Webbasierte Anwendungen 2
Phase 2: Dokumentation

Verteiltes System "Fridgemanager"

Sommersemester 2013
Fachhochschule Köln
Betreut von Kristian Fischer

Beteiligte dieser Ausarbeitung:
Simon Klinge 11082448
Julia Warkentin 11082815

Inhaltsverzeichnis:

1. Aufgabenstellung

Eine zusammenhängende, textuell ausformulierte Aufgabenstellung wurde niemals der Allgemeinheit zugänglich hinterlegt. Deshalb lohnte es sich im ersten Schritt das Ziel und seinen Kontext zu spezifizieren, um sich immer wieder bewusst werden zu können, welche Intention eigentlich mit diesem Projekt verfolgt werden soll. Für den Fall, dass in dieser Ausarbeitung ein anderer Weg eingeschlagen wurde als der von den Lehrenden dieses Moduls beabsichtigte, bleibt die ganze Dokumentation trotzdem in sich schlüssig, da sich im Entstehungsprozess immer wieder auf den, in der Aufgabenstellung definierten, intendierten Zustand gestützt werden konnte.

Die Aufgabe bei dieser Arbeit war es ein verteiltes System, das in der Lage ist synchrone und asynchrone Nachrichten, zwischen seinen physisch voneinander getrennten Komponenten, auszutauschen, vollständig konzeptioniert und weitestgehend implementiert zu haben.
Es sollte nicht nur die reine Kommunikation zwischen den technischen Elementen des Systems möglich gemacht werden, sondern darüber hinaus auch den Transaktionen die Semantik zu einem selbst zu definiertenden Oberthema, das Bezug zur realen Welt besitzt, gegeben werden. Einzige Bedingung an jenes Oberthema war es, Potenzial für einen möglichst frequenten Datenaustausch bieten zu können. Dabei war es möglich die Architektur entweder aus bereits vorhandenen Paradigmen zu wählen, komplett neu zu gestalten oder sowohl bekannte als auch unbekannte Elemente in das Architekturdesign mit aufzunehmen. Außerdem gab es keine Begrenzung bei der Wahl der vorstellbaren Endgeräte. Möglich war es demnach das System in Form einer Applikation für ein mobiles Engerät, als Browseranwendung o.ä. zu realisieren, solange die Implemetierung und Architektur alle Charakteristika eines verteilten Systems erfüllen.
Der Entwicklungsprozess sollte dabei in sieben vorgeschriebenen Teilaufgaben, den sogenannten "Meilensteinen", dokumentiert werden. Diese Steps lauteten wie folgt:

0. Konzeptioneller Meilenstein Kommunikationsabläufe und Interaktionen
1. Meilenstein Projektbezogenes XML Schema / Schemata
2. Meilenstein die Semantik der HTTP-Operationen für das Projekt
3. Meilenstein RESTful Webservice
4. Meilenstein Konzeption + XMPP Server einrichten
5. Meilenstein XMPP - Client
6. Meilenstein Client – Entwicklung

Durch diese Vorgaben wurde die Modalität der Realisierung des verteilten Systems weiter eingeschränkt und noch zusätzliche Anforderungen an sie gestellt. Unteranderem bestand die Pflicht mindestens einen Client und mindestens einen XMPP-Server, auf dem die benötigten Ressourcen auch als XML-Dateien mit zugehörigen XSD- bzw. Schemadateien gespeichert werden sollen , zu erstellen und einen RESTful Werbservice für den synchronen Teil der Kommunikation zu erschaffen. Die Einschränkungen sind selbsterklärend und können der oben stehenden Liste entnommen werden.

2. Ideenfindung

Nach einem kurzen Brainstorming standen folgene Oberthemen für den Dienst fest, zwischen denen es sich zu entscheiden galt.

1. Ein System zur Verwaltung des Kühlschranks (bzw. eine WG-App)
2. Browsergame/ Multiplayerapp/ Rollenspielapp
3. Nachrichtensystem (Publish-Subscribe-Paradigma)
4. Chatsystem (Synchrone und asynchrone Nachrichten möglich)

Die Idee, die ohne großes Nachdenken zuerst in den Sinn kam, war eine Applikation, die den Inhalt eines Kühlschranks verwaltet. Ihre grundsätzlichen Funktionen und Alleinstellungsmerkmale hatten sich ebenfalls schon unmittelbar manifestiert.

Die anderen drei Einfälle waren alle nur sehr schemenhaft skizziert und boten nur abstrakte Ansätze in der Funktionalität, aber ihre zentrale Daseinsberechtigung wurde eigentlich nicht gerechtfertigt. Sie stellten nur oberflächlich beschriebene Standardbeispiele für ein verteiltes System dar., für die das, von der Aufgabenstellung geforderte, Oberthema, im Gegensatz zu der ersten Idee, noch nicht gewählt worden war.
Diese Ansätze waren generell nur als Aorta gedacht, für den Fall, dass die erste Idee in den Augen der Betreuer kein Potential für ein verteiltes System liefern konnte.
Glücklicherweise wurde der erste Ansatz für die Weiterentwicklung zugelassen noch bevor die letzteren weiter ausgearbeitet werden konnten. Es wurde zusätzlich gesagt, die Idee solle noch etwas an Umfang gewinnen.

3. Projektbeschreibung "Fridgemanager"

Die primäre Funktionalität des "Fridgemanagers" liegt, wie sein Name bereits besagt, darin, den Kühlschrank eines Haushaltes und alle seine beinhalteten Nahrungsmittel zu verwalten. Durch diverse Scanner werden eingeladene und entnommene Waren direkt vom Kühlschrank erkannt und deren zugehörige Daten wie Inhaltsstoffe, Nährwerte, Produktbeschreibungen, Preise und Menge gespeichert. Für jedes Produkt werden zusätzlich noch Einlesedatum, Datum des Verzehrs, Verfallsdatum und Agabe über den aktuellen Status gesichert. Die Status könnten sich auf das Vorhandensein im Kühlschrank, "Grad der Verbauchung", oder "Konsumiert von Benutzer XY" beziehen.
Der Fridgemananager ist speziell für Wohngemeinschaften, oder Haushalte mit mehr als nur einem Bewohner angedacht, in denen oft Missverständnisse autreten können, da jeder Konsument seine, teilweise separat bezahlten, Lebensmittel in einem zentralen Aufbewahrungsystem lagert, diese Lebensmittel jedoch ohne instantane Kenntniss des Inhabers von anderen entnommen und verzehrt werden können. Um dem entgegen zu wirken, wird für jeden Nutzer eine Art Account angelegt, in dem eine Liste seiner kürzlich erworbenen und konsumierten Waren gespeichert wird.
Jedes eingelesene Produkt enthält eine Referenz zum Profil seines Besitzers. Sobald ein Artikel unberechtigterweise von einem Benutzer, der nicht Inhaber ist, entnommen wird, erhält dieser eine Warnmeldung, dass es sich bei dem entnommenen Artikel nicht um sein Eigentum handelt. Ebenfalls erhält der eigentliche Besitzer eine Warnmeldung, in der er über den Vorfall aufgeklärt wird.
Zusätzlich können für jedes Profil eine Liste an favorisierten Artikeln angelegt und der Mindestbestand eines Produktart angegeben werden, anhand dieser der Fridgemanager weiß, welche Lebensmittel er vollautomatisiert nachbestellen soll. Fällt die Anzahl der existenten Produkte unter die im Mindestbestand definierte Untergrenze, wird jenes Produkt nachbestellt. Auch ohne die Favoritenliste erkennt der Fridgemanager Muster in der Produktwahl und lernt permanent, welche Waren häufiger eingekauft werden und demnach auch beliebter sind. Mit diesen Daten wird dem System erneut eine Grundlage für die automatisierte Nachbestellung geliefert. Natürlich können diese Daten entweder pro Profil, oder für den durchschnittlichen Geschmack aller Bewohner der Wohngemeinschaft ausgewertet werden.
Die automatische Nachbestellfunktion muss nicht zwingend aktiviert werden. Eine Fridgemanager-Application kann auch einfach nur als koexistente Lösung für einen Einkaufszettel benutzt werden. Bei Spontaneinkäufen kann jederzeit eine Liste der, sich im Kühlschrank befindlichen, Nahrungsmittel angefragt werden, und somit Rückschluss auf die noch benötigten Artikel gezogen werden.
Das als letztes zuerwähnende Feature ist der "Fitnesstrainer". Parameter, über die der Body-Mass-Index eines Menschen berechnet werden kann, bzw. Parameter, die den Grad der Fitness eines Menschen bestimmen, können für jedes Profil gespeichert werden. Mit der Angabe des Nähstoffgehalts für jedes Produkt und des Liste der zuletzt konsumierten Nachrungsmittel, kann für jedes Profil die Zunahme von lebenswichtigen und gesundheitsschädlichen Inhaltsstoffen errechnet und in Beziehung zu den

Parametern eines durchschnittlichen Verbrauchsers gesetzt werden.
Noch ist die Realisierung auf einem mobilen Endgerät geplant, da der mobile Zugriff auf die Verwaltungsdaten wie den Lebensmittelbestand im Kühlschrank abfragen eigentlich nur dann Sinn ergibt, wenn man sich nicht direkt im selben Raum wie der Kühlschrank befindet.

3.1 Szenarien

Im vorherigen Abschnitt wurden die geplanten Funktionen nur grob aufgezählt und die anzusprechende Focal Group namentlich gemacht. Es handelt sich dabei aber wirklich nur um die Ergebnisse des allerersten Brainstormings.
Im nächsten Schritt sollte versucht werden noch Schwächen oder Verbesserungspotentiale zu finden, da die Wahrscheinlichkeit viel zu gering ist, beim ersten Skizzieren der Funktionen schon tief genug in die Materie eingedrungen zu sein, alle wichtigen Eventualitäten bedacht haben zu können.

Es existiert eine vielzahl an Dokumentationsarten, um die Vollständigkeit und die Genauigkeit der Funktionalitäten des Systems zu überprüfen und sicherzustellen. Als erstes sollte die Zielgruppe aus der Menge der gesamten Menschheit lokalisiert werden. In diesem Fall handelt es sich bei der Focal Group um Personen, die in einem Haushalt mit mehreren anderen Menschen zusammen leben und den selben Kühlschrank nutzen. Danach kann diese eingeschränke Menge an Personen noch weiter unterteilt, Stakeholder für sie analysiert und consolidiert, (essential oder concrete) User Profiles erstellt, Personae und/oder Real People ausfindig gemacht werden. Sobald die potentiellen Nutzer analysiert wurden, kann das System maßgeschneidert auf exakt diese Ziegruppe ausgearbeitet, bisherige Funktionen ergänzt und verbessert werden.
Diese extrem detailreiche Auswertung des Nutzungskontextes ist für die Arbeit im Rahmen dieses Moduls sicher nicht gefragt, da das zentrale Wesen des Moduls sich nur auf das Entwickeln von verteilten System und nicht auf das Lokalisieren aller wichigen Funktionen des bereitzustellenden Dienstes und der Analyse der Stakeholder beschränkt. Dies kann an dieser Stelle leider nicht hundertprozentig vorausgesetzt werden, da die Bewertungskriterien dieses Projektes niemals bekannt gegeben wurden. Trotzdem würde eine solche Analyse den zeitlichen Rahmen des Moduls sprengen, deshalb wird sich im Laufe dieser Ausarbeitung nur auf das wesentliche und fristgemäß machbare konzentriert.

Eine gängige Vorangehensweise, um am Anfang eines Projektes mit geringem Aufwand einen möglichst großen Designschritt zu tätigen und Designpotentiale zu adressieren, liegt in dem Dreischritt "Descriptive Task Model", "Claim Analysis" und "Presciptive Task Model". Zuerst wird anhand eines konkreten Anwendungsfalls der Ist-Zustand der Arbeit beschrieben und daraus auch implizit enthaltene Schwächen analysiert werden, die dann im dritten Schritt als Designpotentiale/Soll-Zustand genutzt werden können. An dieser Stelle ist es sinnig ein Verfahren eines deskriptiven Modells anzuwenden, da in erster Linie nur Schwächen ausfindig gemacht werden sollen.

Unter den Begriff "Descriptive Task Model" fällt auch die Methodik der Szenarien. Szenarien sind extrem konkrete und auf den Kontext bezogene Instanzen eines Anwenungsfalls.
Es wurde sich für eine Mischung aus Problem- und Soll-Szenario entschieden, da sowohl Probleme der realen Kühlschranknutzer als auch derer, die den fiktiven Kühlschrank nutzen könnten, evident gemacht werden sollen.

Ein weiter Grund Szenarien aufzustellen besteht darin die bisher angedachte Funktionalität des Fridgemanagers im Bezug auf die zu versendenen Daten zu analysieren, um für den konzeptionellen Meilenstein (siehe 5.1 Konzeptioneller Meilenstein: Kommunikationsabläufe und Interaktionen) gewappnet zu sein.

3.1.1 Szenario: Nahrungsmittel einlesen und selbst kallibrierbare Parameter ändern

Petra Müller und Hans Müller wohnen seit jahren zusammen und benutzen den selben Kühlschrank. Petra Müller kommt von ihrem wöchtentlichen Einkauf und vollen Tragetaschen zurück in die Wohnung. Durch die Scanner am Kühlschrank können die neu erworbenen Nahrungsmittel umgehend und ohne weitere Arbeitsschritte eingelagert werden. Die Produkte werden direkt erkannt. Ihre Informationen müssen nicht erst in einer langwierigen Prozedur für jeden einzelnen Artikel manuell von Hand eingegeben werden.
Hans hat von der automatischen Nachbestellfunktion gehört und stellt mithilfe der Applikation den Mindestbestand für Milch, Butter und Käse jeweils auf zwei Einheiten. Nun prüft der Kühlschrank permanent, ob und welche Lebensmittel langsam zuneige gehen und nachbestellt werden müssen. Petra Müller müsste jetzt nicht mehr jede Woche selbst einkaufen gehen und die vielen, schweren Taschen selbst nach Hause schleppen.

3.1.2 Szenario: Spontaneinkauf und Daten mobil erfragen

Workaholic Luisa Stein wohnt in einer Wohngemeinschaft mit drei anderen, ebenfalls sehr beschäftigten Berufstätigen zusammen. Oft ist niemand in der Wohnung. Einkaufen kommt für sie eigentlich nur nach einem langen, harten Arbeitstag in Frage. Doch bis dahin ist schon vergessen, bei welchen Lebensmittel überhaupt Nachkaufbedarf besteht. Normalerweise könnte sie sich morgens vor dem Weg zur Arbeit einen Einkaufszettel schreiben, aber dank des Fridgemanagers kann sie mobil und jederzeit den Inhalt ihres Kühlschranks abfragen.
Kurz nach Feierabend genügt ein Blick auf die Applikation auf ihrem Handy und schon weiß sie, an welchen Nahrungsmittel es in ihrem Kühlschrank mangelt.

