender FoodCoops mit speziellen Fähigkeiten gefunden und um die Erledigung von Aufgaben angefragt werden. Es ist zu beachten, dass die Angabe des Besitzes generischer Fähigkeiten bzw. der Betroffenheit von generischen Einschränkungen für sämtliche Bereiche möglich ist – jede FoodCoop, jede Person kann diese vergeben. Weiters ist aus Datenschutzsicht zu berücksichtigen, dass diese Angaben sichtbar sind, sobald eine indirekte Beziehung besteht.

Subtypen

Es kann Sinn machen, Fähigkeiten und Einschränkungen als spezielle Ausprägungen bestimmter Fähigkeiten bzw. Einschränkungen zu kennzeichnen. Die Attribuierung eines Subtyps bedeutet dabei indirekt die Attribuierung der übergeordneten Fähigkeit bzw. Einschränkung und kann dadurch Verwaltungsaufwand einsparen. Beispiele dafür sind:

- „Programmieren in Java“ als Subtyp der Fähigkeit „Programmieren“
- „Blind“ als Subtyp der Einschränkung „eingeschränktes Sehvermögen“
- „Einlernen anderer beim Einkistln“ als Subtyp der Fähigkeit „Erfahrung beim Einkistln“

Die Verschachtelung von Subtypen wurde jeweils über das Fremdschlüsselattribut *derivedFrom* (reflexive Many-to-One-Beziehung) realisiert.

Attribuierung

Für die Abbildung des Besitzes einer Fähigkeit wurde der Entitätstyp *SkillPossession* als aufgelöste Many-to-Many-Beziehung zwischen *Skill* und *SocialEntity* definiert. Analog wurde für die

Betroffenheit von einer Einschränkung der Entitätstyp *ImpairmentAffection* ergänzt. Beide erhalten außerdem das Attribut *note*, um Bemerkungen anfügen zu können.

Einschränkungen und Fähigkeiten können auch temporär sein, z. B. wenn ein Gerät defekt oder nur geliehen ist. Um vorzubeugen, dass solche Einschränkungen und Fähigkeiten angelegt werden und es versäumt wird diese danach wieder zu entfernen, wurden *ImpairmentAffection* und *SkillPossession* mit den zeitlichen Attributen *startsAt* (verpflichtend), *endsAt* und *endConfirmed* (beide optional) ausgestattet.

Die Attribuierung zu Sozialen Entitäten lässt grundsätzlich zu, neben natürlichen Personen auch ganzen Arbeitsgruppen oder FoodCoops Fähigkeiten oder Einschränkungen zuzuschreiben. Dafür gibt es auch reale Anwendungszwecke. Bei bestimmten Fähigkeiten und Einschränkungen ist die Attribuierung jedoch ausschließlich für bestimmte Typen Sozialer Entitäten (SETs, beispielsweise natürliche Personen) sinnvoll. Um dies forcieren zu können und unpassende Soziale Entitäten bei der Zuschreibung nicht anzubieten, wurden bei *Skill* und *Impairment* jeweils die Many-to-Many-Beziehung *LimitedToSocialEntityTypes* ergänzt.

5.3.4. Abläufe

Die Umsetzung von Abläufen wurde auf Basis des aus coOps stammenden Konzepts verschachtelter *Operation Patterns*, *Occurrence Rules* und *Operations* gestaltet. Als deutsche Begriffe wurden in coOps lite „Vorgang“ für *Operation* und „Muster“ für *Operation Pattern* definiert. „Auftrittsregel“ für *Occurrence Rule* wurde von coOps übernommen.

Es wurde analysiert, welche verschiedenen Typen von Vorgängen und Mustern es gibt und in welchen Eigenschaften sie sich unterscheiden. Das Datenmodell konnte auf diesem Gebiet nicht fertiggestellt werden.

Aufgaben

Als Aufgabe (*task*) wird ein manueller Vorgang oder Mustervorgang betrachtet, dessen Erledigung primär den Zielen einer Gemeinschaft dient und einen gewissen Grad an Verbindlichkeit aufweist. Beispiele sind:

- Lager putzen
- Einkistln
- Begleichen einer offenen Rechnung

Gelegenheiten

Etwas, das nach Wunsch in Anspruch genommen werden kann und primär persönlichen Zielen dient, wird als Gelegenheit (*opportunity*) bezeichnet. Beispiele sind:

- Mitbestellen bei einer Sammelbestellung
- Anmeldung zur Teilnahme an einer Exkursion (die Teilnahme nach erfolgter Anmeldung könnte hingegen als verbindlich gewertet werden)

Die Grenze zwischen Aufgaben und Gelegenheiten ist in der Realität fließend. Viele Vorgänge haben sowohl einen Nutzen für die Gemeinschaft als auch einen Vorteil für die ausführende Person. Es gibt Vorgänge, die manchen Menschen Spaß machen und für andere unangenehm sind – die Beurteilung eines persönlichen Vorteils ist also häufig subjektiv. Die Einordnung in Aufgaben und Gelegenheiten soll eine übersichtlichere Darstellung in der Benutzeroberfläche ermöglichen, indem sie in separaten Listen aufgeführt werden können. Eine Darstellung von Vorgängen auf einem Spektrum zwischen Aufgabe und Gelegenheit wäre dort nicht praktikabel.

Automatische Aktionen

Ein Vorgang oder Mustervorgang, der vollautomatisch durch Ausführung von benutzerdefiniertem Code erledigt wird, wird als Automatische Aktion (*automatic*) eingeordnet. Beispiel ist das Auslesen des Webshops einer Lieferant:in.

Zur Abbildung teilautomatischer Aktionen – bei der zunächst Eingaben erforderlich sind oder die manuell ausgelöst werden sollen – ist kein eigener Typ erforderlich. Stattdessen werden dafür zwei Schritte angelegt: Eine Aufgabe und eine davon abhängige Automatische Aktion. Beispiel ist das Hochladen einer zugesandten Preisliste, die anschließend automatisiert verarbeitet wird.

Externe Ereignisse

Etwas, das von keiner in coOps lite eingebundenen Sozialen Entität gesteuert werden kann, sondern von Externen oder von höherer Gewalt abhängig ist, wird als Ereignis (*happening*) bezeichnet. Beispiel ist der Erhalt einer Preisliste oder Lieferung von einer Lieferant:in, die keinen Zugang zur Software hat bzw. dort nicht selbst Aufgaben abhaken möchte. In der Business Process Modelling Notation werden solche Schritte als *Events* bezeichnet. Da dieser Begriff jedoch in der Ontologie von coOps bereits für Veranstaltungen vorgesehen ist, wurde ein anderer Begriff gewählt.

Verschachtelung von Vorgängen und Mustern

Die Verschachtelung in Schritte, Unterschritte usw. ist dahingehend einzuschränken, dass Automatische Aktionen nur andere Automatische Aktionen als Schritte haben dürfen. Ebenso dürfen

Ereignisse nur aus Ereignissen als Schritte bestehen. Bei Aufgaben und Gelegenheiten ist keine Einschränkung vorzusehen. Beispielsweise kann durch die Inanspruchnahme einer Gelegenheit die Durchführung einer Automatischen Aktion oder einer Aufgabe (möglicherweise durch jemand anderen) notwendig werden, oder es muss ein Ereignis abgewartet werden.

Muster und deren Iterationen

Vorgänge, die Iterationen (Durchläufe) eines Musters sind, verweisen mit *iterationOfPattern* auf das Muster, auf dem sie basieren. Hat es den Wert *null*, handelt es sich um einmalige Vorgänge.

Zeitfenster zur Durchführung von Vorgängen

Vorgänge müssen häufig nicht zu einem fix vorgegebenen Zeitpunkt, sondern in einem Zeitfenster mit gewisser Flexibilität durchgeführt werden.

Sie können einen frühestmöglichen Zeitpunkt haben – beispielsweise darf die interne Abrechnung einer Sammelbestellung erst nach Ablauf einer Reklamationsfrist für Mitglieder abgeschlossen werden. Viele Vorgänge haben keinen frühestmöglichen Zeitpunkt und können jederzeit durchgeführt werden, dies ist daher optional zu behandeln. Der frühestmögliche Zeitpunkt (*earliestDatetime*) ist für Aufgaben, Gelegenheiten und Automatische Aktionen relevant. Für Ereignisse ist die Relevanz zweifelhaft.

Außerdem können Vorgänge einen angestrebten Zeitpunkt (*targetDatetime*) haben, bis zu dem sie durchgeführt sein sollten. Bei einer Aufgabe lässt sich davon sprechen, bis wann diese fällig ist; bei Gelegenheiten davon, bis wann diese in Anspruch genommen werden kann. Da Automatische Aktionen alsbald möglich vollautomatisch ausgeführt werden, wäre ein *targetDatetime* hier allenfalls für fortgeschrittene Prioritätsbehandlung relevant. Bei Ereignissen liegt eine Relevanz vor, da einstellbar sein soll, bis wann beispielsweise eine Lieferung erwartet wird. Da viele Vorgänge keinen Zielzeitpunkt haben, ist dieser ebenfalls optional zu behandeln.

Es gibt Vorgänge, die bei Überschreitung von *targetDatetime* hinfällig werden und solche, die nachträglich erledigt werden können bzw. sollen. Beispielsweise ist die Vorbereitung einer Vereinsversammlung nach Verstreichen des Termins hinfällig. Die Begleichung einer überfälligen Rechnung sollte jedoch unbedingt nachgeholt werden. Daher ist zwischen einem angestrebten und strikten Zielzeitpunkt zu unterscheiden.

Analog könnte zwischen einem empfohlenen frühesten Zeitpunkt und einem strikten frühestmöglichen Zeitpunkt unterschieden werden.

Abhängigkeiten von Vorgängen

In FoodCoops kommt es häufig vor, dass ein Vorgang vor einem anderen durchgeführt werden muss. Manchmal muss dazwischen eine bestimmte Zeitspanne liegen, beispielsweise sollte eine Sammelbestellung eine gewisse Zeit offen sein, bis das Bestellergebnis abgeschickt werden muss, damit Mitglieder ihre Bestellung eintragen dürfen. Ähnlich müssen Mitglieder, nachdem Ware zur Abholung bereitliegt, eine ausreichende Reklamationsfrist haben, bevor die interne Abrechnung durchgeführt wird.

In der Netzplantechnik werden außerdem Ende-Anfang-Beziehungen (Normalfolge), Ende-Ende-Beziehungen (Endfolge), Anfang-Anfang-Beziehungen (Anfangsfolge) und Anfang-Ende-Beziehungen (Sprungfolge) unterschieden. Zur Begrenzung der Komplexität wird in coOps lite lediglich die Normalfolge abgebildet. Zur feineren Abstimmung von Abhängigkeiten können Vorgänge ohnehin verschachtelt und z. B. der erste Schritt eines Vorgangs in Abhängigkeit zum ersten Schritt eines anderen Vorgangs gestellt werden. So lassen sich beispielsweise auch Anfangsfolgen umsetzen.

Es wird vorgesehen, dass nicht nur Schritte, die beide Teil desselben Vorgangs sind, in Abhängigkeit gestellt werden können, sondern auch Abhängigkeiten zu anderen Vorgängen im gleichen Bereich eingestellt werden können.

Reihen von Vorgängen

Es gibt einerseits Vorgänge, die einmalig bzw. eine feste Anzahl durchzuführen bzw. durchführbar sind, beispielsweise:

- Aufgabe: Eine bestimmte Rechnung muss genau einmal beglichen werden
- Aufgabe: Es gibt fünf Aufbewahrungskübel zu reinigen
- Gelegenheit: Es gibt acht Kürbisse zu verschenken

Um das mehrfache Anlegen derartiger Vorgänge zu ersparen, und gleichzeitig abbilden zu können, wie viele Durchführungen noch übrig sind, wäre ein ganzzahliges Attribut *stack* sinnvoll. Bei Automatischen Aktionen ist der Nutzen einer mehrfachen Durchführung mit den gleichen Werten zweifelhaft, die Einstellmöglichkeit schadet aber nicht. Bei Ereignissen ist Nutzen denkbar.

Andererseits gibt es Vorgänge, die *von jedem* Mitglied (bzw. von anders in Beziehung stehenden Sozialen Entitäten) durchgeführt werden sollen bzw. können:

- Aufgabe: Alle Mitglieder sollen eine interne Umfrage ausfüllen
- Gelegenheit: Alle Mitglieder können an einer bestimmten Sammelbestellung teilnehmen

Es wäre unpassend, wenn derartige Vorgänge nach Durchführung durch ein Mitglied als beendet markiert und den restlichen Mitgliedern nicht mehr angezeigt würden. Das (automatisierte) Anlegen derartiger Reihen als eigenständige Vorgänge für die zum Zeitpunkt der Erstellung bestehenden Mitglieder hätte den Nachteil, dass sie für später beigetretene Mitglieder nicht zur Verfügung stehen würde. Außerdem würden Daten (wie Bezeichnung, Beschreibung und diverse weitere Attribute) redundant gespeichert und unnötig viele Datensätze angelegt werden. Deshalb wird für derartige Reihen eine Many-to-Many-Beziehung *byEachSocialEntityOfRelations* zu *SocialEntityRelation* vorgesehen. Bei Automatischen Aktionen und Ereignissen ist dies nicht relevant, da diese nicht von Sozialen Entitäten durchgeführt werden.

Weiters gibt es Vorgänge, die für jedes Mitglied (bzw. für anders in Beziehung stehenden Sozialen Entitäten) durchgeführt werden sollen bzw. können:

- Aufgabe: Jedes Mitglied soll (von z. B. einer Arbeitsgruppe) mündlich befragt werden
- Automatische Aktion: Für jedes Mitglied soll das Profil in einer externen Software synchronisiert werden
- Ereignis: Die Vereinsbehörde muss alle kürzlich gewählten Vorstandsmitglieder bestätigen

Dafür wird analog eine weitere Many-to-Many-Beziehung *forEachSocialEntityOfRelations* vorgesehen.

Referenzieren anderer Vorgänge

Es soll die Funktionalität umgesetzt werden, dass beispielsweise im wöchentlichen Ablauf „Einkistln“ alle seit dem letzten Einkistln eingetroffenen Lieferungen kontrolliert und aufgeteilt werden sollen, und dabei derjenige Schritt („Einkistln dieser Bestellung“) aus dem Ablauf der jeweiligen Sammelbestellung samt seiner Beschreibung und Unterschriften (was bei Lieferungen jener Lieferant:in zu beachten ist) eingebunden wird.

Dafür werden Many-to-Many-Beziehungen (etwa *ReferenceRule*) geplant:

- zwischen Vorgängen und Mustern, um sämtliche Vorgänge bestimmter Muster referenzieren zu können
- reflexiv zwischen Mustern als Voreinstellung für deren Iterationen

Weiters wird das Referenzieren konkreter Vorgänge mittels *operationReference* (reflexive Many-to-Many-Beziehung von Vorgang) angedacht.

Labels und Tags

Zur Sortierung und Filterung von Vorgängen und Mustern wird der Entitätstyp *Label* definiert. Neben der Bezeichnung (*title*) soll eine Bemerkung (*note*) notiert werden können. Eine Bereichs-übergreifende Sortierung von Vorgängen erscheint nicht sinnvoll, daher werden hier keine generischen Elemente zugelassen, sondern Labels müssen stets zu einem bestimmten Bereich gehören. Um die Erstellung häufig verwendeter Labels in neu erstellten Bereichen zu erleichtern, wird stattdessen der Entitätstyp *SampleLabel* mit *title* und *note* geplant. Sie dienen gleichzeitig als Vorlagen und als Beispiele. Bei der Erstellung von einem auf einem *SampleLabel* basierenden *Label* wird das Label bewusst denormalisiert gespeichert, damit das Label von eventuellen späteren Änderungen am *SampleLabel* unabhängig ist.

Die Markierung von Vorgängen bzw. Mustern mit Labels wird *Tag* genannt und jeweils über eine Many-to-Many-Beziehung vorgesehen. Auf Beziehungsattribute wie die Reihenfolge der Tags wird verzichtet, daher bleibt sie im logischen Modell unaufgelöst. Tags sind für alle Arten von Vorgängen und Muster relevant.

Eingabefelder und Daten

Für Funktionalitäten wie das Hochladen einer einzulesenden Preisliste oder der Dokumentation von Ergebnissen eines Vorgangs werden Eingabefelder (*inputFields*) eingeplant, deren Ausfüllen für die Durchführung eines Vorgangs entweder verpflichtend oder optional ist. Eingabefelder können für Aufgaben und Gelegenheiten relevant sein; bei Automatischen Aktionen dürfen diese nicht direkt enthalten sein, da die Aktion sonst nicht mehr vollautomatisch laufen könnte; bei Ereignissen könnten sie zur Dokumentation von Ergebnissen sinnvoll sein, dies kann jedoch auch als separater Schritt (Aufgabe) betrachtet werden.

Die mittels Eingabefeldern eingegebenen oder durch eine Automatische Aktion generierten Daten werden in *CustomDataFields* gespeichert. Sie treffen auf alle Arten von Vorgängen zu (bei Ereignissen, sofern diese Eingabefelder enthalten können).

Einstellungen zur Ausführung von benutzerdefiniertem Code

Dies trifft nur auf Automatische Aktionen zu. Es müssen eine Ausführungsumgebung, ein auszuführender Code sowie möglicherweise Variablenwerte konfiguriert werden.

Zuteilung von Vorgängen

Die Zuteilung einer Sozialen Entität zur Durchführung eines Vorgangs wird als *Assignment* bezeichnet. Dabei soll unterschieden werden, ob jemand fix zugewiesen oder zunächst angefragt wurde. Weiters soll gespeichert werden, wann und von wem jene Zuweisung bzw. Anfrage vorgenommen wurde und

wann die Anfrage ggf. angenommen oder abgelehnt wurde. Tauschanfragen sollen ebenfalls ermöglicht werden. Außerdem soll erfasst werden, von wem ein Vorgang tatsächlich durchgeführt wurde. Die Dokumentation des Zuteilungsverlaufs mittels *AssignmentAction* wird in Erwägung gezogen, dies wurde aber bislang nicht finalisiert. Damit verbunden ist die Abbildung des Durchführungsstatus (offen, abgeschlossen, storniert).

Da ein Vorgang mehreren Sozialen Entitäten zugewiesen werden können soll, handelt es sich bei *Assignment* um eine aufgelöste Many-to-Many-Beziehung. Zuteilung Sozialer Entitäten ist nur bei Aufgaben (werden erledigt) und Gelegenheiten (werden in Anspruch genommen) sinnvoll, nicht jedoch bei Automatischen Aktionen und Ereignissen.

Automatisierte Zuteilung

Für die automatische Zuteilung ist zu speichern, wie lang im Voraus (bezogen auf *targetDatetime* eines Vorgangs) sie stattfinden soll und ob es sich um eine fixe Zuweisung oder eine Anfrage handeln soll. Außerdem sollte die Rotationsgewichtung einstellbar sein, d. h. wie stark der Abstand zwischen zwei Zuweisungen gegenüber dem Verhältnis zwischen der Anzahl erledigter Aufgaben und der Teilnahmedauer gewichtet wird. In Task-Rotation wurde dies *Zuweisungs-Abstandsfaktor* genannt.

Erforderliche Fähigkeiten

Fähigkeiten können ebenfalls nur bei Aufgaben und Gelegenheiten relevant sein.

Neben der Angabe, dass eine Fähigkeit erforderlich ist, soll angegeben werden können, wie viele der durchführenden Sozialen Entitäten sie mindestens besitzen müssen. Ebenso ist es möglich, dass alle durchführenden SEs sie besitzen müssen und es somit keine bestimmte Anzahl gibt. Hierfür wird der Entitätstyp *SkillRequirement* als Many-to-Many-Beziehung zwischen Vorgang bzw. Muster definiert. In einem Ganzzahl-Attribut kann die Anzahl der nötigen Sozialen Entitäten mit dieser Fähigkeit eingeschränkt werden – ist es *null*, müssen alle die Fähigkeit besitzen.

Kompatible und inkompatible Einschränkungen

Einschränkungen können ebenfalls nur bei Aufgaben und Gelegenheiten relevant sein.

Bei Einschränkungen wird angenommen, dass Aufgaben und Gelegenheiten häufig nicht auf ihre Eignung für Soziale Entitäten mit Einschränkungen beurteilt werden und eine bloße Auflistung entweder der kompatiblen oder der inkompatiblen Einschränkungen daher keine verlässlich vollständige Angabe wäre. Daher wird hierfür die explizite Angabe von kompatiblen sowie von inkompatiblen Einschränkungen vorgesehen. Der Entitätstyp *ImpairmentCompatibilityEvaluation* bildet dies mit dem Boolean-Attribut *compatible* ab. Zusätzlich gibt es wie bei *SkillRequirement* ein Ganzzahl-Attribut

zur Einschränkung der Anzahl der Sozialen Entitäten, für die diese Einschränkung kompatibel bzw. inkompatibel ist.

Automatische Attribuierung von Fähigkeiten

Die automatische Zuschreibung von Fähigkeiten ist bereits aus Task-Rotation bekannt: Z. B. „wenn jemand zweimal an einem erfolgreich abgeschlossenen Einkistln beteiligt war, soll die Fähigkeit „Erfahrung beim Einkistln“ automatisch verliehen werden“. Hierfür wird der Entitätstyp *SkillAcquisitionTrigger* mit Fremdschlüssel auf ein Muster sowie auf eine Fähigkeit sowie einem Ganzzahl-Attribut zur Konfiguration der benötigten Anzahl Teilnahmen definiert.

Eine analoge automatische Zuschreibung von Einschränkungen erscheint nicht sinnvoll.

Auftrittsregeln für Mustervorgänge

Es wird geplant, mehrere Auftrittsregeln für ein Muster konfigurieren zu können. Auf die Wiederverwendbarkeit von Auftrittsregeln in mehreren Mustern wird verzichtet. Somit ergibt sich eine Many-to-One-Beziehung.

In einer Auftrittsregel soll konfiguriert werden können, dass ein Muster sich in einer bestimmten Anzahl von Zeitperioden (Minuten, Stunden, Tage, Wochen, Monate, Jahre) wiederholt, also beispielsweise *jede (1) Woche*. Zusätzlich soll optional ein Ende der Auftrittsregel zu einem bestimmten Zeitpunkt oder nach einer bestimmten Anzahl an Iterationen eingestellt werden können.

Polymorphe Abbildung von Vorgängen und Mustern

Zusammengefasst teilen sich die vier verschiedenen Arten von Vorgängen und Mustern viele Beziehungen zu anderen Entitätstypen, auch wenn nicht alle Beziehungen und Attribute für alle vier Arten anwendbar sind. Eine Abbildung als gemeinsame Entitätstypen *Operation* bzw. *OperationPattern* mit Enum-Attribut *type* bedeutet eine Verletzung der dritten Normalform, da andere Attribute von *type* abhängig den Wert *null* haben müssen und somit transitiv abhängig sind.

Als Alternative wurde die Spezialisierung der vier Arten mittels *is-a*-Beziehungen in Erwägung gezogen:

- *TaskOperation, OpportunityOperation, ... (jeweils) is an Operation*
- *TaskOperationPattern, ... (jeweils) is an OperationPattern*

Dies birgt jedoch die Problematik, dass bei Verwendung herkömmlicher Fremdschlüssel diverse Beziehungen mehrfach abgebildet werden müssten und die Verschachtelung von Vorgängen und Mustern des gleichen Typs nicht mehr einfach als reflexive Beziehungen umsetzbar wären. Die Komplexität würde sich deutlich erhöhen und die Wartbarkeit des Codes verschlechtern.

Vereinfacht werden könnte dies durch die Verwendung polymorpher Beziehungen (Many-to-Any): Auf Subtypen verweisende Fremdschlüssel könnten neben der *id* den Namen der Tabelle als zusätzliches Attribut angeben (polymorphe Fremdschlüssel bzw. Assoziationen). Durch die hohe Anzahl der auf *Operation* bzw. *OperationPattern* verweisenden Entitätstypen würde dies insgesamt viele zusätzliche Attribute nach sich ziehen. Daher wurde diese Variante nicht weiter verfolgt.

Beim tabellarischen Vergleich der Attribute und Beziehungen (siehe Anhang 12.2) fällt auf, dass es mehrere gibt, die nur auf Aufgaben und Gelegenheiten zutreffen:

- SkillRequirements
- ImpairmentCompatibilityEvaluations
- AutoAssignmentConfigurations (für automatisierte Zuteilung)
- byEachSocialEntityOfRelations
- InputFields
- Assignments (nur Vorgänge)
- SkillAcquisitionTriggers (nur Muster)

Ihre Gemeinsamkeit besteht darin, dass sie nur auf *zuweisbare* Vorgänge bzw. Muster anwendbar sind. Es erscheint daher sinnvoll, diese Beziehungen in separate Entitätstypen *AssignableOperation* bzw. *AssignableOperationPattern* auszulagern, die jeweils mittels One-to-One-Beziehungen (*is-a*- oder *in-a*-) an *Operation* bzw. *OperationPattern* geknüpft werden. Diese Erkenntnis wurde erst spät im Projekt erlangt und findet sich noch nicht in der Implementation wieder.

Da sich Einstellungen zur Ausführung von benutzerdefiniertem Code nur auf Automatische Aktionen beziehen, werden hierfür analog *AutomaticOperation* bzw. *AutomaticOperationPattern* geplant.

Bei den verbliebenen transitiven Attributen wird die Verletzung der dritten Normalform in Kauf genommen. Zur Wahrung der Konsistenz ist die Verwendung geeigneter Constraints bzw. Geschäftslogik notwendig.

5.3.5. Benachrichtigungen

Benachrichtigungen sollen aus einer Kurzfassung und bei Bedarf aus einem Beschreibungstext sowie einem Link bestehen. Außerdem sollen sie als gelesen markiert werden können. Der Entitätstyp *Permission* wurde dafür vorgesehen, aber wegen Zeitmangels nicht genauer ausgearbeitet.

5.4. API-Design

Das API für die Kommunikation zwischen Backend und Frontend wurde unter der Maxime erstellt, alle für die Anzeige eines bestimmten Menüs erforderlichen Daten mit einem einzelnen *GET*-Request abrufen zu können. Außerdem sollen die Daten bereits vom Backend möglichst gut für die Benutzeroberfläche aufbereitet werden, um Rechenleistung auf dem Endgerät der Benutzer:in einzusparen, und nur Daten zurückgegeben werden, die die Benutzer:in auch erhalten darf.

Die API-Spezifikation erfolgte mit TypeSpec. Als HTTP-Statuscodes wurden 200/201/204 sowie 400, 401, 403 und 404 verwendet. Eine Dokumentation der API-Spezifikation ist nach Start des Backend unter */swagger-ui/index.html* abrufbar. Nicht alle spezifizierten Endpoints wurden implementiert.

5.4.1. Authentifizierung

Für den Mock-Login mittels *POST /api/auth/mock-Login* muss nur eine *userId* angegeben werden. Zurückgegeben werden soll ein Objekt mit folgenden Informationen:

- Name der Benutzer:in
- Bereiche, auf die die Benutzer:in Zugriff hat (die ihr im Bereichswechsel-Menü angezeigt werden sollen)
 - Basis-Info über den Bereich (Name, Logo)
 - Als *wer* die Benutzer:in im jeweiligen Bereich agieren darf (*asSocialEntities*), jeweils mit *welchen Berechtigungen* und welche Beziehung zum Bereich zugrunde liegt
 - Unterbereiche, jeweils erneut mit diesen Angaben
- Serverweite Berechtigungen: Als *wer* die Benutzer:in *welche Aktionen* durchführen darf

Alle anderen Endpoints stehen nur für eingeloggte Benutzer:innen mit entsprechenden Berechtigungen zur Verfügung.

5.4.2. Grundlegende Daten eines Bereichs

In der Benutzeroberfläche muss an mehreren Stellen abgerufen werden können, welche grundlegende Daten es in einem Bereich gibt:

- Verfügbare Labels (etwa zum Markieren eines Vorgangs oder Musters mit einem Label)
- Verfügbare *SocialEntityTypes*, *SocialEntityRelationTypes*, *SocialEntityRelationRoleTypes* und *PermissionSets* (beim Bearbeiten von Gruppenstrukturen und z. B. Einschränken von Fähigkeiten auf bestimmte SETs)

Dafür wurden jeweils folgende Endpoints geplant: (am Beispiel von *SocialEntityType*, ähnlich für die anderen Datentypen)

- *GET /api/social-entity-types/{socialEntityTypeId}*: Alle verfügbaren Elemente als verschachtelte Liste (Basis-Informationen)
- *GET /api/social-entity-types/detail/space/{spaceId}*: Alle verfügbaren Elemente als verschachtelte Liste mit sämtlichen Eigenschaften
- *GET /api/social-entity-types/{socialEntityTypeId}*: Detaillierte Informationen für ein einzelnes Element
- *POST /api/social-entity-types/{socialEntityTypeId}*: Anlegen eines neuen Elements mit allen erforderlichen Eigenschaften im Request Body
- *PATCH /api/social-entity-types/{socialEntityTypeId}*: Bearbeiten eines Elements; es sind nur die Eigenschaften im Request Body zu übermitteln, die zu ändern sind
- *PATCH /api/social-entity-types/archive/{socialEntityTypeId}*: Archivieren eines Elements

Auf ein echtes Löschen solcher Elemente wurde beim Minimum Viable Product verzichtet, da diese in vielen Fremdschlüsseln verwendet werden und fallweise entschieden werden müsste, ob die Beziehung aufzuheben wäre bzw. ob verknüpfte Elemente zu löschen wären. Da die Verwendung von *DELETE* zum Archivieren nicht der HTTP-Norm entspräche, wurde hierfür eine separate *PATCH*-Route definiert.

5.4.3. Orga – Verwaltung

Mit *GET /api/manage-space/{spaceId}* werden die anzuzeigenden Daten des Bereichs sowie dessen Typ-Zuweisungen (SETAs) abgefragt. Mit *PATCH /api/manage-space/{spaceId}* können Daten des Bereichs geändert werden.

GET /api/manage-space/type-attributions/{typeAttributionId} liefert Details zu einer Typ-Zuweisung. Unter der gleichen Route können mit *PATCH* Details geändert und mit *DELETE* die Typ-Zuweisung aufgehoben werden.

5.4.4. Orga – Können

Durch *GET /api/abilities/space/{spaceId}* wird ein Objekt mit jeweils einer verschachtelten Liste der Fähigkeiten und der Einschränkungen übermittelt, bestehend aus generischen Elementen, Elemente aus übergeordneten Bereichen und Elementen aus dem Bereich selbst. Je Eigenschaft (*Trait* wird hier als Oberbegriff für *Skill* und *Impairment* verwendet) werden Daten wie zugehöriger Bereich und Zuschreibungen mitangegeben.

Durch *POST /api/skills* können Fähigkeiten angelegt, durch *PATCH /api/skills/{skillId}* bearbeitet und durch *PATCH /api/skills/archive/{skillId}* archiviert werden. Dies gilt analog für Einschränkungen.

Für die Zuschreibung von Eigenschaften muss das Frontend zunächst wissen, für welche Soziale Entitäten sie zuschreibbar sind, um diese der Benutzer:in als Optionen anbieten zu können. Für Fähigkeiten lässt sich dies mit *GET /api/skills/possessions/attributable-to-skill/{skillId}* abfragen. Eine Zuschreibung kann mit *POST /api/skills/possessions* angelegt werden. Die Endpoints zum Bearbeiten und Löschen einer Zuschreibung folgen dem etablierten Schema. Für die Zuschreibung von Einschränkungen gibt es ähnliche Endpoints.

Außerdem wurden Endpoints zum Erstellen, Bearbeiten und Löschen von *SkillAttributionTriggers* definiert.

