Technology
Arts Sciences
TH Köln

Entwicklungsprojekt Interaktive Systeme
Wintersemester 2018/19

Implementationsdokumentation/ Installationsdokumentation

Dozenten:
Prof. Dr. Gerhard Hartmann
Prof. Dr. Kristian Fischer

Mentoren:
Ngoc-Anh Gabriel
Lena Wirtz

von
Kristian Czepluch - (11112444)
Denise Kübler - (11119149)

Vorwort:

Bei der Implementierung des Projektes war uns bereits von Beginn bewusst, dass es nicht möglich und unrealistisch ist ein fertiges Gesamtprodukt zu erstellen. Unser Ziel war es einen vertikalen Prototypen zu entwickeln, welcher besonders die Alleinstellungsmerkmale und die Anwendungslogik im Fokus hat. Somit wurde sich größtenteils auf die Verwaltung der frischen Lebensmittel, die Berechnung der Haltbarkeitsdaten und optimalen Lagerbedingungen, sowie das Finden von geeigneten Angeboten in der Nähe, durch Berechnung des Abholwerts konzentriert.

Client:

Die Implementierung des Clients hat den größten Aufwand mit sich gebracht, da in der Entwicklung mit Android noch kaum Erfahrung bestand. Besonders die komplexen User Interface Elemente, sowie die Nutzung von Sensoren und Netzwerkanfragen stellten sich beim Client besonders schwierig dar. Im Folgenden wird eine grobe Übersicht der erstellen Klassen gegeben:

BaseApplication.java:

Diese Klasse stellt in dem Sinne unsere Grundlegende Applikation dar, denn Sie wird aufgerufen, sobald die Anwendung gestartet wird. In dieser Klasse wird einmalig ein JobScheduler aufgerufen, sowie die Notification Channels erstellt, welche für die Darstellung von Benachrichtigungen auf Android Geräte mit einem API Level von 26 und höher benötigt werden. Diese werden ebenfalls nur dann erstellten, wenn der Client sich in diesem Versionsbereich befindet.

LoginActivity.java, RegistrierenActivity.java:

Diese Klassen realisieren das Login System, wie es der Name der Klassen bereits schließen lässt. An dieser Stelle wurde, wie geplant mit der Authentication von Firebase gearbeitet, um diese Funktionalität zu realisieren. Bei der Registrierung wird neben dem anlegen des Kontos auf der Firebase Konsole, ein Benutzerkonto auf dem Server erstellt, wobei an dieser Stelle direkt das von Firebase generierte Token für die asynchrone Kommunikation mitgesendet wird. Dem Benutzer steht es frei ein Profilbild zu erstellen. Dies wird zunächst als Byte-stream umgewandelt und im Anschluss als Base64 String decodiert und in dieser Form übertragen und gespeichert. Ein Shared Memory Segment hält fest, ob sich der Benutzer bereits eingeloggt hat. Falls ja, so muss er diesen Vorgang nicht wiederholen, sondern wird direkt zur Anwendung weitergeleitet.

Hauptbildschirm.java:

Diese Activity stellt das Zentrum unseres Systems dar. In dieser Activity wurde das Lebensmittelinventar des Benutzers implementiert, welches durch Verwendung von einer durch CardViews zugeschnittenen ListView realisiert wurde. Beim starten der Activity