3.1.3 Szenario: Produkt unbefugt entnehmen, Warnung, Rechte an Produkt erfragen

Klaus Einfach wohnt unteranderem mit Neil Malls im selben Haushalt. Eigentlich kauft jeder seine benötigten Produkte selbst und stellt sie eigenhändig in den Kühlschrank. Teilweise werden Grundnahrungsmittel durch diese Organisation mehrfach in der selben Ausführung eingekauft und unverbraucht weggeworfen. Durch die Verwaltung des Fridgemanagers, kann ihr Kaufverhalten aktiv verbessert werden.
Am abend möchte Klaus ein Glas Milch trinken. Leider hat er für sich selbst keine Milch gekauft und der Aldi hat schon geschlossen. Neils Milchpakung steht glücklicherweise noch ungeöffnet im Kühlschrank. Da Neil Nachtschicht hat und sich aufgrunddessen nicht in der Wohung befindet, beschliesst Klaus die Milch erstmal ansich zu nehmen und seinem Mitbewohner am nächsten Morgen eine neue Pakung zu kaufen.
Sobald er die Milchpakung unberechtigterweise herausnimmt, erhält er eine Warnmldung durch die Applikation, die ihn darauf aufmerksam macht, dass das Ansichnehmen nicht rechtens ist. Auch Neil bekommt instantan eine Warnmeldung, in der er über den Diebstahl aufgeklärt wird. Über die Applikation erklärt Klaus sein Handeln und die beiden können sich darauf einigen, dass Neil ihm die Rechte an der Milch überträgt, unter der Bedingung, dass er am nächsten Tag eine neue Pakung kauft, für die er Neil als Besitzer einträgt.

3.1.4 Szenario: Fitnesstrainer

Nach der Schwangerschaft ihrer Zwillinge hat Molly Eillaftuid ordentlich zugelegt. Nun verlässt sie sich auf den Fridgemanager, um ihre Ernährung zu überwachen und zu verbessern. Auf ihrem eigenen Account im Kühlschrank gibt sie charakteristische Parameter ihres Körpers, Alter, Größe, und auch ihr Wunschgewicht, an. Aus denen wird ihr persönlicher BMI berechnet. Nun kann sie sich anzeigen lassen, welche Produkte und in welcher Menge sie sie pro Tag zu sich genommen hat. Zu jedem Artikel sind auch seine Nährwerte eingespeichert. Aus diesen Informationen wird zusätzlich berechnet wie viel Molly

von welchen Nahrungsmittel essen darf, um ihr Traumgewicht gesund zu erreichen. Die Applikation schlägt auch selbst immer wieder Lebensmittel vor, die konsumiert werden können. Diese Lebensmittel werden nach dem Kriterium, dass die Tagesration an Nährwerten durch den Konsum des vorgeschlagenen Produktes nicht überschritten wird, vom System ausgewählt. Molly hat am Abend schon 2500 kcal zu sich genommen. Ihr werden nur noch kalorienarme Lebensmittel wie Magerquark empfohlen. Trotzdem greift sie noch mal zur Colaflasche. Das System registriert die Wahl, ergänzt die bisher zunenommenen Nährwerte mit denen der Cola und vergeicht diese mit der empfohlenen Tagesration. In diesem Fall sind die aktuell berechneten Nährwerte höher als die Tagesration, demnach schickt das System ihr eine Warnmeldung.

3.2 Lokalisierte Schwächen

Durch die Erstellung der Problemszenarien konnten wie erwartet sehr viele vorher noch undurchdachte Unstimmigkeiten adressiert werden.

3.2.1 Sämtliche Scanner nur als Simulation

Die geplante Allokation, der User muss nur die eingekaufte Ware in den Kühlschrank stellen oder höchstens ein paar Einstellungen in der Applikation tätigen und das System kümmert sich unteranderem durch Scanner um den ganzen Rest der Verwaltung, würde zwar vom Benutzer akzeptiert werden, ist aufgrund mangelnder Ressourcen Technik und Zeit nicht auf die geplante Art realisierbar. Mit Allokation ist die Relation zwischen den Aufwändungen des Nutzers und denen des Systems gemeint. Natürlich wäre es in der heutigen Zeit technisch gesehen möglich spezielle Scanner zu entwickeln, die nur dazu dienen Produkte anhand ihrer äußerlichen Erscheinung zu analysieren und klar einzuordnen. Als Student fehlen aber die Mittel dazu, deshalb wird der Output dieser Scanner für das zuimplementierende verteilte System nur virtuell simuliert.
Es war zwar geplant bei jedem Einlagern und Entnehmen von Lebensmittel in oder aus dem Kühlschrank immer eine Besitzerprüfung zu vollführen, um z.B. Diebstahl vorzubeugen und jedem Artikel genau einen Besitzer zu ordnen zu können, aber wie genau diese Prüfung vonstatten gehen soll, darüber wurde noch nicht nachgedacht. Bei der ersten Variante könnte die Besitzerprüfung auf die Scanner vertagt werden, damit die bisherige Allokation bestehen bleibt. Auf irgendeine Weise z.B. durch Scannen des Fingerabdrucks wird die Identität des aktuell Benutzendens sichergestellt. Dieser Scanner müsste, aufgrund oben genannt Probleme, ebenfalls simuliert werden. Bei der zweiten Variante müsste jeder User vor der Benutzung des Kühlschranks jedesmal erneut in seiner Applikation angeben, in wiefern er den Kühlschrank gedenkt zu nutzen, damit der Fridgemanager weiß, wer die nächtste Aktion an ihm vollführt. Durch diese manuelle Bedienung wird die Allokation in eine unbeabsichtigte Richtung verzerrt. Da bietet sich die erste Variante viel eher an.
Bisher befinden sich die Scanner nur am Kühlschrank, jedoch gibt es auch Lebensmittel, die nie oder erst ab einem bestimmten Zeipunkt im Kühlschrank verwahrt werden. Soll sich das System nur um jene kümmern, die in den Kühlschrank eingelesen werden, oder um die Gesamtheit der Lebensmittel der Wohngemeinschaft? Falls alle Lebensmittel berücksichtigt werden und sich die Scanner im Kühlschrank befinden, kann es sein, dass Benutzer abundzu vergessen, alle neu erworbenen Artikel eizulesen, die nicht zwnagsweise im kühlschrank gelagert werden müssen.
Der Fitnesstrainer würde versagen, wenn nur die Waren gespeichert werden, die im Kühlschrank gelagert werden. Demnach wird die Verwaltung auf alle Lebensmittel im Haushalt erweitert. Der Fitnesstrainer ist schliesslich eines der wesentlichsten Alleinstellungsmerkmale.
Auf Lebensmittel, die außerhalb des Gebäudes gekauft und direkt konsumiert werden, kann nur mühsam Rücksicht genommen werden, auch wenn der Fitnesstrainer Werte dieser Produkte braucht, um korrekte Werte zu berechnen. Der Verzehr derartiger Artikel könnte manuell an der Applikation eingegeben werden. Deren Nährwerte könnten aus Durchschnittswerten eines solchen Produktes aus dem Internet bezogen werden.
Ein ganz anderes Problem stellen Waren dar, die nicht von der Lebensmittelindustrie nach genormten

Designkriterien gestaltet sind z.B. Selbst angepflanzes Obst oder Gemüse. Selbst wenn der Scanner diese Waren erkennt, können Nährwerte nur problematisch ermittelt werden. Für den Fitnesstrainer sind diese Werte aber überlebenswichtig. Es gibt auch andere Produkte, die keine Informationen über ihren Nährstoffgehalt Preis geben z.B. Brötchen vom Bäcker. Über die Behandlung dieser Produkte soll nicht besonders intensiv eingegangen werden. Es sollen lediglich mögliche Varianten und Probleme, die sich bei der Ausarbeitung dieser Aufgabe gestellt haben, evident gemacht werden, um zu zeigen, dass sich viele Gedanken über die Funktionalität und Anforderungen des Systems gemacht wurden.

3.2.2 Grad der Verbrauchung

Dieser Punkt stellt die wohl lästigste Schwäche überhaupt dar. Mit dem "Grad der Verbrauchung" ist gemeint, zu welchem Prozentsatz ein existierendes Produkt bereits verzehrt wurde. Es stellt sich die Frage, ob dieser Wert überhaupt abgespeichert werden muss. Bei Produkten, die nur einer Person gehören spielt dieser Wert zunächsteinmal keine sonderlich große Rolle. Es gibt aber auch Lebensmittel z.B. Milch oder Butter, die die von jedem nur zum Teil und insgesamt von allen Bewohnern eines Haushaltes verzehrt werden können. Jeden Bewohner seine eigene Butter kaufen lassen ist nicht besonders ökonomisch. Aus diesem Grund bietet sich die Einführung eines zentralen Fridge-Accounts, der allen Bewohnern der Gemeinschaft gleichermaßen gehört, an. Jetzt müsste zusätzlich zu jedem Produkt der Gemeinschaft auch sein aktueller Grad der Verbauchung und wer wie viel davon verzehrt hat gespeichert werden. Ansonsten würde die geplante Kostenmanagementfunktion ihren Zweck verfehlen. Außerdem braucht der geplante Fitnesstrainer die genaue Mengenangabe des verzehrten Artikels, sonst würde hier ebenfalls der Zweck verfehlt werden.

Die Einführung eines solchen Wertes würde, die Implementierung noch um einiges schwieriger machen, als sie zu diesem Zeitpunkt ohnehin schon ist. Falls der Verbrauchungsgrad eingeführt werden würde, müssten ihn entweder erneut simulierte Scanner für jedes eingelesene Produkt feststellen oder manuell bei jeder partiellen Konsumierung neueingegeben werden. Wie stellen die Scanner fest, ob es sich bei dem eingelesenen Lebensmittel um ein partiell konsumiertes handelt und nicht um ein ungeöffnetes?

3.2.3 Online-Shop und automatische Nachbestellfunktion

In Verbindung mit der automatisierten Nachbestellfunktion sollte sich auch die Frage stellen, wo genau der Kühlschrank die Waren neu nachbestellt. Soll ein extra Online-Shop für den Fridemanager eingerichtet werden? Wird neue Ware von bereits bestehenden Online-Shops bestellt? Kann der Nutzer sich vielleicht sogar aussuchen, welchen Online-Shop er dafür in Erwägung ziehen möchte?
An dieser Stelle soll auch erwähnt sein, dass geplant ist, dass der Kühlschrank bei Unterschreitung des Mindestbestandes eines bestimmten Artikels zwar direkt eine Nachbestellung einreicht, aber es noch keinen Ort gibt, an dem die Nachbestelle Menge speziefiziert werden kann. Ein Maximalbestand bzw. eine Nachbestellmenge sollte analog zu dem Mindestbestand bei der Nachbestellfunktion für jeden Artikel eingeführt werden.

3.2.4 Unabsehbare WG-Strukturen

In manchen Wohngemeinschaften herrschen interne Regeln, auf die Rücksicht genommen werden könnte. Unteranderem sollen Waren im Kühlschrank in einer speziellen, vereinbarten Art und Weise angeordnet sein. Jedes Mitglied der Wohngemeinschaft darf nur sein für ihn vorgesehenes Fach im Kühlschrank belegen. Wohngemeinschaften können auch mehr als nur einen Kühlschrank besitzen. Rücksicht auf diese Sonderfälle würde die Implementierung nur noch um einiges umständlicher machen.

Als letzte Schwäche wurde eine fehlende Möglichkeit der Kommunikation zwischen den Benutzern der selben Fridgemanagers lokalisiert. Eine kleine Chatfunktion wäre angebracht, z.B. wenn Rechte für einen Artikel erfragt werden oder für ähnliche denkbare Szenarien.

Die Schlussfolgerung der Schwächenanalyse ist, dass sich nur auf die Schlüsselfunktionen fixiert werden soll, auch wenn anfangs gesagt wurde, die Idee müsse noch um einiges an Funktionalität dazu gewinnen, denn blickend auf die zur verfügung stehenden Ressourcen, ist klar erkennbar, dass die Umsetzung aller hier genannten Ideen schier unmöglich erscheint.

4. Zielsetzung

Eine präzise ausformulierte Zielsetzung eignet sich als stabiles Fundament für die Bearbeitung dieser Aufgabe, da das Formulieren einer Zielsetzung dazu führt, dass grundlegende Vereinbarungen getroffen werden, auf die im Entwicklungsprozess wieder eingegangen und mittels derer für oder gegen eine bestimmte Variante argumentiert werden kann. Außerdem stellt sie sicher, dass sich mit dem basalen Hintergrundwissen, den Fragen und Hindernissen bei der Bewältigung der Aufgabe beschäftigt wurde.
Im Gegensatz zu der Aufgabenstellung hat die Zielsetzung einen zeitlosen Charakter. Eine Aufgabe wird einem von anderen Personen, von außen, in einem expliziten Kontext gestellt. Ebenfalls ist eine Aufgabe durch diesen vorgeschriebenen Kontext schon sehr konkret. Der Kontext schreibt also schon teilweise den Weg zum Ziel vor. (vgl. 1. Aufgabenstellung)
Die Zielsetzung hingegen ist viel abstrakter und darf auch von einer Person nur für sich selbst formuliert werden. Bei der Zielsetzung spielt nur das Endprodukt eine Rolle, nicht aber der Weg dorthin bzw. der Kontext. Eine Zielsetzung kann demnach eine Menge mehr an Aspekten berücksichtigen als eine reine Aufgabenstellung, geht also eher in die Vertikale als in die Horizontale.
Zusammengefasst dient sie dazu Entscheidungen effektiver treffen und den Betreuern zeigen zu können, dass sich mit der grundlegenden Thematik verteilter Systeme vorab beschäftigt wurde.

Die Zielsetzung umfasst zwei große Dimensionen: Anforderungen an das zu erstellende System und Anforderungen der Beteiligten dieser Ausarbeitung an sich selbst.
Auf der einen Seite werden Anforderungen, Designkriterien und Alleinstellungsmerkmale für das System gewählt. Das zu erstellende verteilte System soll den Grundsätzen der RESTful Architektur (wird im nächsten Abschnitt beschrieben) genügen. Es sollte eine möglichst lose Kopplung der kommunizierenden Komponenten sichergestellt, Interoperabilität ermöglicht Performance und Skalierbarkeit durchdacht werden. Redundante Datenspeicherung soll möglichst vermieden werden. Die Kommunizierenden Elemente sollten sich, wenn möglich, eher kürzere Nachrichten schicken, um an Performanz zu gewinnen. Insgesamt soll eine Allokation zwischen Mensch und Maschine geplant werden, die auch in der realen Welt Akzeptanz finden würde. Diese Planung braucht aber nicht zwangsweise umgesetzt zu werden, da, wie bereits erwähnt, die nötigen Ressourcen Zeit und Technik nicht gegeben sind.