5.4.5. Orga – Struktur

Mit *GET /api/structure/space/{spaceId}* wird für den ersten Aufruf des Menüs eine Standard-Filterung angewandt. Die Filterung (*selection*), zur Verfügung stehende Filter sowie Ergebnisse werden zurückgegeben. Ergänzt durch */filter* und entsprechende Pfadvariablen können Soziale Entitäten abgefragt werden, die etwa in bestimmten Beziehungstypen zum Bereich stehen oder darin bestimmte Berechtigungssets haben.

Details einzelner Sozialer Entitäten können mittels *GET /api/structure/space/{spaceId}/social-entity/{socialEntityId}/relations-and-types* bzw. */abilities*, */permissions-overview*, */permissions-details* und */operations-overview* abgefragt werden.

Weiters wurden Endpoints für folgende Zwecke erstellt:

- Erstellen, Bearbeiten und Löschen von *PermissionSetAttributions* (direkt sowie *by-relation-type*)
- Erstellen, Bearbeiten und Beenden von Beziehungen (*SocialEntityRelations*) sowie deren Pausierungen und Rollen

In dieser Kategorie wäre außerdem ein *POST*-Endpoint zum Erstellen Sozialer Entitäten anzusiedeln.

Endpoints für weitere Menüs (Muster, Konkret und Menüs außerhalb von Orga) wurden aufgrund von Zeitmangel nicht spezifiziert.

5.5. Implementierung des Datenmodells

Mittels JPA-Entitäts wurde ein Großteil der im Logischen Datenmodell entworfenen Entitätstypen umgesetzt. Die objektorientierten Java-Entitäts werden von Hibernate auf PostgreSQL-Relationen abgebildet.

5.5.1. Primärschlüssel

Als *id* wurden jeweils einheitlich fortlaufend vergebene numerische Werte des 8-Byte-Datentyps *Long* verwendet, der bis zu 9 Trillionen Einträge zulässt. (Oracle) In PostgreSQL entspricht das dem Datentyp *bigint*. (PostgreSQL a) Eine Ausnahme bildet die Entity *DeviatingNameBetweenSocialEntities*, die einen zusammengesetzten Primärschlüssel (*SocialEntityPair*) verwendet und dadurch dessen Einzigartigkeit sicherstellt. Many-to-Many-Beziehungen ohne Beziehungsattribute wurden nicht als eigene Entitäts abgebildet, sondern über die *@ManyToMany* Annotation erstellt. In PostgreSQL müssen sie jedoch als eigene Tabellen abgebildet werden. Die Namen der Tabelle und der beiden Spalten werden in *@JoinTable* definiert. (Abbildung 26)

```
@ManyToMany
@JoinTable(
    name = "operation_pattern_tags",
    joinColumns = @JoinColumn(name = "operation_pattern_id"),
    inverseJoinColumns = @JoinColumn(name = "label_id")
)
private Set<Label> tags;
```

Abbildung 26: Many-to-Many-Attribut tags in model.OperationPattern

5.5.2. Constraints

Textattribute, die aus mindestens einem lesbaren Zeichen (kein Leerzeichen) bestehen sollen, wurden mit der Annotation *@NotBlank* versehen, die einen Bean-Validation-Constraint abbildet. Es ist zu berücksichtigen, dass dies keinen Constraint auf Datenbankebene erzeugt.

Auf die Umsetzung weiterer Constraints wurde zunächst verzichtet. An einigen Stellen wurde im Code darauf hingewiesen, dass dies noch zu implementieren ist.

5.5.3. Datum und Uhrzeit

Für kombinierte Datum-Uhrzeit-Werte wurde der Java-Datentyp *ZonedDateTime* verwendet, der eine Zeitzone wie *Europe/Vienna* mitspeichert. Die örtliche Zeitzone wird insbesondere bei Musterabläufen mit Auftrittsregel relevant sein, da in manchen Regionen eine Sommer-/Winter-Zeitungstellung berücksichtigt werden muss. Beispielsweise verwenden Österreich und Tunesien im

Winter beide die Zeitzone UTC+1, aber nur Österreich im Sommer UTC+2. In PostgreSQL gibt es jedoch standardmäßig keinen Datentyp, der diese Information abbildet. JPA mappt *ZonedDateTime* auf den PostgreSQL-Datentyp *timestamp with time zone*, der eine örtliche Zeitzoneinformation zwar annehmen kann, aber nur die punktuelle Abweichung zur UTC-Zeit (z. B. *+01:00*) speichert. (PostgreSQL b) Die örtliche Zeitzone müsste in einem separaten Feld gespeichert werden. Um Datum-Uhrzeit-Werte einfacher vergleichen zu können, wurde in Betracht gezogen, einen eigenen Datentyp (User-defined composite type) anzulegen, der eine örtliche Zeitzoneangabe mitspeichert, und eine Verknüpfung mit *ZonedDateTime* oder einem abgeleiteten Datentyp herzustellen. Dies wurde aufgrund von Zeitmangel bislang nicht umgesetzt.

5.5.4. Entity-Views

Beim Vergleich einer JPA-Entity (Abbildung 27) mit der zugehörigen PostgreSQL-Tabelle (Abbildung 28) fällt auf, dass die Spalten anders sortiert sind. So stehen Textfelder wie *name* ganz rechts, was die Menschenlesbarkeit von Daten zu Entwicklungszwecken erschwert (Scrollen ist nötig um herauszufinden, um welche Soziale Entität es sich handelt).

```
@Entity
@Data
@Table(name = "social_entities")
@NoArgsConstructor
public class SocialEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank
    private String name;

    private String abbreviation;

    private String fullName;

    @ManyToOne
    private SocialEntity parent;
```

Abbildung 27: JPA-Entity *SocialEntity* (Auszug)

	anonymized_at	created_at	id	parent_id
1	<null>	2026-09-07 13:39:...	1	<null>
2	<null>	2026-09-07 13:39:...	2	1
3	<null>	2026-09-07 13:39:...	3	<null>
4	<null>	2026-09-07 13:39:...	4	3
5	<null>	2026-09-07 13:39:...	5	<null>
6	<null>	2026-09-07 13:39:...	6	5

Abbildung 28: PostgreSQL-Tabelle *social_entities* (Auszug)

Hibernate sortiert die Spalten seit Version 6.2 standardmäßig primär nach ihrer Größe in Bytes und sekundär alphabetisch, davor rein alphabetisch. Die Sortierung lässt sich mittels Definition einer eigenen *ColumnOrderingStrategy* ändern, jedoch würde eine Sortierung gemäß der Reihenfolge der in Java definierten Attribute eine erneute Definition der Spalten in der erwünschten Reihenfolge erfordern, was die Wartbarkeit aufgrund von Code-Redundanz verschlechtern würde. Außerdem hat die Sortierung der Spalten Performance-Auswirkungen. (Niestroj, 2023)

Um die Menschenlesbarkeit der Datenbank dennoch zu verbessern, wurden sogenannte Entity-Views eingeführt: Mittels Java Reflection API werden die Entity-Klassen gescannt und für jede JPA-Entity eine PostgreSQL-View erstellt. (Abbildung 29)

	id	name	abbreviation	full_name	parent_id
1	1	FoodCoop EggenLend	EL	FoodCoop EggenLend - G...	<null>
2	2	Leitungskreis			1
3	3	Foodkoop KrautKOOPf	KK	KrautKOOPf - Verein fü...	<null>
4	4	Finanzteam			3
5	5	IG FoodCoops	IG	IG FoodCoops - Interes...	<null>
6	6	Koordinationsteam	KT		5

Abbildung 29: PostgreSQL-View *ev_social_entities*

5.6. Abfragen und Datenbankentwicklung

5.6.1. Technologieauswahl für Datenbankkommunikation

Aufgrund der vielen reflexiven Beziehungen im Datenmodell ist bei Abfragen an vielen Stellen rekursive Logik notwendig. Beispiele hierfür sind:

- Schritte, Unterschritte usw. eines Vorgangs (*Operation*) ermitteln
- Hierarchischen Pfad (übergeordnete Vorgänge) eines Vorgangs ermitteln
- Alle untergeordneten Bereiche eines Bereichs (*Space bzw. SocialEntity*) ermitteln
- Hierarchischen Pfad (übergeordnete Bereiche) eines Bereichs ermitteln

Sollen nun beispielsweise für die Darstellung des Menüs „Orga – Konkret“ alle Vorgänge sowie deren Schritte in einem bestimmten Bereich sowie dessen Unterbereichen abgefragt werden, bieten die verwendeten Java-Frameworks hierfür keine elegante Möglichkeit. Für die Ermittlung aller Unterbereiche gäbe es folgende Varianten:

1. *findById(spaceId)* -> *space.getChildren()* abrufen -> darüber iterieren, rekursiv bis keine *children* gefunden. Das würde $O(n)$ Datenbank-Roundtrips bedeuten.
2. *findById(spaceId)* -> *findAllByParentIdIn(...)* -> pro Ebene wiederholen, bis keine Ergebnisse gefunden. Dadurch würde sich die Anzahl der Datenbank-Roundtrips auf die Anzahl der Ebenen beschränken.
3. *findAll()* -> in Java sortieren. Dabei wäre nur eine einzelne Datenbankabfrage nötig, also $O(1)$, aber die abgerufene Datenmenge und Anzahl der zu erstellenden Java-Objekte wäre deutlich größer.

Beim anschließenden Abrufen der Vorgänge und deren Schritte (ebenfalls als *children* abgebildet) wäre Variante 2 nur anwendbar, wenn die Verschachtelung der Vorgangs-Objekte anschließend mit Java-Logik rekonstruiert wird. Dabei müsste darauf geachtet werden nicht auf „lazy“ nachladende Beziehungen wie *operation.getChildren()* zurückzugreifen. Bei der Ermittlung der jeweils benötigten Fähigkeiten, erfolgten Zuweisungen und weiterer darzustellender Eigenschaften ist es ähnlich.

Andere Abfragen sind mitunter noch komplexer, etwa die Ermittlung der Bereiche, auf die eine Benutzer:in aufgrund von Attribuierungen wie *PermissionSetAttributionByRelationType* Zugriff hat. Dabei muss u. a. rekursiv ermittelt werden, welche übergeordneten Beziehungstypen zutreffen.

Es wurde befürchtet, dass bei manchen Abfragen $O(n)$ oder mehr Datenbank-Roundtrips nötig wären oder die Vermeidung davon so komplexe Java-Logik erfordern würde, dass sowohl Code-Wartbarkeit als auch Performance beeinträchtigt werden könnten.

PostgreSQL bietet indessen mit *WITH RECURSIVE* eine elegante und performante Möglichkeit rekursive Abfragen direkt auf Datenbankebene durchzuführen. Diese sind auch etwa in HQL verfügbar (Hibernate). Allerdings ermöglichen JPA und Hibernate keine komfortable Einbindung von Unterabfragen, mehrfach verwendete Unterabfragen müssten also dupliziert werden. Auch die Nutzung von PostgreSQL-Views in JPA ist kompliziert.

Als Alternative wurde ein Umstieg von JPA auf JOOQ (jooq.org) erwogen, das die Nutzung mehr PostgreSQL-nativer Funktionalitäten ermöglichen würde. Dagegen sprach jedoch der nötige Einarbeitungsaufwand wegen fehlender Erfahrung mit JOOQ, die Umstellung vorhandener Entity-Klassen und die Einschränkungen der Open Source Edition. Eine Bindung an eine kostenpflichtige Edition wäre problematisch.

Schließlich wurde beschlossen, Stored Functions, Views und weitere Elemente nativ über DDL-Skripte anzulegen und Geschäftslogik auf Datenbankebene zu verlagern. Entscheidend waren folgende Vorteile:

- Nur Beschäftigung mit PostgreSQL nötig, statt zusätzlich im ORM-Framework
- Funktionen und Views können sich gegenseitig referenzieren
- Views können später bei Bedarf zu Materialized Views geändert werden, um die Performanz zu verbessern
- Erfahrung mit Stored Functions und Procedures ist bereits vorhanden
- Anzahl der Datenbank-Roundtrips ist besser kontrollierbar und lässt sich auf $O(1)$ reduzieren

Die schlechtere Menschenlesbarkeit ist ein Nachteil komplexer nativer SQL-Abfragen. Jedoch findet Implementierung heutzutage ohnehin häufig mit Hilfe von KI-Agenten statt und die Notwendigkeit menschlicher Wartbarkeit tritt zunehmend in den Hintergrund.

5.6.2. Hierarchie von DDL-Skripten

Da während der Entwicklungsphase laufende Änderungen an DDL-Skripten zu erwarten sind, wurde auf eine Aufzeichnung der vielen Migrationen (etwa mittels Liquibase oder Flyway) verzichtet. Stattdessen werden die DDL-Skripte als Teil der Codebasis behandelt und der Versionsverlauf über Git dokumentiert. Bei jedem Start des Backend werden alle Datenbank-Objekte mit *DROP ... CASCADE* entfernt und neu aufgebaut.

Bei der Ausführung von DDL-Skripten ist die Reihenfolge wichtig, denn wenn z. B. in einer View ein Feld mittels einer Funktion berechnet werden soll, muss die Funktion zuerst erstellt werden. Eine Funktion könnte aber wiederum auf eine andere View zurückgreifen. Views und Funktionen befinden sich somit auf der gleichen Hierarchieebene und es braucht eine Reihenfolge darin. Prozeduren hingegen können nicht in Views oder Funktionen aufgerufen werden, nur umgekehrt. D. h. Prozeduren können generell später erstellt werden.

Um bestimmte Stored Functions und Views in SQL-Formeln von virtuellen Feldern verwenden und duplizierten Code dort vermeiden zu können, wird bei Functions und Views unterschieden, ob diese vor oder nach Erstellung der Entity-Views geladen werden müssen.

Insgesamt ergibt sich folgende Hierarchie:

/0_before_entity_views/

(10_domain_types_etc, brauchen wir aktuell noch nicht)

20_functions+views

Diese werden vor den Entity-Views geladen, können also in *@Formula* verwendet werden.

Custom Operators wären auf der gleichen Ebene.

/2_after_entity_views/

23_functions+views

Diese werden nach Entity-Views geladen und diese verwenden.

30_procedures

40_trigger_functions

50_triggers

Die Hierarchie wurde als Verzeichnisstruktur in *resources/ddl* abgebildet. Die Verzeichnisse und die darin liegenden DDL-Skripte werden im Service *DbInit* gemäß ihrer Nummerierung aufgerufen und ausgeführt. Sollten später weitere Ebenen erforderlich sein, können diese zwischen der bestehenden Nummerierung eingefügt werden.

DDL-Skripte werden nach folgendem Schema benannt: Dreistellige Nummer, Kürzel für den Objekttyp, Name des Objekts. Skripte dürfen nur DDL-Objekte aus Skripten mit niedrigeren Nummern im gleichen Verzeichnis und aus Skripten in Verzeichnissen mit niedrigerer Nummer referenzieren. Als Kürzel werden verwendet: *f* für Function, *v* für View, *mv* für Materialized View, *p* für Procedure, *t* für Trigger.

5.6.3. Abfragesprache und Kennzeichnung von Volatilität

Stored Functions können in PostgreSQL unter anderem in PL/pgSQL und purem SQL geschrieben werden. PL/pgSQL bietet mehr Möglichkeiten, wird jedoch bei jedem Abruf prozedural abgearbeitet und ist deshalb deutlich weniger performant. Außerdem kann PostgreSQL SQL-Funktionen, die andere SQL-Funktionen und Views aufrufen, zu einer einzelnen Abfrage („inlined“) optimieren. (Schönig, 2021)

Obwohl empfohlen wird, Funktionen nicht für Datenbank-Schreibzugriffe zu verwenden, lässt PostgreSQL dies aus historischen Gründen zu. Unter anderem aus diesem Grund werden Funktionen standardmäßig als *VOLATILE* behandelt, d. h. sie werden bei jedem Aufruf erneut ausgeführt. Die Kennzeichnung als *STABLE* kann unnötige zusätzliche Aufrufe verhindern, indem gültige Ergebnisse wiederverwendet werden. Als *IMMUTABLE* sollten hingegen nur Funktionen gekennzeichnet werden, die nicht von veränderbaren Daten oder dem Zeitpunkt der Ausführung abhängig sind. (Belz, 2026) Funktionen ohne Seiteneffekte oder bestimmter zeilenbasierter Sicherheitsimplikationen können zudem als *LEAKPROOF* gekennzeichnet werden, um die Performanz weiter zu verbessern. (Albe, 2026)

Ein Beispiel für ein DDL-Skript ist

`0_before_entity_views/20_functions+views/000f_is_timeframe_current.sql`, das eine simple SQL-Funktion erstellt. (Abbildung 30)

```
DROP FUNCTION IF EXISTS
    is_timeframe_current(TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH TIME ZONE)
    CASCADE;

CREATE OR REPLACE FUNCTION is_timeframe_current
(
    p_starts_at TIMESTAMP WITH TIME ZONE,
    p_ends_at  TIMESTAMP WITH TIME ZONE
)
RETURNS BOOLEAN
LANGUAGE sql
STABLE LEAKPROOF
AS $function$
    SELECT
        (p_ends_at IS NULL OR p_ends_at >= now())
        AND p_starts_at <= now();
$function$;
```

Abbildung 30: Beispiel eines DDL-Skripts

Die SQL-Funktion kann beispielsweise in einem virtuellen Feld eingesetzt werden. (Abbildung 31)

```

@Generated
@Formula("""
    NOT EXISTS (
        SELECT 1
        FROM social_entity_relation_pauses p
        WHERE p.relation_id = id
              AND is_timeframe_current(p.starts_at, p.ends_at)
    )
    AND is_timeframe_current(starts_at, ends_at)
""")
private boolean isCurrentAndActive;

```

Abbildung 31: Virtuelles Feld in SocialEntityRelation.java

Zur weiteren Verbesserung der Performanz sollten Indizes eingeführt werden, bevor coOps lite im Produktivbetrieb eingesetzt wird.

Bei manchen DDL-Skripten wurde PL/pgSQL zur Erstellung von Views oder SQL-Functions eingesetzt, um Berechtigungs-Spalten dynamisch referenzieren zu können, statt diese im Skript aufzuzählen und bei jeder Änderung definierter Berechtigungen in diversen Skripten anpassen zu müssen. Dies wird jedoch nur bei Start des Backend prozedural ausgeführt. Ein Beispiel ist die Funktion `get_permissions_as_array` (Abbildung 32).

```

DO
LANGUAGE plpgsql
$$
BEGIN
EXECUTE '
    CREATE OR REPLACE FUNCTION get_permissions_as_array
    (
        p_user_id BIGINT,
        p_within_social_entity_id BIGINT,
        p_representing_social_entity_id BIGINT
    )
    RETURNS TEXT[]
    LANGUAGE sql
    STABLE LEAKPROOF
    AS $function$

```

Abbildung 32: Dynamische Erstellung einer reinen SQL-Funktion (Auszug) in /23_functions+views/015f_get_permissions_as_array.sql

Bei Verwendung von dynamischem SQL wurde darauf geachtet, dass keine Eingaben von Benutzer:innen in SQL-Statements eingebaut werden, was ein Einfallstor für SQL-Injection wäre.

5.6.4. Implementierung der Endpoints

PostgreSQL bietet die Möglichkeit, Abfrageergebnisse als JSON- oder JSONB-Sets zurückzugeben. Dadurch können die abgefragten Daten direkt in den Datenbankfunktionen in die Struktur gebracht werden, die in der API-Spezifikation definiert und in *responseToClient*-Records umgesetzt ist. Die Umwandlung in Instanzen des jeweiligen Record erfolgt generisch mittels *service.JsonbResultSetExtractor*.

GET-Endpoints verwenden ausschließlich Funktionen und Views und verändern die Datenbasis somit nicht. Bei *POST*-, *PATCH*- und *DELETE*-Endpoints kommen Prozeduren zum Einsatz.

Implementiert wurden die Endpoints der Kategorien *Auth (mock-login)*, *General space data*, *Orga – Verwaltung* und *Orga – Können*.

Beispiel

Im Folgenden wird die Funktion *list_labels_minimal* (Abbildung 33) für den Endpoint *GET /api/labels/space/{spaceId}* erläutert.

Als Parameter werden übergeben, von welcher Benutzer:in die Anfrage stammt und auf welchen Bereich sie sich bezieht. Als welche Soziale Entität die Benutzer:in agiert, spielt bei einer *GET*-Anfrage keine Rolle.

Im CTE-Block *state* wird ermittelt, ob der Bereich existiert und ob die Benutzer:in, von der die Anfrage stammt, darin die notwendigen Berechtigungen besitzt. Da zum Sehen von Labels keine speziellen Berechtigungen erforderlich sind, genügt es, wenn ihr ein beliebiges Berechtigungsset zugeschrieben ist. Da diese Prüfung bei mehreren Endpoints erforderlich ist, wurde die Logik in eine Hilfsfunktion ausgelagert.

Im CTE-Block *result* werden passende HTTP-Status und ggf. Fehlermeldungen vergeben. Ggf. werden die Labels des Bereichs und der übergeordneten Bereiche (in Hilfsfunktion *get_social_entity_ancestor_ids* mit rekursiver Logik ermittelt, Abbildung 34) abgefragt und als JSONB-Liste mit jeweils *id* und *title* verarbeitet. Falls keine Labels gefunden werden, wird statt eines *null*-Wertes eine leere Liste konstruiert.

Schließlich werden die in der *WITH*-Klausel ermittelten Status, Daten und eventuelle Fehlermeldung zu einem JSONB-Objekt zusammengesetzt und zurückgeliefert.

```

WITH state AS (
  SELECT
    EXISTS(SELECT 1 FROM social_entities se WHERE se.id = p_space_id) AS space_exists,
    COALESCE(user_has_basic_view_access_within_SE(p_user_id, p_space_id), FALSE) AS has_access
),
result AS (
  SELECT
    CASE
      WHEN NOT s.space_exists THEN 'not_found'
      WHEN NOT s.has_access THEN 'forbidden'
      ELSE 'ok'
    END AS status,
    CASE
      WHEN NOT s.space_exists THEN 'Space not found'
      WHEN NOT s.has_access THEN 'Missing permission to view this space'
      ELSE NULL
    END AS message,
    CASE WHEN s.space_exists AND s.has_access THEN (
      SELECT COALESCE(
        JSONB_AGG(
          JSONB_BUILD_OBJECT('id', l.id, 'title', l.title)
          ORDER BY l.id
        ),
        '[]'::jsonb
      )
      FROM labels l
      WHERE l.belongs_to_social_entity_id IN (SELECT id FROM get_social_entity_ancestor_ids(p_space_id))
        AND l.archived_at IS NULL
    ) ELSE NULL END AS data
  FROM state s
)
SELECT JSONB_BUILD_OBJECT(
  'status', r.status,
  'data', CASE WHEN r.status = 'ok' THEN COALESCE(r.data, '[]'::jsonb) ELSE NULL END,
  'message', r.message
)
FROM result r;

```

Abbildung 33: Funktionskörper von `list_labels_minimal` (`p_user_id` BIGINT, `p_space_id` BIGINT) RETURNS JSONB

```

WITH RECURSIVE ancestors AS (
  SELECT se.id, se.parent_id
  FROM social_entities se
  WHERE se.id = p_id

  UNION ALL

  SELECT se.id, se.parent_id
  FROM social_entities se
  JOIN ancestors a ON se.id = a.parent_id
)
SELECT a.id FROM ancestors a;

```

Abbildung 34: Funktionskörper von `get_social_entity_ancestor_ids` (`p_id` BIGINT) RETURNS TABLE (`id` BIGINT)

5.7. Backend-seitig umgesetzte User Stories

Nr.	User Story	Stichwort
6	Als Admin möchte ich Berechtigungen von Mitgliedern verwalten können.	PSs, PSAs, PSABRTs
8	Als Admin möchte ich verschiedene Arten von Mitgliedern unterscheiden können (benutzerdefiniert, z.B. Probemitglied, Kernmitglied, Gast ...)	SERTs
15	Als Admin möchte ich die Stammdaten eines Space einsehen und bearbeiten können.	Space, manage
20	Als Koordinator:in möchte ich Fähigkeiten, die innerhalb einer Organisation/Space relevant sind, verwalten können (anlegen, bearbeiten, löschen).	Skills, Impairments
21	Als Koordinator:in möchte ich nachschauen können, welche Mitglieder eine bestimmte Fähigkeit haben.	SkillPossessions, ImpairmentAffections
25	Als Koordinator:in möchte ich Fähigkeiten an Mitglieder manuell vergeben und entziehen können.	
26	Als Koordinator:in möchte ich einstellen können, dass eine Fähigkeit nach "n erledigten Aufgaben vom Typ y" automatisch erlangt wird.	SkillAcquisitionTriggers
74	Als User möchte ich zwischen verschiedenen Organisationen (IG FoodCoops, meinen FoodCoop(s)) hin- und her wechseln können, ohne mich neu einloggen zu müssen.	Spaces

5.8. Testing

Tests dienen dazu, Fehler zu erkennen und das erwartete Verhalten einer Anwendung zu überprüfen. Dabei können einzelne Komponenten isoliert oder mehrere Komponenten im Zusammenspiel untersucht werden. Die automatisierten Backend-Tests von coOps lite wurden mit JUnit umgesetzt und umfassen beide Ansätze.

5.8.1. Zielsetzung und Testumfang

Die Tests von coOps lite prüfen erfolgreiche Abläufe und den Umgang mit Fehlern. Ein Organisationsbereich soll beispielsweise korrekt umbenannt werden. Gleichzeitig muss das Backend ungültige Eingaben und Änderungen ohne ausreichende Berechtigung zurückweisen. Deshalb werden sowohl zurückgegebene Daten als auch Fehlerzustände und HTTP-Statuscodes überprüft.

Im Backend des Projektes sind 66 Testmethoden in zehn Testklassen implementiert. Sie liegen unter *backend/src/test/java/at/foodcoops* und sind überwiegend den Paketen *controller* und *service* zugeordnet. Tabelle 2 zeigt ihre Verteilung. Davon wurden 45 Testmethoden ausgeführt.

Bereich	Testmethoden
Authentifizierung und JWT-Zugriffschutz	20
Verwaltung von Organisationsbereichen	20
Fähigkeiten und Einschränkungen	21
Allgemeine Bereichsdaten	3
Zentrale Fehlerbehandlung	1
Laden des Anwendungskontexts	1
Gesamt	66

Tabelle 2: Implementierte Backend-Tests

5.8.2. Aufbau der automatisierten Tests

Die Tests verwenden das Java-Testframework JUnit. Mit *@Test* gekennzeichnete Methoden bilden einzelne Testfälle ab. Assertions wie *assertEquals* und *assertThrows* prüfen erwartete Ergebnisse und Fehlerfälle. Beispielsweise vergleicht *assertEquals* einen zurückgegebenen Wert mit der Erwartung. Mit *assertThrows* lässt sich prüfen, ob eine fehlerhafte Eingabe die erwartete Ausnahme auslöst.

Für die isolierte Prüfung einzelner Klassen wird die Java-Bibliothek Mockito eingesetzt. Sie ersetzt benötigte Abhängigkeiten durch simulierte Objekte, sogenannte Mocks. Im *AuthControllerTest* wird beispielsweise vorgegeben, welches Token der simulierte *AuthService* zurückgeben soll. Der Test

prüft dann, ob der Controller dieses Token korrekt in seine Antwort übernimmt. Die eigentliche Token-Erstellung wird separat im *AuthServiceTest* untersucht.

Der *SecurityJwtFlowTest* verwendet das Spring-Testwerkzeug *MockMvc*. Damit lassen sich HTTP-Anfragen an die Anwendung simulieren, ohne einen eigenen Webserver zu starten. Die Sicherheitskonfiguration wird dabei berücksichtigt. So prüft der Test beispielsweise, ob eine Anfrage ohne gültiges Token mit dem HTTP-Status *401 Unauthorized* abgelehnt wird.

Die Integrationstests prüfen das Zusammenspiel der Services mit der Datenbank. Sie laden die benötigten Spring-Komponenten und führen die tatsächlichen SQL-Funktionen und gespeicherten Prozeduren aus. Damit wird beispielsweise untersucht, ob eine Änderung gespeichert und anschließend korrekt zurückgegeben wird.

Dafür ist eine vorbereitete Testdatenbank mit den erwarteten Demodaten erforderlich. Die Integrationstests für Fähigkeiten und Einschränkungen machen ihre Änderungen durch *@Transactional* nach dem Test automatisch rückgängig. Bei den Integrationstests der Bereichsverwaltung ist dies nicht eingerichtet. Zusätzlich verwendet die Anwendung die Datenbankeinstellung *create-drop*, die Tabellen beim Start neu anlegt und beim Beenden entfernt. Um die Daten der laufenden Anwendung nicht zu verändern, wurden die datenbankabhängigen Tests im dokumentierten Lauf nicht ausgeführt.

5.8.3. Ausgewählte Testfälle

Ablehnung eines abgelaufenen Tokens

Der Test *protectedManageSpaceEndpointRejectsExpiredToken* sendet eine Anfrage mit einem abgelaufenen Token an einen geschützten Endpunkt. Erwartet wird die Ablehnung mit *401 Unauthorized*. Das Token kann korrekt signiert sein und trotzdem wegen seines Ablaufzeitpunkts ungültig werden. Weitere Tests prüfen einen fremden Signaturschlüssel und einen abweichenden Aussteller. Der zugehörige Anmeldeablauf und die Postman-Aufnahmen sind in Kapitel 5.2 dokumentiert.

Rückgabe des aktualisierten Bereichsnamens

Der Integrationstest *editSettingsReturnsFreshlyUpdatedName* benennt einen vorhandenen Organisationsbereich über den *ManageSpaceService* um. Anschließend prüft er den Status *ok*, die Bereichskennung und den neuen Namen in der Antwort. Ein erfolgreicher Status allein genügt hier nicht: Auch der zurückgegebene Inhalt muss die Änderung bereits enthalten.

Der Test dient gleichzeitig als Regressionstest, damit dieser bereits behobene Fehler nicht erneut auftritt. Im vorliegenden Fall konnte die Antwort noch den alten Namen enthalten, obwohl die Änderung

bereits durchgeführt worden war. Die Datenbankprozedur führt deshalb die Änderung und das Zusammenstellen der Antwort in zwei getrennten SQL-Anweisungen aus. Der Test kontrolliert, dass danach der neue Wert zurückgegeben wird.

Verhindern einer doppelten Fähigkeitszuordnung

Der Test *createSkillPossessionTwiceReturnsBadRequestOnSecondAttempt* ordnet einer sozialen Entität eine Fähigkeit zu und versucht anschließend, dieselbe Zuordnung erneut anzulegen. Der erste Aufruf soll *ok* zurückgeben. Beim zweiten erwartet der Test *bad_request* und eine Fehlermeldung zur bereits vorhandenen Zuordnung. Ein vergleichbarer Test existiert für Einschränkungen. Beide prüfen die Regeln direkt im Backend, unabhängig von möglichen Eingabeprüfungen der Benutzeroberfläche.

Durchführung und dokumentierte Ergebnisse

Am 10. September 2026 wurden die sieben datenbankunabhängigen Testklassen gemeinsam in IntelliJ IDEA ausgeführt. Alle 45 ausgewählten Testmethoden waren erfolgreich. Der XML-Export weist keine fehlgeschlagenen oder übersprungenen Tests aus. Die ausgewiesene Testdauer beträgt 3,107 Sekunden.

Geprüft wurden unter anderem die Ablehnung abgelaufener Tokens, die Übernahme der Benutzerkennung aus dem Token und die Rückgabe erwarteter HTTP-Statuscodes bei simulierten Fehlerfällen.

Die 21 datenbankabhängigen Testmethoden waren nicht Bestandteil dieses Laufs. Dazu gehören auch die beiden beschriebenen Integrationstests zum aktualisierten Bereichsnamen und zur doppelten Fähigkeitszuordnung. Für diese Tests liegt kein aktuelles Ausführungsergebnis vor.