werden alle Lebensmittel, welche der Benutzer eingetragen hat aus der SQLite Datenbank geladen. Die Funktionen “berechneOptimal(…)“ und “berechneAnhandUserInput(…)“ stellen die Anwendungslogik in unserer Anwendung dar, wobei die Namen selbst beschreibend sind. Nachdem der QR-Code-Scanner die Informationen erfolgreich ausgelesen hat, werden diese in ***Lebensmittel.java*** Elemente zusammengefasst und stehen für die weitere Berechnung bereit. Bei der Berechnung wird zu Beginn das aktuelle Inventar ausgelesen und mit den neuen Lebensmitteln in einer Liste zusammengefasst. Diese werden dann zusammen mit ***LebensmittelInformation.java***, welche aus der Datenbank ausgelesen werden zu einem ***LebensmittelVerarbeitungsObjekt.java*** zusammengefasst. Dies hätte an dieser Stelle schöner gelöst werden können und hat den Ursprung, dass die Datenbank erst nach der Anwendungslogik hinzugefügt wurde. Es hätte auf das Zwischenobjekt verzichtet werden können. Das LebensmittelVerarbeitungsObjekt fasst alle Informationen zusammen, welche für die Berechnung und die anschließende Erstellung von LebensmittelEinträge notwendig ist. Die eigentlichen Berechnungen wurden in die Klasse ***Haltbarkeitsrechner.java*** ausgelagert, um dies übersichtlicher zu gestalten. Der generelle Ablauf der Berechnung wurde bereits in Meilenstein 1 und 2 ausführlich dargestellt. Grundsätzlich setzt sich die Berechnung aus einer Grundhaltbarkeit + (optional) Kühlungszuschlag – Distanzabzug zusammen. Hinzu kommt ein weiterer optionaler Abzug durch ethylenhaltige Lebensmittel in der Umgebung von 30%. Da somit 2^2 Möglichkeiten entstehen, wurden vier Funktionen definiert, welche diese abdecken: berechneOhneEthylenMitKühung(…), berechneMitEthylenMitKühlung(…),berechneMitEthylenOhneKühlung(…) und berechne OhneEthylenOhneKühlung(…). Diese werden in der berechneOptimal Methode verwendet, während die Methoden berechneImKühlschrank(…) und berechneAusserhalbKühlschrank(…) die Berechnungen anhand der manuellen Benutzereingabe übernehmen. Die Klasse ***Distanzrechner.java*** realisiert die Entfernungsberechnung für den Distanzabzug für die Lebensmittel. Grundlage hierzu sind Längen und Breitengrad angaben, welche Teil der Datenbank sind. Alle Lesevorgänge, welche im Zusammenhang mit der Berechnung von Haltbarkeitsdaten auftreten, wurden durch eine abstrakte Schicht ***Datenbank.java*** zusammengefasst, welche auf der ***LebensmittelDBHelper.java*** aufbaut. Ein weiterer wichtiger Bestandteil dieser Activity ist die ***EintragAdapter.java***, welche die Einträge für die ListView ermöglicht. Hierbei handelt es sich, um einen Custom Adapter, welcher die aktuelle Ansicht verwaltet. Die Kühlung kann durch Tippen auf die einzelnen Item Elemente verändert werden und wird automatisch neu berechnet. Durch „swipen“ oder durch das kleine Optionsmenü an den Listeneinträgen lässt sich zusätzlich mit den Elementen interagieren. Grundsätzliche funktioniert das Verwalten der frischen Lebensmittel reibungslos. Der einzige Fehler der vorliegt, und nach längerer Recherche ein bekannter Fehler seitens Android ist, ist, dass wenn man das letzte Element löschen möchte, dieses fälschlicherweise weiter angezeigt wird, obwohl es nicht mehr existiert. Durch berühren des Elements oder drehen des Bildschirms wird dieses Element erst entfernt. Laut Internet-Recherche sei dies jedoch ein bekannter Fehler in höheren API Versionen.

LebensmittelDBHelper.java, LebensmittelContract.java:
Diese Klassen stellen wie bereits erwähnt die grundlegende Verbindung zur Datenbank dar, wobei hier eine zusätzliche Klassen LebensmittelContract.java angelegt wurde, um eine Übersicht über Konstanten zu haben. Dies hat bei der Erstellung der Tabellen und dem

anlegen von Daten besonders geholfen, da hier kleinste Schreibfehler, zum sofortigen abstürzen der Anwendung geführt haben. Der LebensmittelDBHelper erstellt beim ersten aufrufen einmalig eine Datenbank, welche dann dauerhaft zur Verfügung steht. An dieser Stelle werden dann ebenfalls die notwendigen Informationen über die Herkunftsländer und die Lebensmittelinformationen angelegt. An dieser Stelle wird die Sprachbarriere, welche sich leider durch unser Projekt zieht deutlich. Die Angaben wurden an dieser Stelle in Englisch gemacht, da die aufbauende Anwendungslogik auf dem Server englischsprachige Dienste verwendet. Im weiteren Verlauf des Projekts und mit zusätzlichen Ressourcen müsste eine Übersetzungsschnittstelle mit in das System integriert werden.