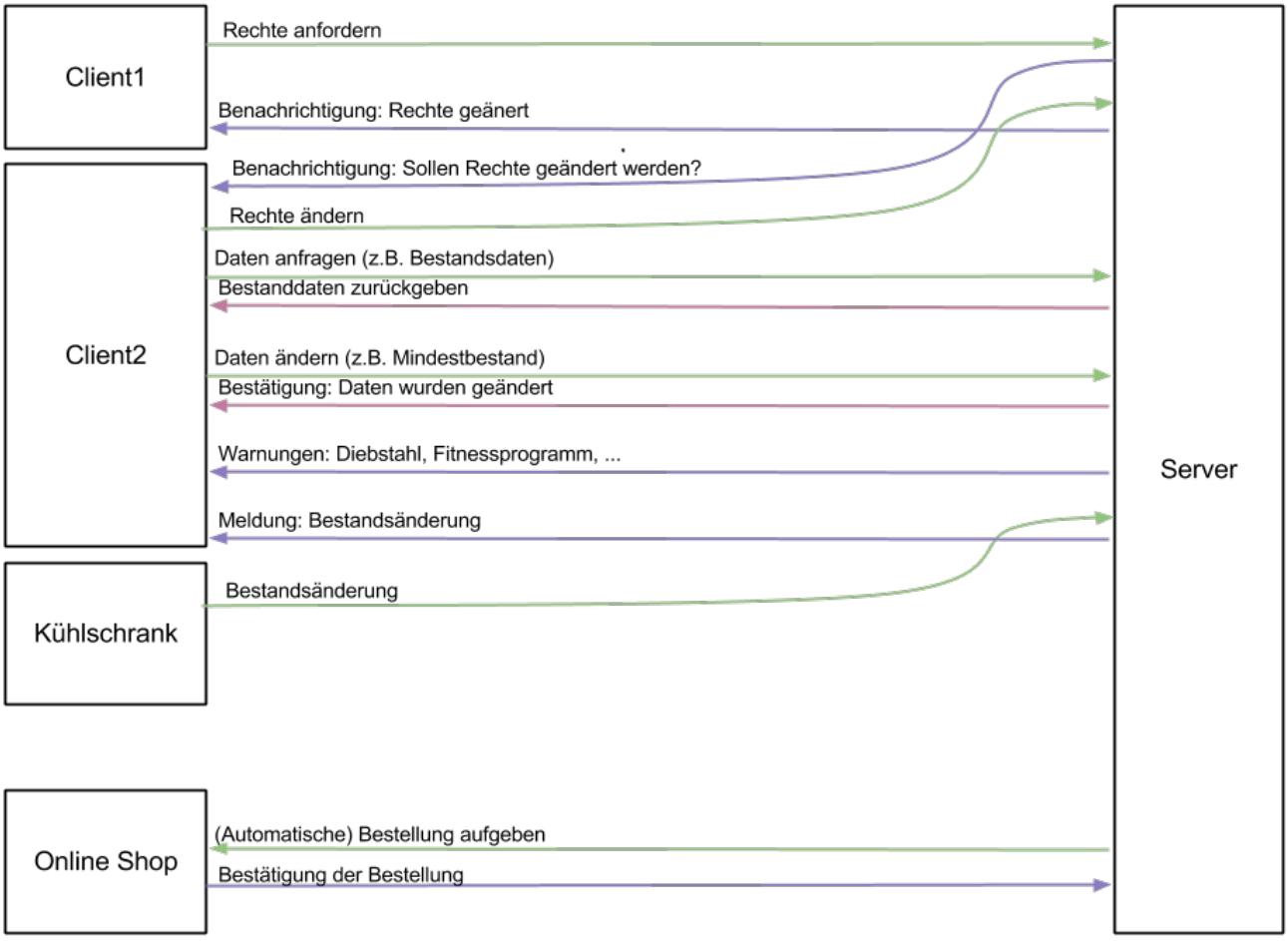
Auf der anderen Seite wird der Erkenntnisgewinn, das Erlangen neuer Kompetenzen im Bereich der verteilten Systeme angestrebt und diese/r auch versucht den Bewertenden zu demonstrieren. Da dieses Modul einen Teil der, im Modul „Entwicklung interaktiver Systeme“ erforderten, Kenntnisse zur Thematik der verteilten Systeme aufweist, soll dieses Projekt auch als indirekte Vorbereitung für das aufwändige Projekt jenes Moduls genutzt werden. Dabei handelt es sich auch um einen weiteren Grund Begriffe wie Szenarien und Stakeholderanalyse erwähnt zu haben.

5. Entstehungsprozess

5.1 Konzeptioneller Meilenstein: Kommunikationsabläufe und Interaktionen

Beschreibung der Aufgabe des Meilensteins
Im ersten Schritt sollen die Interaktionspartner und ihre kommunizierten Nachrichten ausfindig gemacht und modellhaft skizziert werden, um einen ersten Überblick des Funktionspaketes des Systems zu gewinnen. Darüber hinaus soll evident gemacht werden, welche Komponeten bei welcher Anfrage mit welchen anderen Komponenten interagiert und welche Daten bei dieser Interaktion verschickt werden sollen. Außerdem soll an dieser Stelle schon bestimmt werden, bei welchen dieser Nachrichten es sich um jeweils synchrone oder synchrone Interaktion handelt, da beide Arten völlig unterschliedlich zu implentieren sind.

Begründung und Beschreibung der Darstellung
Als abstrakte Kommunikationskomponenten wird zunächst die kleinstmögliche Anzahl von zwei Interaktionspartnern gewählt, da aus der Sicht des Nutzers im Grunde eben nur diese zwei Partner existieren. Das muss aber nicht zwangsläufig heißen, dass das System am Ende nur aus zwei unterschiedlichen Kommunikationspartnerarten besteht. In diesem Schritt soll lediglich abstrahiert werden. Gemeint ist eine Kommunikation nach dem Client-Server-Paradigma. Der Benutzer fragt durch seine Applikation Daten von einer zentralen Recheneinheit an und erwartet eine serverseitige Antwort. Das sich hinter der zentralen Recheneinheit ein viel komplexeres System als nur ein Server verbirgt, soll dieser nicht zwangsweise zu Gesicht bekommen. Es wird die abstrakte Sicht auf die Kommunikationsabläufe, wie sie ein Benutzer hätte, gewählt, um im ersten Schritt einmal einen groben Überblick über alle Funktionalitäten und ihrer jeweiligen Direktion zwischen Benutzer und, im weitesten Sinne, den Dienstanbieter zu gewinnen.



In der vorangegangenen Abbildung ist der erste Entwurf eines solchen Interaktionsdiagrammes zu sehen. Die Pfeile und deren Farbe bestimmen die Richtung und den Modus der Nachricht. Grüne Pfeile stellen Anfragen vom Client an den Server dar. Hell violette Pfeile repräsentieren die aus dieser Perspektive als synchron erscheinende Nachrichten des Servers an den Client und dunkel violette die asynchronen.

Synchone Nachrichten
Sämtliche Anfragen von Client zu Server, die dieser direkt beantworten kann ohne eine weitere Instanz ansprechen zu müssen, die eventuell zu dem Zeitpunkt nicht erreichbar sein könnte wie z.B. Geräte, die nicht durch einen Endnutzer gesteuert werden, werden als synchron modelliert. Synchrone Anfragen erwarten eine zeitnahe Bearbeitung und Rückantwort. Dieser Bearbeitung kann nicht nachgegangen werden, wenn der Kommunikationspartner erst gar nicht erreichbar ist. An dieser Stelle wird angenommen, dass der Server als wichtigste und zentrale Instanz des verteilten Systems möglichst zu jeder Zeit ansprechbar ist. Unter die synchronen Funktionen fallen generell alle Anfragen, die den **Datenbestand zu Profilen**, **Kühlschränken** und **Produkte** der angenommenen Datenbank **auslesen**, **erweitern**, **ändern** und **verringern**, da der Server hier mit keiner weiteren, unzuverlässigen, menschlichen Instanz kommunizieren muss, sondern nur mit der Datenbank, bei der ebenfalls versucht wird, sie permanten erreichbar zu erhalten.

Asynchrone Nachrichten
Bei den restlichen Funktionen handelt es sich um asynchrone Transaktionen, da bei ihnen der Server nur als Medium zwischen zwei Endbenutzern dient, die wechselseitig Nachrichten austauschen ohne dabei eine anhaltende Internetverbindung zu sichern. Daher sind konkrete, asynchrone Funktionen in **Benachrichtigungen** und **Warnungen** gegliedert. Benachrichtigungen umfassen Nachrichten, die sich in der Welt zweier Profile abspielen. Nutzer1 stellt Nutzer2 die Anfrage, ob Nutzer2 die Rechte an einem definierten Produkt in seinem Eigentum auf den fragenden Nutzer1 übertragen möchte. Auch die darauf folgende, serverseitige Benachrichtigung an beide Profile, wenn die Rechte tatsächlich übertragen wurden, und die geplanten Chatnachrichten, stellen asynchrone Kommunikation dar.
Produktwarnungen werden unteranderem an alle Inhaber des selben Kühlschranks ausgeworfen, sobald das Mindeshaltbarkeitsdatum eines Produktes eben dieses Kühlschrankes in einer gegenwartsnahen Zeitspanne abläuft. Falls einer der beiden menschlichen Akteure keine Verbindung zum globalen Netz vorweisen kann, muss ein Mechanismus den Verlust der nicht zustellbaren Nachricht verhindern und sie persistieren bis der Empfänger eine fähige Netzanbindung vorweisen kann. Dieser Mechanismus wird in einem späteren Meilenstein realisiert.

Variante: Jegliche Kommunikation asynchron
Asynchronität bietet den mächtigen Vorteil, gegenüber der Synchronität, Nachrichten auch entkoppelt von dem zeitlichen Aspekt an den Empfänger übermitteln zu können und dabei auch an mehrere Empfänger adressiert zu sein. Wäre es nicht sinnvoll diese mächtigen Vorteile für alle Nachrichten innerhalb des Systems, für den Fall, dass der Server kurzzeitig nicht erreichbar ist, zu nutzen? Natürlich wäre es implementierbar, aber neben den Vorteilen zieht die Asynchronität auch einige Nachteile mit sich. Asynchrone Daten müssen persistent gespeichert werden, damit sie auch erst zu einem anderen Zeitpunkt zugestellt werden können. Alle Nachrichten als asynchrone Kommunikation zu implementieren würde eine enorme Steigerung der zuspeichernden Daten bedeuten, die zwar nur bis zum Zeitpunkt ihrer Versendung gespeichert werden müssten, aber dies würde das Netzwerk unnötig belasten und den Verwaltungsaufwand um einiges steigern. Wiegt man diesen beträchtlichen Aufwand mit zu der geringen Häufigkeit der serverbetreffenden Netzwerkausfälle, so fällt der Entschluss nicht schwer zu versuchen einen an die jeweilige Nachricht angepassten Mittelweg einzuschlagen, zum mal es auch vorkommen kann, dass die Verbindung zur Zwischenspeicherinstanz auch simultan mit der des Servers abbrechen kann. In dem Fall wäre auch der Vorteil der asynchronen Kommunikation nicht mehr vorhanden.

Kühlschrank früher eigenständiger Client
Da es hier hierbei nur um die erste Skizze handelt, kann nicht gewährleistet werden, dass exakt diese Funktionen in ihrer hier dargestellen Ausprägung auch ebenso im implementierten System wieder gefunden werden kann. In der ersten Variante, war der Kühlschrank ein eigenständiger Client, da er, durch seine sensorischen Anbringungen, die Änderungen seiner Umgebung wahrnimmt, sprich das Entnehmen oder Einlagern von Lebensmitteln, und Daten zu diesen Vorfällen an den Server weiterleitet. In der Realität kann es auch so gehandhabt werden können, aber da das zeitliche und technische Pensum nicht ausreicht, wird die Rolle des Kühlschranks auf den Server adaptiert und simuliert.

Online-Shop
Auch der geplante Online-Shop ist in der ersten Skizze existent. Trotzdem erhält die Realisierung dieses Features zunächst eine niedrigere Priorität, da es sich nicht um eine der Kernfunktionalitäten handelt, aber durchaus einen außerordentlichen konzeptionellen und programmiertechnischen Aufwand bedingen würde. Nachrichten zwischen ihm und dem Server könnten synchron modelliert werden, wenn sichergestellt werden kann, dass der Online-Shop permanent erreichbar ist.
Nachrichten zwischen Server und Online-Shop könnten aber auch in dem Sinne asynchron modelliert werden, dass der Server erstmal Bestellanfragen des Clients entgegen nimmt, zwschenspeichert, dann zuächst eine verbindungsorientierte Interaktion zum Online-Shop sicherstellt und erst im letzten Schritt die Bestellaufträge an den Online-Shop übermittelt. Dadurch soll eine unbeabsichtigte Bestellung bedingt durch einen Verbindungsabbruch verhindert werden.
Die architekturelle Ausarbeitung dieser Komponente stellt sich als sehr problematisch dar, da noch auf einige Aspekte mehr eungegangen erden muss: wird dieser von den selben Personen bereitgestellt und verwaltet, die auch die Capability dieses Systems bereitstellen? Oder handelt es sich bei dem Online Shop um eine losgelößte Komponente, die einer völlig anderen Struktur folgt, sowohl auf architektureller Ebene als auch auf Datenebene? In diesen Fällen müsste eine geeignete Schnittstelle zu bereitsvorhandenen Shops des Internet realisiert werden.
Aufgrund der geringeren Priorität wird an dieser Stelle das Konzept seiner Logistik noch nicht fixiert.

5.2 Meilenstein 2: die Semantik der HTTP-Operationen für das Projekt

Beschreibung der Aufgabe dieses Meilensteins
Die Aufgabe dieses Meilensteins besteht darin benötigte Ressourcen zu bestimmen und die Semantik aller vier HTTP-Methoden für jede Ressource festzulegen.

Begründung Vertauschung der Meilensteine
Durch scharfes hinsehen, kann festgestellt werden, dass die vorgegebene Reihenfolge der Meilensteine an dieser Stelle nicht hunderprozentig beachtet wird. Insofern ist ein Austauschen des ersten und zweiten Meilensteins sinnvoll, dass im zweiten alle Ressourcen bestimmt und erst dann im ersten für jede einzelne ein XML-Schema angelegt werden. Woher kann man wissen, zu welchen Ressourcen im ersten Meilenstein die XSD-Dateien zu erstellen sind, wenn diese noch gar nicht feststehen, ohne die Lösung für den ersten Meilenstein im Nahinein noch ergänzen zu müssen? Oder anders gesagt: Meilenstein 2 "Semantik der HTTP-Operatoren" betrachtet das Projekt von oben und Meilenstein 1 "XML-Schemata" geht noch ein wenig tiefer auf die Ebene der Programmierung. Der Top-Down-Ansatz erscheint angemessener.

5.2.1 Festlegung aller Ressourcen

Geplante Ressourcen:

Bezeichnung der Listenessource	Bedeutung
Fridges	Speichert Daten zu **allen Kühlschränken**, die das System verwalteten kann.
Profiles	Speichert Daten zu **allen Accounts**, die aktuell in der Gesamtheit aller Kühlschränke registriert sind.
Producttypes	Speichert Daten zu **allen Produktarten**, die aktuell erwerblich sein können.
Products	Speichert Daten zu allen aktuell verwalteten, konkreten **Instanzen eines Producttypes**.
Notifications	Speichert **alle Benachrichtigungen** und **Warnungen**, die in einem bestimmten Zeitraum im Rahmen der Gesamtheit aller Kühlschränke verschickt wurden.
(Fitnesstrainer)	Speichert Daten über Nährstoffe etc, über die die Fitness eines Menschen berechnet werden kann, für alle aktuell existierenden Accounts.