Testklasse	Anzahl	Ergebnis
<i>AuthServiceTest</i>	12	Erfolgreich
<i>AuthControllerTest</i>	3	Erfolgreich
<i>SecurityJwtFlowTest</i>	5	Erfolgreich
<i>ManageSpaceControllerTest</i>	17	Erfolgreich
<i>AbilitiesControllerTest</i>	4	Erfolgreich
<i>GeneralSpaceDataControllerTest</i>	3	Erfolgreich
<i>ApiExceptionHandlerTest</i>	1	Erfolgreich
Gesamt	45	Erfolgreich

Tabelle 3: Ergebnisse des Testlaufs vom 10. September 2026
(XML-Export aus IntelliJ IDEA, „Test Results - AuthServiceTest_and_6_more.xml“)

Ergänzend zeigen die in Kapitel 5.2 enthaltenen Postman-Aufnahmen einen manuellen Zugriff auf */api/auth/user-info*: Ohne Token wird die Anfrage mit *401 Unauthorized* abgelehnt, mit gültigem Token werden Benutzerinformationen und *200 OK* zurückgegeben. Dieser Nachweis betrifft die beiden dargestellten Anfragen und ersetzt keinen vollständigen Testlauf.

5.8.4. Grenzen der Überprüfung

Die vorhandenen Tests betreffen ausgewählte Backend-Funktionen. Eine Messung der Codeabdeckung liegt nicht vor. Aus der Zahl der Testmethoden lässt sich nicht ableiten, welcher Anteil des Anwendungscode tatsächlich geprüft wird.

Im dokumentierten Projektstand sind keine automatisierten Frontend-Tests vorhanden. TypeScript hilft dabei, Typfehler bereits während der Entwicklung zu erkennen, während ESLint den Quellcode auf mögliche Fehler und festgelegte Code-Regeln prüft. Bedienabläufe werden damit jedoch nicht getestet.

Auch die vorgesehene SSO-Anmeldung ist noch nicht vollständig prüfbar. Der zugehörige Controller-Test kontrolliert nur, dass der vorbereitete Endpunkt mit *501 Not Implemented* antwortet. Sobald die SSO-Anmeldung umgesetzt ist, sollte auch der gesamte Ablauf von der Anmeldung über den Bereichswechsel bis zum Speichern und erneuten Anzeigen einer Änderung getestet werden.

6. Umsetzung des Frontend

Das Frontend hat die Aufgabe, organisatorische Abläufe und andere Informationen für die Benutzer:innen verständlich und übersichtlich darzustellen sowie bearbeitbar und visuell nachvollziehbar zu machen.

6.1. Forschungsthema

Im Zentrum steht dabei folgende Forschungsfrage:

Wie kann das React-Framework mit Drag-and-Drop-Funktion genutzt werden, um flexible Abläufe in kollaborativen Umgebungen zu visualisieren und zu bearbeiten?

Zur Beantwortung dieser Forschungsfrage werden zunächst die grundlegenden Anforderungen an Benutzerfreundlichkeit und die Gestaltung von Benutzeroberflächen betrachtet. Danach folgt eine Beschreibung der eingesetzten Frontend-Technologien, der Struktur der Frontend-Implementierung sowie ausgewählter technischer Komponenten, Funktionen und Lösungsansätzen. Ein besonderer Fokus liegt auf der visuellen und interaktiven Darstellung von Vorgängen mithilfe von React und Drag-and-Drop. Da die dafür erforderlichen Backend-Funktionen zum Zeitpunkt der Entwicklung noch nicht vollständig verfügbar waren, wird die Drag-and-Drop-Funktion anhand einer prototypischen Umsetzung mit realitätsnahen Mock-Daten untersucht und dargestellt.

6.2. Zielsetzung

Nutzer:innen sollen schnell erkennen können, welche Aufgaben aktuell relevant sind, welche Abläufe existieren und welche Möglichkeiten zur Bearbeitung zur Verfügung stehen.

Außerdem sollen komplexe Prozesse in eine möglichst einfache Benutzeroberfläche übertragen werden. Vorgänge, Muster, Mitglieder, Fähigkeiten und weitere Informationen sollen abhängig vom jeweiligen Kontext und von den vorhandenen Berechtigungen dargestellt werden.

Darüber hinaus soll das Frontend es ermöglichen, Abläufe auch zu erstellen, zu bearbeiten und interaktiv anzupassen. Dabei sollen eine klare Navigation, wiedererkennbare Benutzeroberflächenelemente und verständliche Rückmeldungen die Bedienung unterstützen.

6.3. Auswahl von Technologien für das Frontend

Als Überblick werden im Folgenden wichtige und nützliche Technologien und Bibliotheken vorgestellt, die im Frontend eingesetzt wurden.

6.3.1. TypeScript

TypeScript ist eine Erweiterung der als JavaScript bekannten Sprache, die eine statische Typisierung bietet, um eine klarere Struktur der in einer Anwendung verwendeten Daten zu erzielen. Statische Typisierung hilft Programmierer:innen, bestimmte Fehler bereits bei der Entwicklung des Codes zu erkennen, statt dass diese erst zur Laufzeit auftreten. (The TypeScript Handbook)

In coOps lite wird TypeScript zusammen mit React verwendet, um die Struktur der Benutzerauthentifizierung mit den jeweiligen persönlichen Daten, Berechtigungen, Bereiche / Sozialer Entitäten und Fähigkeiten sowie Vorgängen zu definieren, die einem Bereich oder Unterbereich entsprechen. Da es verschiedene Typen und Schnittstellen gibt, arbeiten die in diesem Projekt erstellten Komponenten mit zuverlässigeren Daten. Außerdem ist es einfacher, eine kohärente Struktur aufrechtzuerhalten, wenn diese Komponenten in React zwischen den verschiedenen Teilen des Frontend wiederverwendet werden.

Aus diesem Grund erfordert TypeScript zwar eine genauere Spezifikation, doch bei einem großen und modularen Projekt verbessert es die Lesbarkeit des Codes und hilft dabei, Fehler oder Inkompatibilitäten zwischen Eigenschaften, Funktionen und anderen Elementen im weiteren Verlauf der Entwicklung zu vermeiden.

6.3.2. React UI Framework

React (react.dev) ist ein leistungsstarkes JavaScript-Frontend-Framework, welches effiziente und zugleich flexible Benutzeroberflächenentwicklung erleichtert. Nach dem Erlernen der Grundprinzipien ermöglicht es strukturierte und skalierbare Umsetzungen interaktiver Benutzeroberflächen. Dies wird vor allem durch das Kombinieren verschiedener Bibliotheken, Komponenten und Abhängigkeiten für den Zusammenbau der interaktiven User Interfaces erreicht.

Aufgrund seiner Reaktionsschnelligkeit und Zugänglichkeit ist es gut geeignet für die Anforderungen und Ziele dieses Projekts. (Roshi, 2026)

Die Kernfunktion unseres Projekts besteht darin, mittels den in Abbildung 35 dargestellten Benutzeroberflächenelementen Abläufe flexibel, interaktiv und unter Berücksichtigung unterschiedlicher Berechtigungen zu visualisieren.

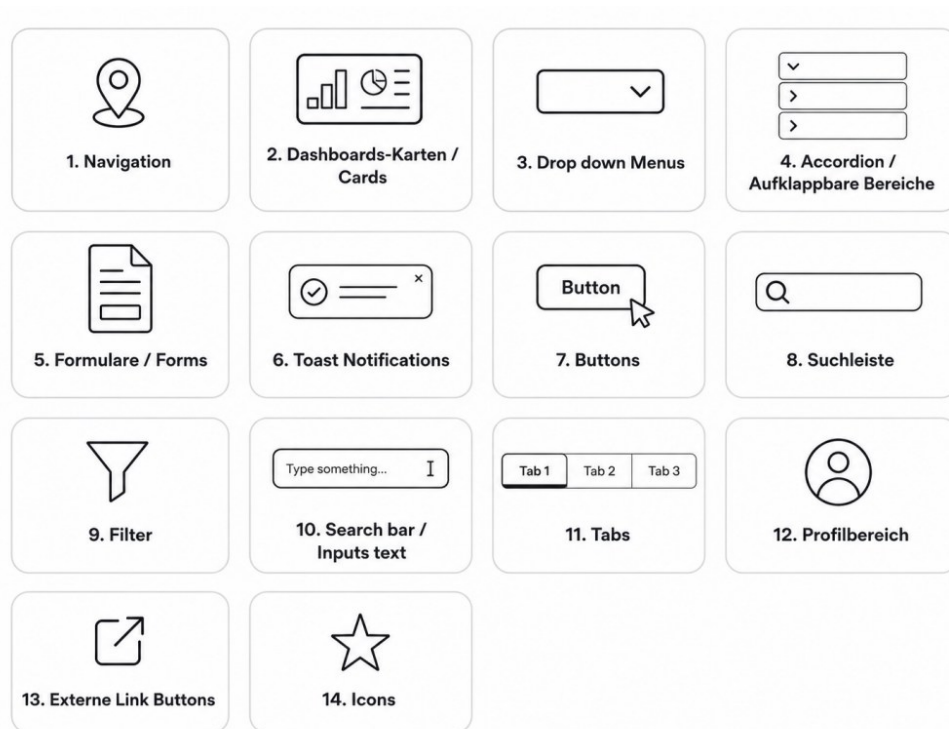


Abbildung 35: Übersicht verwendeter Benutzeroberflächenelemente (eigene Darstellung)

Für coOps lite sind vor allem übersichtliche Navigationselemente, Dashboard-Karten, filterbare Listen, Detailansichten, Formulare, Benachrichtigungen, Statusanzeigen und Drag-&Drop-Komponenten hilfreich.

6.3.3. Tailwind CSS

Tailwind CSS ist ein Open Source CSS Framework, das keine vordefinierten Klassen für Elemente wie Buttons oder Cards bietet, sondern vordefinierte Klassen zur Steuerung von Größe, Rahmen, responsivem Verhalten und anderen Stilelementen bereitstellt. (Tailwind Docs)

Für das visuelle Design der App wurde daher dieses Framework als Werkzeug ausgewählt. Seine Verwendung ermöglicht es, die verschiedenen HTML-Elemente flexibel an unterschiedliche Bildschirmgrößen anzupassen. Außerdem eignet sich dieses Tool sehr gut für andere Komponenten, die bei der Entwicklung der Anwendung zum Einsatz kamen.

```
<main className="flex-1 min-h-0 pt-15 pb-16 overflow-y-auto">
  <div className="w-full lg:max-w-7xl lg:mx-auto lg:px-6 xl:px-8">
    <AnimatedPage>
      <Outlet />
    </AnimatedPage>
  </div>
</main>
```

Abbildung 36: Layout je nach Größe des Geräts

Abbildung 36 zeigt ein Code-Beispiel, bei dem je nach Größe des Geräts unterschiedliche Ränder oder Abstände in dem markierten Bereich angewendet werden. Auf diese Weise können alle visuellen Seiten der App effizienter gestaltet werden.

6.3.4. Shadcn/ui

Bei shadcn/ui (ui.shadcn.com) handelt es sich um eine Plattform, die vollständig offenen Quellcode bereitstellt. Das bedeutet, dass sie hauptsächlich aus einer Reihe wiederverwendbarer und bearbeitbarer Komponenten besteht, die individuell angepasst werden können. (shadcn/ui Documentation) Da diese Tailwind verwenden, eignet sich die Kombination ideal. Die Komponenten wurden an das Design angepasst, um ein einheitliches Erscheinungsbild in der gesamten Anwendung zu gewährleisten.

6.3.5. Drag and Drop - dnd-kit

Drag & Drop ist eine technische Interaktionsfunktion, mit der verschiedene Elemente oder Objekte – wie beispielsweise Dateien, Symbole, HTML-Zeilen oder sogar eine Komponente mit einer Reihe von Elementen – innerhalb der visuellen Benutzeroberfläche einer Anwendung an eine andere Position gezogen und dort abgelegt werden können. (Brunken, 2022)

In unserem Projekt ist diese Funktion von großer Bedeutung für die Bearbeitung von Abläufen, um das Verschieben von Schritten intuitiv zu realisieren.

In coOps lite wird diese Funktionalität durch die Bibliothek *dnd-kit* umgesetzt, die speziell für diese Art von Implementierungen in React-Anwendungen entwickelt wurde.

6.3.6. Formulare und Validierung

Um neue Daten in einer Anwendung anzulegen, werden in der Regel Formulare verwendet, in denen Eingabefelder zum Ausfüllen dieser Daten enthalten sind. Zu diesem Zweck nutzt coOps lite *React Hook Form* und *Zod*. *React Hook Form* ist eine Bibliothek, die Formulare speziell für React verwaltet und dabei unnötige Renderings minimiert, um eine bessere Interaktion zu ermöglichen, während *Zod* für die Validierung der Regeln zuständig ist, denen die eingegebenen Daten entsprechen müssen.

Im umgesetzten Prototyp dient die Anwendung in erster Linie der Erstellung von Vorgängen. Eingabefelder im Formular unterliegen teilweise bestimmten Anforderungen. Werden diese nicht erfüllt, zeigt *Zod*, je nach Konfiguration, Fehlermeldungen an, um eine fehlerhafte Eingabe zu vermeiden und die Benutzeroberfläche entsprechend den Anforderungen der Anwendung effizient zu gestalten. (Pigeon, 2025)

6.3.7. Progressive Web App (PWA)

Das coOps lite Frontend wurde als Progressive Web App, kurz PWA, umgesetzt. Diese kann plattformunabhängig (beispielsweise auf Desktop-PCs und Handys) verwendet werden.

Für die Initialisierung der PWA als Frontend-Projekt wurde folgender Befehl verwendet:

```
npm create @vite-pwa/pwa@latest coOps Lite -- --template react-ts
```

Intern erweitert *@vite-pwa* die Browser-Funktionalität bzw. die Anwendung im Web und basiert auf Workbox, einer Google-Bibliothekssammlung. Diese arbeitet wiederum mit einem Service Worker, also einem Skript, das auch im Hintergrund ausgeführt wird, wenn die App gerade nicht aktiv verwendet wird. Die Funktionsweise dient der Automatisierung von Aufgaben zur Verwaltung des Caches. Darüber hinaus stellt sie Offline-Ressourcen bereit. Das bedeutet, dass die App auch dann noch verfügbar ist, obwohl eine Benutzer:in auf dem Gerät keine Internetverbindung hat, allerdings nur mit den bereits gespeicherten Daten.

Außerdem ist zu berücksichtigen, dass für das Lesen der Daten über die App spezifische und minimale Informationen bereitgestellt werden, die der Browser benötigt, um die Anwendung besser zu verstehen. Diese Daten sind in *vite.config.ts* unter *VitePWA({})* definiert, zum Beispiel die Hintergrundfarbe, eine Beschreibung der App und der Display-Modus. Letzteres ist besonders wichtig, da dadurch definiert wird, wie unsere App je nach Größe der Oberfläche sowie in verschiedenen Browsern bei Bedarf dargestellt wird, auch wenn der CSS- und JavaScript-Code nicht immer auf identische Weise umgesetzt wird. Diese Daten werden in Browser-Tools unter *Application (Manifest und Storage)* angezeigt.

In der Entwicklungsversion müssen die Ressourcen noch nicht verfügbar sein. Deshalb ist es sinnvoller, das PWA-Plugin erst später vollständig umzusetzen.

Zusammenfassend gilt: Bei einer Progressive Web App sind die Assets grundsätzlich dieselben wie bei einer Website. Sie bietet jedoch zusätzliche Möglichkeiten: Wenn Nutzer:innen online sind, werden Inhalte schneller geladen, und sie sind auch offline verfügbar. (MDN contributors, 2025)

6.3.8. Toast-Bibliothek

Toast ist eine Open-Source-Bibliothek, deren Ziel es ist, Meldungen mit verschiedenen Funktionen zu gestalten. Sie ist für den Einsatz in React-Anwendungen konzipiert. Sie enthält ein `toast()`-API. Mit dieser Funktion lassen sich verschiedene Arten von Erfolgs-, Fehler- oder Lademeldungen hinzufügen, die mit unterschiedlichen Stilen, Animationen und Symbolen gestaltet werden können, um die Bedeutung der jeweiligen Meldung besser zu verdeutlichen.

(Abbildung 37)

```
toast('', {
  duration: null,
  position: '',
  style: {},
  className: '',
  icon: '',
  iconTheme: {
    primary: '',
    secondary: '',
  },
  ariaProps: {
    role: '',
    'aria-live': '',
  },
  removeDelay: null,
  toasterId: '',
});
```

Abbildung 37: Toast-Beispiel (react-hot-toast)

Darüber hinaus ist `<Toaster />` die Hauptkomponente von Toast, die für die visuelle Darstellung der Meldungen zuständig ist. Sie akzeptiert verschiedene Eigenschaften und Optionen, mit denen sich das Erscheinungsbild und das Verhalten der Meldungen anpassen lassen. Dadurch kann die Darstellung an die jeweiligen Anforderungen angepasst und somit die Benutzererfahrung zu verbessert werden. (Omotayo, 2024)

Diese Komponente unterstützt verschiedene Instanzen innerhalb der App. Es ist jedoch wichtig zu beachten, dass bei einer mehrfachen Verwendung von `<Toaster />` ohne eine entsprechende ID jede Instanz auf denselben globalen Store zugreift. Infolgedessen wird dieselbe Nachricht mehrfach angezeigt werden. In einem effizienten Anwendungsfall soll die Komponente jedoch nur einmal eingebunden werden. Dafür eignet sich der Root der App am besten, genauer in *Main* auf derselben Ebene wie *BrowserRouter*. Auf diese Weise steht sie auf allen Routen einschließlich des Logins zur Verfügung, ohne dass sie auf jeder Seite erneut eingebunden werden muss. (Abbildung 38)

```
main.tsx > ...
<Toaster position="top-center" toastOptions={{ duration: 1000 }} />
```

Abbildung 38: Einbindung des Toaster

Motion

Motion (motion.dev) ist eine Animationsbibliothek für React. Sie ermöglicht es, Seitenwechsel, Einblendungen und andere UI-Elemente flüssig zu animieren. Motion wird in der Anwendung beispielsweise genutzt, um den Wechsel zwischen verschiedenen Seiten weicher und übersichtlicher zu gestalten.

6.4. Aufbau des Frontend

6.4.1. Projektstruktur

Das Frontend ist als React-Projekt mit TypeScript und Vite umgesetzt, wobei eine deutliche Trennung zwischen den folgenden Bereichen besteht:

- Hauptkomponente Main
- App-Routing und Layouts
- Seiten innerhalb der App
- Wiederverwendbare UI-Komponenten
- Daten- und Authentifizierungslogik
- Assets

Die zentrale Struktur befindet sich im *src*-Verzeichnis, wobei das grundlegende App-Setup in *main.tsx* und die Routen in *App.tsx* festgelegt werden. (Abbildung 39)

Die wichtigsten Verzeichnisse und deren Aufgaben sind:

1. *src*

Dies ist das Hauptverzeichnis für den gesamten Quellcode der Anwendung. Hier liegen alle selbst erstellten Komponenten, Seiten, Logik und CSS-Stile.

2. *Auth*

In diesem Ordner befindet sich die Authentifizierung sowie die zentrale Anwendungslogik rund um Typen, Berechtigungen, Kontext und Mockdaten.

Da die App Benutzerstatus, Login, Bereiche und Berechtigungen zentral verwaltet, ist dieser Ordner besonders wichtig.

Hier sind unter anderem enthalten:

- Auth-Kontext
- Typdefinitionen
- Mockdaten
- Berechtigungen
- Operations-Kontext

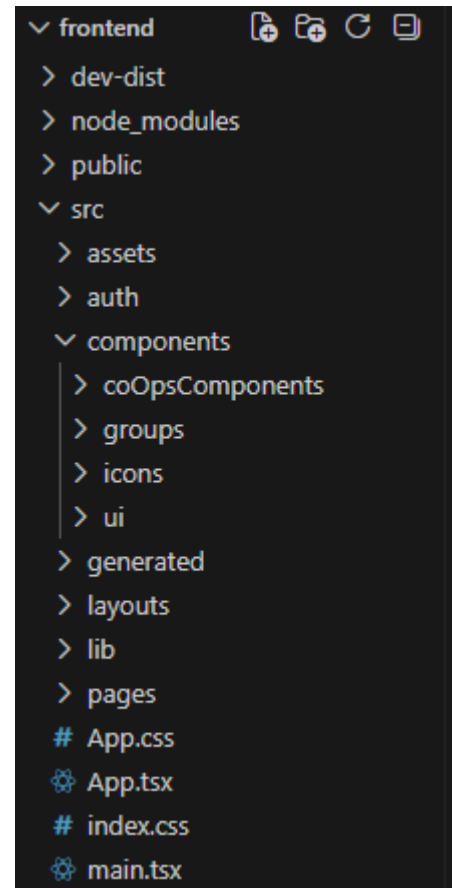


Abbildung 39:
coOps lite Frontend Struktur

3. components

Dieser Ordner enthält wiederverwendbare UI-Bausteine und funktionale Komponenten. Die Komponenten sind so aufgebaut, dass sie in verschiedenen Teilen der Anwendung genutzt werden können, ohne sie mehrfach zu schreiben. Er ist in Unterbereiche aufgeteilt, zum Beispiel:

- *components/ui*: allgemeine Bausteine wie Buttons, Dialoge oder Eingabefelder
- *components/icons*: wiederverwendbare Icons
- *components/coOpsComponents*: anwendungsspezifische Komponenten wie Filter, Karten, Formulare oder Navigationselemente
- *components/groups*: Komponenten als Datentypen für fachliche Objekte wie Operation, Pattern und Step für Mockdaten
- *components/ui*: allgemeine, wiederverwendbare UI-Elemente
- *components/icons*: SVG-Icons

Dies sorgt für eine modulare Struktur und hilft, Duplikationen zu vermeiden.

4. Layouts

Hier befinden sich die globalen Seitenrahmen und Layouts, also Strukturen wie Bereichs-Templates, Footer und Navigation. Beispiele sind:

- *Layout.tsx*: Hauptlayout mit Start, Benachrichtigungen und Profil
- *OrgaLayout.tsx*: Layout für den Organisationsbereich
- *ParticipationLayout.tsx*: Layout für den Teilnahmebereich

So müssen wiederkehrende Strukturen nicht auf jeder Seite neu implementiert werden.

5. pages

In diesem Ordner liegen die eigentlichen Bildschirmseiten beziehungsweise Views der Anwendung.

- *Login*
- *start*
- *participation*
- *orga*
- *profile*
- *error*

Jeder Bereich steht für eine bestimmte fachliche Funktion der App. Das ist der Bereich, der für den Nutzer sichtbar ist und die konkrete Business-Logik der jeweiligen Seite enthält.

6. *lib*

Hier werden kleine Hilfsfunktionen und allgemeine Utility-Tools abgelegt. Typische Beispiele sind:

- Formatierungen
- Klassen-Helper
- Toast- oder Benachrichtigungsfunktionen

In diesem Projekt ist *toast.ts* ein gutes Beispiel dafür.

6. *assets*

In diesem Ordner liegen Ressourcen wie Bilder, Grafiken und Design-Assets. Dabei handelt es sich nicht um UI-Komponenten, sondern um Bildmaterial oder visuelle SVG-Elemente, die in TypeScript umgewandelt und der App zur Verfügung gestellt werden.

7. *public*

Dieser Ordner enthält öffentliche, direkt ausgelieferte Dateien. Typische Inhalte sind:

- Icons
- Manifest-Dateien
- Statische Ressourcen, die direkt im Browser verfügbar sein sollen

8. *generated*

Hier werden automatisch generierte Typdefinitionen oder API-Modelle abgelegt, die aus der Backend- oder API-Spezifikation übernommen werden.

9. *dev-dist*

In diesem Ordner befinden sich generierte Build- und PWA-Dateien, die für die lokale Ausführung oder Service-Worker-Funktionen benötigt werden. Die wichtigsten Dateien sind:

- *main.tsx*: startet die App, setzt Provider und Router
- *App.tsx*: definiert die Routen der Anwendung
- *Layout.tsx*: globales Layout mit Navigation

Anhand dieser drei Dateien lässt sich der Aufbau der App gut nachvollziehen:

1. Die App wird gestartet
2. Das Routing wird aktiviert
3. Die Seiten werden über Layouts und Komponenten gerendert

Diese Struktur orientiert sich an den Prinzipien moderner Frontend-Architekturen und bietet dadurch:

- Trennung von Datenlogik und UI
- Wiederverwendbarkeit durch Komponenten
- klare Zuständigkeiten für jeden Bereich
- gute Skalierbarkeit für größere Projekte
- zentrale Authentifizierungslogik
- modulare Gliederung nach Features und Layouts

Komponentenstruktur und Wiederverwendbarkeit

Die wiederverwendbaren Komponenten sind im Ordner *components* organisiert. Dabei wird zwischen allgemeinen UI-Komponenten und fachlich spezifischen Komponenten, die bereits zuvor erwähnt wurden, unterschieden.

Die Komponenten werden in den Seiten und Layouts importiert und über Eigenschaften, sogenannte *props*, mit Daten versorgt. Dadurch können sie an verschiedenen Stellen wiederverwendet werden, ohne dass die gleiche Benutzeroberfläche mehrfach implementiert werden muss.

In coOps lite gibt es im gesamten Projekt zahlreiche wiederverwendbare Komponenten. Um diese Funktion konkret zu erläutern, dient die Komponente *GroupSwitcher* als Beispiel dafür, wie gut sich React und TypeScript in diesem Kontext ergänzen.

TypeScript in React ermöglicht es, die Datentypen verschiedener Werte, Parameter und Rückgabewerte zu spezifizieren. Der Datentyp *type* ist eine Bezeichnung, mit der verschiedene Eigenschaften oder Funktionen beschrieben werden können, die dieser Wert aufweist, bzw. jeder Wert, der einen Datentyp enthält. (TypeScript Tutorial)

Das ist es, was in der Komponente mit *Props* gemeint ist: Eigenschaften, die aus einer anderen Komponente mit anderen Werten wiederverwendet werden. (Abbildung 40)

Dem Gedanken der Wiederverwendung folgend lässt sich erkennen, dass die Eigenschaft *spaces* ein Array als Datentyp enthält, da sie weitere Werte kapselt. (Abbildung 41)

```
type Props = {
  status: "authenticated" | "loading" | "unauthenticated";
  user: User | null;
  spaces: SpaceInfo[];
  activeSpace: SpaceInfo | null;
  setActiveSpaceId: (id: number) => void;
  logout?: () => void;
  onOpenSettings?: () => void;
  onRefresh?: () => void;
};
```

Abbildung 40: type Props in coOpsComponents/GroupSwitcher

```
spaces: SpaceInfo[];
```

Abbildung 41: spaces in coOpsComponents/GroupSwitcher

Vor diesem Hintergrund wird deutlich, dass nicht nur Komponenten, sondern auch Eigenschaften von einer Datei oder Komponente in einer anderen wiederverwendet werden können.

Um diese Eigenschaft in einer Komponente zu nutzen, wird die Array-Methode `map()` verwendet (Abbildung 42). Diese erstellt bei ihrer Anwendung ein neues Array, das mit den verfügbaren Daten gefüllt wird und indem jedes Element des entsprechenden Arrays aufgerufen wird. Diese Funktion wird auch als Mapping oder Zuordnung bezeichnet. (MDN contributors, 2026)

```
{spaces.map((space) => {
```

Abbildung 42: `map()` in `coOpsComponents/GroupSwitcher`

Und innerhalb eines HTML-Elements kann der Name dieses Bereichs angezeigt werden. (Abbildung 43)

```
<div className={
  {space.name}
}>
```

Auf diese Weise kann man beispielsweise jeden Bereich und seine jeweiligen Eigenschaften darstellen und visualisieren.

Abbildung 43: Anzeige des Bereichsnamen in `coOpsComponents/GroupSwitcher`

Kontext und globale Zustände

Context ist eine React-Technologie, die darauf ausgelegt ist, Daten innerhalb eines Komponentenbaums bereitzustellen, ohne dass diese Eigenschaften beziehungsweise Informationen auf jeder Ebene manuell weitergegeben werden müssen. (React b)

Für die Erstellung und Verwendung eines neuen Kontextes werden zwei wichtige Funktionen benötigt: `createContext()` und `useContext()`

`createContext()` ist eine Funktion, die einen *Context* erstellt, dessen Werte über einen Provider bereitgestellt und mithilfe von `useContext()` von untergeordneten Komponenten abgerufen werden können.

Dieser Provider wird beispielsweise verwendet, um Zustände bei der Authentifizierung eines Benutzers, Funktionen, die bestimmte Informationen zu rendern oder andere relevante Informationen für die darunterliegenden Komponenten bereitzustellen.

In `coOps lite` wird dafür z. B. ein `AuthContext` (Abbildung 44) verwendet.

```
export const AuthContext =
  createContext<AuthContextType | null>(null);
```

Abbildung 44: `auth/AuthContext`

Über den zugehörigen Provider (Abbildung 45) können anschließend Zustände und Funktionen zentral bereitgestellt werden.

```
export function AuthProvider({ children }: { children: ReactNode }) {
  return (
    <AuthContext.Provider value={value}>
      {children}
    </AuthContext.Provider>
  );
}
```

Abbildung 45: auth/AuthProvider

Dadurch können beispielsweise Informationen zur Authentifizierung, zum aktuellen Anwendungskontext oder weitere gemeinsam benötigte Zustände für die darunterliegenden Komponenten verfügbar gemacht werden.

`useContext()` ist ein React-Hook, dessen Funktion darin besteht, um diese bereitgestellten Werte innerhalb anderer im Frontend verwendeter Komponenten zugreifen zu können. In coOps lite erfolgt dieser Zugriff beispielsweise über einen eigenen `useAuth()` Hook (Abbildung 46).

```
// Hook für Zugriff
export function useAuth() {
  const context = useContext(AuthContext);
```

Abbildung 46: useAuth in auth/AuthContext

Andere Komponenten können über diesen Hook auf die möglichen Informationen zugreifen. (Abbildung 47)

```
const { mockLogin, status } = useAuth();
```

Abbildung 47: Verwendung von useAuth in pages/Login

Dies ermöglicht es, gemeinsam benötigte Zustände zentral zu verwalten und in verschiedenen Bereichen des Frontend wiederzuverwenden.

Services und Backend-Anbindung

Die Kommunikation mit dem Backend ist von der Darstellungskomponente getrennt. React-Komponenten kümmern sich in erster Linie um die Benutzeroberfläche und die Interaktionen der Benutzer:innen. HTTP-Anfragen laufen im zentralen Kontext ab. Die erhaltenen Daten werden über *Props* oder *Hooks* an die jeweiligen Komponenten weitergegeben.

Im aktuellen Projekt wird diese zentrale Verwaltung vor allem durch den *AuthProvider* und den *OperationsProvider* umgesetzt. So lädt der *AuthProvider* zum Beispiel User-Informationen mit `fetch("/api/auth/mock-login")` und stellt diese anschließend über den Hook `useAuth()` verschiedenen Komponenten zur Verfügung. Dadurch entfällt für die einzelnen Komponenten die Notwendigkeit, eigene HTTP-Anfragen zu implementieren.

6.4.2. Besonderheiten (Komponenten, Snippets)

In diesem Kapitel werden ausgewählte technische und funktionale Aspekte näher erläutert. Zudem wird beschrieben, wie bestimmte User Stories im Frontend verwirklicht und wie die Benutzeroberfläche auf Benutzerinteraktionen reagiert. Anhand ausgewählter Codeausschnitte wird die Funktionalität des Frontends erläutert und dargestellt, welche Zusammenhänge zwischen den Funktionen und der UI bestehen.