LebensmittelJobService.java:
Diese Klasse stellt einen JobService bereit, welcher in einem Intervall von einem Tag das Lebensmittelinventar des Benutzers kontrolliert und ihn bei zukünftig ablaufenden Lebensmitteln über eine Notification benachrichtigt (2 Tage vorher). Dieser Service wurde dem AlarmManager vorgezogen, da dieser auch nach der Terminierung der Anwendung, als auch nach einem Reboot, weiterhin aktiv bleibt. Wie bereits erwähnt wird dieser einmal in der BaseApplication gestartet und durch ein SharedMemory Segment abgesichert.

AllergienDialog.java, ErnaehrungstypDialog.java, AllergienDialog.java, LebensmittelDialog1.java, MobilitätsDialog.java:
Diese Klassen wurden als Fragmente implementiert und Stellen die verschiedenen Dialoge der Anwendung dar. Diese werden für die Custom Dialogs verwendet, welche sich über unsere gesamte Anwendung erstrecken. Jeder dieser Dialoge besitzt ein entsprechendes Layout-File, welches die View definiert. Grundsätzlich sind dies lediglich Interface Elemente.

HTTPUrlConnectionService.java:
Diese Klasse ermöglicht die Kommunikation mit dem Client und wurde als einzelner Service implementiert, da viele Activities diese Funktionalität benötigen, und da Netzwerkanfragen als Hintergrundoperation ausgeführt werden müssen, um den Hauptthread nicht zu blockieren. Dieser Service beinhaltet einen Broadcast Receiver, welcher beim Auslösen eine Anfrage an die spezifizierte URL versendet und der anfragenden Activity über einen Intent antwortet. Bevor dieser Service antwortet überprüft er jedoch zunächst den Statuscode, welcher vom Server zurückgeliefert wurde. Je nach Statuscode erhält die aufrufende Activity eine andere Nachricht und hat so die Möglichkeit entsprechend zu reagieren. Auch einem Timeout oder bei einem fehlerhaften Verbindungsaufbau wird die anfragende Activity entsprechend in Kenntnis gesetzt, um so eine gewisse Stabilität zu gewährleisten. Grundsätzlich war eine Kommunikation über das HTTPS-Protokoll geplant gewesen, jedoch hat sich die Implementierung des Services erheblich umfangreicher als geplant herausgestellt, weshalb nach einem vergeblichen Proof of Concept und zusätzlichem Zeitdruck auf den Fallback zurückgegriffen werden musste und die Kommunikation über HTTP gewählt wurde. Positiv ist dennoch, das keine zusätzliche Abhängigkeit an dieser Stelle entstehen musste und die Kommunikation auch mit grundlegenden in Java fest integrierten Funktionen realisiert werden konnte.

AngeboteSuchen.java, AngeboteErstellen.java, DetailedAngebot.java, Angebot.java:
Diese Klassen stellen die verschiedenen Interaktionen mit Angeboten dar, wobei die auf dem Server vorgegebene Struktur eines Angebots durch die Klasse ***Angebot.java*** abgebildet wurde. Das Erstellen der Angebote findet in der AngeboteErstellen.java statt, wobei hier lediglich die Informationen aufgenommen und an den HTTPUrlConnectionService weitergeleitet werden. Beim versenden der Bilder wurde an dieser Stelle ebenfalls, wie beim erstellen des Benutzerprofilbild vorgegangen. In der Klasse ***AngeboteSuchen.java*** werden die Ergebnisse der Anwendungslogik des Server visualisiert. Innerhalb einer ListView sind die verschiedenen Angebote mit ihrem Abholwert aufgelistet, wobei durch verschiedene Dialoge die Parameter manipuliert werden können. Die Einträge innerhalb der Liste werden durch die ***AngebotEintragAdapter.java*** realisiert. Angefragte Bilder werden zwischen der Activity und dem Adapter durch das interne File-System kommuniziert. Die benötigten Standortinformationen werden an dieser Stelle durch den im Betriebssystem integrierten Standortservice ermittelt. Die Berechtigung diesen Dienst zu Nutzen findet während der Registrierung statt. Bei der Installation sollten diese jedoch sicherheitshalber manuell in den Einstellungen aktiviert werden. Die Klasse ***DetailedAngebot.java*** stellt alle Informationen zu einem Angebot dar, wobei zusätzlich an dieser Stelle die Google Maps API wie im Architekturdiagramm spezifiziert, eine Karte für den Benutzer dar.