Bei diesen Ressourcen handelt es sich konsequent um Listenressourcen, die als Subressourcen jeweils die einzelnen Instanzen dieser Listen enthalten, vergleichbar mit der Beziehung zwischen Klassen und Objekten in Java:

Bezeichnung der Subressource	Bedeutung
Fridges/{id}	Speichert Daten einer **konkreten Ausprägung** eines aktuellen, verwaltbaren **Kühlschranks** mit der unverwechselbaren Kennung "id".
Profiles/{id}	Speichert die Daten einer **konktreten Ausprägung** eines aktuell aktiven **Accounts** unter der Gesamtheit aller Accounts und mit der eindeutig identifizierenden Kennung "id".
Producttypes/[id}	Speichert die Daten einer **konkreten Ausprägung** einer bestimmten **Produktart** mit der eindeutig identifizierenden Kennung "id".
Products/{id}	Speichert die Daten einer **konkreten Ausprägung** eines expliziten, aktuell verwalteten **Produktes** mit der eindeutig identifizierenden Kennung "id".
Notifications/[id}	Speichert die Daten zu einer **konkreten Ausprägung** einer **Benachrichtigung** oder **Warnung**, die in einem gegebenen Zeitraum entstanden ist mit der eindeutig identifizierenden Kennung "id".

Listenressourcen mit Subressourcen und ID
Die Pragmatik bei dieser Wahl ist es, jeweils Elemente des gleichen Ressourcentypens zusammenzufassen und gemeinsam, aber auch jedes Listenelement einzeln ansprechen zu können. Diese URI-Struktur kann Vorteile bieten, unteranderem erleichert das URI-Hacking durch gezielt manuelles Bearbeiten der URI-Zeile im Browser dem Sysetmprogrammierer die Arbeit und macht sie komfortabler. Mittels ID-Ressourcen kann zielstrebig auf einen Sohn einer Listenressource zugegriffen werden, ohne die Daten der restlichen Brüder mitübertragen zu müssen. Beispielsweise sollen Daten zu genau einem konkreten Produkt angezeigt werden. Durch die ID kann eine Repräsentation dieser Daten direkt adressiert werden. Es muss nicht die detailsreiche Liste aller Profile versendet werden. Da dadurch kleine Datenpakete übertragen werden können, wird das Netzwerk weniger stark pro Aufruf einer Ressource belastet. Diese

Datenstruktur hat im Bereich der RESTful-Architekturen eine begründet essenzielle Daseinsberechtigung.

Unterscheidung Product und Producttype, Produktinstanz und Produkttypinstanz
Variante1: Unterscheidung der vier, oben angegebenen, Ressourcen

Producttypes Producttypes/{id} Products Products/{id}

Es wird zwischen dem einzelnen Produkt und seiner zugehörigen Produktart differenziert, da bei Produkten der selben Art zwar teilweise gleiche Informationen wie enthaltene Inhaltsstoffe und Nährstoffe vorliegen, sie sich aber zusätzlich in einigen Punkten wie Verfallsdatum, Erwebsdatum oder Inhaber unterscheiden können. Es ist für den Benutzer auch zweckmäßiger nur alle erwerbbaren Lebensmittelarten anzeigen zu lassen und sich erst im nächsten Schritt auf die detailiertere Ebene ihrer konkreten Instanzen zu begeben. Dadurch sollten unnötige Redundanzen unterschiedlicher Produktinstanzen aber gleichen Typs bei der Übermittlung eliminiert werden. Leider erfordert diese Variante die Installierung von ganzen vier Ressourcen, alleine um Produkte gezielt filtern zu können.

Variante 2: Unterscheidung zwischen Listen- und Subressource für Produkte

Products	<-Kombination aus der ehemahligen "Produts"- und "Producttypes"-Ressource
Produs/{id}	<-Kombination aus der ehemahligen "Products/{id}"- und der "Producttypes/{id}"-Ressource

Man kann auch jeweils beide Ressourcen zu einer Products-Ressource und Products/{id}-Ressource zusammenfassen. Eine Repräsentation der Products/{id}-Ressource enthält dann Informationen zu einer expliziten Produktart und listet zusätzlich alle Vertreter dieser Produktart samt jeweils zugehörigen Informationen wie Verfallsdatum etc. pro konkrete Produktinstanz auf. Diese Variante komprimiert die Anzahl der bereitzustellenden Ressourcen und korrelierend damit auch die Anzahl der zuimplementierenden Klassen des REST-Webservices. Dies muss aber partout keinen Vorteil darstellen. Für den Systemadministrator ist die erleichtere Darstellung durch Differenzierng zwischen Produktart und Produktinstanz bei der Wartung der Daten wichtiger als eine kompaktere Anzahl an zuimplementierenden Ressourcen.

Variante 3: Eine einzelne Ressource für Produkte

Products <- Kombination aus allen vier ursprünglichen Ressourcen

Eine weitere Möglichkeit bestünde darin, alle vier Ressourcen zu einer riesigen Ressource zu verinen und damit die essenzielle Struktur in Listenressourcen zu umgehen. Dabei müssten erneut weniger Ressourcen implementiert, aber die einzelne große Ressource mit mehr HTTP-Operationen versehen werden. Auf den ersten Blick erscheint es so, als müsse bei dieser Variante immer die gesamte Liste aller auf der Welt existierenden Produkte mit samt ihren Details übertragen werden, sei die eigentlich benötigte Menge an Informationen noch so gering. Produkte können anscheinend nicht einzeln angesprochen werden. Dieses Manko kann aber wiederum durch die Einsetzung von Queryparams, die z.B. Produkte nach bestimmten Typen filtern behoben werden.

Entscheidung: Variante 1
Die Aufteilung in vier scheint am sinnigsten, wenn das hohes Aufkommen an Daten der Produkte der gesamten Welt in Betracht gezogen wird, das außerdem auch schnell sehr unübersichtlich wird. Die

gezielte Filterung könnte zwar auch durch die Queryparams realisiert werden, jedoch wird sich für die Möglichkeit der URI-Manipulation zwischen den vier separierten Ressourcen beabsichtigt. Die Quryparams werden für eine noch feinere Selektion eingesetzt.

Varianten: Fitnesstrainer
Variante 1: Keine eigene Ressource, nur Teil der Profile-Ressource
Die Ressource "Fitnesstrainer" wird hier zwar aufgeführt, weil sie einen Teil einer frühren Variante des Ressourcendesigns darstellte, aber im Allgemeinen lassen sich seine Daten perfekt mit denen der Profil-Ressource verbinden, gerade weil der Fitnesstrainer exakt einem Profil zugeschrieben wird. Eine Repräsentation einer Profil-Ressource kann daher auch direkt Informationen zur Fitness enthalten, ohne eine separate Ressource für die Fitnessdaten vorauszusetzen.

Variante 2: Ressourcenpaar Fitnesstrainer & Fitnesstrainer{id}

Fitnesstrainer Fitnesstrainer/{id}

Selbstverständlich kann auch diese Ressource konsequenterweise erneut in ein Ressourcenpaar "Fitnesstrainer" und "Fitnesstrainer/{id}" aufgeteilt werden. Dabei enthält jedes Teilelement, eine Referenz auf das zugehörige Profil. Konforme IDs zwischen Fitnesstrainer-ID und seiner zugehörigen Profile-ID können die Verbindung der beiden erleichern.

Variante 3: eine separate Ressource (leichtes Auswerten von Fitnessdaten)

Fitnesstrainer

Durch eine große Ressource "Fitnesstrainer", die Fitness-Informationen zu allen Profilen enthält, kann dazu benutzt werden das Essverhalten und die Fitness der großen Maße komfortabel zu analysieren, ohne das Netzwerk und die Serverleistung durch eine häufige Sendung kleiner Nachrichten zu belasten, da nur eine große Auflistung alles Fitnessdaten übertragen werden kann. Dieser Aspekt ist sehr wahrscheinlich für die meisten Endverbraucher irrelevant. Da stellt sich die Frage, ob für den Entwickler oder für die Benutzer desingt werden soll.

Notifications direkt in zugehörigen Kühlschrank- oder Profilinformationen speichern
Bei dieser Ressource stellen sich, aufgrund der Tatsache, dass Notifications sich immer speziell auf einen Kühlschrank oder ein Profil beziehen, die gleichen Fragen wie beim Fitnesstrainer. Ist es angemessen die Benachrichtigungen von seinem zugehörigen Vaterelement zu trennen? Bei dieser Ressource erscheint es tatsächlich zweckvoller diese beiden Gruppierungen von einander zu trennen, da, bei einer Anrage der Profilressource, allenfalls eine erhebliche Menge an Benachrichtigungen mitübertragen werden müssten, obwohl nur die Informationen bezüglich eines Profils abgerufen werden sollten. Dies belastet das Netzwerk. Jedoch könnten die, für den Moment irrelevanten, Daten der Benachrichtigungen durch Queryparams wieder heraus gefiltert und nur die interessanten Profilinformationen übertragen werden.

Ressourcendesign prinzipiell nicht so wichtig
Das Ressourcendesign muss nicht haarklein ersonnen und durchdacht sein. Es ist auch nicht unabdingbar, dass das Ressourcendesign, die gleiche Speicherstruktur der persistenten Datenspeicherung reflektiert. Die Essenz liegt größten Teils in der Speicherstruktur und Informationsmenge auf der Datenbank und eines gezielten Erdenkens von zeckgemäßen Java-Methoden bezüglich dieser Daten. Denn durch das Vorhandensein der richtigen Daten in der Datenbank, können alle nötigen Informationen einer Repräsentation einer Ressource errechnet und beliebig zusammen gesetzt werden. Die Ressource beschreibt lediglich den Informationsgehalt und

Informationsstrukturierung einer transienten Nachricht.
Beispielsweise kann die Datenbank feingranular strukturiert sein, währenddessen nur eine Ressource definiert ist, die absolut alle Daten der Datenbank abbilden kann, und je nach Anwendungsfall auch nur eine kleine Teilmenge der Daten enthält. Der umgekehrte Fall ist natürlic auch implementierbar. Auf der Datenbank befindet sich eine einzelne Datei, in der alle Informationen gespeichert sind. Diese kann durch eine vielzahl an Ressourcen immer wieder anders gefiltert und dargestellt werden.
Eine Ressource kann auch so variabel definiert werden, dass sie auf das Projekt des "Fridgemanagers" bezogen und mit vielen Filtermechanismen geäußert durch z.B. Queryparams, sowohl ein Profil als auch eine Liste von Produkten als eine mögliche Repräsentation liefert. Dann wäre es möglich auch mit nur einer einzigen Ressource für ein komplettes System auskommen.
Diese Varianten mögen nicht sehr sinnvoll erscheinen, aber diese Beispiele sollten auch nur verdeutlichen, dass das Ressourcenesign keine sonderlich große Rolle bei der Implementierung eines verteilten Systems darstellt.
Generell zieht ein Webentwickler mehr Vorteile aus einer URI-Struktur, die leicht manuel zu hacken ist, als der Endverbraucher. Zumal er, als Person mit Domänenwissen der Informatik, den Aufbau einer URI eher kennt als ein durchschnittlicher Endverbraucher. Für den ihn könnte diese unterumständen kryptisch erscheinen. Außerdem ist der "Fridgemanager" als Applikation für ein mobiles Engerät geplant, das voraussichtlich kein URI-Hacking unterstützt.
Grundsätzlich wird sich für die Variante entschieden, die es dem Systemprogrammierer erleichtert die Übersicht zu behalten, da die gewünschte Funktionalität für den Benutzer durch die spätere Implementierung der Java-Methoden auch noch nachträglich sichergestellt werden kann. Alle Komponenten, Client-Applikation, Server, REST-Schnittstelle und Datanbank, müssen lediglich im Einklang zu einander stehen und aufeinander abgestimmt sein.

5.2.2 URI-Design – Hierarchisierung der Ressourcen

Variante 1: Alle Daten von Kühlschrank abhängig
Eine extrem verschachtelte Variante der möglichen URI-Struktur mit vielen Subressourcen und Subressourcen von Subressourcen zeigt sich wie folgt:

```
Fridges/
Fridges/{id}
Fridges/{id}/Producttypes
Fridges/{id}/Producttypes/{id}
Fridges/{id}/Producttypes/{id}/Produtcs
Fridges/{id}/Producttypes/{id}/Produtcs/{id}
Fridges/{id}/Profiles
Fridges/{id}/Profiles/{id}
Fridges/{id}/Profiles/{id}/Notifications
Fridges/{id}/Profiles/{id}/Notifications/{id}
```

Bei diesem URI-Design wird Wert daraufgelegt alle Daten genau einem Kühlschrank zuordbar zu machen und sie auch nur über diesen einen Kühlschrank abfragen zu können. Die Idee, die dahinter stand, ist es die Sicht des Benutzters in das URI-Design mit einfliessen zu lassen, der sich speziell nur für seinen eigenen Kühlschrank interessiert und auch aus Datenschutzgründen gar nicht befugt sein darf, Daten anderer Kühlschränke einzusehen. Durch die Hierarchische Anordnung entsteht eine Struktur, die die Beziehungen der einzelnen (Sub-)Ressourcen zueinander mit einem Blick erfassen lässt. Das ist neben einer erleichterten URI-Manipulation quasi der einzige Vorteil. Nachteilig dabei wäre der denkbare Fall, dass ein Benutzer nicht nur einen, sondern auch mehrere Kühlschränke verwalten möchte. Denn dann könnte dieser seine Produkte beispielsweise nicht ohneweiteres Kühlschrank übergreifend einsehen und müsste auch umständlich für jeden Kühlschrank ein neues Profil mit neuen Logindaten anlegen, das wahrscheinlich auch nicht equivalente Daten zu dem ersten Profil beinhaltet. Funktionen wie der

Fitnesstrainer würden nicht Profil übergreifend anwendbar sein, und somit ihren Zweck verfehlen.
Um die Beziehungen der Ressourcen zueinander herzustellen, müssen sie jedoch nicht ineinander verschachtelt sein. Es kann auch das Konzept der Hyperlinks genutzt werden. Aus der starr hierarchischen Struktur kann dadurch eine flexibel netzartige werden. Dieser Gedanke fürht zur nächsten Variante.