Änderung des Aufgabenstatus und visuelle Fortschrittsanzeige

Die Statusänderung wird durch die Komponente *StateStepButton* umgesetzt. Der aktuelle Status wird über die Prop *currentState* an die Komponente übergeben. Die Funktion *getAvailableStates()* bestimmt anschließend, welche Statusänderungen abhängig vom aktuellen Zustand möglich sind. (Abbildung 48)

```
const getAvailableStates = (): {  
  state: StepSituation;  
  label: string;  
}[] => {  
  const currentLevel = getCurrentLevel();  
  const currentLevelIndex = progressLevels.indexOf(currentLevel);  
  
  const isInProgress =  
    currentState.toString().startsWith("inProgress");  
  
  const isStopped =  
    currentState.toString().startsWith("restart");  
  
  // ...  
};
```

Abbildung 48: *getAvailableStates* in *components/coOpscomponents/stateStepSelected*

Der aktuelle Zustand wird als *State* gezeigt und als *Label* wird anhand dieses Status die entsprechende Auswahlbeschreibung dargestellt.

Befindet sich ein Schritt beispielsweise in Bearbeitung, werden die Optionen „Fertig“, „Stop“ und die nächste Fortschrittsstufe angeboten. (Abbildung 49)

```
if (isInProgress) {
  const options: { state: StepSituation; label: string }[] = [
    { state: "done", label: "Fertig" },
    { state: `restart-${currentLevel}` as StepSituation, label: "Stop" },
  ];
  const nextLevel = progressLevels[currentLevelIndex + 1];

  if (nextLevel) {
    options.push({
      state: `inProgress-${nextLevel}` as StepSituation,
      label: `In bearbeitung (${nextLevel}%)`,
    });
  }
  return options;
}
```

Abbildung 49: components/coOpscomponents/stateStepSelected: In Bearbeitung

Für pausierte Schritte werden hingegen die Optionen „Fortsetzen“ und „Erledigt“ angezeigt. (Abbildung 50)

```
if (isStopped) {
  return [
    {
      state: `inProgress-${currentLevel}` as StepSituation,
      label: "Fortsetzen"
    },
    { state: "done", label: "Erledigt" },
  ];
}
```

Abbildung 50: components/coOpscomponents/stateStepSelected: Pausiert

Dadurch werden der Benutzer:in nur sinnvolle Statusänderungen angeboten. Ein bereits erledigter Schritt kann beispielsweise nur neu gestartet werden.

Die Auswahl eines neuen Status wird durch *handleStateChange()* verarbeitet: (Abbildung 51)

```
const handleStateChange = (newState: StepSituation) => {
  onStateChange(newState);
  setIsOpen(false);
};
```

Abbildung 51: components/coOpscomponents/stateStepSelected: Neuer Status

Die Funktion ruft zunächst *onStateChange* auf. Dieser „Callback“ übergibt dann den neuen Status an die übergeordnete Komponente. Anschließend wird das Auswahlmenü geschlossen.

Damit können wir für User Story 81 („Als Benutzer möchte ich meine zugewiesenen Aufgaben sehen und bearbeiten können“) die Anzeige und Bearbeitung des Status erfüllen.

Die visuelle Darstellung des Status erfolgt über die Funktion *StateStep()*. Sie erhält den aktuellen Status und gibt abhängig davon ein passendes Icon zurück: (Abbildung 52)

```
export const StateStep = (
  state: StepSituation | string
) => {
  return state === "InProgress-0"
    ? <IconInProgress0 />
    : state === "InProgress-5"
    ? <IconInProgress5 />
    : state === "InProgress-15"
    ? <IconInProgress15 />
    : state === "InProgress-30"
    ? <IconInProgress30 />
    : state === "restart-0"
    ? <IconRestart0 />
    : state === "done"
    ? <IconStepDone />
    : state === "initiate"
    ? <IconInitiate />
    : <p>Noch zu definieren...</p>;
};
```

Abbildung 52: components/coOpscomponents/stateStepSelected: visuelle Icons

Die Statuswerte *InProgress-0*, *InProgress-5*, *InProgress-15* und *InProgress-30* stehen für unterschiedliche Fortschrittsstufen. Für pausierte und erledigte Schritte werden eigene Symbole verwendet.

Das Icon wird im Button der Komponente angezeigt:

```
/* Button mit Icon */
<button
  ref={buttonRef}
  onClick={() => setIsOpen(!isOpen)}
  disabled={disabled}
  className={`p-2 rounded-md transition-colors ${disabled
    ? "opacity-50 cursor-not-allowed"
    : "hover:bg-gray-100 cursor-pointer active:bg-gray-200"}
  `}
  title="Klickst du hier, um den Status zu ändern"
>
  {StateStep(currentState)}
</button>
```

Abbildung 53: components/coOpscomponents/stateStepSelected: klickbare Status-Änderung

Wenn die Benutzer:in einen neuen Status auswählt, wird der Callback *onStateChange* aufgerufen. Die übergeordnete Komponente aktualisiert dadurch den Status. Beim nächsten Rendern erhält *StateStep()* den neuen Status und zeigt automatisch das dazugehörige Icon an.

Dies betrifft User Story 90: „Als User möchte ich andere Schritte in einem ‚Projekt‘, in dem ich für einen Schritt zuständig bin, ansehen können, sofern sie mich betreffen - wer zuständig ist, ob/wann erledigt, wann fällig, ausgefüllte Formularfelder ...“

Asynchrone Aktionen und Benutzerfeedback mit Toast-Nachrichten

Viele Aktionen der Anwendung kommunizieren mit dem Backend und werden deshalb asynchron ausgeführt. Dazu gehören beispielsweise das Laden von Bereichsdaten, das Aktualisieren von Eingabefeldern oder die Zuordnung eines Bereichstyps. Während diese Aktionen ausgeführt werden, soll die Benutzer:in unmittelbar erfahren, ob die Aktion erfolgreich war oder ein Fehler aufgetreten ist.

Für diese Rückmeldung wird die Bibliothek *react-hot-toast* verwendet. Der zentrale Toaster wird in *main.tsx* einmal global eingebunden: (Abbildung 54)

```
<Toaster  
  position="top-center"  
  toastOptions={{ duration: 1000 }}  
>
```

Abbildung 54: *src/main* : zentrale toast Funktion

Dadurch können Toast-Nachrichten von verschiedenen Komponenten aus angezeigt werden, ohne dass jede Komponente einen eigenen Toast-Container benötigt.

Zentrale Toast-Funktionen

Die Datei *toast.ts* stellt einheitliche Hilfsfunktionen bereit: (Abbildung 55)

```
export const notify = {  
  success: (message: string) => toast.success(message),  
  error: (message: string) => toast.error(message),  
  loading: (message: string) => toast.loading(message),  
};
```

Abbildung 55: *src/lib* : Hilfsfunktionen um verschiedene Meldungen zu laden

Für asynchrone Speicheraktionen wird zusätzlich *withSaveToast()* verwendet. Die Funktion wartet mit *await* auf den Abschluss der Aktion und zeigt abhängig vom Ergebnis eine Erfolgs- oder Fehlermeldung an: (Abbildung 56)

```
export async function withSaveToast<T>({
  action: () => Promise<T> | T,
  messages?: {
    success?: string;
    error?: string;
  },
}): Promise<T | undefined> {
  try {
    const result = await action();

    notify.success(
      messages?.success ?? "Änderung gespeichert."
    );

    return result;
  } catch (error) {
    notify.error(
      messages?.error ?? "Speichern fehlgeschlagen."
    );

    console.error(error);
    return undefined;
  }
}
```

Abbildung 56: src/lib : Asynchrone Funktion

Beispiel: Space-Daten speichern

In folgender Logik wird diese Funktion verwendet, wenn ein Eingabefeld des Bereichs gespeichert wird: (Abbildung 57)

```
const handleSaveSpaceField = async (
  field: "name" | "avatar" | "abbreviation" | "fullName" | "parent"
) => {
  await withSaveToast(async () => {
    if (field === "name") {
      await onUpdateSpaceField?.(
        "name",
        spaceFieldDrafts.name
      );
    }

    if (field === "abbreviation") {
      await onUpdateSpaceField?.(
        "abbreviation",
        spaceFieldDrafts.abbreviation
      );
    }

    if (field === "fullName") {
      await onUpdateSpaceField?.(
        "fullName",
        spaceFieldDrafts.fullName
      );
    }
  });
};
```

Abbildung 57: components/coOpscomponents/SpaceManagementCard:
handleSaveSpaceField

Beim Aufruf von *onUpdateSpaceField* wird eine asynchrone Backend-Anfrage ausgeführt. Ist die Anfrage erfolgreich, erscheint die Meldung „Änderung gespeichert.“. Tritt ein Fehler auf, wird stattdessen „Speichern fehlgeschlagen.“ angezeigt.

Ein weiteres Beispiel ist das Hinzufügen einer Typzuordnung: (Abbildung 58)

```
const result = await withSaveToast(
  () => onCreateAttribution?.(typePreview.id, ""),
  {
    success: "Typ hinzugefügt.",
    error: "Typ konnte nicht hinzugefügt werden.",
  }
);

if (result !== undefined) {
  setTypePickerOpen(false);
  resetTypePicker();
}
```

Abbildung 58: *components/coOpscomponents/SpaceManagementCard:*
Typ hinzugefügt

Das Formular wird hier erst geschlossen, wenn die asynchrone Aktion erfolgreich abgeschlossen wurde. Bei einem Fehler bleibt die Benutzer:in im aktuellen Kontext und erhält eine entsprechende Rückmeldung.

Damit können wir User Story 15 erfüllen: „Als berechtigte Person möchte ich die Stammdaten eines Spaces einsehen und bearbeiten können“.

Die Toast Nachrichten verbessern dadurch das Benutzerfeedback, da die Benutzer:in nicht auf eine neue Seite wechseln oder den aktuellen Zustand selbst überprüfen muss, um zu erkennen, ob ihre Eingabe erfolgreich gespeichert wurde.

Typsichere Formularvalidierung mit Zod und React Hook Form

Für das Formular zum Erstellen eines neuen Vorgangs (*Operation*) werden React Hook Form und Zod gemeinsam eingesetzt. React Hook Form übernimmt die Verwaltung der Formulardaten und der Benutzerinteraktionen. Zod definiert hingegen die Regeln, die die eingegebenen Daten erfüllen müssen. Durch die Kombination beider Bibliotheken werden die Eingaben sowohl strukturell als auch inhaltlich validiert.

Definition des Validierungsschemas

Im ersten Schritt wird mit Zod ein Schema für die Operation definiert: (Abbildung 59)

```
const formSchema = z.object({
  title: z
    .string()
    .min(1, "Der Titel darf nicht leer sein.")
    .max(50, "Der Titel darf höchstens 50 Zeichen lang sein."),

  description: z
    .string()
    .min(1, "Die Beschreibung darf nicht leer sein.")
    .max(150, "Die Beschreibung darf höchstens 150 Zeichen lang sein."),
});
```

Abbildung 59: components/coOpscomponents/createOperationForm: Titel und Beschreibung

Dabei wird beispielsweise festgelegt, dass der Titel nicht leer sein darf und maximal 50 Zeichen enthalten darf. Auch das Datum wird auf ein gültiges Datumsformat geprüft. Der Vorgangstyp darf nur einen der im Schema definierten Werte enthalten.

Verbindung mit React Hook Form

Das Zod-Schema wird über `zodResolver` mit React Hook Form verbunden: (Abbildung 60)

```
const form = useForm<z.infer<typeof formSchema>>>({
  resolver: zodResolver(formSchema),
  mode: "onSubmit",
  defaultValues: {
    title: "",
    description: "",
  },
});
```

Abbildung 60: components/coOpscomponents/createOperationForm:
Verbindung über zodResolver

`z` erzeugt automatisch den TypeScript-Type aus dem Zod-Schema. Dadurch müssen die Formulardaten nicht zusätzlich manuell typisiert werden. Änderungen am Schema werden somit direkt in den TypeScript-Type übernommen.

Anbindung eines Eingabefelds

Die einzelnen Felder werden mit *Controller* an React Hook Form angebunden: (Abbildung 61)

```
<Controller
  name="title"
  control={form.control}
  render={({ field, fieldState }) => (
    <Field data-invalid={fieldState.invalid}>
      <FieldLabel>Titel</FieldLabel>

      <Input
        {...field}
        placeholder="Gib einen Titel ein..."
        aria-invalid={fieldState.invalid}
      />

      {fieldState.invalid && (
        <FieldError
          errors={[fieldState.error]}
        />
      )}
    </Field>
  )}
/>
```

Abbildung 61: *components/coOpscomponents/createOperationForm:*
Anbindung React Hook - Controller

Controller verbindet das Eingabefeld mit dem Formularzustand. *field* enthält unter anderem den aktuellen Wert und die Änderungsfunktion. Über *fieldState.invalid* und *fieldState.error* kann die Benutzeroberfläche direkt auf Validierungsfehler reagieren.

Dadurch erhält die Benutzer:in beispielsweise beim Verlassen oder Absenden des Formulars eine konkrete Fehlermeldung, wenn der Titel leer ist oder die maximale Zeichenzahl überschritten wird.

Dynamische Formulare abhängig vom Vorgangstyp

Abhängig vom ausgewählten Vorgangstyp wird innerhalb derselben Komponente ein unterschiedliches Formular angezeigt. Dies wird über den Wert *selectOperationType* gesteuert. (Abbildung 62)

```
const selectOperationType = form.watch("operationType");

return (
  <>
    {selectOperationType === "task" ? (
      <TaskForm />
    ) : selectOperationType === "opportunity" ? (
      <OpportunityForm />
    ) : selectOperationType === "happening" ? (
      <HappeningForm />
    ) : (
      <span>Wählt bitte eine Option aus...</span>
    )}
  </>
);
```

Abbildung 62: createOperationForm: Auswahl des Vorgangstyps

In der tatsächlichen Umsetzung werden die Formulare direkt innerhalb von *CreateOperationForm* bedingt gerendert. Dadurch muss keine separate Komponente für jeden Vorgangstyp erstellt werden. Gleichzeitig können die Eingabefelder an die jeweiligen fachlichen Anforderungen angepasst werden. Eine *task* kann beispielsweise zusätzliche Schritte mit Titel und Beschreibung enthalten, während andere Operationstypen nur die allgemeinen Felder wie Titel, Beschreibung, Datum, Zuweisung, Labels und Fähigkeiten benötigen.

Abhängige Validierung von Schritten

Zusätzlich wird eine abhängige Validierung mit *superRefine* umgesetzt: (Abbildung 63)

```
.superRefine((data, ctx) => {
  const needsSteps =
    data.steps > 0 &&
    data.operationType === "task";

  if (
    needsSteps &&
    (!data.stepName ||
      data.stepName.trim().length < 10)
  ) {
    ctx.addIssue({
      code: "custom",
      path: ["stepName"],
      message:
        "Der Schrittname muss mindestens 10 Zeichen lang sein.",
    });
  }
});
```

Abbildung 63: createOperationForm: Validierung

In diesem Fall müssen Schrittnamen und Schrittbeschreibung nur dann ausgefüllt werden, wenn eine Operation vom Typ *task* mehrere Schritte enthält. Die Validierung berücksichtigt somit nicht nur einzelne Felder, sondern auch die Beziehung zwischen mehreren Eingaben.

Verarbeitung des Formulars

Beim Absenden wird die Funktion *handleSubmit* verwendet: (Abbildung 64)

```
<form
  onSubmit={form.handleSubmit(onSubmit)}
>
```

Abbildung 64: createOperationForm: Absenden

Erst wenn alle Validierungsregeln erfüllt sind, wird *onSubmit* ausgeführt: (Abbildung 65)

```
function onSubmit(
  data: z.infer<typeof formSchema>
) {
  const newOperation = {
    id: operations.length > 0
      ? Math.max(...operations.map((op) => op.id)) + 1
      : 1,
    title: data.title,
    description: data.description,
    type: data.operationType,
    targetDatetime: data.date,
    assigned: data.assignedUsers,
    requiredSkills: data.requiredSkill,
    labels: [],
    currentSteps: [],
  };

  addOperation(newOperation);
  onClose?();
  form.reset();
}
```

Abbildung 65: createOperationForm: Validierung

Die validierten Formulardaten werden anschließend in eine neue *Operation* umgewandelt und über *addOperation()* hinzugefügt. Danach wird das Formular geschlossen und zurückgesetzt.

Da noch kein *POST*-Endpoint für die Übermittlung ans Backend existiert, kann der Vorgang bislang nicht dauerhaft gespeichert werden. User Story 48 („Als Koordinatorin möchte ich einzelne (konkrete) Aufgaben/Projekte erstellen, duplizieren, bearbeiten und löschen können - auf Basis einer Musteraufgabe/-ablauf oder von Null auf.“) ist Frontend-seitig teilweise erfüllt.

Ansichtswechsel zwischen Fähigkeiten und Einschränkungen

Die Komponente *SkillsOverview* bildet die gemeinsame Übersichtsseite und stellt zwei Navigationsmöglichkeiten bereit: „Fähigkeiten“ und „Einschränkungen“. Der Wechsel zwischen den beiden Ansichten erfolgt über *NavLink*: (Abbildung 66)

```
<nav className="flex justify-between gap-4 border-b border-sky-200">
  <NavLink to="." end>
    <Lightbulb size={16} />
    Fähigkeiten
  </NavLink>

  <NavLink to="impairments">
    <ShieldAlert size={16} />
    Einschränkungen
  </NavLink>
</nav>
```

Abbildung 66: pages/orga/SkillsOverview: Navigation zwischen Fähigkeiten und Einschränkungen

Beim Klick auf „Fähigkeiten“ wird die Standardroute */organize/skills* geöffnet. Beim Klick auf „Einschränkungen“ wird zur Route */organize/skills/impairments* gewechselt.

Die aktuell ausgewählte Ansicht wird über die Eigenschaft *isActive* erkannt und visuell hervorgehoben: (Abbildung 67)

```
className={({ isActive }) =>
  `border-b-2 ${
    isActive
      ? "border-[#7d2647] text-[#7d2647]"
      : "border-transparent text-slate-400"
  }`
}
```

Abbildung 67: pages/orga/SkillsOverview: *isActive*

Damit erkennt die Benutzer:in direkt, welcher Bereich aktiv ist. Die jeweilige Unterseite wird anschließend über *<Outlet />* innerhalb von *SkillsOverview* dargestellt.

Die verschachtelten Routen werden in *App.tsx* definiert: (Abbildung 68)

```
<Route path="skills" element={<SkillsOverview />}>
  <Route index element={<Skills />} />
  <Route path="impairments" element={<Impairments />} />
</Route>
```

Abbildung 68: src/App.tsx: Routen für Skills und Impairments

Dadurch bleiben die gemeinsame Navigation und die Darstellung der Unterseiten voneinander getrennt. Beide Ansichten verwenden außerdem eine Berechtigungsprüfung (Abbildung 69):

```
if (!hasPermission("view_skills_and_impairments"))
```

Abbildung 69: Berechtigungsprüfung Skills & Impairments ansehen

Nur Benutzer:innen mit der Berechtigung *view_skills_and_impairments* können die Daten anzeigen. Die eigentlichen Einträge werden anschließend über *AbilityAccordion* als aufklappbare Elemente dargestellt.

Globale Seitenanimationen beim Navigieren

Um den Seitenwechsel innerhalb der Anwendung angenehmer zu gestalten, wird eine zentrale Komponente namens *AnimatedPage* verwendet. Sie umschließt den jeweiligen `<Outlet />` in den Layouts und animiert dadurch den Inhalt jeder untergeordneten Route.

Definition der Animation

In *AnimatedPage.tsx* werden die Zustände der Animation mit *motion* definiert (Abbildung 70):

```
const pageVariants: Variants = {
  hidden: {
    opacity: 0,
    y: 12,
  },
  visible: {
    opacity: 1,
    y: 0,
    transition: {
      duration: 0.35,
      ease: "easeOut",
    },
  },
};
```

Abbildung 70: components/coOpscomponents/AnimatedPage: pageVariants

Der Zustand *hidden* beschreibt den Anfangszustand. Die Seite ist zunächst leicht nach unten verschoben und transparent. Im Zustand *visible* wird sie an ihre normale Position bewegt und vollständig sichtbar dargestellt.

Animierte Komponente

Die Varianten werden auf ein *motion.div* angewandt:

Beim Rendern startet der Inhalt im Zustand *hidden* und wird anschließend in den Zustand *visible* überführt. Dadurch erscheint der neue Seiteninhalt nicht abrupt, sondern wird mit einer kurzen Bewegung und einem Fade-in eingeblendet. (Abbildung 71)

```
export function AnimatedPage({
  children,
}): {
  children: ReactNode;
}) {
  return (
    <motion.div
      initial="hidden"
      animate="visible"
      exit="hidden"
      variants={pageVariants}
    >
      {children}
    </motion.div>
  );
}
```

Abbildung 71:
components/coOpscomponents/AnimatedPage:
motion.div

Einbindung in die Layouts

Die Komponente wird in den zentralen Layouts um den *Outlet* gelegt.

Der *Outlet* stellt dabei den Inhalt der aktuell ausgewählten Route bereit. Da *AnimatedPage* in *Layout*, *OrgaLayout* und *ParticipationLayout* verwendet wird, erhalten die Seiten dieser Bereiche eine einheitliche Übergangsanimation. (Abbildung 72)

```
<main>
  <div>
    <AnimatedPage>
      <Outlet />
    </AnimatedPage>
  </div>
</main>
```

Abbildung 72: src/main.tsx:
AnimatedPage/Outlet

Wirkung für die Benutzeroberfläche

Beim Wechsel zwischen den Bereichen oder Unterseiten wird der neue Inhalt mit einer Bewegung von unten nach oben und zunehmender Transparenz dargestellt. Die Animation dauert 0,35 Sekunden und verwendet die Übergangsfunktion *easeOut*.

Die Animation verbessert dadurch die Orientierung innerhalb der Anwendung und macht den Wechsel zwischen den Ansichten weicher. Die eigentliche Navigation wird dadurch nicht verändert. Es handelt sich ausschließlich um einen visuellen Aspekt der Benutzeroberfläche.

6.5. Umgesetzte Frontend-Funktionen und User Stories

6.5.1. Verwendete Endpoints anhand ausgewählter Stories

Mehrere User Stories können denselben Endpoint verwenden. Daher werden die Stories im Folgenden nach der technischen Backend-Funktion gruppiert. Die Rollen Admin, Koordinator:innen und User beziehen sich auf die Berechtigungen der jeweiligen Benutzergruppe.

Die folgenden Tabellen zeigen nicht alle in der App realisierten User Stories, sondern nur eine Auswahl derjenigen, die den genannten Endpoints zugeordnet werden können. Viele Funktionen sind bereits im Frontend umgesetzt worden, teils mit echten Backend-Aufrufen und teils mit Mock-Daten oder lokalem Zustand. Dadurch ließen sich auch Stories präsentieren, deren Backend-Anbindung noch nicht komplett fertiggestellt ist. Gerade im Bereich der Vorgänge wurden Anzeige, Filterung und Statusanzeige nur im Frontend umgesetzt. Die Tabelle soll vor allem den Zusammenhang zwischen bestimmten Stories und deren technischer Umsetzung verdeutlichen und erhebt keinen Anspruch auf Vollständigkeit aller umgesetzten Stories.

6.5.2. Implementierte bzw. im Frontend verwendete Endpoints

Bereich	Endpoint	Zweck	Zugehörige ausgewählte Stories
Login / Anmeldeseite	<i>POST</i> <i>/api/auth/mock-Login</i>	Anmeldung einer Benutzer:in und Ausgabe eines Authentifizierungs-Tokens	Testbenutzer:in kann sich für Entwicklungszwecke anmelden (teilweise US 68, 72)
Layouts	<i>GET</i> <i>/api/auth/user-info</i>	Laden der Benutzerdaten, Bereiche, Soziale Entitäten und Berechtigungen nach dem Login	Benutzer:in sieht ihre Bereiche; Benutzer:in kann nach einem Seitenaufruf angemeldet bleiben; Berechtigungen werden geladen (US 74)
Management / Verwaltung	<i>GET</i> <i>/api/manage-space/{spaceId}</i>	Laden der Informationen eines Bereichs und seiner Typzuordnungen	Benutzer:in kann Bereichsdaten ansehen. (US 15)
Management / Verwaltung	<i>PATCH</i> <i>/api/manage-space/{spaceId}</i>	Ändern der Stammdaten eines Bereichs	Admin oder berechnigte Koordinator:in kann Name, Abkürzung, Avatar oder übergeordneten Bereich bearbeiten (US 15)

Management / Verwaltung	<i>POST</i> <i>/api/manage-space/type-attributions</i>	Anlegen einer Zuordnung zwischen Bereich und Social-Entity-Typ	Koordinator:in kann einen Bereich als Typ markieren (US 15)
Management / Verwaltung	<i>PATCH</i> <i>api/manage-space/type-attributions/{attributionsId}</i>	Ändern einer bestehenden Typ-Zuordnung	Koordinator:in kann die Zuordnung oder deren Notiz bearbeiten (US 15)
Management / Verwaltung	<i>GET</i> <i>/api/social-entity-types/{socialEntityTypeId}</i>	Laden detaillierter Informationen zu einem Social-Entity-Typ	Koordinator:in kann die Beschreibung und Herkunft eines Bereichstyps einsehen (US 15)
Skills / Können	<i>GET</i> <i>/api/abilities/space/{spaceId}</i>	Laden von Fähigkeiten und Einschränkungen eines Bereichs	Benutzer:in kann Fähigkeiten und Einschränkungen anzeigen (US 20)
Management / Verwaltung	<i>GET</i> <i>api/social-entity-types/space/{spaceId}</i>	Laden einer Liste aller Social-Entity-Types innerhalb eines Bereichs	Eine berechtigte Person kann einem Bereich einen passenden Attribution-Typ zuordnen (US 15)

Außerdem wird der Aufruf für Fähigkeiten nur ausgeführt, wenn die Berechtigung *view_skills_and_impairments* vorhanden ist.

6.5.3. Prototypische User Stories mit Mock-Daten

Bereich	Zweck	Zugehörige ausgewählte Stories	Status
Start	Relevante Benachrichtigungen, Aufgaben und Gelegenheiten laden und anzeigen	User sieht relevante Aufgaben oder Möglichkeiten auf der Startseite und in den Benachrichtigungen. (US 76, 77, 78)	Frontend vorbereitet
Orga: Konkret	Vorgänge in Orga-Perspektive anzeigen und ihre Eigenschaften darstellen	Koordinator:in kann Aufgaben mit Eigenschaften wie Titel, Beschreibung, Typ, Status, Fälligkeit, Labels, Skills und Zuweisungen einsehen. (US 50, 101, 103, 112)	Anzeige im Frontend vorhanden; Daten stammen derzeit aus Mock-Daten bzw. lokalem Zustand
Orga: Konkret	Die angezeigten Vorgänge lokal nach bestimmten Kriterien einschränken	Koordinator:in kann Vorgänge nach Text und Labels gefiltert auflisten lassen. (US 68, 73)	Teilweise im Frontend umgesetzt; keine serverseitige Filterung über den Endpoint
Orga: Konkret	Das Formular zum Erstellen eines neuen Vorgangs öffnen	Koordinator:in kann eine neue Aufgabe erstellen und die entsprechenden Eigenschaften erfassen. (US 48)	Benutzeroberfläche vorhanden; keine bestätigte Speicherung über zum Beispiel <i>POST /api/operations</i>
Mitmachen: Aufgaben	Aufgaben anzeigen, die für die Benutzer:in relevant sind	User sieht zugewiesene relevante Aufgaben. (US 76, 77, 78)	Anzeige im Frontend vorhanden; derzeit mit Mock-Daten bzw. lokalem Zustand

Mitmachen: Aufgaben	Die angezeigten Aufgaben anhand ihres Prozessstatus in unterschiedliche Bereiche einordnen	User sieht je nach Prozessstatus unterschiedliche Aufgaben, zum Beispiel aktive, offene oder aktuelle Aufgaben. (US 76, 77, 79)	Im Frontend umgesetzt; die Statusfilterung erfolgt lokal
Mitmachen: Aufgaben	Den aktuellen Bearbeitungsstatus einzelner Schritte anzeigen und verändern	User kann einen Schritt starten, fortsetzen, pausieren, erledigen oder neu beginnen. (US 81)	UI mit <i>StateStepButton</i> vorhanden; dauerhafte Speicherung im Backend noch nicht umgesetzt
Mitmachen: Aufgaben	Details einer Aufgabe und ihrer Schritte anzeigen	User kann Informationen zu einer Aufgabe bzw. <i>Operation</i> und deren aktuellem Fortschritt sehen. (US 90, 101, 103, 112)	Darstellung im Frontend vorhanden; Daten basieren derzeit auf Mock-Daten
Start, Mitmachen, Orga	Eine gemeinsame Liste von <i>Operations</i> für mehrere Seiten bereitstellen		Frontend-seitig über <i>OperationsContext</i> vorbereitet

6.5.4. Authentifizierung

Der Authentifizierungsprozess findet zunächst auf der Anmeldeseite von coOps lite statt. Um die Grundlagen zu verstehen, wird untersucht, wie die Struktur der API-Antwort aus dem Backend aufgebaut ist (Abbildung 73):

```
{
  "token": "...",
  "userInfo": {
    "user": {
      "id": "...",
      "firstName": "...",
      "lastName": "..."
    },
    "spaces": [
      {
        "id": "...",
        "name": "...",
        "abbreviation": "...",
        "avatar": "...",
        "asSocialEntities": [
          {
            "id": "...",
            "name": "...",
            "abbreviation": "...",
            "relations": "...",
            "permissions": [
              "permission_1",
              "permission_2",
              "..."
            ],
            "notificationCount": "..."
          }
        ],
        "subspaces": [
          {
            "id": "...",
            "name": "...",
            "abbreviation": "...",
            "relations": "...",
            "permissions": [
              "permission_1",
              "permission_2",
              "..."
            ],
            "notificationCount": "..."
          }
        ],
        "subspaces": "..."
      }
    ]
  },
  "notificationCount": "..."
},
]
```

Abbildung 73: Struktur der API-Antwort bei erfolgreicher Benutzeranmeldung

Mit diesen Informationen und mithilfe von TypeScript lassen sich die Properties für die endgültige Datenstruktur im Frontend präzise definieren (Abbildung 74):

```
export type SpaceInfo = {
  id: number;
  name: string;
  abbreviation: string;
  avatar: string | null;
  asSocialEntities: SocialEntityInfo[];
  subspaces: SpaceInfo[];
};
```

```
export type User = {
  id: number;
  firstName: string;
  lastName: string;
};
```

```
export type UserLoginInfo = {
  user: User;
  spaces: SpaceInfo[];
};
```

Abbildung 74: auth/types

Mit *AuthState* (Abbildung 75) lässt sich die endgültige Form für einen übersichtlicheren Sitzungsstatus definieren:

```
export type AuthState = {
  status : "loading" | "unauthenticated" | "authenticated";
  user: User | null;
  spaces: SpaceInfo[];
  activeSpaceId: number | null;
  activeSocialEntityId: number | null;
}
```

Abbildung 75: *auth/AuthContext*

Die App kann sich zu jedem Zeitpunkt nur in einem von drei Zuständen befinden, die als String-Literale dargestellt werden. Auf diese Weise werden Inkonsistenzen wie ein Zustand „authentifiziert, aber ohne Benutzerdaten“ vermieden.

Die Komponente *AuthProvider* ist für die Verwaltung dieses Zustands zuständig. Dazu speichert sie mithilfe der *useState()* Funktion den Wert *AuthState* und stellt außerdem Funktionen bereit, um diesen Zustand durch einen Aufruf an das Backend zu ändern.