MyFireBaseService.java:
Dieser Service stellt die Schnittstelle für die asynchrone Kommunikation mit Firebase dar. Das versenden von Nachrichten über den Server konnte, jedoch nicht bzw. kaum umgesetzt werden. Hauptfokus auf der Anwendungslogik und dem Alleinstellungsmerkmal war.

MeinAccount.java:
Diese Klasse stellt eine direkte Verbindung mit der Benutzer Ressource auf dem Server dar und bildet diese ab und lässt weiterhin zu, diese zu bearbeiten. Die Anfragen werden über den HTTPUrlConnectionService an den Server weitergeleitet, welcher die eigentlichen Aufgaben übernimmt.

MeineErnährungsprofil.java:
Auch diese Klasse stellt eine Direkt Verbindung mit dem Server dar, jedoch wird an dieser Stelle die Subressource „User/Einschraenkungen“ gesteuert. Der Benutzer hat die Möglichkeit hier festzulegen, welche Einschränkungen bei der Auswahl der geeigneten Angebote beachtet werden sollen. Die komplexen View-Elemente haben das synchronisieren besonders erschwert. Aus diesem Grund ist diese Klasse etwas umfangreicher.

Was lief beim Client nicht so gut?

Grundsätzlich hätten wir gerne mehr Ansichten implementieren können, jedoch hat hierfür der kurze Entwicklungszeitraum nicht ausgereicht, weshalb sich auf einen kleineren, jedoch ausschlaggebenden Bereich konzentriert wurde. Ebenfalls hätte HTTPUrlConnection Service weiter ausgebaut werden können, um die Verwendung des HTTPS Protokolls. Ebenfalls sind viele Code Redudanzen im Code vorzufinden, welche wir gerne in einem Optimierungsdurchlauf beseitigt hätten. Auch können wir nicht abschätzen wie sich das User Interface auf anderen Bildschirmgrößen darstellt, da lediglich ein Testgerät für die

Entwicklung zur Verfügung stand. Teilweise sind beim Server noch mehrere Funktionierende Routen vorhanden, welche jedoch aus zeitlichen Gründen nicht umgesetzt werden können wie z. B. das Abrufen der eigenen Angebote und Reservierungen, sowie die Bearbeitung dieser. Oder das Versenden und abrufen von Nachrichten. Grundsätzlich wurde die Schnittstelle zu Firebase implementiert, konnte jedoch nicht komplett ausmodelliert werden.

Server:

Bei der Implementierung des Servers konnte weitgehend all das, was geplant wurde auch umgesetzt werden. Die größte Herausforderung war hierbei zu Beginn die Implementierung der MongoDB Datenbank, da zuvor noch keine Erfahrung in der Arbeit mit einer Datenbank bestand.
Das anfängliche Erstellen der Datenbank stellte kein Problem dar. Erst als die zuvor definierten Ressourcen modelliert werden sollten, damit diese nach bestimmten Schemata in der Datenbank abgespeichert werden können, kam es zu kurzen Schwierigkeiten. Dies lag daran, dass die Modellierung der Sub-Ressourcen innerhalb der Primär-Ressourcen etwas komplizierter war als erwartet. Doch auch diese Schwierigkeiten konnten schnell überwunden werden und so konnte die Datenbank wie ursprünglich geplant realisiert werden.
Die Anwendungslogik des Servers wurde grundlegend bereits im Prototypen aus Meilenstein 1 implementiert und musste nur noch um den in Meilenstein 2 ermittelten Aspekt der Mobilität ergänzt sowie mit der Datenbank verbunden werden. Außerdem wurde ein offensichtlicher Faktor, der in Meilenstein 2 übersehen wurde, aber in den Zielen aus Meilenstein 1 festgelegt wurde, nachträglich in der Implementierung ergänzt - Hierbei handelt es sich um das festlegen des Radius in dem sich die Angebote die angezeigt werden befinden sollten, damit es dem Benutzer so möglich ist, individuell anzugeben, wie weit die Angebote höchstens von ihm entfernt sein dürfen.