Variante 2: Produkte & Profile vom Kühlschrank entkoppeln

```
Fridges/
Fridges/{id}
Producttypes
Producttypes/{id}
Producttypes/{id}/Produtcs
Producttypes/{id}/Produtcs/{id}
Profiles
Profiles/{id}
Profiles/{id}/Notifications
Profiles/{id}/Notifications/{id}
```

Durch die Abkopplung der Ressourcen "Profiles", "Products" und "Producttypes" von der Primärressource "Fridges" wird sowohl die Verbindung von mehren Kühlschränken zu einem Profil gefördert, als auch die Möglichkeit einem Kühlschrank mehrere Profile zuweisen zu können beibehalten. Darüber hinaus können die drei Ressourcen unbeachtet ihrer zugehörigen Kühlschränke angesprochen und eine Repräsentation ihrer Daten angefordert werden. Die logische Verbindung zwischen den einzelnen Instanten werden durch Hyperlinks realisiert. So enthält beispielsweise jeder Kühlschrank die Referenzen auf alle Profile, die er verwaltet und umgekehrt auch jedes Profil Referenzen aller Kühlschränke, deren Informationen es bezieht.

Variante 3: Notifications ebenfalls unabhängig einer Primärressource

An dieser Stelle stellt sich die Frage, ob sowohl die Products als auch die Notifications separiert von ihren jetzigen Superressourcen aufgeführt werden sollen, da sie eventuell auch gesondert angesprochen werden sollen.

```
Fridges/
Fridges/{id}
Producttypes
Producttypes/{id}
Produtcs
Produtcs/{id}
Profiles
Profiles/{id}
Notifications
Notifications/{id}
```

Für die Notifications ist es denkbar, da auch Benachrichtigungen infrage kommen, die sich teilweise nicht nur an einen Empfänger richten und daher auch keiner deterministischen Supressource untergeordnet werdne können. Da greift die gleiche Pragmatik, die bei der Veranlassung der Entkopplung von Produkten und Profilen von der Primärressource ausschlaggebend ist: Realisierung einer m-zu-m-Beziehung zwischen Benachrichtigungen und zugeordenten Profilen. Eine Benachrichtigung, die sich an mehrere Profile richtet, ist beispielsweise ein asynchrone Nachricht, die alle Benutzer desselben Kühlschrankes auf die Überschreitung des Mindesthaltbarkeitsdatums eines sich im Kühlschrank

befindlichen Produktes hinweist.
Ein zentrales Kühlschrankprofil repräsentativ für die Gesamtheit der Benutzer desselben Kühlschrankes könnte Abhilfe verschaffen, wenn Nachrichten nicht nur einem Profil sondern allen Profilen eines Kühlschrankes zugehörig sind.
Um Notifications besser aswertbar zu machen und konsequent die übersichtliche Struktur von Listen- und Subressource beizubehalten, werden die Ideen dieser Variante bezüglich der Benachrichtigungen übernommen.

Im Falle der Beziehung zwischen Produkten und ihren Typen, kann nicht auf diese Weise argumentiert werden, da es sich hierbei tatsächlich um eine hierarchische Beziehung handelt. Jede konkrete Produktinstanz gehört auch nur einer konktreten Produktart an. Daher kann ein existenzabhänginge Struktur von Superressource "Producttypes" zu Subressource "Products" als zweckentsprechend angesehen werden. Aus einer anderen Perspektive kann wiederum argumentiert werden, dass diese hierarchische Verbindung erneut durch Querverweise realisiert werden kann. Aber welcher Anwendungsfall ist denkbar, bei dem eine komplette Liste aller konkreten Produktinstanzen abgefragt werden?

Fitnesstrainer

```
Fridges/
Fridges/{id}
Producttypes
Producttypes/{id}
Producttypes/{id}/Produtcs
Producttypes/{id}/Produtcs/{id}
Profiles
Profiles/{id}/Fitnesstrainer (?)          <- Variante 2
Notifications
Notifications/{id}
Fitnesstrainer (/{id}) (?)                <-Variante 1
```

Variante 1: Fitnesstrainer als einzelne Primärressource oder Paar aus Listen- und Subressource
Eine Variante die Daten des Fitnesstrainers abfragen zu können, bestünde darin, ihn als Primärressource anzubieten. Der Bezug zu dem jeweils zugehörigen Profil könnte erneut über Hyperlinks und nicht durch die hierarchische Anordnung der URI-Struktur erreicht werden. Eine komplette Liste der Daten aller Profile über ihre Fitness, kann ihre Auswertung ungemein erleichtern, da nicht für jedes Profil einzeln Daten zur Fitness übermittelt werden müssen. Diese Auswertungen können für die Industrie oder Forschungszwecke benutzt werden. Natürlich müsste bei der Wahl dieser Variante eine zusätzliche Ressource mit eigenen Methoden implementiert werden. Dies bedeutet mehr Aufwand für einen Designschritt, der auch auf eine simplere Weise umgesetzt werden kann und trotzdem ein zweckgemäßes Ergebnis liefert.
Aus Konsistenzgründen könnte der Fitnesstrainer auch wieder in das essentielle Paar von Listenressource und Subressource unterteilt werden. Dadurch würde aber der einzige Vorteil, diese Daten erleichtert auswerten zu können, außer Kraft gesetzt werden, da trotzdem jede XML-Nachricht, bezüglich der Fitness-Informationen eines Profils, einzeln aufgerufen werden müsste.

Variante 2: Fitnesstrainer als Subressource eines konkreten Profils
Die Fitnessdaten bezieht sich ohnehin auf exakt ein konkretes Profil, daher ergibt es Sinn sie auch rein Profil abhängig anzubieten. Der Vorteil der erleichterten Auswertung der Daten geht dabei verloren.

Variante 3: Fitnesstrainer als Inhalt eines Profils
Der Fitnesstrainer braucht auch keine eigenständige Ressource sein. Seine Daten können auch mit denen

eines Profils übertragen werden, da er sich erstens nur auf genau ein Profil bezieht und zweitens auch keine sonderlich große Menge an ergänzenden Daten umfasst.
Dadurch wird der Vorteil der erleichterten Auswertung sogar doppelt ausgelöscht. Auf der einen Seite müssen alle Profile durchgegangen werden, um die Fitness-Informationen aller Personen zu sammeln, auf der anderen Seite müssen diese Daten noch einer Filterung unterzogen werden, um Fitness-Informationen von den restichen Profildaten zu trennen. Jedoch erfordert diese Variante keine Implementierung einer weiteren Ressource. Der Aufwand scheint dadurch oberflächlich gesehen geringer zu sein.
Letztendlich zeigen alle drei Varianten Vor- und Nacheile, die sich quasi die Waage halten. An dieser Stelle müsste näher festgelegt werden, auf welchen Aspekt am meisten Wert gelegt werden soll. Entweder soll das System eine simple Wartung der Daten sicherstellen, oder eine nachvollziehbarere URI-Struktur bereitstellen. Jedoch stellt der Fitnesstrainer vergleichsweise mit dem Online-Shop keine zentrale Kernfunktion dar, daher erhält seine Umsetzung ebenfalls eine niedrigere Wichtigkeit als andere Funktionen.

5.2.3 Semantik der HTTP-Operatoren

Folgende Semantik wurde den REST-Operationen der festgelgten Ressourcen zugewiesen:

Bezeichnung der Ressource	Semantik der HTTP-Opearionen
Fridge	**GET**: liefert Liste aller verwaltbaren Kühlschränke **POST**: Erstellung einer neuen Fridge-Instanz
Fridges/[id}	**GET**: liefert Daten einer konkreten Kühlschrankes mit der angegebenen id **PUT**: Äderung der Daten des Kühlschrankes mit der angegebenen id **DELETE**: Löschung des Kühlschrankes mit der eingegebenen id
Profiles	**GET**: liefert Liste aller aktuell verwalteten Accounts **POST**: Erstellung eines neuen Accounts
Profiles/{id}	**GET**: liefert Daten eines konkreten Accounts mit der angegebenen id **PUT**: Änderung der Daten des Accounts mit der angegebenen id **DELETE**: Löschung des Accounts mit der angegebenen id
Producttypes	**GET**: liefert Liste aller aktuell verwaltbaren Produktarten **POST**: Erstellung einer neuen Produktart
Producttype/{id}	**GET**: liefert Daten einer konkreten Produktart mit der angegebenen id **PUT**: Änderung der Daten eines Produktart mit der angegebenen id **DELETE**: Löschung einer Produktart mit der angegebenen id
Products	**GET**: liefert Liste aller aktuell verwalteten Produkte **POST**: Erstellung eines neuen Produktes einer ausgewählten Produktart
Products/{id}	**GET**: liefert Daten eines konkreten Produktes eines konkreten Produkttypens mit der angegebenen id **PUT**: Änderung der Daten eines konkreten Produktes mit der angegebenen id
Notifications	**GET**: liefert Liste aller in einem bestimmten Zeitraum erstellten Benachrichtigungen und Warnungen (Client kann keine Benachrichtigung selbst erstellen)
Notifications/{id}	**GET**: liefert Daten einer konkreten Benachrichtigung eines bestimmten Zeitraumes mit der angegebenen id (Client kann keine Benachrichtigung im Nachhinein verändern oder löschen)

Vereinbarung bezüglich der Operatoren
GET kann auf alle Primär- und Subressourcen angewendet werden. Der Client kann demnach auf alle Listenressourcen und deren einzelnen Instanzen gezielt zugreifen. Dies erscheint angemessen, da die Unterscheidung zwischen Listenressource und ihrer Subressource nur gewählt wurde, um sowohl die komplette Liste anzufragen, als auch gezielt eines ihrer Elemente.
POST wird nur auf die primären Listenressourcen angewendet und bewirkt die **Erstellung** einer neuen Instanz zu der ausgewählten Primärressource. PUT wäre an dieser Stelle auch möglich, aber bei einem PUT wird auf eine Instanz mit einer konkreten ID bzw. einer vorher bestimmten URI zugegriffen. Für die Erstellung einer neuen Instanz sollte nicht der Client die ID bestimmen müssen, sondern der Server, da nur dieser weiß, welche IDs noch verfügbar sind. Für den Client und auch für das Netzwerk würde es einen enormen Aufwand darstellen alle möglichen IDs durchzugehen und immer wieder zu erfragen, ob diese noch verfügbar ist.
PUT wird nur für die **Änderung** einer bereits vorhandenen Ressource benutzt, zum einen weil diese Methode die URI einer konkreten Instanz einer Ressource verlangt und nicht auf die gesamte Listenressource angewendet werden kann, und zum anderen um die beiden Operationen PUT und POST konsequent von einander zu trennen.
DELETE gilt auch nur konkrete Instanzen einer Listenressource. Um eine komplette Liste zu löschen müssen alle ihre Elemente einzeln gelöscht werden. Diese Konvention wurde getroffen, da der Anwendungsfall "lösche eine komplette Liste" beim "Fridgemanager" niemals auftritt. Der Server soll generell auch bereits verbrauchte Produkte persistent speichern, um eine Art Produkt-History, die . Daher ist clientseitig keine Löschung vorgesehen, um die Produkt-History nicht so zuverfälschen. Andernfalls wird der Fitnesstrainer seine Daten nachträglich nicht mehr beziehen können. Möglich ist es auch die Fitness-History nicht nachträglich anfragen zu können, sondern die Daten jedes Tages direkt in dem jeweiligen Profil zu speichern. Diese Variante würde die Profil-XML nur weiter mit Daten belasten, die auch ausgelagert werden können.
Außerdem ist diese Produktliste auch für die Berechnung des Einkaufszettels, die Produkte und deren Preise beschreibt, die ein Nutzer im Laufe einer gegebenen Zeitspanne in der Vergangenheit erworben hat, relevant. Daher wird sich gegen einen clientseitigen DELETE-Mechanismus entschieden.

Notifications:
Die Notifications werden grundsätzlich nur serverseitig ausgeworfen, da der Server imstande die benötigten Daten dafür auszuwerten und gegebenenfalls daraus resultierende Benachrichtigungen, über z.B. die akute Überschreitung des Mindesthltbarkeitsdatums eines Produktes, mitzuteilen.
Natürlich können serverseitig generierte Nachrichten auch nicht nachträglich verändert werden.
Es ist auch keine clientseitige Löschung von Benachrichtigungen vorgesehen, wei die Notification-History dadurch verfälscht werden würde. Die persistente Speicherung von nicht wahrheitsgetreuen Daten liefert dem Nutzer keinen Informationsgehalt und ist somit unssinnig. Der Server soll Nachrichten nach einer bestimmten Zeit selber löschen, um den begrenzten Speicher nicht durch eine gegen Unendlich konvergierende Anzahl an Daten strapaziert, die nach Ablauf einer gewissen Zeitspanne keinen Informationsgehalt mehr für den Benutzer darstellen.
Daher ist nur die Anwendung der Get-Methode für diese Ressourden möglich.

POST von usergenerierten Chat-Nachrichten möglich
Bei Einführung eines Chatsystems würde diese POST-Sperre jedoch aufgehoben. Es könnten auch clientseitige Messages verschickt werden. Auf REST-Ebene würde die Erstellung einer Chatnachricht durch einen POST-Befehl auf eine Ressource, die alle Chatnachrichten verwaltet, realisiert werden. Diese Funktionalität kann entweder im Rahmen der Notifications verwirklicht werden oder bedürfe einer ergänzenden Chat-Messages-Ressource.

POST und PUT auf Produkttypen
Ein Client muss trivialerweise in der Lage sein neue Profile zuerstellen und bereits vorhandene zu verändern.

Aber sollen Benutzer auch noch Produkte und Produkttypen durch einen POST erstellen bzw. Verändern können?
Der Fridgemanager soll in der Lage sein alle Lebensmittel des Haushaltes verwalten zu können, ansonsten würde unteranderem das Fitnesstrainer-Feature nicht in gänze funktionieren. Daher erscheint die autonome Erstellung von individuellen Produkten auf Seite des Nutzer angemessen, da dieser so die Möglichkeit erhält auch Produkte die nicht nach den Normen der Industrie gestaltet sind zu verwalten, z.B. einen selbst gebackenen Kuchen.
Trotzdem kann nicht sichergestellt werden, dass der Benutzer alle Daten seines selbst erstellten Produkttypen wie Calorien- oder Nährwertgehalt überhaupt kennt und der Realität entsprechend ausfüllen kann. Somit würde der Fitnesstrainer erneut seinen Zweck nicht erreichen können. Außerdem müsste man sich auf eine Welle an neu erstellten Produktarten gefasst machen, die alle zwar nicht identisch aber ähnlich zu einander sind. Für diese Semiredundazen müsste noch ein Mechanismus gefunden werden, der ungewollten Spam verhindert. Parallel zu der Erstellung eines neuen Produkttypen kann das System eine Prüfung durchführen, ob ein ähnlicher Produkttyp bereits von einem anderen Nutzer angelegt wurde, und dem aktuellen Nutzer die Wahl lassen, ob dieser den bereits vorhandenen Produktypen übernommen werden soll.Solche Alorithmen für eine Maschine zu entwickeln, ist jedoch schwierig.
Die beste Lösung besteht darin, den User keine neuen Produkttypen selbst erstellen, sondern ihn auf bereits vorgefertigte Produkttypen zurückgreifen zu lassen, deren Liste permanent ergänzt werden, da den Nutzer sowohl die Möglichkeit geboten wird ihre individuellen Produkte zu verwalten, als auch der Spam dadurch unterbunden wird.