Die Funktion *mockLogin()* führt diesen Aufruf an das Backend-API tatsächlich durch. Wenn sich die Benutzer:in mit ihrer *id* anmeldet, erhält sie vom Backend ein Token. Diese Funktion ist asynchron, das heißt, die Benutzeroberfläche friert nicht ein, während auf eine Antwort vom Backend gewartet wird. Nach Erhalt der Antwort führt sie zwei Schritte in dieser Reihenfolge aus:

1. Sie empfängt das Token und speichert es persistent ein, sodass die Sitzung eine Aktualisierung der Seite überstehen kann, ohne dass die Anmeldedaten verloren gehen.
2. *LoginAs(userInfo)* empfängt den *User* und seine Bereiche, die vom Backend kommen, und wandelt sie in einen neuen Zustand *AuthState* um. (Abbildung 76)

```

async function mockLogin(userId: number): Promise<void> {
  try {
    const response = await fetch(`${API_BASE_URL}/api/auth/mock-login`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ userId })
    });
    if (!response.ok) {
      throw new Error("Mock Login failed")
    }
    const data = await response.json();

    saveAuthToken(data.token);
    loginAs(data.userInfo);
  } catch {
    console.log("Error loading the login...")
  }
}

```

Abbildung 77: auth/AuthProvider: Aufruf ans Backend

Das heißt, die Funktion *LoginAs()* ruft keine Daten aus dem Netzwerk ab, sondern bildet diesen Endzustand aus den bereits empfangenen Daten. Auf diese Weise wird der Zustand als *authenticated* markiert, wobei der standardmäßig aktive Bereich oder der aus der vorherigen Sitzung gespeicherte Bereich berücksichtigt wird (Abbildung 77):

```

export function AuthProvider({ children }: { children: ReactNode }) {
  function loginAs(loginInfo: UserLoginInfo) {
    setState({
      status: "authenticated",
      user: loginInfo.user,
      spaces: loginInfo.spaces,
      activeSpaceId: selectedSpace?.id ?? null,
      activeSocialEntityId: selectedSocialEntity?.id ?? null,
    });
  }
}

```

Abbildung 76: auth/AuthProvider: loginAs

Zusammenfassend lässt sich sagen, dass *mockLogin* die Daten vom Server abrufen und das Token des *User* speichern, während *LoginAs* diese Daten in den Zustand umwandelt, in dem sich eine angemeldete Benutzer:in befindet. Wie zu erkennen ist, sind beide Funktionen voneinander getrennt. Dadurch kann ausschließlich die Funktion *LoginAs()* wiederverwendet werden, ohne dass die Daten erneut abgefragt oder das Token gespeichert werden müssen.

Auf diese Weise muss mittels eines `useEffect()` lediglich der Zustand neu aufgebaut werden, indem `/api/auth/user-info` mit der Autorisierung durch das zuvor gespeicherte Token aufgerufen wird. (Abbildung 78)

```
async function loadUserInfo() {
  try {
    const response = await fetch(`${API_BASE_URL}/api/auth/user-info`, {
      headers: {
        Authorization: `Bearer ${token}`
      },
      signal: controller.signal
    });
    const userInfo: UserLoginInfo = await response.json();
    loginAs(userInfo);
  }
}
```

Abbildung 78: auth/AuthProvider: loadUserInfo

Wie beim Login wird diese gesamte Logik über `useContext()` bzw. `useAuth` – das nur einen Wrapper für `useContext(AuthContext)` darstellt – genutzt (Abbildung 79):

```
// React Context erstellen
export const AuthContext =
  createContext<AuthContextType | null>(null);

// Hook für Zugriff
export function useAuth() {
  const context = useContext(AuthContext);
}
```

Abbildung 79: auth/AuthContext

Auf der Seite `Login.tsx` wird dieser gesamte Kontext gelesen, ohne dass Eigenschaften manuell zwischen Komponenten übergeben werden müssen. Über den Button „Login“ wird die asynchrone Funktion ausgelöst, die wartet, bis der gesamte Zyklus abgeschlossen ist, um den vollständigen Status des User in React aufzubauen. (Abbildung 80)

```
const { mockLogin, status } = useAuth();

async function handleMockSsoLogin() {
  await mockLogin(userInput);
}

useEffect(() => {
  if (status === "authenticated") {
    navigate("/");
  }
}, [status, navigate]);
```

Abbildung 80: pages/Login

Da *MockLogin()* diesen internen Status des *AuthProvider* über *setState* aktualisiert, wird durch den entsprechenden *useEffect()*, der die Statusabfrage überprüft, erst dann zur Startseite navigiert und alle gespeicherten Daten visuell geladen, wenn sich der Status auf *authenticated* geändert hat.

Abbildung 81 zeigt ein Diagramm, das die Authentifizierung in coOps lite veranschaulicht:

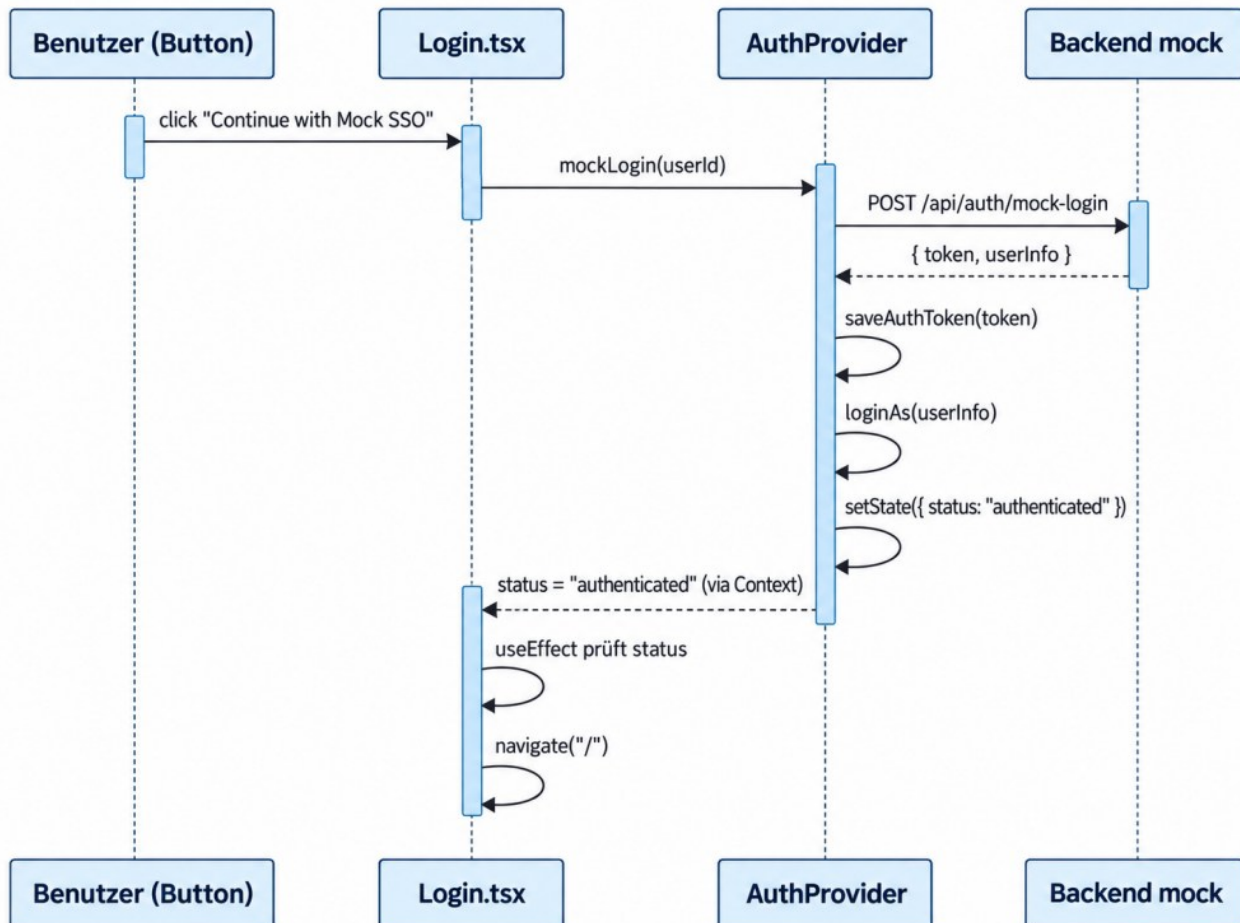


Abbildung 81: Authentifizierungszyklus im Frontend (eigene Darstellung)

Kontextwechsel

In coOps lite wird Kontextwechsel verwendet, um Informationen global zu verwalten, insbesondere im Zusammenhang mit der Benutzerauthentifizierung: Den Bereich, in dem sich die Benutzer:in befindet, mögliche aktive Soziale Entitäten, verfügbare Fähigkeiten und andere relevante Daten abhängig vom aktuellen Kontext der Anwendung. Auf diese Weise kann die Benutzeroberfläche bei einem Kontextwechsel zwischen den verschiedenen Komponenten konsistent aktualisiert werden, ohne dass es zu Konflikten zwischen ihnen kommt.

Für diesen Kontextwechsel in der App wurde die Komponente *GroupSwitcher* erstellt, die in den Layouts wiederverwendet wird, in denen der Kontextwechsel – genauer gesagt, der Wechsel des aktiven Bereichs – stattfindet. Der entscheidende Punkt ist, dass sich dieser aktive Bereich nicht innerhalb der

wiederverwendbaren Komponente selbst befindet, sondern im globalen Kontext der Authentifizierung verwaltet wird. Daher benötigt *GroupSwitcher* bestimmte Eigenschaften, *Props*, um den entsprechenden aktiven Bereich visuell darzustellen. (Abbildung 82)

```
type Props = {
  status: "authenticated" | "loading" | "unauthenticated";
  user: User | null;
  spaces: SpaceInfo[];
  activeSpace: SpaceInfo | null;
  setActiveSpaceId: (id: number) => void;
  logout?: () => void;
  onOpenSettings?: () => void;
  onRefresh?: () => void;
};
```

Abbildung 82: components/coOpscomponents/GroupSwitcher: Props

Das heißt, die Komponente übernimmt nur den aktuellen Wert und löst den Wechsel aus (Abbildung 83)

```
onClick={() => {
  setActiveSpaceId(space.id);
  setOpen(false);
}}
```

Abbildung 83: components/coOpscomponents/GroupSwitcher: Wechsel auslösen

Wenn die Benutzer:in einen anderen Bereich auswählt, wird die entsprechende Funktion aufgerufen und der Bereich festgelegt, der in dem dafür vorgesehenen Feld visuell angezeigt wird. Diese Funktion *setActiveSpaceId* befindet sich im globalen Kontext bzw. im Provider *AuthProvider*, wo die tatsächliche und globale Änderung dieses Zustands stattfindet.

Dadurch wird der Zustand des authentifizierten Benutzers aktualisiert (Abbildung 84):

```
setState((previousState) => {
  if (previousState.status !== "authenticated") {
    return previousState;
  }

  const selectedSpace = findSpaceById(
    previousState.spaces,
    spaceId,
  );

  if (!selectedSpace) {
    return previousState;
  }

  const firstSocialEntity =
    selectedSpace.asSocialEntities[0] ?? null;

  return {
    ...previousState,
    activeSpaceId: selectedSpace.id,
    activeSocialEntityId: firstSocialEntity?.id ?? null,
  };
});
```

Abbildung 84: auth/AuthProvider: Zustand aktualisieren

Es wird überprüft, ob die Benutzer:in authentifiziert ist. Der ausgewählte Bereich wird anhand seiner *id* ermittelt und bei Bedarf aktualisiert. Außerdem wird die Soziale Entität – also die Rolle, die der Benutzer in diesem Bereich innehat – bereitgestellt.

In den Layouts erhält die Komponente *GroupSwitcher* die Daten aus dem Kontext über *useAuth()*. (Abbildung 85)

All diese Werte und Funktionen sind in den jeweiligen Layouts benötigt, da ein aktueller globaler Status erforderlich ist, um eine Synchronisation zwischen diesen Eigenschaften zu gewährleisten. So wird sichergestellt, dass beim Navigieren innerhalb der App und beim Wechsel von einem Layout zum anderen der aktuelle Status erhalten bleibt und die aktuellen Benutzerdaten des jeweiligen Bereichs nicht verloren gehen oder verändert werden.

```
const {
  status,
  user,
  spaces,
  activeSpace,
  setActiveSpaceId,
  logout,
} = useAuth();
```

Abbildung 85: layouts

Anschließend wird gerendert (Abbildung 86):

Zusammenfassend lässt sich sagen, dass die angezeigten Daten aus dem aktuellen Kontext stammen. Beim Wechsel des Bereichs werden alle Layouts, die vom aktiven Bereich abhängen, automatisch aktualisiert, ohne dass manuelle Eigenschaften erforderlich sind.

Damit wird User Story 74 erfüllt: „Als User möchte ich zwischen verschiedenen Organisationen (IG FoodCoops, meinen FoodCoop(s)) hin- und her wechseln können, ohne mich neu einloggen zu müssen.“

```
<GroupSwitcher
  status={status === "authenticated"
    ? "authenticated"
    : status === "loading"
    ? "loading" : "unauthenticated"}
  user={groupSwitcherUser}
  spaces={spaces}
  activeSpace={activeSpace}
  setActiveSpaceId={setActiveSpaceId}
  logout={logout}
  onRefresh={() => {
    // optional : refetch
    console.log("refresh")
  }}
/>
```

Abbildung 86: Wechsel des Bereichs (Komponente)

6.5.5. Berechtigungsabhängige Benutzeroberfläche

Die Benutzeroberfläche wird abhängig von den Berechtigungen der angemeldeten Benutzer:in angepasst. Jede Benutzer:in kann je nach dem Bereich, in dem sie sich befindet, unterschiedliche Berechtigungen haben. Berechtigungen beziehen sich immer auf einen Bereich, da jemand in einer FoodCoop Koordinator:in und in einer anderen FoodCoop oder Arbeitsgruppe einfacher User sein kann.

Diese Berechtigungen sind im Authentifizierungsbereich im Ordner *auth* verfügbar und können über den Authentifizierungskontext an die Komponenten weitergegeben werden.

Über die Funktion `hasPermission()` kann geprüft werden, ob eine Benutzer:in eine bestimmte Aktion ausführen darf. (Abbildung 87)

```
/**Check if the user has a specific permission */
function hasPermission(permission: Permission): boolean {
  return permissions.includes(permission);
}

const canEdit = hasPermission("edit_social_entity");
```

Abbildung 87: auth/AuthProvider: Berechtigungsfunktion

Abhängig vom Ergebnis werden beispielsweise auf der Verwaltungsseite Bearbeitungs-Buttons ein- oder ausgeblendet. (Abbildung 88)

So kann eine Benutzer:in ohne die Berechtigung `edit_social_entity` keine Stammdaten des Bereichs bearbeiten.

Bei fehlender Berechtigung wird die Darstellung des betreffenden Bearbeitungsfelds verhindert.

```
{canEdit ? (
  <button
    type="button"
    className="rounded-
    onClick={onEdit}
    aria-label={` ${labe
  >
    <IconEdit />
  </button>
) : null}
```

Abbildung 88: Berechtigungsverwendung

6.5.6. Navigation und Menüstruktur

Hierarchie der Routen

Die Navigation von coOps lite ist hierarchisch aufgebaut und besteht aus drei Ebenen:

1. Hauptmenü: Das Hauptmenü befindet sich im Footer des *Layout.tsx*. Es ermöglicht den Wechsel zwischen den zentralen Bereichen „Start“, „Benachrichtigungen“, „Mitmachen“, „Orga“ und „Profil“.
2. Kontextbezogene Untermenüs: Die Bereiche „Mitmachen“ und „Orga“ besitzen jeweils ein eigenes Layout mit einem zusätzlichen Untermenü. Diese Navigation wird in *ParticipationLayout.tsx* beziehungsweise *OrgaLayout.tsx* definiert.
3. Zugeordnete Routen: Jeder Menüpunkt ist mit einer Route verbunden. Über diese Route wird die entsprechende Seite innerhalb des Layouts geladen.

Initialisierung der Navigation

Die Hierarchie des Komponenten- und Routenbaums beginnt in *main.tsx*. Dort werden die globalen Provider und der *BrowserRouter* definiert. Der *AuthProvider* stellt den Authentifizierungszustand

bereit, während der *OperationsProvider* die Vorgangsdaten und deren Zustand verwaltet. Anschließend werden die Login-Route und die geschützte Anwendung geladen. (Abbildung 89)

```
<AuthProvider>
  <OperationsProvider>
    <BrowserRouter>
      <Routes>
        <Route path='login' element={<Login />} />
        <Route path='/*' element={<App />} />
      </Routes>
    </BrowserRouter>
    {/* Zentrale Toast-Anzeige für die gesamte App */}
    <Toaster position="top-center" toastOptions={{ duration: 1000 }} />
  </OperationsProvider>
</AuthProvider>
```

Abbildung 89: Initialisierung der Anwendung und Einbindung des *BrowserRouter* in *main.tsx*

Routen in *App.tsx*

In *App.tsx* wird die eigentliche Navigationsstruktur der Anwendung definiert. Die Komponente *PrivateRoute* überprüft, ob ein Benutzer authentifiziert ist. Ist dies nicht der Fall, wird er zur Login-Seite weitergeleitet.

Die geschützten Bereiche sind in drei Hauptbereiche aufgeteilt. (Abbildung 90)

1. */:*
allgemeiner Anwendungsbereich
mit *Layout*
2. */participate:*
Bereich „Mitmachen“ mit
ParticipationLayout
3. */organize:*
Bereich „Orga“ mit *OrgaLayout*

Zusätzlich enthält *App.tsx* eine Fallback-Route mit dem Pfad ***. Diese wird verwendet, wenn eine nicht vorhandene Route aufgerufen wird, und zeigt eine Fehlerseite an.

```
<>
  <PrivateRoute>
    <Routes>
      <Route path='/' element={<Layout />} >
        <Route index element={<Start />} />
        <Route path="notifications" element={<News />} />
        <Route path="profile" element={<Profile />} />
      </Route>

      <Route path="participate" element={<ParticipationLayout />} >
        <Route index element={<DoTasks />} />
        <Route path="use" element={<ParticipateTasks />} />
        <Route path="actuate" element={<ParticipateWorkflows />} />
        <Route path="my-archive" element={<ParticipateArchive />} />
      </Route>

      <Route path="organize" element={<OrgaLayout />} >
        <Route index element={<Tasks />} />
        <Route path="patterns" element={<Processes />} />
        <Route path="members" element={<Members />} />
        <Route path="skills" element={<SkillsOverview />} >
          <Route index element={<Skills />} />
          <Route path="impairments" element={<Impairments />} />
        </Route>
        <Route path="manage" element={<Management />} />
      </Route>

      {/* Catch-all route for 404 Not Found */}
      <Route path="*" element={<NotFoundPage />} />
    </Routes>
  </PrivateRoute>
</>
```

Abbildung 90: Definition der geschützten Haupt- und Unterrouen in *App.tsx*

Aufbau der Layouts

Jedes Layout besteht grundsätzlich aus einem Header, einem Hauptbereich und einem Footer. Der Header enthält den *GroupSwitcher*, über den beispielsweise der aktive Bereich gewechselt und die Benutzeraktion ausgeführt werden können.

Der zentrale Bereich enthält den React-Router-Outlet: (Abbildung 91)

Der *Outlet* dient als Platzhalter für die jeweils untergeordnete Route. Dadurch kann beispielsweise *Tasks* innerhalb von *OrgaLayout* angezeigt werden, ohne dass Header und Footer erneut implementiert werden müssen.

```
<main className="flex">
  <div className="w-full">
    <AnimatedPage>
      <Outlet />
    </AnimatedPage>
  </div>
</main>
```

Abbildung 91: Verwendung von Outlet zur Darstellung verschachtelter Routen

Hauptmenü mit NavLink

Die Links im Footer werden mit *NavLink* aus *react-router-dom* umgesetzt. Dadurch kann automatisch erkannt werden, welcher Menüpunkt aktuell aktiv ist. Der aktive Link erhält eine eigene CSS-Klasse und wird dadurch visuell hervorgehoben.

NavLink übernimmt dabei die Navigation, während *isActive* den aktuellen Navigationsstatus liefert. Das zugehörige Icon unterstützt zusätzlich die visuelle Orientierung der Benutzer:in. (Abbildung 92)

```
<NavLink className={({ isActive }) =>
  `button-a flex flex-col justify-center items-center ${isActive ? "bg-white text-[#7d2647]" : "text-black"}`
>
  to="/"
  end
  <IconStart /> Start
</NavLink> { " " }
```

Abbildung 92: Menüpunkt „Mitmachen“ im Footer des Hauptlayouts

Kontextbezogene Untermenüs

Die Untermenüs werden direkt in den jeweiligen Layouts definiert. Im *ParticipationLayout* sind die Bereiche „Erledigen“, „Teilnehmen“, „Starten“ und „Archiv“ erreichbar. Im *OrgaLayout* stehen „Konkret“, „Muster“, „Struktur“, „Können“ und „Verwaltung“ zur Verfügung.

Da beide Layouts ebenfalls ein *Outlet* enthalten, werden die zugehörigen Unterseiten innerhalb des jeweiligen Bereichs dargestellt.

Zusammenfassung der implementierten Navigation

Bereich	Funktionalität	Routen
Hauptmenü	Wechsel zwischen Start, Benachrichtigungen, Mitmachen, Orga und Profil	<i>/</i> <i>/notifications</i> <i>/participate</i> <i>/organize</i> <i>/profile</i>
Mitmachen	Zugriff auf Aufgaben erledigen, Teilnehmen, Ablauf starten und Archiv	<i>/participate</i> <i>/participate/use</i> <i>/participate/actuate</i> <i>/participate/my-archive</i>
Orga	Verwaltung von Aufgaben, Mustern, Mitgliedern, Fähigkeiten/Einschränkungen und Stammdaten des Bereichs	<i>/organize</i> <i>/organize/patterns</i> <i>/organize/members</i> <i>/organize/skills</i> <i>/organize/manage</i>
Können	Trennung zwischen Fähigkeiten und Einschränkungen	<i>/organize/skills</i> <i>/organize/skills/impairments</i>
Authentifizierung	Schutz der Anwendung und Weiterleitung zur Login-Seite	<i>/login</i> <i>PrivateRoute</i>
Fehlerbehandlung	Anzeige einer Fehlerseite bei unbekannten Routen	<i>*</i>

6.6. Umsetzung von Usability-Prinzipien

Um ein einheitliches und effektives Design in coOps lite zu erreichen, wurde den zehn Usability-Heuristiken von Jakob Nielsen für die Gestaltung von Benutzeroberflächen besondere Bedeutung beigemessen. Das einheitliche Design der Interaktionselemente und eine intuitive Benutzerführung sorgen dafür, dass sich die Benutzer:in besser im System zurechtfindet und die kognitive Belastung minimiert wird. (Abbildung 93)

Jakob Nielsen, ein Pionier der Usability-Forschung, hat die Heuristiken in den 1990er-Jahren aufgestellt. Bis heute sind sie ein unverzichtbares Werkzeug für UX-Designer:innen. (Nielsen, 1994)

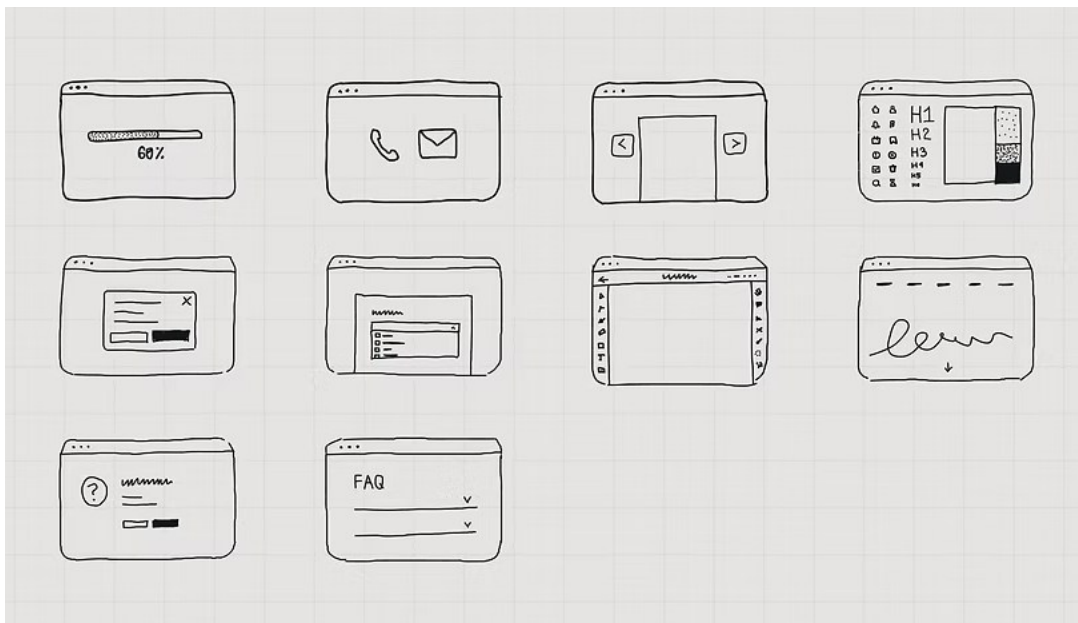


Abbildung 93: Zehn Usability Heuristiken (kreativviertel.at, 2025)

Im Folgenden werden Screenshots des coOps lite Designsystems gezeigt, die darstellen, welche Funktionen es bietet.

6.6.1. Transparente Rückmeldung zum Systemzustand

Das System informiert Nutzer:innen darüber, was nach einer Aktion passiert. In coOps lite wird dies zum Beispiel durch Rückmeldungen nach dem Ändern einer Eigenschaft in der Verwaltung umgesetzt. (Abbildung 94)

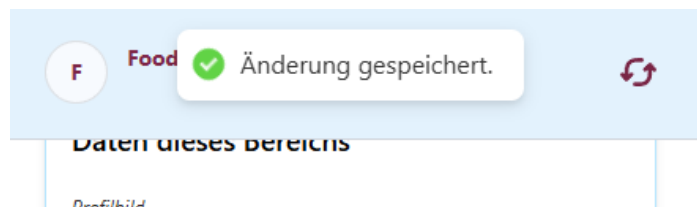


Abbildung 94: Erfolgsmeldung in Orga – Verwaltung

Eine Erfolgsmeldung erscheint erst, wenn die Änderung tatsächlich erfolgreich abgeschlossen ist. So sieht die Nutzer:in, ob ihre Aktion verarbeitet wurde. (Nielsen, 1994)

6.6.2. Kontrolle über eigene Aktionen

Nutzer:innen können begonnene Aktionen abbrechen oder Eingaben anpassen. In den Formularen von coOps lite stehen dafür etwa die Funktionen *Zurücksetzen* und *Schließen* bereit. (Abbildung 95) Beim Zurücksetzen des Formulars werden die zuvor eingegebene Werte gelöscht. Zusätzlich kann das Formular jederzeit über *Schließen* oder durch Klicken außerhalb des Formulars verlassen werden. Dadurch behalten Nutzer:innen die Kontrolle über den Bearbeitungsvorgang und können unbeabsichtigte Aktionen korrigieren oder schnell abbrechen. Eine Wiederherstellung bereits zurückgesetzter Daten ist dabei allerdings nicht möglich. (Nielsen, 1994)

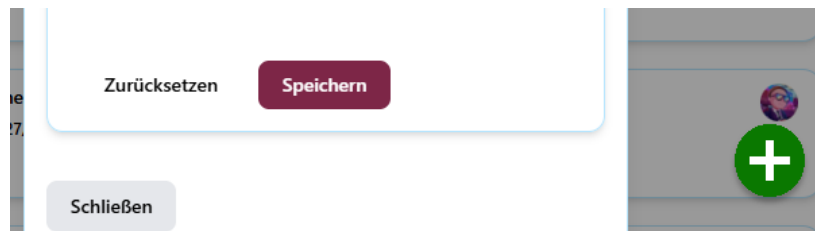


Abbildung 95: Zurücksetzen oder Speichern (Orga – Konkret)

6.6.3. Konsistente Bedienung

Wiederkehrende Interaktionselemente sind innerhalb der Anwendung einheitlich gestaltet und verhalten sich ähnlich. (Nielsen, 1994) In coOps lite werden gemeinsame Gestaltungsmuster für Buttons, Eingabefelder und andere Komponenten genutzt. Einheitliche Farben, Abstände und Button-Varianten helfen bei der Wiedererkennung und erleichtern die Bedienung verschiedener Ansichten. Konkret gibt es für *Button* verschiedene Stilvarianten, die direkt im Code genutzt werden können. Beispielsweise wurde der Code um die Variante *coopslite* erweitert. (Abbildung 96)

```
coopslite: "rounded-xl px-3 py-2 bg-[#7d2647] text-white text-sm font-semibold hover:opacity-90",
```

Abbildung 96: components/ui/Button

6.6.4. Fehler frühzeitig verhindern

Fehler sollen möglichst schon während der Eingabe erkannt werden, bevor ungültige Daten verarbeitet werden. Beim Erstellen oder Bearbeiten eines Vorgangs werden daher Pflichtfelder und weitere Eingabeanforderungen beachtet. (Abbildung 97) Wenn eine Eingabe nicht den festgelegten Anforderungen entspricht, zeigt das System direkt am betroffenen Feld eine verständliche Fehlermeldung an. Bereits eingegebene Daten bleiben dabei erhalten, sodass Nutzer:innen ihre Eingaben korrigieren können. (Nielsen, 1994)

Titel

Der Titel darf nicht leer sein.

Beschreibung

Korrekte Annahme der gelieferten Ware,
 Qualitätskontrolle für die Mitglieder.

0/150 characters

Enthält genauere Angaben, um zu verstehen, worum es geht.

Die Beschreibung darf nicht leer sein.

Abbildung 97: Formular „Vorgang erstellen“

Auf diese Weise wird verhindert, dass ein Eingabefeld leer bleibt, damit Informationen klar und nachvollziehbar sind.

6.6.5. Sichtbare Auswahlmöglichkeiten statt Gedächtnisbelastung

Nutzer:innen sollen verfügbare Optionen sehen können, anstatt sich deren Bezeichnungen merken zu müssen. In coOps lite werden u. a. Vorgangstypen, vorhandene Labels und Soziale Entitäten über Auswahlfelder angezeigt. Wie Abbildung 98 zeigt, können vorhandene Labels über ein Auswahlfeld gewählt werden, um Vorgänge nach ihnen zu filtern. So lassen sich die gewünschten Werte direkt auswählen, ohne interne Bezeichnungen manuell eingeben zu müssen. (Nielsen, 1994)

Label

Dienste für alle

Label auswählen...
 Dienste für alle
 Lagerartikel
 Mitgliederverwaltung
 Sammelbestellung

Abbildung 98: Label-Filter

6.6.6. Unterschiedliche Wege zur effizienten Nutzung

Heutzutage nutzt fast jeder Mensch sein Smartphone bereits als Arbeitsinstrument oder für soziale Kontakte, aber bei coOps lite legen wir Wert darauf, so präzise wie möglich zu sein – unter Berücksichtigung aller in der App verfügbaren Routen. Die App soll sowohl für neue als auch für erfahrene Nutzer:innen effizient bedienbar sein.

Abbildung 99: Button zur Suchfunktion

Aus diesem Grund haben wir im Frontend vordefinierte Tools für die verschiedenen Menüs sowie Funktionen wie *useNavigate()* verwendet, die für visuelle Komponenten wie Buttons in der App genutzt werden können und zu einem bestimmten Pfad innerhalb der App führen. Dazu kommen noch weitere implementierte Optionen wie Suchfunktionen (Abbildung 99), Filter, direkte Auswahlmöglichkeiten und Drag-and-Drop-Interaktionen bei Abläufen. Dadurch ermöglicht das System unterschiedliche Bedienungswege, um dasselbe Ziel zu erreichen, und kann an unterschiedliche Nutzungsgewohnheiten angepasst werden. (Nielsen, 1994)

6.6.7. Übersichtliches und minimalistisches Design

Bei der Entwicklung von coOps lite wurde darauf geachtet, nur die Informationen und Funktionen anzuzeigen, die im jeweiligen Kontext tatsächlich relevant sind. Aus diesem Grund sieht jede Benutzer:in, wenn sie einen bestimmten Bereich öffnet, als erstes eine Übersicht der für sie persönlich relevanten Benachrichtigungen und Aufgaben („Start“).