PUT und POST auf Products
Der Client muss imstande sein neue Produkte eines Produkttypens zu erstellen, damit diese auch verwaltet werden können. Im Idealfall werden diese Produkte nicht manuell vom Benutzer erstellt, sondern automatisch bei der Erfassung durch die Scanner am Kühlschrank. Da die Scanner nicht vorhanden sind, werden die Funktionen POST und PUT dem Benutzer zur manuellen Manipulation in der Applikation bereitgestellt. Das Produkt wird nicht durch DELETE gelöscht sondern nur der aktuelle Status durch PUT von "inside" auf "consumed" gesetzt, da der Zugriff auf diese Daten noch für die Berechnung der Einkaufsliste pro Monat oder für den Fitnesstrainer erhalten bleiben müssen.

Auswirkungen von Idempotenz
Da GET, PUT und DELETE aufgrund ihrer Idempotenz bei einer mehrfachen Anwendung immer wieder das gleiche Resultat liefern und die gleichen Seiteneffekte veranlassen, kann bei einem unerwarteten Netzwerkabbruchs die gleiche Methode erneut ausgeführt werden, ohne unbeabsichtigte Seiteneffekte hervorzurufen. Dieses Wissen ist in sofern wichtig, als das bei einem Abbruch der Verbindung während der Client eine HTTP-Methode auf eine Ressource anwendet,
1. die Anfrage entweder gar nicht den Server erreicht hat, oder
2. sie bereits von diesem ausgeführt wurde und bloß keine Antwort mehr zurück an den Client versendet werden konnte. Der Client kann ohne weiteres nicht erahnen, welcher dieser beiden Fälle eingetreten ist, müsste aber bei einer nicht idempotenten Operation, das Resultat kennen, um entscheiden zu können, ob eine erneute Ausführung der Operation getätigt werden muss, oder diese bereits beim ersten Mal den Server erreicht hat und daher nicht noch einmal ausgeführt werden sollte, um unbeabsichtigte Seiteneffekte zu vermeiden. Für nicht idempotent verschickte Nachrichten können als sichernde Maßnahne Selbstbestätigungsketten eingeführt werden. Durch diese kann immer bestimmt werden, welcher der beiden oben beschriebenen Fälle bei einem Verbindungsabbruch eingetreten ist.
Bei den idempotenten Methoden würde in beiden Fällen eine erneute Anwendung das gleiche gewünschte Resultat ohne unbeabsichtigte Seiteneffekte liefern, daher ist es bei diesen Methoden nicht zwingend gefordert den eingetretenen der beiden Fälle zu bestimmen, zumal eine Einführung von Selbstbestätigungsketten für Ausnahmslos jede Nachricht den Traffic drastisch erhöhen würde.

5.3 Meilenstein 1: Projektbezogenes XML Schema / Schemata

Schemadateien der synchronen Kommunikation
Die vorher definierten Ressouren helfen enorm bei Fertigstellung dieses Meilensteines, da zu jeder REST-Ressource eine XML-Schemadatei erstellt werden kann, weil das Anbieten einer Ressource auch die Übermittlung der zu ihr passenden Daten impliziert. Ansonsten würde die Ressource ihrem zentralen Zweck, abrufbar zu sein, nicht gerecht werden.

Folgende Schemadateien sind geplant:
(Beschreibung der Alternativen im Anschluss an die rein deskriptive Tabelle)

Bezeichnung der Ressource	Zugehörige XSD-Bezeichnung	Beschreibung ihres Aufbaus
Fridge	Fridges.xsd	Liste: -aller Kühlschrank-IDs -Namen aller aktuellen Kühlschränke -eine Referenz auf jeden konkreten Kühlschrank
Fridges/[id}	Fridge.xsd	Daten zu einem konkreten Kühlschrank: -Kühlschrank-ID -Name des Kühlschranks Accountliste aller von ihm verwalteten Profile mit: -Account-ID, -Profilnamen und -Refezenz auf das jeweilige Profil Produkttypenliste: -Auflistung aller aktuell in diesem Kühlschrank befindlichen Produktarten mit Produktart-ID, Name, Anzahl der Instanzen dieser Produktart im aktuellen Kühlschrank, Mindestbestand, Maximalbestand und Referenz auf die jeweilige Produktart und eine weiter verschachtelte Produktliste zu jedem Produkttypen mit: -seinem jeweiligen Status (verbraucht oder im Kühlschrank enthalten) -Refezenz auf konkrete Produkte, - Produkt-IDs und - Produkt Namen
Profiles	Profiles.xsd	Liste aller Accounts mit: -Accountname, -Account-ID -Referenz auf Profil Liste zugehöriger Kühlschränke mit: -Referenz auf Kühlschrank -Kühlschrankname

		-Kühlschrank-ID
Profiles/{id}	Profile.xsd	Daten zu einem konkreten Account mit: -Account-ID -Accountname -Geburtsdatum -Geschlecht -Größe -Gewicht Liste zugehöriger Kühlschränke mit: -Referenz auf Kühlschrank -Kühlschrankname -Kühlschrank-ID Liste aller zugehörigen Produkte mit: -Referenz auf das Produkt -Produkt-ID -Produktname (-Einkaufsliste aller jemals erworbenen Produkten) (-Informationen des Fitnesstrainers)
Producttypes	Producttypes.xsd	Liste aller verwalteten Produkttypen mit jeweils: -Produkttypen-Name & Referenz auf den konkreten Produkttypen
Producttype/{id}	Producttype.xsd	-Produkttypen-Name -Produktbeschreibung -Inhaltsstoffe -Nährwerte -Barcode -Preis
Products	Products.xsd	Liste alle verwalteten Produkte mit jeweils: -zugehöriger Produkttypname & Referenz -zugehöriger Kühlschrank-name & Referenz -Status -Referenz auf konkrete Produktinstanz
Products/{id}	Product.xsd	-Produktypenname -Referenz auf Produkttypen -zugehöriger Kühlschrank-Name & Referenz -Einlesedatum -Datum des Verzehrs -Mindesthaltbarkeitsdatum -Name und Referenz des Besitzer-Profils -Preis -Stauts (konsumiert oder im Kühlschrank enthalten)
Notifications	Notifications.xsd	Liste mit Daten zu allen Notifications innerhalb eines bestimmten Zeitraumes: -Notification-ID -Referenz aufEmpfänger-Profil -Name des Empfänger-Profils -ID des Empfänger-Profils -Versanddatum der Notification

		-Art der Notification (Warning oder Benachrichtigung) -Überschrift der Nachricht -Referenz auf den eigentlich Inhalt der Nachricht
Notifications/ {id}	Notification.xsd	-Notification-ID -Referenz aufEmpfänger-Profil -Name des Empfänger-Profils -ID des Empfänger-Profils -Versanddatum der Notification -Art der Notification (Warning oder Benachrichtigung) -Überschrift der Nachricht -textueller Inhalt der Nachricht

Grundsätzliche Strukturprinzipien
In Listen-Schemata sollen nur die wesentlichen identifizierenden Merkmale der einzelnen Elemente dargestellt und so weit wie möglich reduziert werden, um das Netzwerk keiner sonderlich großen Belastung auszusetzen. Der Kontrast zwischen XML-Dateien, denen eine grober Auflistung innewohnt und denen mit detailreichen Informationen, sollte möglichst zum Vorschein kommen. Ansonst hätte man sich die abstrahierende Listenressource direkt sparen und direkt eine Auflistung aller Instanzen mit allen ihren Details verschicken können. Daher enthalten die Schemata der Listenressourcen nur die nötigsten Informationen. Daten, die auch an einer anderen Stelle wiedergefunden werden können, werden weitestgehend durch Querverweise markiert.
Man könnte auch darauf achten, wichtigere Informationen möglichst weit oben in der Datei zu platzieren. Im Falle eines akuten Verbindungsabbruchs während der Übertragung einer XML-Datei, kann sichergestellt werden, dass bereits übermittelte Daten interpretiert werden können., ohne die gesamte Datei versendet zu haben.

Fridge.xsd
Die Fridges-Schemadatei ist die komplexeste und wichtigste, da ihr die Aufgabe zugeteilt wird, alle Daten kombiniert und in Relation zueinander darzustellen. Über sie erhält der Client nicht nur die Informationen, welche Produkte sich aktuell im Kühlschrank befinden und welche Profile dieser verwaltet, sondern auch die Referenzen auf jede einzelne deklarierte Instanz, und dient somit als Schnittstelle zu (fast) allen anderen Ressourcen. Darüber hinaus werden diese Daten noch zusätzlich mit Kühlschrank spezifischen Informationen über Anzahl aller im Kühlschrank existierenden Produkte eines Produkttypen und dem jeweiligen Status eines konkreten Produktes angereichert. Sie beinhaltet demnach einen sehr unterschiedlich gegliederten Aufbau. Im Normalgebrauch werden durch diese XSD-Datei die voraussichtlich die größten XML-Dateien geniert werden.
Daher ist wichtig das Design eben dieser Schemadatei besonders behutsam abzuwägen. Bei welchen Attributen ist es wirklich nützlich ihre Ausprägung aufzuführen anstatt das Datenvolumen mit einer kleinen Hyperlink einzudämmen? Bei allen Kühlschrank spezifischen Daten ist das Aufführen innerhalb der Kühlschrank-XML trivial.
Beispielsweise könnten konkrete Produkte eines Kühlschrankes auch in der Repräsentation der Kühlschrank-Ressource mit einem Inhaber versehen werden, obwohl dieser auch durch die Liste zugehöriger Produkte in seiner Profil-Ressource einsehbar ist. Hier muss zwischen Datenmenge pro Übertragungsvorgang und Nutzen der zusätzlichen Information abgewägt werden.
Es wurde sich für die minimalistischte und referenzenreichste Anordnung Enschieden, um das Datenvolumen einer zu übermittelnden Fridge-Datei möglichst gering zu halten und das Netzwerk nicht zu strapazieren.

Profile.xsd
Neben Fridge.xsd diese Datei das zweit komplexeste Schema des "Fridgemanagers". Sie enthält ebenfalls viele unterschiedliche Strukturen, und kombiniert auch zwischen mehreren Ressourcen. Und wird dementsprechend auch mit Vorsicht gestaltet. An dieser Stelle wurde sich infolge zuvor genannten

Gründen auch für die minimalistische Struktur entschieden.

Referenz auf Notification
Bisher wurde noch nicht festgelegt, ob sich der Einstiegpunkt zu einer Liste von Notifications im Bereich eines konkreten Fridges oder eines Profils befindet, da sowohl Nachrichten existieren, die sich nur auf ein Profil beziehen, und welche, die an die Gesamtheit aller Benutzer eines Kühlschrankes gesendet werden sollen. Die Erstellung eines repräsentativen Profils für alle Nutzer desselben Kühlschrankes kann die Entscheidung abnehmen. Die Liste aller Notifications kann so an die Profil-XSD gebunden werden. Es msste nur noch realisiert werden, dass Nachrichten mit dem Fridge-Profil als Empfänger an alle Nutzer des Kühlschranks gesendet werden.
Eine andere Möglichkeit bestünde darin, das Datenvolumen der Fridge- und Profile-Dateien nicht noch weiter aufzublähen und die Verwaltung der Notifications, allein durch den dort angegebenen Empfänger zu überwachen. Bei der Abfrage aller Nachrichten eines bestimmten Profils müsste die Liste aller Notifications zunächst einmal nach der gewünschten Nutzer-ID mühlehlig selektiert werden.

Schemadateien der Datenbank
Das waren bisher nur die Schemadateien zu den XML-Nachrichten, die nachher zwischen Client und Server kommuniziert werden sollen.
Zusätzlich wurden noch Schemadateien für die Datenspeicherung in Servernähe in Form einer XML-Datenbank erstellt:

Beizeichnung der lokal gespeicherten XML-Datei	Zugehörige XSD-Bezeichung	Ihre zugehörige Datenstruktur
FridgeLOCAL.xml	FridgesLOCALxsd	Liste aller Kühlschränke: -mit Name -Fridge-ID -verschachtelte Liste der von ihm verwöteten Profile mit: -Schlüssel zu Profil -Profil-ID - verschachtelte Liste der von ihm verwalteten Produkttypen mit: - Schlüssel - Bestandsinformationen -erneut verschachtelte Liste mit: -Schlüsseln zu den einzelnen Produktinstanzen
ProfileLOCAL.xml	ProfilesLOCAL.xsd	Liste aller verwalteten Profile: -Schlüssel zu Profil -Name, Geburtstag, Geschlecht, Größe, Gewicht -Liste aller zugehörigen Kühlschränke mit jeweils: Schlüssel zu Kühlschrank -Liste aller zugehörigen Produkte mit jeweils: Schlüssel zu Produkten (-Einkaufsliste aller jemals erworbenen Produkten) (-Informationen des Fitnesstrainers)

ProducttypeLOCAL.xml	ProducttypesLOCAL.xsd	Liste aller verwalteten Produkttypenmit jeweils: -Schlüssel zu Produkttyp -Produkttypen-Name -Produktbeschreibung -Inhaltsstoffe -Nährwerte -Barcode -Preis
ProductLOCAL.xml	ProductsLOCAL.xsd	Liste aller Produkte mit jeweils: -Produkt-ID - Schlüssel zu Produkttyp, Kühlschrank & Besitzerprofil -Status -Einlesedatum - Datum des Verzehrs -Mindesthaltbarkeitsdatum -Preis
NotificationLOCAL.xml	NotificationLOCAL.xsd	Liste aller jemals gesendeten Benachrichtigungen mit folgenden Daten je Eintrag: -Notification-ID -Schlüssel zum Empfänger-Profil - Art der Benachrichtigung -Versendedatum -Überschrift der Nachicht -eigentlicher Inhalt der Nachricht

Diese Unterscheidung ist notwendig, da es sich bei den, zwischen Client und Server versendeten, Nachrichen nur um transiente Daten handelt. Sie sind nur während des Übertragungsvorganges vorhanden. Die Daten, die nach den Vorgaben einer Locals-XSD generiert werden, unterliegen einer permanenten Speicherung in einer XML-Daenbank. Außerdem kann sich die Speicherstruktur der Daten der Datenbank von denen, die kommuniziert werden unterscheiden.
Zu jedem Ressourcenpaar (Listenressource und ihre Subressource), aus der Arbeit des vorangegangenen Meilensteins, besteht eine XML-Datei für die persitente Datenspeicherung. Damit ist die lokale Datenspeicherung zwar mit der URI vergleichbar, aber nicht identisch. Um Redundanzen zu vermeiden, wurde die Existenzunabhängigkeit der Ressourcen untereinander beibehalten. Beispielsweise bleiben Profile unabhägig von dem Kühlschrank, dem sie zu geordnet sind und Produkttypen können Kühlschrank übergreifend gespeichert sein. Hierbei wird sehr viel Wert auf eine Speicherung mit möglichst wenig Redundanz gelegt, um den begrenzten Speicherplatz effektiv zu nutzen. Diese Zielsetzung impliziert die wohldurchdachte Benutzung von Datenbank Schlüsseln.

Gewählte Granularität
Bei der Wahl der Granularität, Umfang einer Daten im Verhältnis zur Anzahl aller Schemadateien, wird, sowohl bei den Locals-Dateien als auch bei den kommunizierten, versucht einen Mittelweg zwischen den beiden Extremen viele kleine Dateien oder einer großen zu finden. Bei einer zu groben Granularität müsste eine riesige XML-Datein, die alle gespeicherten Daten vereint, bei jeder noch so "kleinen" Operation geladen und in ihrer vollen Größe die Netzwerkdarstellung gebracht werden. Dieser hochgradig unperformante Vorgang würde zu dem auch noch sehr häufig wiederholt werden. Dies würde nicht nur den Arbeitsspeicher belasten sondern auch das Netzwerk.
Mit einer feineren Granularität würde der Aufruf von "kleinen" Funktionen keine negativen Asuwirkungen haben. Jedoch müssten bei der Anfrage einer kompletten Liste, z.B. eine Liste aller Produkte im Kühlschrank, die Datei zu jedem einzelnen Produkt separat durchgegangen und immer wieder aufs neue in die Netzwerkdarstellung gebracht werden. Beide Varianten sind hochgradig unperformant.

FridgeLOCAL.xsd
Ebenso wie die Fridge.xsd besteht auch die FridgeLOCAL.xsd aus einer komplexesten Datenstrukturen des Systems. Außerdem werden hier die nötigen Informationen beider XSDs "Fridges" und "Fridge" zu einer zusammengefügt, was die Feinheit der Strukturierung und die Menge an Daten noch weiter erhöht. Aufgrunddessen wird versucht hier nur Informationen zu speichern, die absolut kühlschrank spezifisch sind, die Profile haben beispielsweise alle eine andere Ausprägung genau wie die Bestandsdaten zu den aktuell verwalteten Produkten. Diese Informationen sind absolut Kühlschrank spezifisch. Wobei erneut die Redundanzen für Profile und Produkte durch Schlüssel eliminiert werden, die auf eine andere Stelle im Speicher verweisen.

Schlüssel und Namen oder nur Schlüssel speichern?
Für eine erleichterte Verständlichkeit und Übersichtlichkeit könnten unteranderem zu jedem Produkt nicht nur die Producttype-ID angegeben werden, sondern darüber hinaus der Produktname. Somit ist die Datenbank schneller einsichtig und kann auch besser vom Administrator gewartet werden. Ansonsten wäre dieser gezwungen zu jeder ID das passende Objekt in einer anderen XML-Datei zu suchen, um den Produkttypen des eingesehenen Produktes zu identifizieren, da die Products-Daten quasi nur aus Zahlen bestehen. Jedoch würde dies den Nachteil einer etwas größeren Redundanz mit sich ziehen. Der Aspekt der Redundanzenvermeidung wiegt mehr als die erleichterte Einsicht der gespeicherten Daten, da sie als primäres Ziel des Datenbankdesigns festgelegt wurde und der Administrator eine erleichterte Einsicht der Datenbank auch durch andere Mechanismen sicher stellen kann.
Dieses Prinzip von Schlüsseln, zu an einer anderen Position genauer spezifizierten Objekten, gilt auch für andere XML-Dateien der Datenbank.

Datentyp der Datenbank
Bevor sich für die Speicherung der Daten in Form von XML-Dateien entschieden wurde, gab es viele gescheiterte Versuche eine Datenbank aus unteranderem TXT-Dateien oder Java-Objekten zu erstellen. Eine XML-Datenbank bietet im Kontext einer verteilten Anwendung, die zu dem auch nur ausschliesslich mit XML-Dateien komminiziert, eine vielzahl an Vorteilen. Daten der XML-Datenbank liegen in einem Format vor, wie die zu kommunizierenden Daten. Sie müssen demnach nicht erst mühsehlig in einen Dateitypen compiliert werden, der vom Server bearbeitet werden kann.Im Rahmen dieses Projektes wurde auch die Benutzung des JAXB-Frameworks, welches speziell die Übersetzung zwischen XML, der Netzwerkdarstellung und Java-Objekten ermöglicht. Durch die zugehörigen XSD-Dateien sind die XML-Dateien der Datenbank validierbar.
Eine Datenbank mit Jave-Objekten wäre ebenfalls denkbar, da JAXB zwischen XML und Java-Objekten übersetzt.
Es wird sich jedoch für eine XML-Datenbank entschieden, da sie die selbe Struktur wie die zu verschickenden Daten hat und daher leichter eingesehen und testeshalber manipuiert werden kann. Außerdem sind XML-Dateien validierbar ohne einen spezifischen Parser selbst schreiben zu müssen.

Andere Dateitypen zur Kommunikation als XML
In dieser Ausarbeitung werden kommunizierte Nachrichten zwischen den Komponenten des veteilten Systems in der Ausprägung von XML-Dateien gewählt. Neben dem XML-Format existieren aber noch unzählige andere wie z.B. JSON, CSV oder Binär-Formate, die ebenfalls für die Kommunikation in Betracht gezogen werden können. Es wurde sich für XML entschieden, da die Erweiterbarkeit durch Namesräume erleichtert und sie eher von anderen Menschen eingesehen und vorstanden werden können als z.B. die binären formate. Der einzige Nachteil ist eine etwas größere Datenmenge, hervorgerufen durch die Tag-Notation, die übermittelt werden muss, aber dieser wird durch den wichtigsten Vorteil wieder entkräftet: die Validierbarkeit durch Schemadateien.

5.4 Meilenstein 3: RESTful Webservice

Aufgabe des Meilensteins
In den vorangegangenen Meilensteinen wurde methodisch darauf hingestrebt einen Webservice zu designen, der die, in der Projektbeschreibung definierten, Anforderungen an das System erfüllt. In diesem Meilenstein soll der synchrone Anteil der Kommunikation innerhalb der verteilten Systems endlich vollständig ermöglicht werden.
Für die Realisierung der synchronen Kommunikation ist nur noch ein letzter Schritt nötig, der eine serverseitige Verarbeitung der erstellten XSD- und XML-Dateien der letzen Meilensteine umsetzt.
Dieser RESTful Webservice sollte in Java umgesetzt werden, mindestens zwei Ressourcen implementiert haben, deren Repräsentationen durch JAXB gemarshallt und umarshallt werden sollten. Zusätzlich sollten die HTTP-Operationen GET, PUT und DELETE implementiert sein und die Benutzung mindestens eines Queryparams und eines Pathparams sichergestellt werden.
Der RESTful Webservice sollte natürlich den Grundprinzipien der REST-Architektur genügen und die geforderte Capability in dem gewünschten Maße bereitstellen.

5.4.2 Java-Klassen des Werbservices für den "Fridgemanager"

Für den eigentlich Webservice können nun eigene, nicht gegenrische Java-Klassen erstellt werden, die die HTTP-Operatoren implementieren und somit die Capability bereitstellen. Da in den vorangegangenen Meilensteinen genug konzeptionelle Vorarbeit geleistet und immer genau eine Variante festgelegt wurde, sind an dieser Stelle vereinzelt nur noch kleine Alternativen bezüglich des Quellcodes möglich.
Folgende Klassen werden erstellt:

Bezeichnung der Java-Klasse	Funktionalität
FridgeRessource.java	Bietet REST-Schnittstelle für die Ressourcen Fridges & Fridges/{id} an
ProfileRessource.java	Bietet REST-Schnittstelle für für die Ressourcen Profiles & Profiles/{id} an
ProducttypeRessource.java	Bietet REST-Schnittstelle für für die Ressourcen Producttypes & Producttypes/{id} an
ProductRessource.java	Bietet REST-Schnittstelle für für die Ressourcen Products & Products/{id} an
NotificationRessource.java	Bietet REST-Schnittstelle für für die Ressourcen Notifications & Notifications/{id} an
MyMarshaller.java	Auslagerung des (Un-/)Marshallingverfahrens

Zusammenfassung zu einer Klasse für zwei Ressourcen
Die Zusammenfassung aller Subressourcen einer einzigen Primärressource zu einer Java-Klasse scheint eine Konvention für REST-Standards darzustellen, jedenfalls resultieren aus dieser Implementierung keine erkennbaren Nachteile. Auf die Subressourcen kann mittels Pathparams zugegriffen werden.

MyMarshaller
Die Umformung des Marshallers zwischen XML- und Java-Dateien ist ein vollkommen generischer Ablauf, der auch immer im gleichen Maße ausgeführt wird. Um Redundanz im Quellcode zu vermeiden, wurde eine zusätzliche Klasse geschrieben, die sich nur auf die Implementierung eines, für diesen Anwendungsfall "Fridgemanager", universellen Marshallers beschränkt und auf den in jeder anderen Webservice-Klasse bei jedem Marshall- und Unmarshall-Vorgang zugegriffen wird.

5.4.3 Implementierung der HTTP-Operationen mittels "Jersey"

Die Implementierung der HTTP-Operationen kann durch die vorgeschlagene API namens "Jersey" erheblich erleichtert werden. Durch Annotationen im Java-Quellcode konnte für eine Java-Methode

markiert werden, ob und um welcheHTTP-Operation es sich bei ihr jandelt handelt. Diese Annotationen sind von Jersey interpretierbar.

Pro HTTP-Operation muss sich immer im Klaren gemacht werden, auf welche lokalen Entitäten der Datenbank zurück gegriffen werden müssen, da sich sich dabei oft um mehr als nur eine XML-Datei handelt.

Da für jeden POST-Befehl bei jeder Listenressource immer wieder aufs neue eine Such nach einer freuen ID nach dem selben Algorithmus hervorruft, kann dieser Ablauf ebenfalls wie der Marshall-Vorgang ausgelagert werden. Die Einigung daruf die einmal angelegten Daten zu einer Subressource erstmal nicht löschbar zu machen, sondern für die History und Marktforschung persistent zu speichern, hätte zur Folge, dass die ID für eine neu zu erstellende Ressource sich immer nur um eine Einheit gegenüber der zuletzt geposteten unterscheidet. Der Such-Algorithmus müsste augrunddessen gar nicht implementiert werden, sondern nur eine Klassenvariable, die die nächste freie ID preisgibt.

5.4.4 Implemetierung von Queryparams und Pathparams

Pathparams
Da nur Primärressourcen als Java-Klassen implementiert wurden, kann mittels Pathparams auf die Subressourcen zugegriffen werden z.B. um in der FridgesRessources-Klasse nicht die zentrale Ressource "Fridges" anzusprechen, sondern die von ihr abhängige "Fridges/{id}"-Ressource. Dabei muss der ID auch eine konkreter Wert übergeben werden. Dieses Prinzip gilt für alle anderen Beziehungen zwischen Primär- und Subressource.

Queryparams
Durch die Queryparams kann einer der wichtigsten Kernfunktionen des Kühlschranks implementiert werden. Gemeint ist das willentliche Filtern von Produkten nach Kriterien, die der Nutzer sich selbst bestimmen kann. Diese query sollen auch beliebig kombiniert werden können. Beispielsweise kann ein Benutzer die Gesamtheit aller Produkte seiner Kühlschränke nach Kriterien wie maximale Mange an enthaltenen Kalorien darf nicht höher als 300kcal betragen, der verwaltende Kühlschrank soll den Namen "Super Fridge" tragen und die Anzahl der Produkte eines solchen Typens muss mindestens gleich zwei Exemplare aufweisen, durchsuchen. Diese Werte müssen jedoch durch die in der Datenbank enthaltenen Informationen berechnet werden können.

Filterung durch kombinierte Queryparams
Hierbei geht es um die Implementierungsvarianten der Filterung durch mehrere verschachtelte Queryparams.
Variante 1: Filterung bezüglich des aktuellsten Queryparam
Bei der ersten Möglichkeit wird sich zunächst nur ein Queryparam angeschaut und alle Daten dieser ressource auf der Datenbank nach Eintragen gesucht, die der Ausprägung des aktuell betrachteten Queraparams genügen. Aus der Anzahl aller passenden Einträge wird eine XML-Datei erstellt. Diese XML-Datei gilt dann als Eingangsmenge für die erneute Filterung bezüglich des zweiten Queryparams. So verhält es sich analog für alle noch folgenden Filterungen. Die komplett gefilterte XML-Datei wird dem Client übermittelt.
Nachteil: Einträge, die den Anforderungen des ersten Queryparams gengen, aber nicht zwangsläufig auch den anderen, werden trotzdem erstmal zwischengespeichert. Diese Zwischenspeicherung kann auch umgangen werden.

Variante 2: Filterung bezüglich aller Queryparams gleichzeitig
Ein Eintrag der Liste wir erst bezüglich des ersten Queryparams überprüft, wenn das Queryparam zutrifft wird der Eintrag auch bezüglich des zweiten Queraparams überprüft usw. Sobald ein Queryparam für einen Listen Eintrag nicht passt, wird dieser Eintrag nicht weiter überprüft, sondern zum nächsten Eintrag

übergegangen und wieder auf alle Queryparams überprüft.
Dieser Vorgang ist wesentlich effizienter, da keine unnötigen XML-Daten extra herausgefiltert und zwischengespeichert werden müssen.

5.5 Meilenstein 4: Konzeption + XMPP Server einrichten und Meilenstein 5: XMPP – Client

Die vorangegangenen Meilensteine galten weitestgehen nur dem Ziel, die synchrone Kommunikation zu konzeptionieren und zu implementieren. Bei diesem Meilenstein wird erstmals etwas genauer auf den asynchronen Part der Kommunikation eingegangen.