Diese Inhalte werden mithilfe von klar strukturierten Bereichen und aufklappbaren Elementen organisiert. Weniger häufig benötigte Informationen werden erst bei Bedarf, beispielsweise über *Accordions*, geöffnet.

Zusätzlich werden bestimmte Verwaltungsfunktionen nur dann angezeigt, wenn Nutzer:innen über die entsprechenden Berechtigungen verfügen. Dadurch bleibt die Oberfläche für reguläre User einfach, während Koordinationspersonen Zugriff auf erweiterte Funktionen erhalten. (Nielsen, 1994)

6.6.8. Fehler verständlich erkennen und beheben

Wenn Fehler auftreten, durch die Nutzer:innen den Zugriff auf die Anwendung verlieren, ohne die Möglichkeit zu haben, zurückzugehen oder das Problem selbst zu beheben – sei es, weil eine Seite bzw. die URL nicht gefunden wurde oder einfach, weil der Server aus bestimmten Gründen nicht erreichbar ist –, sollte eine klare Meldung ausgegeben werden, die den Eindruck einer modernen und professionellen Anwendung vermittelt. Dadurch können Frustration und weitere Probleme vermieden werden.

Anwendungsbeispiel: Bei Aufruf einer beliebigen Route, die in der App nicht verwendet wird, wird zu einer Fehlerseite geroutet. (Abbildung 100)

```
{/* Catch-all route for 404 Not Found */}
<Route path="*" element={<NotFoundPage />} />
```

Abbildung 100: Route zur NotFoundPage

Mögliche Lösungen sind das Hinzufügen von Buttons zum Zurückkehren zur Startseite und/oder zum Neuladen der Seite sowie die Angabe von Kontaktinformationen auf dieser Fehlerseite, damit die zuständigen Verantwortlichen über das Problem informiert werden können und dadurch der Fehler so schnell wie möglich behoben werden kann. (Abbildung 101)

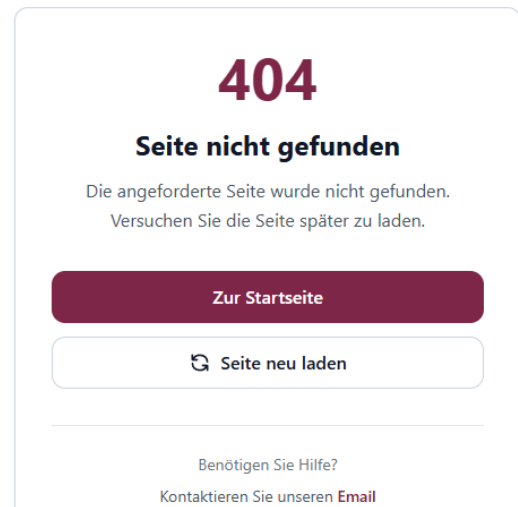


Abbildung 101: /pages/error

6.7. Drag-and-Drop-Interaktion innerhalb eines Ablaufs

In diesem Abschnitt werden prototypisch die Interaktionen und das Verhalten zwischen Aufgaben in einer Vorgehensweise gezeigt. Die hier gezeigten Funktionen basieren zum großen Teil auf ausgewählten User Stories und wurden ausschließlich im Frontend implementiert und funktionsfähig umgesetzt. Ergänzend wird eine prototypische Drag-and-Drop-Interaktion betrachtet, die einen direkten Bezug zur Forschungsfrage aufweist. Die gezeigten Abbildungen sind direkt aus dem implementierten Frontend aufgenommen und stellen den aktuellen Stand des Entwicklungsdesigns dar.

Als Anwendungsfall wurde der Bereich „Mitmachen“ gewählt. Hier werden eine Vielzahl von Operationen mit zugehörigen Schritten dargestellt, sodass eine gemeinsame Veranschaulichung aller möglichen Beziehungen und Veränderungen der Zustände möglich wird.

Hierzu gehörende technische Logik und konkrete Durchführung des Konzepts finden sich in Kapitel 6.4.2 (Definition der Animation).

User Story 76: Meine Aufgaben

„Als User möchte ich eine Liste an konkreten Aufgaben/Projekten, für die ich verantwortlich bin (insb. dringende), angezeigt bekommen.“

User Story 90: Schritte ansehen

„Als User möchte ich andere Schritte in einem ‚Projekt‘, in dem ich für einen Schritt zuständig bin, ansehen können, sofern sie mich betreffen - wer zuständig ist, ob/wann erledigt, wann fällig, ausgefüllte Formularfelder ...“

Aufgaben der Benutzer:in werden aufgelistet. Beim Klicken auf einen Vorgang werden die zugehörigen Schritte in einer aufklappbaren Liste als einzelne Aufgaben angezeigt. (Abbildung 102)

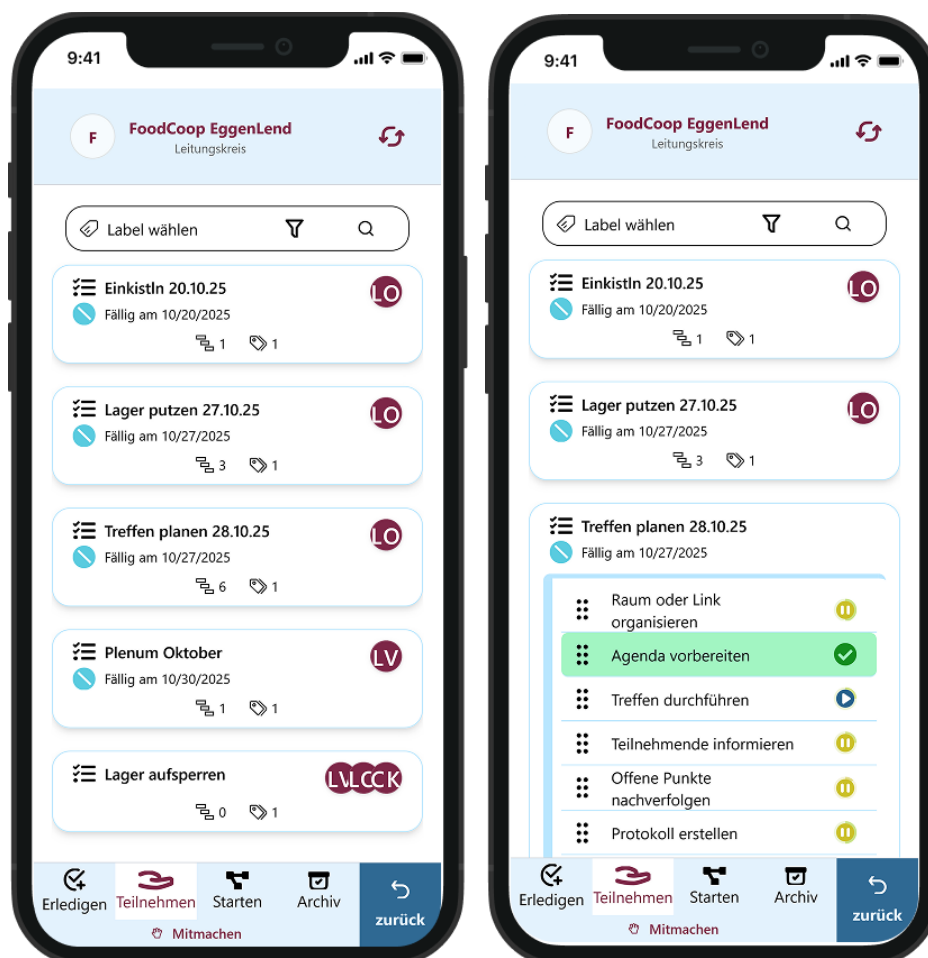


Abbildung 102: Aufgeklappter Vorgang mit zugehörigen Arbeitsschritten im Bereich „Mitmachen“

User Story 81: Bearbeitungsstatus ändern

„Als User möchte ich eintragen können, wenn ich eine konkrete Aufgabe begonnen/pausiert/fortgesetzt/erledigt habe.“

Es werden Vorgänge und ihre Schritte dargestellt. Der Bearbeitungsstatus eines Schrittes kann beispielsweise gestartet, pausiert oder als erledigt markiert werden. Durch einen Klick auf eine Aufgabe bzw. einen Schritt öffnet sich ein Auswahlménü, über das der Status geändert werden kann.

(Abbildung 103)

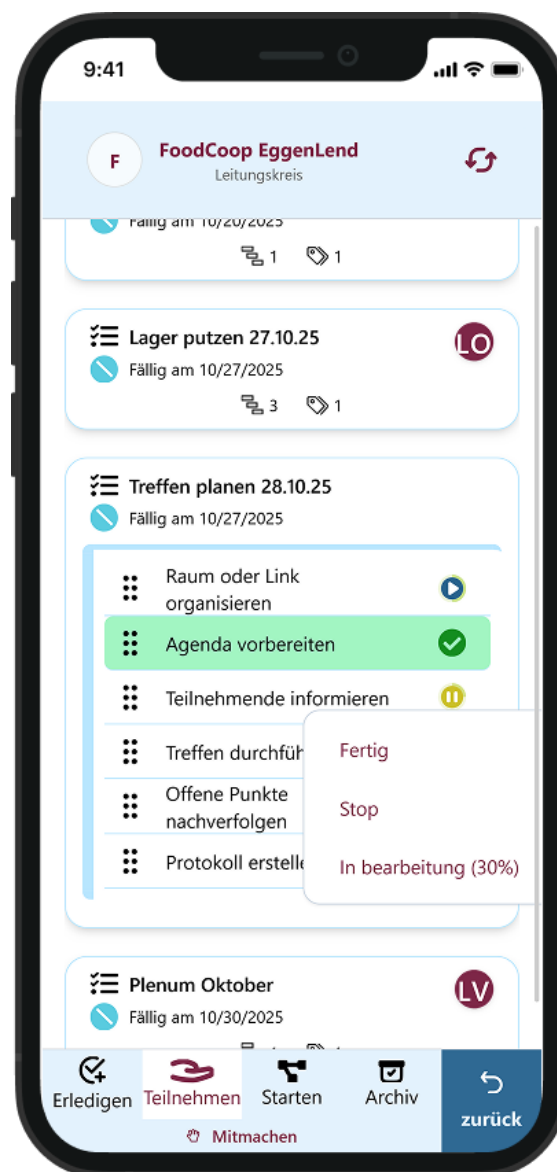


Abbildung 103: Ändern des Bearbeitungsstatus eines Arbeitsschrittes

Je nach aktuellem Zustand werden unterschiedliche Auswahlfelder angeboten. Das heißt, auch wenn die Aufgabe schon als erledigt markiert wurde, kann der Bearbeitungsstatus erneut korrigiert werden. (Abbildung 104)

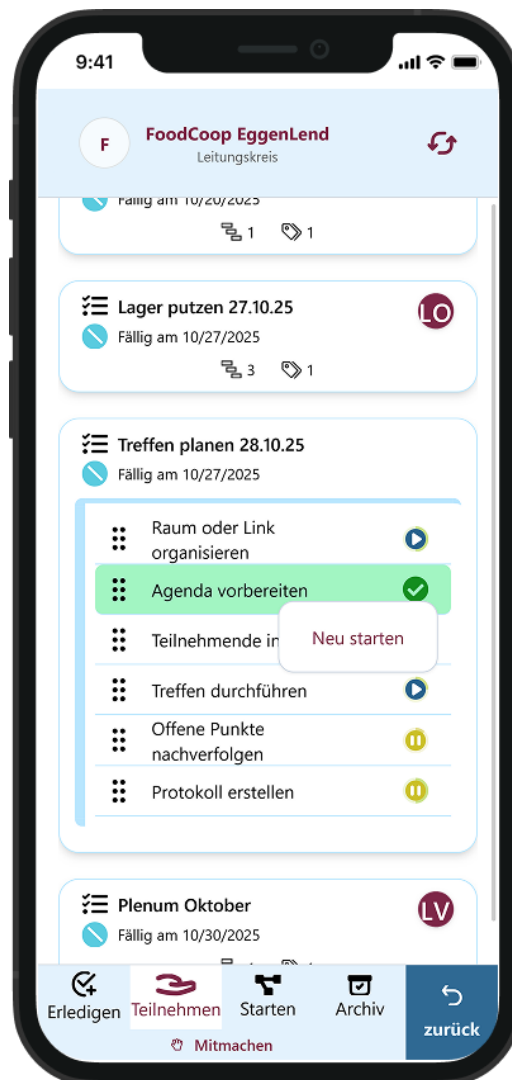


Abbildung 104: Zurücksetzen des Bearbeitungsstatus

Ergänzend zu den beschriebenen User Stories wurden für das Bearbeiten der Schritte innerhalb eines Vorgangs eine Drag-and-Drop-Interaktion vorgesehen. Jede Aufgabe enthält dafür ein Symbol mit sechs Punkten, das als visuelles Bedienelement für die verschiebbare Interaktion dient. Wird dieses Element verwendet, kann die jeweilige Aufgabe innerhalb der dargestellten Struktur per Drag-and-Drop bewegt beziehungsweise neu angeordnet werden. Dadurch wird die direkte visuelle Bearbeitung der Schritte unterstützt und die Interaktion für Nutzer:innen klar erkennbar gemacht.

6.8. Beantwortung der Forschungsfrage

6.8.1. Ziel und Abgrenzung

Die Untersuchung konzentriert sich auf die Möglichkeiten und die Benutzerfreundlichkeit bei der Organisation von Aufgaben und Arbeitsabläufen unter Berücksichtigung der Nutzerbedürfnisse innerhalb eines Frameworks wie React. Diese Prozesse, die mehrere Aufgaben umfassen, beinhalten ein Fälligkeitsdatum und einen Aufgabenstatus. Das heißt, es wird angezeigt, wie sich jede einzelne Aufgabe entwickelt. Aus diesem Grund wird die Drag-and-Drop-Funktion untersucht, da sie den Nutzer:innen hilft, die zahlreichen anstehenden Aufgaben innerhalb der Anwendung selbstständig und bequemer organisieren zu können.

Mit solchen Funktionen sollen mehr Flexibilität, Übersichtlichkeit und Zusammenarbeit zwischen den Nutzer:innen eines Bereichs in coOps lite erreicht werden. Die Technologie *dnd-kit* ist für dieses Ziel besonders geeignet, da sie sich sehr gut in die bereits vorhandenen technischen Abhängigkeiten der App integrieren lässt.

6.8.2. React als Grundlage interaktiver Benutzeroberflächen

Komponentenbasierte Architektur

React bietet die Möglichkeit, wiederverwendbare Elemente beziehungsweise Komponenten zu erstellen; tatsächlich besteht sie im Wesentlichen aus Komponenten. Diese können wiederum ein bestimmtes Erscheinungsbild, ein bestimmtes Verhalten oder eine bestimmte Logik beinhalten. Mit „wiederverwendbar“ ist gemeint, dass dieselbe Komponente auf unterschiedliche Weise, für einen anderen Zweck oder einfach zur Darstellung eines anderen Erscheinungsbildes verwendet werden kann. Dadurch wird zusätzlicher Code vermieden und die Entwicklung vereinfacht. (React a)

Wie bereits erwähnt, wurde die Implementierung im Backend nicht vollständig umgesetzt. Im Frontend wurde jedoch ein Prototyp erstellt, der für die Erstellung und Darstellung von Aufgaben mithilfe der Drag-and-Drop-Funktion in wiederverwendbarer Form konzipiert wurde. Dadurch wird lediglich eine wiederverwendbare Komponente benötigt, um die Drag-and-Drop-Funktionalität für unterschiedliche Aufgaben beziehungsweise Operationen bereitzustellen.

Damit diese Logik wiederverwendet werden kann, können Komponenten und Funktionen in React beziehungsweise TypeScript exportiert werden. (Abbildung 105) Anschließend werden sie in einer anderen Datei importiert und können dort

```
export function OperationCard({
  operation,
  expanded = false,
  onToggleExpanded,
  onOpenSettings,
  onOpenPatternTemplate,
  className,
}): {
```

Abbildung 105: components/coOpsComponents/OperationCard

von weiteren wiederverwendbaren Komponenten genutzt werden, die diese „Draggable“- und „Sortable“-Logik in ihr Verhalten übernehmen.

Der Import muss den richtigen Pfad enthalten, damit die entsprechende Komponente beziehungsweise Funktion korrekt eingebunden werden kann (Abbildung 106):

```
import { OperationCard } from "@components/coOpsComponents/OperationCard";
```

Abbildung 106: Import in orga/konkret

In einer anderen Komponente oder auf einer anderen Seite können anschließend die entsprechenden Funktionen/Properties verwendet werden. (Abbildung 107)

```

{ /* Operations */ }
<div className="space-y-5">
  <div className="rounded-xl bg-white">
    {operationsExist.map((operation) => (
      <OperationCard
        key={operation.id}
        operation={operation}
        expanded={expandedWorkflowId === operation.id}
        onToggleExpanded={toggle}
        onOpenPatternTemplate={(patternId) =>
          navigate(`/orga/patterns${patternId}`)
        }
      />
    ))}
  </div>

```

Abbildung 107: Verwendung in orga/konkret

6.8.3. Wiederverwendbare UI-Komponenten

React in Kombination mit TypeScript eignet sich sehr gut für wiederverwendbare Komponenten in unserem Projekt, da wir damit Werte und/oder Properties sehr konkret definieren können, die anschließend in eine andere Komponente integriert werden können. Auf diese Weise wird kann der Code mit unterschiedlichen Daten wiederverwendet werden.

Mithilfe dieser Properties (Abbildung 108) lassen sich der aktuelle Status und eine mögliche Änderung entsprechend der Aufgabe, zu der dieser Status gehört, erfassen, ohne dass die Properties für jede einzelne Aufgabe separat definiert werden müssen.

```

interface StateStepButtonProps {
  currentState: StepSituation | string;
  onStateChange: (newState: StepSituation) => void;
  disabled?: boolean;
}

```

Abbildung 108: components/coOpsComponents/stateStepSelected

6.8.4. Zustandsverwaltung und dynamische Darstellung

Interaktive Benutzeroberflächen müssen auf Änderungen reagieren können, die während der Nutzung einer Anwendung entstehen. In React können solche veränderlichen Informationen mithilfe eines *State* verwaltet werden. Der *State* beschreibt dabei den aktuellen Zustand bestimmter Daten einer Komponente. Wird dieser Zustand verändert, rendert React die betroffene Komponente erneut und stellt die Benutzeroberfläche entsprechend dem neuen Zustand dar. (React c) (Abbildung 109)

```
const [operations, setOperations] = useState<Operation[]>(() =
```

Abbildung 109: auth/OperationContext

Einfluss des aktuellen Status auf die Darstellung

Die visuelle Darstellung eines Elements kann direkt vom aktuellen *State* abhängig gemacht werden. Bei einer Aufgabe könnte der *State* beispielsweise angeben, ob diese noch offen, in Bearbeitung, pausiert oder abgeschlossen ist. Abhängig von diesem Zustand können unterschiedliche Texte, Icons, Labels oder andere Bedienelemente angezeigt werden.

Dadurch entsteht eine direkte Verbindung zwischen den zugrundeliegenden Daten und ihrer visuellen Darstellung. Ändert sich der *State* einer Aufgabe, kann React automatisch die dazugehörige Darstellung aktualisieren. Der aktuelle Zustand muss somit nicht separat in der Benutzeroberfläche angepasst werden, sondern ergibt sich aus den gespeicherten *State*-Werten. (React c)

Darstellung der möglichen Zustände

Neben dem aktuellen Zustand können in einer Benutzeroberfläche auch mögliche Folgezustände dargestellt werden. Dabei ist es nicht immer sinnvoll, zu jedem Zeitpunkt alle vorhandenen Zustände anzubieten. Welche Zustandsänderungen möglich sind, kann deshalb vom aktuellen *State* abhängig gemacht werden.

Befindet sich eine Aufgabe beispielsweise im Zustand „offen“, könnten als nächste Möglichkeiten „in Bearbeitung“ oder „abgebrochen“ angeboten werden. Eine bereits abgeschlossene Aufgabe kann hingegen andere oder keine weiteren Zustandsänderungen zulassen. Die Benutzeroberfläche kann somit dynamisch nur jene Optionen darstellen, die im jeweiligen Zustand sinnvoll oder erlaubt sind.

Für *Steps-Operations* ist dieses Prinzip besonders relevant, da einzelne Arbeitsschritte („Steps“) häufig einer bestimmten Reihenfolge oder definierten Zustandsübergängen folgen. (React c)

Benutzerinteraktion verändert den State

Eine Zustandsänderung entsteht häufig durch eine direkte Interaktion der Nutzer:in mit der Benutzeroberfläche. Dies kann beispielsweise durch das Anklicken eines Buttons, die Auswahl eines Eintrags oder eine Drag-and-Drop-Interaktion erfolgen.

Durch diese Benutzeraktion wird eine Funktion ausgelöst, die den bisherigen „State“ durch einen neuen Wert ersetzt. Nach der Aktualisierung rendert React die betroffene Benutzeroberfläche erneut. Dadurch wird die Änderung für die Nutzer:in unmittelbar sichtbar.

Dieser Prozess zur Anzeige des Aufgabenstatus ist ein sich wiederholender Vorgang, der entsprechend dem Fortschritt einer Operation aktualisiert wird. (Abbildung 110)



Abbildung 110: Benutzerinteraktion verändert den State

6.8.5. Drag-and-Drop als Interaktionskonzept

Funktionsweise

Wie Microsoft schreibt, bedeutet „Drag and Drop“ auf Deutsch „Ziehen und Ablegen“ (Microsoft, 2026). Je nachdem, um welche Art von Inhalt es sich handelt, kann die Funktionsweise jedoch unterschiedlich umgesetzt werden.

Die „Drag-and-Drop“-Funktion ist hilfreich, wenn zunächst ein Ablageort festgelegt wird, auf den ein bestimmter Inhalt gezogen werden kann. Es gibt auch andere mögliche Situationen, in denen diese Technik, beispielsweise auf einem Laptop, nützlich sein kann.

In unserem Fall ist React die ausgewählte Technologie. Die wichtigsten Grundprinzipien, an denen wir uns dabei orientieren müssen, sind:

- **Draggable Element:** Ein Element, das die Drag-Funktionalität besitzt und auf eine Drag-Interaktion reagieren kann
- **Drop-Ziel:** Der Ablageort beziehungsweise ein Bereich, der auf eine Drag-Interaktion reagiert und einen „Drop“ verarbeiten kann

- Drag-Zustände: Während der Interaktion existiert ein *active*-Element sowie ein mögliches Ziel (*over*) (Abbildung 111)

```
const isDragging = active?.id === id;
const isOver = over?.id === id;
```

Abbildung 111: Drag-Zustände in *components/coOpsComponents/SortableTask*

- Kollisionserkennung: Die Benutzeroberfläche muss erkennen, über welchem Element bzw. Drop-Ziel sich die verschiebbare Aufgabe gerade befindet.
- Datenübertragung: Während des Drag-and-Drop Vorgangs können auch Daten des verschiebbaren Elements für die Interaktion bereitgestellt und verarbeitet werden.
- State Management: In React wird der Zustand nach einer Interaktion aktualisiert, um beispielsweise die Position eines Schrittes innerhalb eines Ablaufs beziehungsweise einer Operation zu verändern. (dnd kit) (Futurezone)

Einsatz bei Steps-Operations

Die Hauptfunktion der Drag-and-Drop Funktion ist in unserem Anwendungsfall „Sortable“, was für die Verwendung in coOps lite das Neuordnen von Aufgaben innerhalb einer *Operation* bedeutet. Dabei ist jede Aufgabe ein „Draggable“-Element, das innerhalb einer Liste neu angeordnet werden kann.

Die Abhängigkeit *dnd-kit/sortable* setzt dieses Verhalten sehr effizient um, da sie individuell anpassbare Funktionen bietet und außerdem keine voreingestellten CSS-Stile enthält, wie sie bei anderen Bibliotheken vorkommen können. Dadurch lässt sie sich nahtlos in das bestehende Tailwind CSS integrieren, ohne fremde Styles überschreiben zu müssen.

Dieses Paket stellt eine Reihe von Funktionen und Hooks bereit, die einen wichtigen Grund für seine Verwendung darstellen. Der wichtigste Hook heißt hier *useSortable* und stellt verschiedene Werte und Funktionen für die Umsetzung eines sortierbaren Elements bereit. Dabei sind unter anderem folgende Aspekte zu beachten:

- *id* und *index*: *id* dient zur eindeutigen Identifizierung eines sortierbaren Elements innerhalb des Drag-and-Drop-Kontexts, während *index* die Position dieses Elements innerhalb einer Liste beschreibt.

- Sensoren: Diese erkennen unterschiedliche Eingabemethoden der Nutzer und ermöglichen die Interaktion mit den verschiebbaren Elementen,

```
const sensors = useSensors(
  useSensor(PointerSensor),
  useSensor(KeyboardSensor, {
    coordinateGetter: sortableKeyboardCoordinates,
  })
);
```

Abbildung 112: *components/coOpsComponents/SortableTask* : *useSensors*

beispielsweise über Maus, Touchscreen oder Tastatur. (Abbildung 112)

- Modifikatoren: Sie modifizieren „Draggable“-Bewegungen mithilfe bereits angepasster Logik, um eine bessere Benutzererfahrung zu ermöglichen. Außerdem können bestimmte Bewegungen begrenzt werden, sodass Komponenten beispielsweise nur horizontal oder vertikal verschoben werden können.
- Transition: Es wird ein animierter Übergang angewandt, wenn sich die Position eines Elements während des Sortiervorgangs verändert.
- Kollisionsdetektor: Diese Funktion eignet sich besonders dazu, Kollisionen bzw. Überschneidungen zwischen „Draggable“- und „Droppable“-Elementen zu erkennen und entsprechend zu behandeln.

Die genannten Funktionen und Eingabemöglichkeiten sind für unsere „Sortable“-Funktion von Aufgaben in coOps lite nützlich. Darüber hinaus gibt es noch viele weitere Funktionen, für deren Verwendung keine umfangreiche Implementierung erforderlich ist. (dnd kit)

Vorteile und Grenzen von Drag and Drop

Die Drag-and-Drop-Funktion ermöglicht eine direkte und visuell intuitive Interaktion mit den Elementen einer Benutzeroberfläche. Insbesondere bei Aufgaben oder Arbeitsschritten, die sortiert werden können, lässt sich die Reihenfolge ohne zusätzliche Formulare oder Eingabefelder ändern. Auf diese Weise können Nutzer:innen beim Verschieben erkennen, welches Element gerade bewegt wird und an welcher Position es abgelegt werden kann. Bibliotheken wie *dnd-kit* unterstützen zudem verschiedene Eingabemethoden wie Maus, Touchscreen oder Tastatur.

Allerdings hat diese Art der Interaktion auch ihre Grenzen. Auf kleineren Touchscreens kann eine präzise Bewegung schwieriger sein. Ebenso können Nutzer:innen mit bestimmten visuellen oder motorischen Einschränkungen möglicherweise nicht sofort erkennen oder bedienen, welche Elemente verschoben werden können. Dies kann zu Problemen oder einer schlechteren Benutzererfahrung führen. Daher sollte im Hinblick auf die Barrierefreiheit die Drag-and-Drop Funktion nicht die einzige Möglichkeit sein, einen Zustand oder eine Reihenfolge zu ändern. Andere Steuerelemente wie Buttons oder die Bedienung über die Tastatur stellen daher eine sinnvolle Ergänzung dar. (W3C Web Accessibility Initiative)

6.8.6. Flexible und kollaborative Aufgaben-Operationen

Visuelle Darstellung und Veränderbarkeit

Drag-and-Drop-Oberflächen ermöglichen es den Nutzer:innen, visuell und intuitiv mit Aufgaben zu interagieren, da sie Inhalte organisieren und Daten nach Bedarf verwalten können. Dieses Thema ist besonders relevant, da dadurch die kognitive Belastung reduziert werden kann.

Ein sehr wichtiger Punkt, den es zu beachten gilt, ist die klare Darstellung dieser verschiebbaren Elemente, da dies den Nutzer:innen hilft, sie leicht zu identifizieren und dadurch Fehler oder Verwirrung bei der Benutzerinteraktion zu vermeiden. Dazu können beispielsweise Animationen, Symbole oder auch unterschiedliche Größen, Farben, Kontraste und/oder Texturen verwendet werden, um hervorzuheben oder anzuzeigen, dass diese Elemente verschiebbar sind.

Ein weiterer wichtiger Punkt, den man unbedingt beachten sollte, ist die Wiederverwendung von Komponenten. Wie bereits zuvor erklärt, lässt sich React sehr flexibel einsetzen, da exportierbaren Komponenten „Properties“ hinzugefügt werden können, um sie mit unterschiedlichen Inhalten und Daten wiederzuverwenden.

Aus diesem Grund wäre es am effizientesten, eine wiederverwendbare Drag-and-Drop-Komponente mit Eigenschaften wie *id*, *children* und möglicherweise *className* zu erstellen, um ihr ein eigenes Styling zu geben. (Abbildung 113)

```
type SortableTaskProps = {
  id: string | number;
  children: React.ReactNode;
  className?: string;
};
```

Abbildung 113:
components/coOpsComponents/SortableTask:
SortableTaskProps

Mithilfe einer *id* kann *dnd-kit* ein Element eindeutig identifizieren und dessen Position innerhalb der sortierbaren Struktur verwalten, wobei *children* ein Schlüsselement für die Wiederverwendbarkeit darstellt. Dadurch wird ein layoutneutrales Design der Drag-and-Drop-Komponente ermöglicht: Die Komponente rendert beispielsweise ein Symbol und reicht den restlichen Inhalt über *children* weiter. Dadurch kann dieselbe Komponente sowohl für *Steps* als auch künftig für andere sortierbare Listen verwendet werden, ohne dass dafür grundlegende Anpassungen an der Komponente erforderlich sind. (AppMaster, 2023)

6.8.7. Einsatz in FoodCoop-Umgebungen

Die Notwendigkeit der Koordination

FoodCoops basieren in hohem Maße auf der gemeinschaftlichen Selbstorganisation ihrer Mitglieder. Dadurch müssen Aufgaben, Zuständigkeiten und Abläufe bzw. *Operations* zwischen mehreren beteiligten Personen mit unterschiedlichen Fähigkeiten koordiniert werden.

Außerdem ist es wichtig zu beachten, dass verschiedene Mitglieder für dieselbe Operation verantwortlich sein können und jedem von ihnen zu diesem Zeitpunkt eine oder mehrere unterschiedliche Aufgaben zugewiesen sein können. Daher ist es wichtig, die beteiligten Nutzer:innen über den Status einer Aufgabe zu informieren bzw. diesen visuell darzustellen. Das bedeutet, sichtbar zu machen, ob eine Aufgabe gerade bearbeitet wird, pausiert ist oder bereits abgeschlossen wurde.

Verbindung zur visuellen Operation-Darstellung

Eine umfassende Darstellung aller wichtigen Daten, die bei jeder *Operation* angezeigt werden sollen – von den einzelnen Schritten bis hin zu den beteiligten Personen – sorgt für mehr Klarheit für jedes am Prozess beteiligte Mitglied.

Das Hauptziel besteht darin, aufzuzeigen, welche Aufgaben verfügbar sind (Transparenz), in welchem Status sich die einzelnen Aufgaben befinden (Übersicht), welche bereits abgeschlossen sind und welche noch zu erledigen sind (Nachvollziehbarkeit).