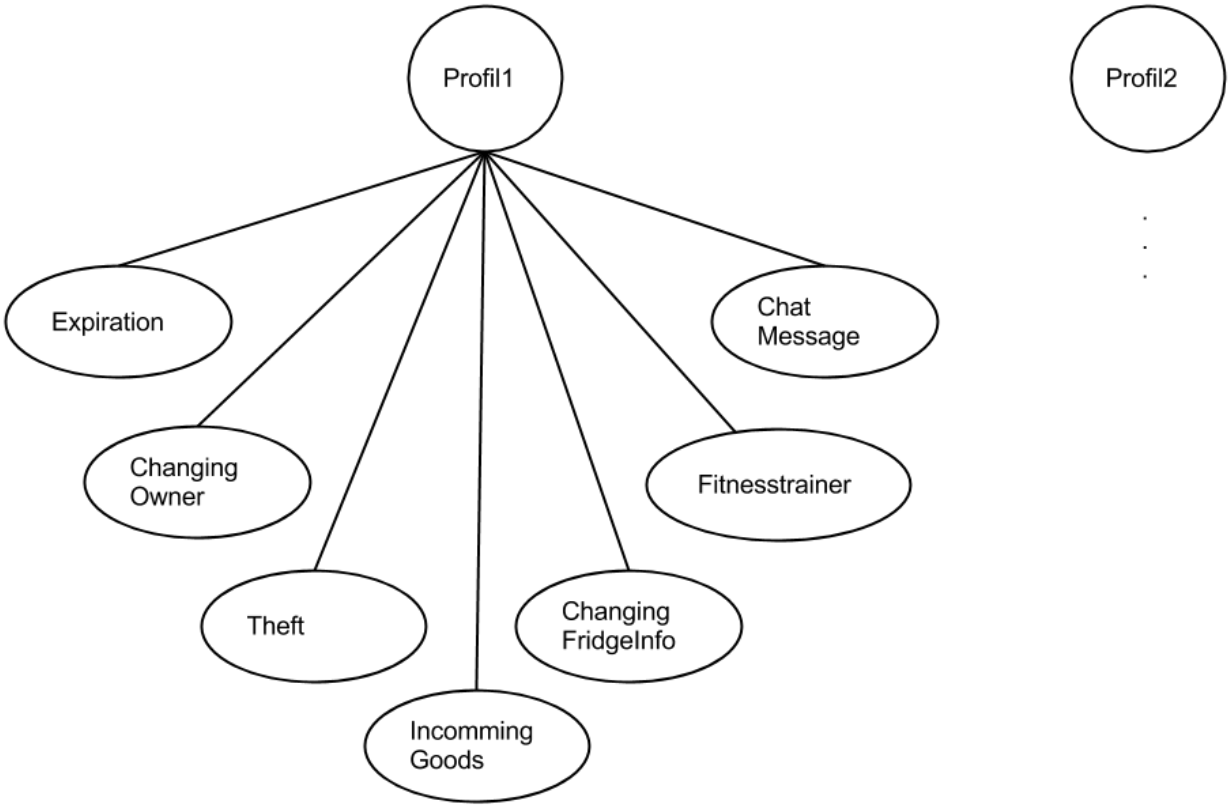
Die asynchrone Kommunikation des "Fridgemanagers" beschränkt sich wie im konzeptionellen Meilenstein beschrieben nur auf die Versendung von Informationen zwischen zwei von Menschenhand gesteurten Clients. Diese Informationen äußern sich nur in der Form von Notifications.
Um das Public.-Subscribe-Paradigma anwenden zu können, und somit auch die asynchrone Kommunikation möglich zu machen, müssen Topics für Notifications verschiedenen Wesens definiert werden. Der Server mit der Datenbank stellt dabei die publizierende Komponente dar und der Client den abonnierenden. Jedes dieser Topics kann von jedem Nutzer individuell und separat abonniert werden.

Topics der Notifcations:

Topic-Bezeichnung	Beschreibung
Expiration	Benachrichtigung über **Überschreitung des Mindesthaltbarkeitsdatums** eines Produktes (mit dem Status "inside") an seinen Besitzer
ChangingOwner	Anfrage, ob **Besitzer eines Produktes geändert** werdes soll, Benachrichtigung über Änderung des Besitzers eines Produktes
Theft	Warnung über **unautorisiertes Entnehmen eines Produktes** (sowohl an den "Dieb" als auch an den rechtmäßigen Besitzer)
IncommingGoods	Benachrichtigung über **eintreffende Waren** (Entweder Ankommen der Lieferung des Online-Shops oder anderer Nutzer lädt grade seinen Einkauf in den Kühlschrank)
ChangingFridgeInfo	Benachrichtigung über **Änderungen bezüglich** aller Informationen aller zugehörigen **Kühlschränke eines Profils** (z.B. neues Profil wurde an einem Kühlschrank erstellt, ein Profil eines Kühlschranks wurde gelöscht, Kühlschrank-Name wurde geändert, Mindesbestandsdaten geändert etc.)
Fitnesstrainer	Warnung bei Entnahme eines Produktes, dessen Verzehr Höhe des zulässigen Tagesbedarfs, für Erreichung der Idealmaße, überschreiten würde
ChatMessage	Chat-Nachricht eines Sender-Profils und (eines oder mehrerer) Empfänger-Profile/s

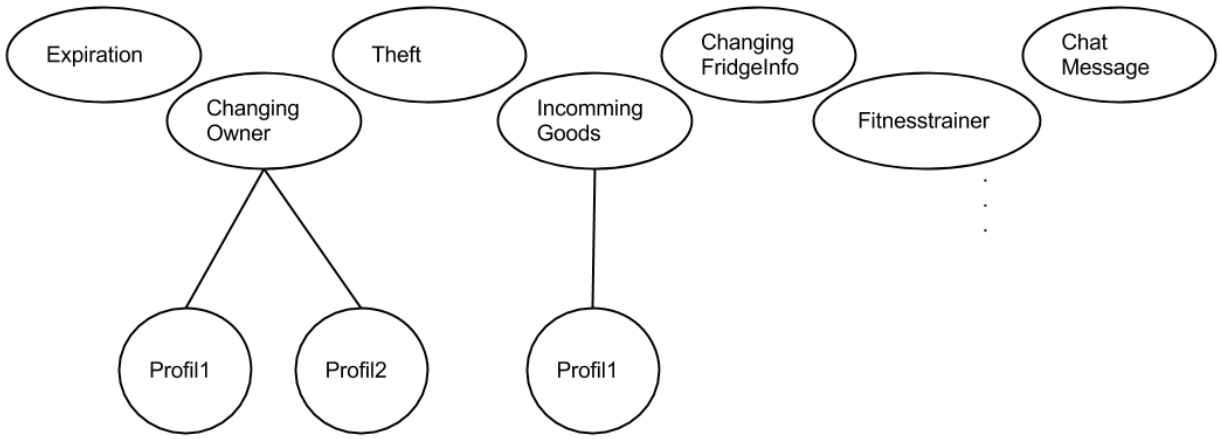
Hierarchie dieser Nodes
Der Nutzer möchte eventuell nicht zu allen diesen Topics regelmäßig Nachrichten empfangen, oder nur eine kleine teilmenge der Nachrichten bezüglich eines Topics. Um dem Nutzer eine Möglichkeit zu bieten gezielt festlegen zu können, welche Nachrichten ihn nur interessieren, sollte die Hierarchie der abonnierbaren Nodes gut durchdacht werden. Generell können alle Leafs zwischen wichtigen Warnungen und eher unwichtigen Benachrichtigungen unterschieden werden. Dementsprechend könnten die Knoten auch unterzeilt werden. Dies erscheint aber nur zweckmäßig, wenn die Nutzer es auch für gutheißen, nur Warnunmeldungen oder nur Benachrichtigungen zu erhalten.

Variante 1: Profiles-Nodes mit jeweils allen Topic-Leafs



Jedes Profil erhält einen Knoten, der defaultmäßig abonniert ist. Von jedem Profil-Knoten existiert für jedes Topic-Leaf ein untergeordnetes Blatt. Durch das Abonnieren des Profil-Knotens werden auch automatisch alle Leaf-Topics abonniert. Diese können nachträglich noch deabonniert werden. Diese Variante lässt die Menge an zuerstellenden Leafs proportional zur Profil-Anzahl anwachsen, was bei vielen Profilen einen hohen Speicher und Verwaltungsaufwand bedeutet.
Diese Variante könnte so abgeändert werden, dass nur zu jedem Kühlschrank ein primärer Knoten existiert. Diese Variante würde die Anzahl der abonnierbaren Knoten zwar senken und somit auch den Speicherbedarf und den Verwaltungsaufwand, würde den Nutzer aber einschränken, da dieser wieder an den Kühlschrank gebunden ist.

Variante 2: Profile-Nodes pro Topic-Node



Umgekehrt können auch die sieben Leaf-Nodes an höchster hierarchischer Position stehen und durch Blätter von allen Profilen ergänzt werden. Hierbei geht der Vorteil, dass der Nutzer default mäßig nur den Primärknoten seines Profils abonnieren kann.
Es müsste zudem noch sichergestellt werden, dass jeder Nutzer nur seine eigenen Profil-Nodes abonnieren kann.
Variante1 weist schon viel eher die Struktur aus Sicht des Nutzers auf. Variante 2 eher die hier weniger wichtige funktionale Sicht.

"**IncommingGoods"-Leaf**
Dieser Blatt kann sich sowohl auf die Anlieferung der Waren des Online-Shops als auch auf das Einlesen von neuen Lebensmitteln eines Mitbewohners beziehen. Beide Anwendungsfälle haben ihre Daseinsberechtigung. Die Benachrichtigung über die Ankuft der gelieferten Ware ist in sofern wichtig, als dass die Nutzer des Kühlschranks direkt darüber aufgeklärt werden und die Ware schnellst möglich in den Kühlschrank einlagern können. Daher sollte diese Nachricht an alle Profile eines Kühlschranks gesendet werden.
Die Benachrichtigung über den anderen Vorfall, ein Mitbewohner kehrt von seinem Einkauf wieder und befüllt den Kühlschrank mit den neu erworbenen Waren, ist wichtig, falls sich ein anderer Mitbewohner zu dieser Zeit auch nicht Zuhause befinder und auch den Drang nach einem Einkauf verspührt. Durch die Benachrichtigung erfährt dieser, dass ein erneuter Einkauf nicht nötig ist.
Evetuell sollten diese beiden Fälle auch separaten Leafs zugeordnet werden, da der Nutzer unter Umständen auch nur Benachrichtigungen zu einem erhalten möchte.

Chat-Message
Das Chat-Message-Leaf ist das einzige Leaf, bei dem es nicht angemessen erscheint, es nur vom eigenen Profil aus abonnieren zu lassen. Denkbar wäre es ein Leaf von allen Abonnieren zu lassen, die an einer Chat-Konversation teilnehmen möchten. Demnach würde es für ein Profil mehrere Chat-Leafs geben müssen.
Oder jedes Profil kann nur das zentrale Kühlschrankprofil abonnieren. So wäre es möglich alle Nutzer eines Kühlschrankes miteinander kommunizieren zu lassen. Dies würde weniger Datenlast bedeuten als die vorangegangene Variante, aber auch eine Einschränkung für die Nutzer.
Da die Ausarbeitung des Chatprogramm keine hohe Priorität erhielt, wird an dieser Stelle noch keine Abwägung der Möglichkeiten unternommen. Dafür müsste sich genauer mit der Zielgruppe und ihren Wünschen und Vorstellungen eines "Fridgemanagers" auseinander gesetzt werden.

Wahl für Fat Ping
Zwischen Light und Fat Ping herrscht eigentlich ein fließender Übergang. In diesem Fall wurde sich für den Fat Ping entschieden, da nur sehr kleine Nachrichten zwischen gespeichert und verschickt werden müssen und es daher keinen sonderlich großen Unterschied macht, ob es zehn oder zwanzig Zeichen mehr oder weniger sind.. In der Praxis wird oft mit Simple Payload gearbeitet, da der Inhalt der zuversendenden Nachricht tatsächlich schon ziemlich groß werden kann. Diese Nachricht müsste solane zwischen gespeichert werden, bis der Empfänger wieder aktiv ist. Bei vielen Nachrichten und vielen inaktiven Empfängern, kann schon eine ganze Last aufkommen. Daher wird teilweise nur eine Referenz in der asynchronen Nachricht auf den eigentlichen Inhalt angegeben z.B. in Form eines Links, oder nur die nötigsten Informationen.
Für die Chat-Nachrichten könnte daher auch der Light Ping in Erwägung gezogen werden, da diese Nachrichten eine höhehre Länge als die anderen Messages erreichen können. Sinnvoll ist auch euch Zeichenbegrenzung pro Chat-Nachricht.

5.6 Meilenstein 6: Client – Entwicklung

Das Ziel dieses Meilensteins ist es alle bisher ausgearbeiteten Ideen und Funktionalitäten im Bereich der synchronen und asynchronen Kommunikation in einer GUI zu vereinen, die nicht nur den Zweck der Testbarkeit des Quellcodes erfüllt, sondern darüberhinaus auch schon einen Prototypen der letztenlichen Client-Oberfläche darstellt.
Dafür werden zuerst die Grundfunktionen lokalisiert, denn diese sind der primäre Zweck des Systems und aufgrund derer wird das System auch genutzt. Daher sollte sich auch das Layout nach diesen Funktionen richtigen und sie dem Benutzer am leichtesten zugänglich machen. Die Kernfunktion besteht darin den oder die Kühlschränke bzw. die Aufbewahrungsorte und ihre beinhalteten Lebensmittel zu verwalten.
Das heißt die wichtigste Funktion umfasst zunächst einmal die **Darstellung aller Produkte**. Zusätzlichen sollen diese Produkte noch noch Kriterien gefiltert werden können, die der Nutzer selbst bestimmen kann, z.B. nach Kühlschränken sortieren, oder nach Inhabern, etc., und die Informationen bezüglich diese Produkte einsehen. Außerdem soll der Nutzer sein **Profil** einsehen und bearbeiten können. Die letze separate Funktion stellt die Einsicht der **Notifications** dar.
Diese drei unterscheidlichen Funktionen werden optisch, durch verschiedene Reiter, voneinander getrennt.
Das Layout ist ein noch sehr variables Konzept.

6. Kritische Reflexion

In diesem Abschnitt möchten wir noch einmal Abstand von der bisherrigen Arbeit gewinnen und sie rückblickend kritisch begutachten. Es sind viele Ideen an Funktionen und Gedanken zu Problemen und ihren Lösungen aufgekommen. Da uns immer gesagt wurde, dass unser System noch zu wenig Funktionen beinhaltet, haben wir quasi versucht das System mit mehr Funktionalität aufzupumpen. Jedoch konnte weder jede Funktionalität, aufgrund der mangelnden technischen und zeitlichen Ressourcen, implementiert werden, noch jede erdachte Variante bei Konzepierung und Implementierung hier aufgezählt und angemessen erörtert werden.
Außerdem wurde in dieser Dokumentation nicht viel Wert auf die Beschreibung des jetzigen Zustands des Systems gelegt, da dies nur redundante Information, zum im Github einsehbaren Quellcode, darstellen würde. Die Dokumentation soll schliesslich kein reines Abbild des Quellcodes sein, sondern andere Sichtweise auf die Ausarbeitung der Aufgabe einnehmen und ergänzende Informationen liefern, um den Quellcode nachvollziehen zu können. Es wurde versucht mögliche Varianten aufzuzählen, diese gegeneinander zu erörtern und zusätzlich mögliche Ideen,die aktuell nicht im Quellcode enthalten sind, die aber auch mit etwas mehr Zeit hätten implementiert werden können, aufzuzeigen. Es wurde demnach versucht sowohl in die Horizontale zu gehen als auch die Vertikale zu beleuchten. Anand des vorliegen Quellcodes kann, unserer Meinung nach, eingesehen werden, dass das Prinzip der verteilten Systeme generell verstanden wurde, und auch auf die noch nicht Implementierten Funktionen adaptiert und angewendet werden kann.
Auch auf Erklärung von trivialen Informationen zu Fachbegriffen und den Konzepten dahinter wurde verzichtet, da diese totes Wissen darstellen, das bei der Ausarbeitung dieser Aufgabe vorausgesetzt wird.

7. Leistungsmatrix

	Simon Klinge	Julia Warkentin
Konzeption	50,00%	50,00%
Implementierung	60,00%	40,00%
Dokumentation	40,00%	60,00%