Flexibilität als Besonderheit von FoodCoops

Da sich verschiedene FoodCoops unterschiedlich organisieren, sollte sich die visuelle Darstellung der Prozesse nicht auf starre Abläufe beschränken. Jede FoodCoop kann Aufgaben anders verteilen wollen, da sich auch die jeweiligen Prozesse unterscheiden können.

Damit ist gemeint, dass es Gruppen gibt, die andere Arten von Tools verwenden, um Prozesse mit Aufgabenzuweisungen zu erstellen. Die ursprüngliche Idee bei coOps lite war es, diese innerhalb der Anwendung übersichtlicher darzustellen. Hierfür ist Drag-and-Drop von großer Bedeutung, beispielsweise wenn ein Dokument in einen Drop-Bereich gezogen und anschließend in eine bestimmte *Operation* importiert werden soll.

Aus diesen Gründen benötigt coOps lite eine größere Flexibilität, als man es von einer herkömmlichen Anwendung gewohnt ist, da sich die Anwendung stärker an unterschiedliche Datenabläufe und Zuständigkeiten anpassen muss. Diese anpassbare Darstellung bietet einen Vorteil gegenüber statischen Komponenten.

6.8.8. Fazit

Diese schriftliche Arbeit zeigt, dass React eine Technologie ist, mit der Komponenten unterschiedliche Daten, Logiken, Funktionen sowie visuelle und interaktive Eigenschaften für einen konkreten Anwendungszweck annehmen können. Genau diese Eigenschaften ermöglichen es, die Darstellung einer Komponente abhängig von ihrem jeweiligen Zustand zu verändern. Diese zustandsbasierte Darstellung bietet eine geeignete Grundlage für interaktive Aufgabenoberflächen.

Drag-and-Drop ergänzt diese dynamische Darstellung von Aufgaben, um eine verbesserte Benutzerinteraktion zu ermöglichen. Dennoch sollte es nicht die einzige Interaktionsmöglichkeit sein, insbesondere in Hinblick auf die Barrierefreiheit. Zusammenfassend bilden React und die Drag-and-Drop-Funktion eine geeignete Kombination für *Steps-Operations*, die nicht ausschließlich statisch dargestellt werden sollen, sondern sich mithilfe verschiedener Zustände anpassen lassen und somit interaktiv bedient werden können.

7. Forschungsthema: Einbindung automatisierter Schritte

Zur Fertigstellung des *Minimum Viable Product* bleibt zunächst Folgendes offen:

- Fertigstellung der Implementation von Vorgängen und Mustern aus Mitmach- und Orga-Perspektive in API-Spezifikation, Backend und Frontend
- Implementation von Backend-Endpoints sowie Frontend-Komponenten zur Verwaltung Sozialer Strukturen
- Implementation der Verwaltung des persönlichen Profils im Kontext eines Bereichs
- Implementation der Verwaltung von Benutzerkonten und Anmeldemethoden wie Single-Sign-On
- Implementation von Benachrichtigungen

Sofern sich das umgesetzte Konzept des MVP im Praxisbetrieb durch FoodCoops als tauglich erweist, kann die Umsetzung weiterer geplanter Funktionalitäten verfolgt werden.

Bereits festgestellt wurde der Bedarf, automatische Schritte in Abläufe einzubinden. Im Folgenden wird untersucht, welche möglichen Probleme zu beachten sind, wenn von Benutzer:innen selbst eingegebener Code ausgeführt wird und mit welchen Strategien diese vermieden werden können. Dafür wurde auch die sichere Speicherung eingegebener Authentifizierungsdaten untersucht.

7.1. Anwendungsfälle für automatische Schritte

Im Kapitel 3.3 wurde analysiert, für welche Prozesse sich die Verwendung von Skripten bei der Foodkoop KrautKOOPf bereits bewährt hat. Zusammengefasst:

- Einlesen übermittelter Preislisten
- Auslesen von Artikeldaten von Webshops
- Übermitteln von Bestellergebnissen an Lieferanten-Webshops
- Durchführen klickintensiver Prozesse in Foodsoft
- Anlegen von Accounts für neue Mitglieder
- Aufbereiten von Daten aus Foodsoft für den öffentlichen Webauftritt der FoodCoop

Außerdem bestünde viel Potenzial zur Reduktion manuellen Aufwandes darin, beim Einkistln die kontrollierten und zugeteilten Mengen digital aufzuzeichnen, statt auf ausgedruckten Bestelllisten zu vermerken. Dadurch könnten sie anschließend automatisiert in die Foodsoft übernommen werden, statt von den Bestelllisten manuell übertragen werden zu müssen. Wie die Kontrolle der Lieferungen im Detail abläuft, unterscheidet sich aber je nach FoodCoop und teilweise je nach Lieferant:in. Eine

Umsetzung mittels dynamisch erstellbarer Abläufe ist daher einer fest programmierten Lösung vorzuziehen.

7.2. Eingabe benötigter Daten

Als essenzielle Voraussetzungen für derartige Automatisierungen haben sich die Eingabe von Standardwerten für Ausführungen sowie die Eingabe von Daten für einzelne Ausführungen herausgestellt.

Die Konfiguration von Standardwerten im Muster einer Automatischen Aktion (*AutomaticOperationPattern*) erspart es, diese entweder vor jedem Durchlauf erneut eingeben oder im Code einprogrammieren zu müssen. So kann der gleiche benutzerdefinierte Code in verschiedenen Abläufen und auch von mehreren FoodCoops eingesetzt werden, was die Wartung des Codes erleichtert.

Die Eingabe von Daten für einzelne Ausführungen ist beispielsweise beim Einlesen einer Preisliste (Hochladen der Datei) oder Anlegen von Accounts auf Basis eines ausgefüllten Anmeldeformulars notwendig. Teilweise könnte dies etwa durch direktes Auslesen des E-Mail-Postfachs obsolet werden. Es ist jedoch nicht davon auszugehen, dass sich alle Übermittlungskanäle (z. B. Messenger-Apps) direkt an coOps lite anbinden lassen.

Außerdem ist für einige Prozesse die Übermittlung von Authentifizierungsdaten notwendig. Beispiele dafür sind:

- Zugangsdaten für einen zugangsgeschützten Lieferanten-Webshop
- Zugangsdaten für interne Tools wie Foodsoft
- API-Schlüssel, sofern auf ein externes API eines Webshops oder internen Tools zugegriffen wird

Da die Authentifizierungsdaten bei der Ausführung der automatischen Aktion im Original benötigt werden, genügt eine Speicherung von Hashwerten nicht. Es werden folgende Varianten in Betracht gezogen:

1. Speicherung der Authentifizierungsdaten in der Konfiguration eines *AutomaticOperationPattern*
2. Eingabe der Authentifizierungsdaten vor jeder Ausführung. Folgt der automatische Schritt direkt auf die Eingabe, müssten die Daten nicht in der Datenbank gespeichert werden.

Im Sinne der Benutzerfreundlichkeit wäre Variante 1 vorzuziehen, damit Benutzer:innen nicht bei jeder Ausführung die Authentifizierungsdaten parat haben müssen. Die Herausforderung, jene Daten sicher

zu verwahren und weiterzugeben würde bei Variante 2 nur an die Benutzer:innen verlagert werden. Außerdem wäre bei Variante 2 keine vollautomatische Durchführung möglich.

Sicherheitsaspekte bei der Speicherung von Authentifizierungsdaten

Eine unverschlüsselte Speicherung von Authentifizierungsdaten in der Datenbank einer auf dem Server der IG FoodCoops betriebenen Instanz von coOps lite würde Sicherheitsrisiken bergen. Datenbank-Administrator:innen hätten Zugriff auf alle gespeicherten Zugangsdaten. Bei einem unbefugten Zugriff, beispielsweise durch Kompromittierung der Zugangsdaten einer Datenbank-Administrator:in, könnten die gespeicherten Authentifizierungsdaten gestohlen und für böswillige Zwecke verwendet werden.

Am Beispiel der Nextcloud-App „Passwords“ wird untersucht, wie Passwort-Manager mit dieser Herausforderung umgehen. In „Passwords“ werden Passwörter verschlüsselt in der serverseitigen Datenbank gespeichert. Beim Start einer Sitzung wird clientseitig die Eingabe eines Masterpasswords verlangt. Zusätzlich wird die Einrichtung einer Zweifaktorauthentifizierung empfohlen. Mithilfe des Masterpasswords werden die Passwörter clientseitig entschlüsselt. (Joos, 2025)

Bei Prozessen, die serverseitig vollautomatisch ablaufen können sollen, ist eine Abhängigkeit von einer benutzerseitigen Sitzung ungeeignet. Das coOps lite Backend müsste hierbei die Rolle des Clients einnehmen. Das Master-Passwort bzw. die Master-Passwörter einzelner FoodCoops müssten nach jedem Neustart des Backend erneut eingegeben werden.

Eine weniger sichere Möglichkeit wäre die Speicherung eines Schlüssels (zur Entschlüsselung von verschlüsselt gespeicherten Daten) als Umgebungsvariable auf dem Server bzw. Docker-Container.

Eine weitere Option wäre die Anbindung eines Secrets-Management-Systems wie HashiCorp Vault. Das coOps lite Backend würde dabei dynamische, kurzlebige Tokens erhalten, die den Zugriff auf die im Secret-Management-System gespeicherten Daten erlauben. (AppRole – HashiCorp Documentation)

Zur Erhöhung der Sicherheit ist auch eine automatisierte regelmäßige Änderung von Passwörtern bzw. API-Schlüsseln in Erwägung zu ziehen. Dies wird als *automated secret rotation* bezeichnet und schränkt die Zeitspanne, innerhalb der kompromittierte Daten für böswillige Zwecke nutzbar sind, ein. (Bodipudi, 2021, S. 291)

7.3. Ausführung von benutzerdefiniertem Code

Die serverseitige Ausführung von durch Benutzer:innen eingegebenen Code ist ein bekanntes Sicherheitsproblem in der Softwareentwicklung. Sofern dies keine beabsichtigte Funktionalität ist, können mittels Reinigung (Sanitization) von Eingabeparametern Gefahren wie Cross Site Scripting oder SQL Injection bekämpft werden. In coOps lite soll die Ausführung von benutzerdefiniertem Code jedoch

bewusst für automatische Prozesse zugelassen werden. Es ist daher zu untersuchen, welche Arten von Risiken und Problemen dies birgt und welche Sicherheitsvorkehrungen getroffen werden können.

Risiken

Benutzerdefinierter Code könnte auf Daten auf dem Server zugreifen. Dabei kann unterschieden werden zwischen Daten, die nicht mit coOps lite zusammenhängen (statische Daten und Daten anderer auf dem Server betriebener Software), Daten von coOps lite und Daten anderer Automatischer Aktionen (etwa anderer Abläufe, auch von anderen FoodCoops). Dazu gehören auch Umgebungsvariablen, API-Keys und Tokens. Daten könnten manipuliert, kopiert oder gelöscht werden. Dies könnte sowohl vorsätzlich als auch unabsichtlich geschehen.

Außerdem ist die Installation von Schadsoftware möglich. Für die Ausführung von benutzerdefiniertem Code kann die Installation bestimmter Versionen von Programmiersprachen und Packages notwendig sein. Zur Vermeidung von Verzögerungen wäre es jedoch vorteilhaft, wenn Benutzer:innen selbständig installieren könnten. Neben der böswilligen Installation von Schadsoftware ist auch die Installation von Packages mit unbekannten Sicherheitslücken oder schadhaftem Code in guter Absicht möglich. Dies wird als Supply-Chain-Risiko bezeichnet. (NIST Glossary)

Bei der Verarbeitung von PDF- und XLS-Dateien ist zu berücksichtigen, dass diese schädliche Makros enthalten können.

Die Ausführung illegaler Aktivitäten durch Benutzer:innen, im böswilligen Fall etwa für Denial-of-Service-Attacken oder das Versenden von Phishing-E-Mails, könnte für den Betreiber des Servers rechtliche Konsequenzen nach sich ziehen. Außerdem könnte dies zu einer Verschlechterung der Domain- und IP-Reputation bis hin zu Blacklisting führen. (Interisle Consulting Group, 2024) Angegriffene Zielseiten könnten zudem die Server-IP-Adresse sperren. Dies wäre auch bei nicht böswilliger Verwendung von Webscraping und Web-Automatisierung mit zu hoher Request-Rate oder Missachtung Anti-Bot-Maßnahmen möglich. (Koster, Illyes, Zeller, & Sassman, 2022)

Die ungebremste Ausführung rechenintensiver Prozesse könnte dazu führen, dass für andere Benutzer:innen und Applikationen nicht mehr genügend Serverkapazitäten zur Verfügung stehen. Dies könnte für Denial-of-Service-Attacken und Zweckentfremdung der Serverleistung missbraucht, aber auch ohne böswillige Absicht ausgelöst werden. Insbesondere unabsichtliche oder böswillige Endlosschleifen sind zu identifizieren.

Maßnahmen

Eine manuelle Kontrolle von neuem bzw. geändertem Code vor der Bereitstellung wäre aufgrund der Verzögerung und des Aufwands für Server-Administrator:innen nicht geeignet. KI-Analysetools könnten Code automatisiert auf problematische Teile prüfen, die Verlässlichkeit wäre jedoch zweifelhaft.

Grundsätzlich ist benutzerdefinierter Code isoliert in von coOps lite separaten und vom Server-Host abgeschirmten Containern auszuführen. Dabei ist auf korrekte Konfiguration der Container und Least-Privilege-Prinzip zu achten. (Souppaya, Morello, & Scarfone, 2017) Container könnten entweder je Bereich (z. B. FoodCoop) erstellt oder dynamisch je individuellem Prozess erstellt und nach Abschluss entfernt werden. Es sind passende CPU-, RAM- und Netzwerk-(Rate-)Limits zu konfigurieren. Für die Verhinderung von Endlosschleifen ist die Konfiguration von Timeouts sinnvoll.

Eine Einschränkung ausgehender Netzwerkanfragen auf bestimmte Ziele (z. B. einer Liste von Webseiten von Lieferant:innen) könnte Missbrauch effektiv verhindern, würde aber Verwaltungsaufwand und Verzögerungen bedeuten. Zumindest ist zu erwägen Netzwerkanfragen verbindlich in Audit-Logs zu dokumentieren, zusammenzufassen und Server-Administrator:innen über verdächtige Muster zu benachrichtigen. Auf diese Einschränkung der Privatsphäre gegenüber Server-Administrator:innen müsste in den Nutzungsbedingungen hingewiesen werden. Auch für die einzelnen Nutzer:innen wäre eine Ansicht ihrer selbst ausgelösten Netzwerkanfragen nützlich.

Für Server-Administrator:innen muss eine Möglichkeit bestehen, Netzwerkziele und Prozesse, die nicht den Nutzungsbedingungen entsprechen, zu sperren. Für besonders harte oder wiederholte Fälle muss auch das Bannen von Benutzer:innen ermöglicht werden.

Ein Leitfaden für benutzerdefinierten Code ist ebenfalls anzudenken. Darin sollte u. a. auf Regeln, die für legales Webscraping zu beachten sind, hingewiesen werden.

8. Fazit und Ausblick

Das Konzept von coOps lite ging über eine reine Aufgabenverwaltung hinaus: Die Anwendung soll Gemeinschaften mit ihren Mitgliedern, Arbeitsgruppen und Beziehungen abbilden, ihre Vernetzung unterstützen und die gemeinsame Organisation erleichtern. Abläufe, Zuständigkeiten und Berechtigungen sollten dabei weitgehend individuell konfigurierbar sein.

Die Umsetzung dieser Anforderungen erwies sich als aufwändiger als zunächst eingeschätzt. Obwohl das Team den Funktionsumfang im Projektverlauf bereits einschränkte, konnte auch dieser reduzierte Umfang innerhalb der Diplomarbeit nicht vollständig umgesetzt werden. Die Unterrichtskenntnisse erleichterten den Einstieg, doch der zusätzliche Lernaufwand, die fehlende Erfahrung mit einem Projekt dieser Größenordnung und die Abstimmung zwischen Frontend und Backend beanspruchten mehr Zeit als vorgesehen. Rückblickend wäre eine konsequentere Eingrenzung der Funktionalitäten und eine frühere Spezifikation der API-Endpoints sinnvoll gewesen.

Mit dem Logischen Datenmodell, der entworfenen Benutzeroberfläche und den bereits implementierten Funktionen wurde eine Grundlage geschaffen, auf deren Basis die Entwicklung fortgesetzt werden kann.

Die FoodCoop-Feedbackgruppe stand dem Vorhaben sehr positiv gegenüber. Ihre Rückfragen machten zugleich deutlich, dass die Anwendung auch für Menschen verständlich sein muss, die sich nicht mit dem zugrunde liegenden Konzept beschäftigt haben. Die vielen Einstellmöglichkeiten dürfen deshalb nicht dazu führen, dass die Organisation mit coOps lite selbst zu einer zusätzlichen Belastung wird.

Als nächster Schritt sollten die begonnenen Funktionen vervollständigt und in einer FoodCoop erprobt werden. Im praktischen Einsatz soll sich zeigen, ob die Verteilung der Aufgaben weniger Abstimmung erfordert und sich mehr Mitglieder an deren Erledigung beteiligen. Diese Erfahrungen sollten bestimmen, welche Erweiterungen anschließend umgesetzt werden.

Der Wert von coOps lite wird sich daran zeigen, ob die Anwendung eine fairere Aufgabenverteilung unterstützt und den Mitgliedern Koordinationsarbeit abnimmt. Die dadurch gewonnene Zeit soll dort ankommen, wo sie gebraucht wird: bei den Menschen, die ihre FoodCoop gemeinsam am Laufen halten.

9. Literaturverzeichnis

- (kein Datum). Abgerufen am 20. 8 2026 von shadcn/ui Documentation: <https://ui.shadcn.com/docs>
- Albe, L. (2026). *When is a function leakproof?* Abgerufen am 22. 7 2026 von Cybertec PostgreSQL Services & Support: <https://www.cybertec-postgresql.com/en/when-is-a-function-leakproof/>
- AppMaster. (12. September 2023). *Designprinzipien für Drag-and-Drop-Schnittstellen*. Abgerufen am 10. September 2026 von AppMaster: <https://appmaster.io/de/blog/designprinzipien-fur-drag-and-drop-schnittstellen>
- AppRole – HashiCorp Documentation*. (kein Datum). Abgerufen am 4. 9 2026 von HashiCorp Developer: <https://developer.hashicorp.com/vault/docs/auth/approle>
- Arhar, E., Seebacher, U., Ostler, L., & Pabst, S. (2023). *FoodCoop 4.0 – Vorstudie*.
- Atlassian a. (kein Datum). *Understand workflows*. Abgerufen am 9. 9 2026 von Atlassian Support: <https://support.atlassian.com/jira-cloud-administration/docs/work-with-issue-workflows/>
- Atlassian b. (kein Datum). *Your complete guide to building an effective risk register*. Abgerufen am 9. 9 2026 von Atlassian: <https://www.atlassian.com/work-management/project-management/risk-register/>
- Belz, M. (2026). *PL/pgSQL-Funktions-Konventionen — Volatilität, RETURNS und die Grenze zur Prozedur*. Abgerufen am 13. 7 2026 von SQL & Beyond – Marcus Belz: <https://sql.marcus-belz.de/plpgsql-funktionen-konventionen/>
- Bodipudi, A. (2021). *Implementing Best Practices and Solutions for Managing Secrets and Credentials in DevSecOps Workflows*. Von https://www.researchgate.net/publication/383414560_Implementing_Best_Practices_and_Solutions_for_Managing_Secrets_and_Credentials_in_DevSecOps_Workflows abgerufen
- Boehm, B. W. (1991). Software risk management: principles and practices. *IEEE Software*, vol. 8, no. 1, 32-41.
- Brunken, Y. (2022). *Was ist Drag and Drop? Einfach erklärt*. Abgerufen am 20. 8 2026 von Chip: https://www.chip.de/ratgeber/software/was-ist-drag-and-drop-einfach-erklaert_bf4e281d-3fc9-44d4-b6e4-526e51f3ab21.html
- dnd kit. (n.d.). *React Quickstart*. Retrieved September 10, 2026, from dnd kit: <https://dndkit.com/react/quickstart/>

dnd kit. (n.d.). *useSortable Hook - React*. Retrieved September 10, 2026, from dnd kit:

<https://dndkit.com/react/hooks/use-sortable/>

FoodCoopShop – Features. (kein Datum). Von FoodCoopShop: <https://www.foodcoopshop.com/die-features-abgerufen>

Hibernate. (kein Datum). *A Guide to Hibernate Query Language*. Abgerufen am 12. 9 2026 von Hibernate Docs: https://docs.hibernate.org/orm/6.5/querylanguage/html_single/#recursive-queries

Hoegl, M., & Gemuenden, H. (2001). Teamwork Quality and the Success of Innovative Projects: A Theoretical Concept and Empirical Evidence. *Organization Science*, 12(4), 435-449.

IG FoodCoops. (2017). Abgerufen am 19. 8 2026 von FoodCoops Österreich: foodcoops.at

IG FoodCoops. (2024). *Software*. Abgerufen am 19. 8 2026 von <https://foodcoops.at/software/>

Interisle Consulting Group. (2024). *Cybercrime Supply Chain 2024: Measurements and Assessments of Cyber Attack Resources and Where Criminals Acquire Them*. Von <https://interisle.net/insights/cybercrimesupplychain2024> abgerufen

Jones, M., & Hardt, D. (2012). *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Internet Engineering Task Force. Von <https://www.rfc-editor.org/rfc/rfc6750.html> abgerufen

Jones, M., Bradley, J., & Sakimura, N. (2015). *JSON Web Token (JWT)*. Internet Engineering Task Force. Von <https://www.rfc-editor.org/rfc/rfc7519.html> abgerufen

Joos, T. (2025). *Passwords-App: Vollwertiger Passwort-Manager für Nextcloud*. Abgerufen am 4. 9 2026 von Cloudcomputing Insider: <https://www.cloudcomputing-insider.de/passwords-app-vollwertiger-passwort-manager-fuer-nextcloud-a-d5294acf471ce81b28c3a1958c525e8d/>

Jovic, B. (2019). *Die Zulässigkeit der wechselseitigen Beteiligung von Kapitalgesellschaften*. Von <https://epub.jku.at/obvulihs/download/pdf/4619943> abgerufen

Koster, M., Illyes, G., Zeller, H., & Sassman, L. (2022). Robots Exclusion Protocol. *RFC 9309*.

MDN contributors. (2025, August 8). *Web application manifest*. Retrieved September 10, 2026, from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Manifest/index.html

MDN contributors. (2026, July 12). *Array.prototype.map() - JavaScript*. Retrieved September 10, 2026, from MDN Web Docs: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map

- Microsoft. (15. July 2026). *Drag & Drop*. Abgerufen am 10. September 2026 von Microsoft Learn:
<https://learn.microsoft.com/de-de/windows/apps/develop/data/drag-and-drop>
- Nielsen, J. (1994, April 24). *10 Usability Heuristics for User Interface Design*. Retrieved September 10, 2026, from Nielsen Norman Group: <https://www.nngroup.com/articles/ten-usability-heuristics/>
- Niestroj, R. (2023). *Ordering columns in a Table in JPA/Hibernate*. Abgerufen am 20. 2 2026 von <https://robertniestroj.hashnode.dev/ordering-columns-in-a-table-in-jpahibernate>
- NIST Glossary*. (kein Datum). Abgerufen am 10. 9 2026 von National Institute Of Standards And Technology: https://csrc.nist.gov/glossary/term/supply_chain_risk
- Omotayo, D. (2024, August 20). *Using React Hot Toast as a Notification Provider for CRUD apps*. Retrieved September 10, 2026, from Refine: <https://refine.dev/blog/react-hot-toast/>
- Oracle. (kein Datum). *Long (Java SE 23)*. Abgerufen am 12. 9 2026 von Oracle Docs: <https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/Long.html>
- Peterjohann, H. (kein Datum). *Das magische Dreieck des Projektmanagements*. Abgerufen am 9. 9 2026 von Peterjohann Consulting: <https://www.peterjohann-consulting.de/magisches-dreieck-des-projektmanagements/>
- Pigeon, T. A. (2025). *React Hook Form with Zod Validation: A Complete Guide with TypeScript*. Abgerufen am 20. 8 2025 von medium.com.
- PostgreSQL a. (kein Datum). *Numeric Types*. Abgerufen am 12. 9 2026 von PostgreSQL Docs: <https://www.postgresql.org/docs/current/datatype-numeric.html>
- PostgreSQL b. (kein Datum). *Date/Time Types*. Abgerufen am 12. 9 2026 von PostgreSQL Docs: <https://www.postgresql.org/docs/current/datatype-datetime.html>
- React a. (kein Datum). *Schnelleinstieg – Komponenten*. Abgerufen am 10. September 2026 von React: <https://de.react.dev/learn#komponenten>
- React b. (n.d.). *useContext*. Retrieved September 10, 2026, from React: <https://react.dev/reference/react/useContext>
- React c. (kein Datum). *useState*. Abgerufen am 10. September 2026 von React: <https://de.react.dev/reference/react/useState>
- react-hot-toast. (n.d.). *<Toaster/> API*. Retrieved September 10, 2026, from react-hot-toast: <https://react-hot-toast.com/docs/toaster>

- Roshi, E. (2026). *9 Best React UI Frameworks for 2026 (Ranked)*. Abgerufen am 18. 8 2026 von <https://codeless.co/best-react-ui-frameworks/>: <https://codeless.co/best-react-ui-frameworks/>
- Schönig, H.-J. (2021). *How the PostgreSQL query optimizer works*. Abgerufen am 13. 7 2026 von Cybertec PostgreSQL Services & Support: <https://www.cybertec-postgresql.com/en/how-the-postgresql-query-optimizer-works/>
- Schwaber, K., & Sutherland, J. (kein Datum). *Der Scrum Guide. Der gültige Leitfaden für Scrum: Die Spielregeln*. Abgerufen am 1. 9 2026 von Scrum Guides: <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-German.pdf>
- Souppaya, M., Morello, J., & Scarfone, K. (2017). Application Container Security Guide. *NIST Special Publication 800-190*, 24-26.
- Spring Security. (kein Datum). *OAuth 2.0 Resource Server JWT*. Abgerufen am 9. 9 2026 von Spring Security Referenzdokumentation: <https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html>
- Tailwind Docs. (kein Datum). *Styling with utility classes*. Abgerufen am 18. 8 2026
- The TypeScript Handbook*. (kein Datum). Abgerufen am 18. 8 2026 von TypeScript: <https://www.typescriptlang.org/docs/handbook/intro.html>
- TypeScript Tutorial. (n.d.). *TypeScript Types*. Retrieved September 10, 2026, from TypeScript Tutorial: <https://www.typescripttutorial.net/typescript-tutorial/typescript-types/>
- Verwijs, C., & Russo, D. (2023). A Theory of Scrum Team Effectiveness. *ACM Transactions on Software Engineering and Methodology*, 32(3), 1-51.
- W3C Web Accessibility Initiative. (n.d.). *Understanding Success Criterion 2.5.7: Dragging Movements*. Retrieved September 10, 2026, from Web Accessibility Initiative (WAI): <https://www.w3.org/WAI/WCAG22/Understanding/dragging-movements>
- Watt, A. (2014). *Project Management*. BCcampus.
- West, D. (kein Datum). *Sprint planning meeting guide*. Abgerufen am 9. 9 2026 von Atlassian.
- Whitten, N. (1999). Duties of the effective project manager. 16.
- Yale University. (n.d.). *Less is More*. Retrieved September 10, 2026, from Usability & Digital Accessibility, Yale University: <https://usability.yale.edu/ux/best-practices/less-is-more>

10. Tabellen- und Abbildungsverzeichnis

Tabellen

Tabelle 1: Endpunkte der Authentifizierung.....	42
Tabelle 2: Implementierte Backend-Tests	82
Tabelle 3: Ergebnisse des Testlaufs vom 10. September 2026 (XML-Export aus IntelliJ IDEA, „Test Results - AuthServiceTest_and_6_more.xml“)	85

Abbildungen

Abbildung 1: Das magische Dreieck des Projektmanagements (Peterjohann, 2026)	9
Abbildung 2: Sechs Dimensionen der Teamwork Quality nach Hoegl und Gemuenden (Eigene Darstellung)	10
Abbildung 3: Schritte des Software-Risikomanagements (Eigene vereinfachte Darstellung in Anlehnung an Boehm (1991, S. 34, Abb. 2); Rückkopplung ergänzt)	13
Abbildung 4: Illustration des FoodCoop-Prinzips (Grafik von gafinen.com unter CC-BY-NC-SA Lizenz)	17
Abbildung 5: Durchlauf einer beispielhaften Sammelbestellung (Klickbarer Prototyp aus FoodCoop 4.0).....	21
Abbildung 6: Ausschnitt aus der User-Story-Datei zum Organisationswechsel (US 74)	26
Abbildung 7: Menüpunkt „Mitmachen“	27
Abbildung 8: Menüpunkt „Orga“	27
Abbildung 9: Gesamtübersicht der in Figma erstellten UI-Entwürfe und Prototypansichten	29
Abbildung 10: coOps lite Logo	29
Abbildung 11: Farbpalette des klickbaren Prototyps	30
Abbildung 12: Hauptmenü im klickbaren Prototyp	30
Abbildung 13: Mitmachen-Submenü im klickbaren Prototyp	30
Abbildung 14: Orga-Submenü im klickbaren Prototyp	30
Abbildung 15: Startseite im klickbaren Prototyp	31
Abbildung 16: Bereichsauswahl im klickbaren Prototyp	32
Abbildung 17: Fähigkeiten-Übersicht unter „Orga – Können“ im klickbaren Prototyp	33
Abbildung 18: Schritte des Musters „Einkistln“ im klickbaren Prototyp	34
Abbildung 19: Auftrittsregel im klickbaren Prototyp	35
Abbildung 20: Automatische Zuweisung mit Rotationsgewichtung im klickbaren Prototyp	36
Abbildung 21: Orga – Konkret im klickbaren Prototyp	37

Abbildung 22: Orga – Verwaltung im klickbaren Prototyp	38
Abbildung 23: Abfrage der Benutzerinformationen ohne Token (Screenshot aus Postman)	44
Abbildung 24: Abfrage der Benutzerinformationen mit gültigem Token	44
Abbildung 25: Entity-Relationship-Diagramm des entworfenen Logischen Modells	46
Abbildung 26: Many-to-Many-Attribut tags in model.OperationPattern	71
Abbildung 27: JPA-Entity SocialEntity (Auszug)	72
Abbildung 28: PostgreSQL-Tabelle social_entities (Auszug)	72
Abbildung 29: PostgreSQL-View ev_social_entities	73
Abbildung 30: Beispiel eines DDL-Skripts	77
Abbildung 31: Virtuelles Feld in SocialEntityRelation.java	78
Abbildung 32: Dynamische Erstellung einer reinen SQL-Funktion (Auszug) in /23_functions+views/015f_get_permissions_as_array.sql	78
Abbildung 33: Funktionskörper von list_labels_minimal (p_user_id BIGINT, p_space_id BIGINT) RETURNS JSONB	80
Abbildung 34: Funktionskörper von get_social_entity_ancestor_ids (p_id BIGINT) RETURNS TABLE (id BIGINT)	80
Abbildung 35: Übersicht verwendeter Benutzeroberflächenelemente (eigene Darstellung)	88
Abbildung 36: Layout je nach Größe des Geräts	88
Abbildung 37: Toast-Beispiel (react-hot-toast)	91
Abbildung 38: Einbindung des Toaster	91
Abbildung 39: coOps lite Frontend Struktur	92
Abbildung 40: type Props in coOpsComponents/GroupSwitcher	95
Abbildung 41: spaces in coOpsComponents/GroupSwitcher	95
Abbildung 42: map() in coOpsComponents/GroupSwitcher	96
Abbildung 43: Anzeige des Bereichsnamen in coOpsComponents/GroupSwitcher	96
Abbildung 44: auth/AuthContext	96
Abbildung 45: auth/AuthProvider	97
Abbildung 46: useAuth in auth/AuthContext	97
Abbildung 47: Verwendung von useAuth in pages/Login	97
Abbildung 48: getAvailableStates in components/coOpscomponents/stateStepSelected	98
Abbildung 49: components/coOpscomponents/stateStepSelected: In Bearbeitung	99
Abbildung 50: components/coOpscomponents/stateStepSelected: Pausiert	99
Abbildung 51: components/coOpscomponents/stateStepSelected: Neuer Status	99
Abbildung 52: components/coOpscomponents/stateStepSelected: visuelle Icons	100
Abbildung 53: components/coOpscomponents/stateStepSelected: klickbare Status-Änderung	100

Abbildung 54: src/main : zentrale toast Funktion	101
Abbildung 55: src/lib : Hilfsfunktionen um verschiedene Meldungen zu laden	101
Abbildung 56: src/lib : Asynchrone Funktion.....	102
Abbildung 57: components/coOpscomponents/SpaceManagementCard: handleSaveSpaceField..	102
Abbildung 58: components/coOpscomponents/SpaceManagementCard: Typ hinzugefügt	103
Abbildung 59: components/coOpscomponents/createOperationForm: Titel und Beschreibung	104
Abbildung 60: components/coOpscomponents/createOperationForm: Verbindung über zodResolver	104
Abbildung 61: components/coOpscomponents/createOperationForm: Anbindung React Hook - Controller	105
Abbildung 62: createOperationForm: Auswahl des Vorgangstyps	106
Abbildung 63: createOperationForm: Validierung	106
Abbildung 64: createOperationForm: Absenden	107
Abbildung 65: createOperationForm: Validierung	107
Abbildung 66: pages/orga/SkillsOverview: Navigation zwischen Fähigkeiten und Einschränkungen .	108
Abbildung 67: pages/orga/SkillsOverview: isActive	108
Abbildung 68: src/App.tsx: Routen für Skills und Impairments.....	108
Abbildung 69: Berechtigungsprüfung Skills & Impairments ansehen.....	109
Abbildung 70: components/coOpscomponents/AnimatedPage: pageVariants.....	109
Abbildung 71: components/coOpscomponents/AnimatedPage: motion.div	110
Abbildung 72: src/main.tsx: AnimatedPage/Outlet.....	110
Abbildung 73: Struktur der API-Antwort bei erfolgreicher Benutzeranmeldung	115
Abbildung 74: auth/types	115
Abbildung 75: auth/AuthContext.....	116
Abbildung 77: auth/AuthProvider: loginAs	117
Abbildung 76: auth/AuthProvider: Aufruf ans Backend	117
Abbildung 78: auth/AuthProvider: loadUserInfo	118
Abbildung 79: auth/AuthContext.....	118
Abbildung 80: pages/Login	118
Abbildung 81: Authentifizierungszyklus im Frontend (eigene Darstellung)	119
Abbildung 82: components/coOpscomponents/GroupSwitcher: Props.....	120
Abbildung 83: components/coOpscomponents/GroupSwitcher: Wechsel auslösen.....	120
Abbildung 84: auth/AuthProvider: Zustand aktualisieren	120
Abbildung 85: layouts.....	121
Abbildung 86: Wechsel des Bereichs (Komponente)	121

Abbildung 87: auth/AuthProvider: Berechtigungsfunktion	122
Abbildung 88: Berechtigungsverwendung.....	122
Abbildung 89: Initialisierung der Anwendung und Einbindung des BrowserRouter in main.tsx	123
Abbildung 90: Definition der geschützten Haupt- und Unterrouen in App.tsx	123
Abbildung 91: Verwendung von Outlet zur Darstellung verschachtelter Routen.....	124
Abbildung 92: Menüpunkt „Mitmachen“ im Footer des Hauptlayouts.....	124
Abbildung 93: Zehn Usability Heuristiken (kreativviertel.at, 2025)	126
Abbildung 94: Erfolgsmeldung in Orga – Verwaltung	126
Abbildung 95: Zurücksetzen oder Speichern (Orga – Konkret).....	127
Abbildung 96: components/ui/Button	127
Abbildung 97: Formular „Vorgang erstellen“	128
Abbildung 98: Label-Filter	128
Abbildung 99: Button zur Suchfunktion	128
Abbildung 100: Route zur NotFoundPage	130
Abbildung 101: /pages/error	130
Abbildung 102: Aufgeklappter Vorgang mit zugehörigen Arbeitsschritten im Bereich „Mitmachen“ ..	131
Abbildung 103: Ändern des Bearbeitungsstatus eines Arbeitsschrittes	132
Abbildung 104: Zurücksetzen des Bearbeitungsstatus	133
Abbildung 105: components/coOpsComponents/OperationCard	134
Abbildung 106: Import in orga/konkret	135
Abbildung 107: Verwendung in orga/konkret.....	135
Abbildung 108: components/coOpsComponents/stateStepSelected.....	135
Abbildung 109: auth/OperationContext.....	136
Abbildung 110: Benutzerinteraktion verändert den State	137
Abbildung 111: Drag-Zustände in components/coOpsComponents/SortableTask	138
Abbildung 112: components/coOpsComponents/SortableTask : useSensors	138
Abbildung 113: components/coOpsComponents/SortableTask: SortableTaskProps	140

11. Abkürzungs- und Symbolverzeichnis

Allgemeine Fachbegriffe

API	Application Programming Interface	Programmierschnittstelle einer Anwendung
CPU	Central processing unit	Prozessor, zentrale Recheneinheit eines Computers
CSS	Cascading Style Sheets	Stylesheet-Sprache für Weboberflächen
CTE	Common Table Expression	SQL-Tabellenausdruck, der innerhalb eines SQL-Statements definiert wird
DDL	Data Definition Language	Sprache zur Erstellung von Datenbankobjekten
EDV	Elektronische Datenverarbeitung	
ERD	Entity-Relationship-Diagram	Grafische Darstellung von Entitäts- und Beziehungstypen in der Datenbankmodellierung
HQL	Hibernate Query Language	SQL-ähnliche Abfragesprache des Java-Framework Hibernate
HTML	HyperText Markup Language	Sprache zur Strukturierung von Dokumenten und Webseiten
HTTP(S)	Hypertext Transfer Protocol (Secure)	Internet-Netzwerkprotokolle
ID	Identifizier	Identifikator-Attribut eines Objekts
JDBC	Java Database Connectivity	Datenbankschnittstelle von Java
JOOQ	jOOQ Object Oriented Querying	Java-Framework für Datenbankannotation
JPA	Jakarta Persistence (API)	Java-Framework für Datenbankannotation
JPQL	Jakarta Persistence query language	SQL-ähnliche Abfragesprache des Java-Framework JPA
JSON(B)	JavaScript Object Notation (Binary)	Textbasiertes Datenformat für Datenaustausch
JWT	JSON Web Token	Offener Standard für sicheren und verifizierbaren Austausch von Claims in Form eines JSON-Objekts
MVP	Minimum Viable Product	Minimal funktionsfähige Iteration einer Software, die Nutzer:innen brauchbaren Nutzen bietet
ORM	Object-Relational Mapping	Verknüpfung von Objekten aus objektorientierten Programmiersprachen und relationalen Datensätzen
O(...)	Big O notation	Landau-Symbol zur Beschreibung der Größenordnung einer Funktion, Domäne oder Anzahl
PL/pgSQL	Procedural Language/PostgreSQL	Prozedurale Programmiersprache von PostgreSQL, ähnlich wie PL/SQL von Oracle
PWA	Progressive Web App	Bestimmter Typ einer Webanwendung, offlinefähig
RAM	Random-access memory	Arbeitsspeicher eines Computers

SSO	Single sign-on	Einmalige Anmeldung einer Benutzer:in für mehrere Anwendungen
SQL		Datenbank-Abfragesprache
SVG	Scalable Vector Graphics	Dateiformat für skalierbare Vektorgrafiken
UI	User Interface	Benutzeroberfläche (einer Software-Anwendung)
US	User Story	Software-Anforderung, aus Nutzersicht formuliert
UX	User Experience	Nutzungserfahrung, etwa bei der Interaktion mit einer Anwendung

Projektspezifische Begriffe

AG	Arbeitsgruppe	
FC	FoodCoop	Zusammenschluss von Personen, die gemeinsam und selbstorganisiert Lebensmittel beziehen
FFG	FoodCoop-Feedback-Gruppe	Gruppe bestehend aus Mitgliedern mehrerer FoodCoops, gebildet um dem Projektteam Feedback zu geben
IG	IG FoodCoops	Interessengemeinschaft der FoodCoops Österreich (Auftraggeber)
Orga	Organisation/Koordination	
PS	Permission Set	Berechtigungsset (Sammlung von Berechtigungen)
PSA	Permission Set Attribution	Direkte Attribuierung eines Berechtigungssets zu einer SE
PSABRT	Permission Set Attribution By Relation Type	Attribuierung eines Berechtigungssets zu diversen SEs auf Basis von Beziehungen eines bestimmten Typs
PSABRRT	Permission Set Attribution By Relation Role Type	Attribuierung eines Berechtigungssets zu diversen SEs auf Basis von Beziehungsrollen eines bestimmten Typs
SE	Social Entity	Soziale Entität – Überbegriff für Personen, Haushalte, FoodCoops, Arbeitsgruppen, Lieferant:innen ... Jede SE bildet einen eigenen Bereich (Space)
SET	Social Entity Type	Art, Rechtsform oder Gegenstand einer SE
SETA	Social Entity Type Attribution	Typ-Zugehörigkeit einer bestimmten SE
SER	Social Entity Relation	Beziehung einer SE zu einer anderen SE
SERT	Social Entity Relation Type	Beziehungstyp
SERR	Social Entity Relation Role	Beziehungsrolle, die eine bestimmte SE bei einer bestimmten Beziehung einnimmt
SERRT	Social Entity Relation Role Type	Beziehungsrollentyp

12. Anhang

12.1. User Stories

Priorisierung	Nr	Als ...	... möchte ich ...
Must have	1	Admin	Stammdaten der Mitglieder verwalten können.
Must have	2	Admin	eintragen können, ob ein Mitglied aktiv oder inaktiv ist.
Should have (später)	3	?	eine Organisation (z.B. FoodCoop) mit wenigen Klicks erstellen können.
Must have	4	Admin	eintragen können, ab wann ein Mitglied aktiv bzw. inaktiv sein wird (sprich, Mitgliedschaftspausen eintragen).
Must have	6	Admin	Berechtigungen von Mitgliedern verwalten können.
Should have	8	Admin	verschiedene Arten von Mitgliedern unterscheiden können (benutzerdefiniert, z.B. Probemitglied, Kernmitglied, Gast ...)
Should have	10	Admin	Kooperationspartner:innen hinzufügen/einladen können.
Should have	12	Admin	eine Synchronisation mit der Foodsoft-Instanz meiner FoodCoop konfigurieren können: Wenn in der Foodsoft ein neues Mitglied angelegt wird, soll es automatisch auch in coOps lite angelegt werden.
Should have	13	Admin	eine Synchronisation mit der Foodsoft-Instanz meiner FoodCoop konfigurieren können: Wenn in coOps lite ein neues Mitglied angelegt wird, soll es automatisch auch in der Foodsoft angelegt werden.
Must have	15	Admin	die Stammdaten eines Space einsehen und bearbeiten können.
Must have	18	Koordinator:in	eine Arbeitsgruppe bzw. Sub-Space mit wenigen Klicks erstellen können.
Must have	19	Koordinator:in	nachschauen können, welche Mitglieder es gibt.
Must have	20	Koordinator:in	Fähigkeiten, die innerhalb einer Organisation/Space relevant sind, verwalten können (anlegen, bearbeiten, löschen).
Must have	21	Koordinator:in	nachschauen können, welche Fähigkeiten ein bestimmtes Mitglied hat.
Must have	24	Koordinator:in	nachschauen können, welche Mitglieder eine bestimmte Fähigkeit haben.
Must have	25	Koordinator:in	Fähigkeiten an Mitglieder manuell vergeben und entziehen können.
Must have	26	Koordinator:in	einstellen können, dass eine Fähigkeit nach "n erledigten Aufgaben vom Typ y" automatisch erlangt wird.
Must have	27	Koordinator:in	Musteraufgaben/-abläufe erstellen, duplizieren, bearbeiten und löschen können.
Must have	28	Koordinator:in	einstellen können, wer eine bestimmte Musteraufgabe/-ablauf bearbeiten darf (sofern ich sie selbst bearbeiten darf).
Must have	29	Koordinator:in	nachschauen können, welche Musteraufgaben/-abläufe es in der FoodCoop/Arbeitsgruppe/Space gibt.
Must have	31	Koordinator:in	abbilden können, dass eine Musteraufgabe/-ablauf aus mehreren Schritten besteht (mit drag & drop).
Must have	32	Koordinator:in	je Schritt wiederum alle Features von Musteraufgaben/-abläufen nutzen können.
Must have	33	Koordinator:in	konfigurieren können, wann eine Musteraufgabe/-ablauf auftritt, z.B. jeden zweitletzten Freitag im Monat.
Must have	34	Koordinator:in	in einer Musteraufgabe einstellen können, wie viele Mitgliederpersonen mindestens nötig sind und wie viele zugewiesen werden sollen.
Must have	35	Koordinator:in	in einer Musteraufgabe einstellen können, welche Fähigkeiten nötig sind.
Must have	36	Koordinator:in	konfigurieren können, dass Iterationen einer Musteraufgabe/-ablauf (also die konkreten Aufgaben/Termine) automatisch an geeignete Mitgliederpersonen zugeteilt werden (nach Fähigkeiten und Fairness).
Must have	37	Koordinator:in	einstellen können, wann eine Musteraufgabe/-ablauf generell fällig wird, z.B. samstags um 12 Uhr bzw. 2 Tage vor/nach einem anderen Schritt.

Must have	38	Koordinator:in	relative Zeitabstände zwischen Schritten in einem Musterablauf einstellen könnten, d.h. ab wann ein Schritt erledigt werden darf und wann er fällig ist – z.B. „darf frühestens 2 Tage, nachdem Schritt x erledigt wurde, erledigt werden“ oder „ist fällig 5 Stunden vor Schritt x“ oder „am Freitag vor/nach Schritt x“
Must have (evtl später)	39	Koordinator:in	in einer Musteraufgabe (Text einzeilig/mehrzeilig/Passwort/E-Mailadresse/etc, Zahl, Datei(en), Auswahlfeld, Datum, Datum+Uhrzeit ...) konfigurieren, die bei jeder Iteration ausgefüllt werden können/müssen, um Ergebnisse zu dokumentieren oder Variablen einzugeben.
Must have (evtl später)	40	Koordinator:in	in einem Musterablauf Fixwerte bzw. Standardwerte konfigurieren, die nicht bei jeder Iteration neu ausgefüllt werden sollen.
Must have (evtl später)	41	Koordinator:in	vollautomatische Prozesse (als Schritt in einem Musterablauf oder eigenständig) konfigurieren, bei denen Custom Code in einer beliebigen Programmiersprache ausgeführt wird und Ergebnisse zurückgeliefert werden.
Must have	42	Koordinator:in	in einer Musteraufgabe, die einen Schritt in einem Musterablauf darstellt, einstellen können, welche anderen Schritte zuvor erledigt werden müssen.
Must have	43	Koordinator:in	verschiedene Typen von Musteraufgaben/-abläufen miteinander verknüpfen können. (Beispiel: Einkistln dieser Woche, zugewiesen an bestimmte Personen, soll Einkistln einzelner Bestellungen derjenigen Woche enthalten)
Must have	44	Koordinator:in	Musteraufgaben/-abläufe für andere FoodCoops/Arbeitsgruppen zum Duplizieren freigeben (d.h. eine andere FoodCoop kann einen Ablauf als Vorlage verwenden).
Must have	45	Koordinator:in	Benachrichtigungen bei Änderungen an ausgewählten Musteraufgaben/-abläufen erhalten.
Must have	46	Koordinator:in	Musteraufgaben/-abläufe mit Labels kennzeichnen.
Must have	47	Koordinator:in	Musteraufgaben/-abläufe nach Labels gefiltert auflisten lassen.
Must have	48	Koordinator:in	einzelne (konkrete) Aufgaben/Projekte erstellen, duplizieren, bearbeiten und löschen können - auf Basis einer Musteraufgabe/-ablauf oder von Null auf.
Must have	49	Koordinator:in	einstellen können, wer eine bestimmte konkrete Aufgabe/Projekt bearbeiten darf (sofern ich sie selbst bearbeiten darf).
Must have	50	Koordinator:in	nachschauen können, welche konkreten Aufgaben/Projekte es in der FoodCoop/Arbeitsgruppe/Space gibt.
Must have	51	Koordinator:in	abbilden können, dass eine konkrete Aufgabe/Projekt aus mehreren Schritten besteht (mit drag & drop).
Must have	52	Koordinator:in	je Schritt wiederum alle Features von konkreten Aufgaben/Projekten nutzen können.
Must have	53	Koordinator:in	Wiederholungsschleifen für Schritte einer konkreten Aufgabe/Projekts (d.h. wiederholen bis abgebrochen wird) konfigurieren können.
Must have	54	Koordinator:in	in einer konkreten Aufgabe/Projekt einstellen können, wie viele Mitgliederpersonen mindestens nötig sind und wie viele zugewiesen werden sollen.
Must have	55	Koordinator:in	in einer konkreten Aufgabe/Projekt einstellen können, welche Fähigkeiten nötig sind.
Must have	56	Koordinator:in	Konkrete Aufgaben/Projekte bestimmten Mitglieder(-personen) oder einer Arbeitsgruppe zuweisen.
Must have	57	Koordinator:in	Zuweisungen von konkreten Aufgaben/Projekten aufheben oder ändern.
Must have	58	Koordinator:in	einstellen können, wann eine konkreten Aufgabe/Projekt fällig wird, z.B. Datum+Uhrzeit bzw. 2 Tage vor/nach einem anderen Schritt.
Must have	59	Koordinator:in	relative Zeitabstände zwischen Schritten in einer konkreten Aufgabe/Projekt einstellen könnten, d.h. ab wann ein Schritt erledigt werden darf und wann er fällig ist – z.B. „darf frühestens 2 Tage, nachdem Schritt x erledigt wurde, erledigt werden“ oder „ist fällig 5 Stunden vor Schritt x“ oder „am Freitag vor/nach Schritt x“
Must have (evtl später)	60	Koordinator:in	in einer konkreten Aufgabe Formularfelder (Text einzeilig/mehrzeilig/Passwort/E-Mailadresse/etc, Zahl, Datei(en), Auswahlfeld, Datum, Datum+Uhrzeit ...) konfigurieren, die ausgefüllt werden können/müssen, um Ergebnisse zu dokumentieren oder Variablen einzugeben.

Must have (evtl später)	61	Koordinator:in	in einer konkreten Aufgabe/Projekt Variablen/-werte setzen, sofern sie von den Standardwerten abweichen bzw. wenn es eine einmalige Aufgabe/Projekt ist.
Must have (evtl später)	62	Koordinator:in	vollautomatische Prozesse (als Schritt in einem "Projekt" oder eigenständig) konfigurieren, bei denen Custom Code in einer beliebigen Programmiersprache ausgeführt wird und Ergebnisse zurückgeliefert werden.
Must have	63	Koordinator:in	in einer konkreten Aufgabe, die einen Schritt in einem konkreten Projekt darstellt, einstellen können, welche anderen Schritte zuvor erledigt werden müssen.
Must have	64	Koordinator:in	verschiedene Stufen von Verbindlichkeit von konkreten "Aufgaben/Projekten" unterscheiden können, von verbindlichen Aufgaben bis hin zu völlig unverbindlichen Einladungen (z.B. an einer Sammelbestellung teilzunehmen).
Must have	65	Koordinator:in	verschiedene Typen von konkreten Aufgaben/Projekten miteinander verknüpfen können.
Must have	66	Koordinator:in	Benachrichtigungen bei Änderungen an ausgewählten konkreten Aufgaben/Projekten erhalten.
Must have	67	Koordinator:in	konkrete Aufgaben/Projekte mit Labels kennzeichnen.
Must have	68	Koordinator:in	konkrete Aufgaben/Projekte nach Labels gefiltert auflisten lassen.
Should have	69	Koordinator:in	Wiederholungsschleifen für Schritte eines Musterablaufs (d.h. wiederholen bis abgebrochen wird) konfigurieren können.
Should have	70	Koordinator:in	verschiedene Stufen von Verbindlichkeit von Musteraufgaben/-abläufen unterscheiden können, von verbindlichen Aufgaben bis hin zu völlig unverbindlichen Einladungen (z.B. an einer Sammelbestellung teilzunehmen).
Should have	71	Koordinator:in	Musteraufgaben/-abläufe anhand von Suchbegriffen finden.
Should have	72	Koordinator:in	konkrete Aufgaben/Projekte anhand von Suchbegriffen finden.
Must have	73	User	mich möglichst unkompliziert anmelden können, wenn ich bereits einen Foodsoft-Account habe.
Must have	74	User	zwischen verschiedenen Organisationen (IG FoodCoops, meinen FoodCoop(s)) hin- und her wechseln können, ohne mich neu einloggen zu müssen.
Must have	75	User	bereits in der Auflistung meiner Organisationen/Spaces sehen können, in welchem Space es etwas neues (Benachrichtigung, Aufgabe ... in verschiedenen Farben) gibt, das mich betrifft.
Must have	76	User	eine Liste an konkreten Aufgaben/Projekten, für die ich verantwortlich bin (insb. dringende), angezeigt bekommen.
Must have	77	User	eine Liste an konkreten Aufgaben/Projekten, für die noch Personen gesucht werden und die für mich geeignet wären (insb. dringende & insb. aus meinen Arbeitsgruppen), angezeigt bekommen.
Must have	78	User	eine Liste an konkreten Aufgaben/Projekten, die anderen Personen zugewiesen sind, aber schon länger überfällig sind und für mich geeignet wären, angezeigt bekommen.
Must have	79	User	ein Archiv an von mir erledigten konkreten Aufgaben/Projekten abrufen können, um z.B. Korrekturen vorzunehmen.
Must have	80	User	mich für eine für mich geeignete konkrete Aufgabe/Projekte eintragen können.
Must have	81	User	eintragen können, wenn ich eine konkrete Aufgabe begonnen/pausiert/fortgesetzt/erledigt habe.
Must have	82	User	meinen Fortschritt bzw. Ergebnisse zu einer konkreten Aufgabe dokumentieren können.
Must have	83	User	erfahren, wenn ich eine mir zugewiesene konkrete Aufgabe nun erledigen kann.
Must have	85	User	automatisierte Prozesse per Klick starten/durchführen können.
Must have	86	User	Ergebnisse von vollautomatischen Prozessen (die für mich relevant sind) angezeigt bekommen.
Must have	87	User	erfahren, wenn mir eine konkrete Aufgabe zugewiesen wurde / welchen Termin ich mir dafür freihalten soll.
Must have	88	User	rechtzeitig erinnert werden, wenn Aufgaben/Termine, für die ich zuständig bin, fällig werden bzw. nahen.

Must have	89	User	Kontaktdaten eines Mitglieds abrufen können, mit dem ich gemeinsam für eine konkrete Aufgabe eingetragen bin oder das für einen anderen Schritt im selben "Projekt" zuständig ist/war.
Must have	90	User	andere Schritte in einem "Projekt", in dem ich für einen Schritt zuständig bin, ansehen können, sofern sie mich betreffen - wer zuständig ist, ob/wann erledigt, wann fällig, ausgefüllte Formularfelder ...
Must have	91	User	an überfällige Aufgaben, für die ich zuständig bin, erinnert werden (bzw. erinnert werden, diese auch als erledigt zu markieren).
Must have	92	User	einstellen können, in welchen Fällen ich Benachrichtigungen bekommen möchte und auf welchen Wegen (Desktop-Benachrichtigung, Push-Notification, E-Mail).
Must have	93	User	Push-Benachrichtigungen erhalten, sofern ich das so eingestellt habe.
Must have	94	User	E-Mail-Benachrichtigungen erhalten, sofern ich das so eingestellt habe.
Should have	95	User	zusätzliche Schritte (Subaufgaben) in einer mir zugewiesenen konkreten Aufgabe einfügen können.
Should have	96	User	melden, wenn ich eine mir zugewiesene Aufgabe nicht wahrnehmen kann bzw. anfragen, die Aufgabe gegen eine für mich geeignete Aufgabe/Termin einzutauschen.
Should have	97	User	erfahren, wenn es eine Zuständigkeits-Tauschanfrage von einem anderen Mitglied an mich gibt und diese annehmen/ablehnen können.
Should have	98	User	erfahren, wenn ich angefragt wurde, eine konkrete Aufgabe zu übernehmen und dies annehmen/ablehnen.
Should have	99	User	erfahren, wenn eine von mir ausgehende Zuständigkeits-Tauschanfrage angenommen bzw. abgelehnt wurde.
Should have	100	User	nachschauen können, welche Fähigkeiten mir anerkannt wurden und eintragen, was ich zusätzlich kann (sofern Selbstzuweisung für diejenige Fähigkeit zugelassen ist)
Must have	101	User bzw. Koordinator:in	den ungefähren Fortschritt einer konkreten Aufgabe/Projekt auf einen Blick sehen können.
Must have	102	User bzw. Koordinator:in	sehen können, welche Voraussetzungen für eine konkreten Aufgabe/Projekt erfüllt bzw. noch nicht erfüllt sind.
Must have	103	User bzw. Koordinator:in	sehen können, wann eine konkrete Aufgabe/Projekt fällig wird bzw. wie lange sie schon überfällig ist.
Should have	104	User bzw. Koordinator:in	erfahren, wenn eine von mir ausgehende Zuständigkeits-Anfrage angenommen bzw. abgelehnt wurde.
Must have	112	Koordinator:in	die Details einer Aufgabe einsehen können, insbesondere die zugewiesenen Benutzer, Labels und erforderlichen Fähigkeiten, damit ich den Ablauf und die Zuständigkeiten vollständig nachvollziehen kann.

12.2. Tabellarischer Vergleich von Vorgangstypen

Englisch, Code	Operation	Task	Opportunity	Automatic	Happening
Deutsch, UI >	Vorgang	Aufgabe	Gelegenheit	Automatische Aktion (nicht in MVP)	Ereignis
Erläuterung >	Überbegriff für Aufgabe, Gelegenheit, Autom. Aktion und Ereignis. Kann verschachtelt sein. Z.B. Projekt, das aus Vorgängen verschiedener Typen besteht.	Etwas, das erledigt werden sollte (manuell), mit Relevanz für die Gruppe & gewisser Verbindlichkeit.	Etwas, das nach Wunsch in Anspruch genommen werden kann, Relevanz primär für einen selbst.	Etwas, das vollautomatisch durch Ausführung von (Custom) Code erledigt wird.	Etwas, das von niemandem in der App gesteuert werden kann, sondern von Externen / von höherer Gewalt abhängig ist.
Beispiele >		- Lager putzen im Juli - Einkistln am 19.05. - Kübel im Lager reinigen	- Mitbestellen bei Biohof XY bis 15.05. - Zusagen für Teilnahme bei Exkursion am 30.06.	- Einlesen der Preisliste von Biohof XY für Bestellung KW30	- Erhalt der Preisliste von Biohof XY für Juni per E-Mail
Attribute, Beziehungen v	Erläuterung v				
assignments [of SocialEntity] oder: assignment Actions	„Wer führt den Vorgang durch? Wer wurde (wann, von wem) zugewiesen/angefragt? zugesagt/abgesagt? Tausch angefragt/angenommen? Tatsächlich durchgeführt?“	ja	ja (wer nimmt es in Anspruch?)	nein	nein
Skill Requirements	„welche Fähigkeiten sind erforderlich & jeweils wie viele derer, die es durchführen, sollten sie erfüllen?“	ja Eigene Entitys, siehe Fähigkeiten		nein	nein
Impairment Compatibility Evaluations	„welche Einschränkungen sind kompatibel bzw. sollten nicht gegeben sein & jeweils wie vielen derer, die es durchführen, dürfen sie nicht haben?“	mögliche Inkonsistenz zwischen „wie viele derer ...“ und „von allen ...“		nein	nein
earliest Datetime	„wann frühestens?“	ja	ja	ja	?
from Datetime Is Strict [boolean]	„bei der früheste-Datum/Zeit handelt es sich um ein striktes Minimum; wenn nicht, wird nur eine Warnung angezeigt“	ja	ja	wenn false: Ausführung vorab auf expliziten Klick startbar	?
target Datetime	„bis wann?“	ja (bis wann fällig / seit wann überfällig)	ja (bis wann möglich / gilt bis wann)	wäre nur für fortgeschrittenes	ja (bis wann sollte dies geschehen)

				Priority handling relevant	
<i>target Datetime Is Strict [boolean]</i>	„bei der bis-wann-Datum/Zeit handelt es sich um eine strikte Frist; ist nachher nicht mehr möglich“	ja (wenn bis dahin nicht erledigt, ist es zwecklos vs. sollte möglichst bis dahin erledigt sein)	ja (ist nur bis dahin möglich bzw. ist möglicherweise auch nachher noch möglich)	wäre nur für sehr fortgeschrittenes Priority handling relevant	ja?
<i>required Operations</i>	„was muss vorher abgeschlossen sein, und zwar jeweils mit welchem zeitlichem Abstand zuvor?“	ja	ja	ja	ja (was muss getan/passiert sein, damit dies geschehen kann)
<i>code Reference</i>	„welcher Code soll ausgeführt werden?“	nein	nein	ja	nein
<i>stack</i>	„soll/kann [bestimmte Anzahl] mal ausgeführt werden“	ja, z.B. „insg. 4 Kübel sind zu reinigen, aber hake ab, dass du 1 gereinigt hast“	ja, z.B. „es liegen 8 Kürbisse im Lager zu verschenken“	selten sinnvoll, schadet aber nicht	ja?
<i>byEach SocialEntity OfRelations</i>	„soll/kann von jedem Mitglied ausgeführt werden“ (oder von anders in Beziehung stehender Soziale Entität)	ja, z.B. „alle Mitglieder sollen an einer internen Umfrage teilnehmen“	ja, z.B. „alle Mitglieder können bei Biohof XY mitbestellen“	nein, da nicht „von“ Mitgliedern ausführbar (vollautomatisch)	nein, da nicht „von“ Mitgliedern ausführbar (nur von Externen / höherer Gewalt)
<i>forEach SocialEntity OfRelations</i>	„soll/kann für jedes Mitglied ausgeführt werden“ (oder von anders in Beziehung stehender Soziale Entität)	ja, z.B. „jedes Mitglied soll [von einer bestimmten AG] mündlich befragt werden“	ja?	ja, z.B. „für jedes Mitglied soll das Profil in einer externen Software synchronisiert werden“	ja, z.B. „Vereinsbehörde muss alle kürzlich gewählten Vorstandsmitglieder bestätigen“
<i>ReferenceRule</i>		ja	ja	ja	ja
<i>tags</i>	„mit welchen Labels wurde dieser Vorgang markiert?“	ja	ja	ja	ja
<i>iterationOf</i>	„basiert dieser Vorgang auf einem Muster und wenn ja, auf welchem?“	ja	ja	ja	ja
<i>children</i>	„welche untergeordneten Vorgänge (Schritte) gibt es?“	ja	ja	nur Autom. Aktionen	nur Ereignisse
<i>inputFields</i>	„welche Eingabefelder gibt es zum Ausfüllen?“	ja	ja	nein	ja? (Ergebnis dokumentieren – oder eigene Aufgabe?)
<i>customData Fields</i>	„welche Daten (z.B. eingegeben oder als Ergebnis einer autom. Aktion) liegen vor?“	ja	ja	ja	?