Für die Implementierung des Servers wurden neben der Klasse Index.js noch 4 weitere Klassen erstellt, in denen verschiedene Aufgaben und Bereiche des Servers verwaltet werden.

db.js

Die Klasse ***db.js*** stellt die direkte Verbindung zu der MongoDB Datenbank dar. Diese Klasse zum einen für das Verbinden mit der MongoDB Datenbank zuständig. Zum anderen werden in dieser Klasse die Schemata festgelegt in denen die späteren Daten – Also Benutzer, Angebote, Reservierungen, Einschränkungen, Einträge, usw - gespeichert werden. Hierbei wurde für jede Ressource ein Schema angelegt und in ein Model gecastet um so später für Objekte der jeweiligen Typen der Ressourcen eine eigene Collection zur Speicherung in der Datenbank verfügbar zu haben.
Außerdem wurden in dieser Klasse alle Methoden angelegt, die die CRUD Operationen für das Arbeiten mit der Datenbank realisieren. Diese Methoden ermöglichen es, Daten aus der Datenbank zu lesen, Daten in der Datenbank zu speichern, zu verändern und zu löschen.
In diesen Methoden wurden außerdem gleichzeitig durch die Library *Joi* die späteren Statuscodes für das Antworten auf spätere HTTP-Anfragen implementiert.

Hierbei wurden alle Statuscodes wie sie geplant umgesetzt. Zusätzlich wurden aufgrund des Feedbacks für den Meilenstein 2 auch noch Statuscodes die ein Serverproblem beschreiben eingefügt.

User.js

In der Klasse ***User.js*** wurden alle HTTP-Requests die den User und dessen Subressourcen (Einträge, Einschränkungen, Nachrichten und Token) betreffen behandelt. Hierbei wurde mit den zuvor erstellten Methoden aus der Klasse db.js gearbeitet um so bei den Anfragen auf die jeweils benötigten Daten in der Datenbank zugreifen zu können. Außerdem war es nötig in dieser Klasse die Validierungs-Schemata für die Ressourcen innerhalb dieser Klasse festzulegen, damit es möglich war, mit den Statuscodes durch die Library *Joi* zu arbeiten. Diese Validierungs-Schemata ermöglichen es festzulegen, welche Attribute bei der Erstellung oder Veränderung eines Datensatzes notwendig sind. Wird bei dem Anlegen eines Benutzerprofils so zum Beispiel vergessen den Benutzernamen anzugeben, so kann Joi anhand von den Validierungs-Schemata mit einem Statuscode antworten, der darüber informiert, dass ein Benutzername angegeben werden muss.

Reservierungen.js

In der Klasse ***Reservierungen.js*** wurden die HTTP-Requests die die Reservierungen betreffen behandelt. Auch hier wurde mit den zuvor erstellten Methoden aus der Klasse db.js gearbeitet. Außerdem wurde hier das Validierungs-Schema für die Reservierungen festgelegt um mit der Library *Joi* arbeiten zu können.

Angebote.js

In der Klasse ***Angebote.js*** wurden die HTTP-Requests die die Angebote sowie deren Subressourcen betreffen behandelt. Auch hier wurde erneut mit den Methoden aus der Klasse db.js gearbeitet und das Validierungs-Schema für die Ressource Angebote festgelegt um mit *Joi* arbeiten zu können.
Außerdem befindet sich in dieser Klasse die Anwendungslogik des Servers – Die Ermittlung des individuellen Abholwertes eines Angebotes für einen Benutzer. Hierbei wurde zum einen eine Wetter API abgefragt um so das momentane Wetter und so den dementsprechenden Wetter-Wert zu ermitteln. Zum anderen wurde anhand von Längen- und Breitengrad der Benutzer und der Angebote die Distanz zwischen den beiden ermittelt. Die Berechnung wurde wie in den Meilensteinen 1 und 2 beschrieben durchgeführt. Alle Angebote die sich außerhalb des spezifizierten Radius befinden werden an dieser Stelle nicht weiter beachtet. Durch einen Vergleich der Benutzer-Einträge mit den Angeboten wurde der Vorlieben-Wert des Benutzers bezüglich eines angebotenen Lebensmittels ermittelt. Hierbei wurde ermittelt ob und wenn ja wie oft der Benutzer ein angebotenes Lebensmittel bereits selbst gekauft hat. Zuletzt wurde noch basierend auf der Mobilität des Benutzers und der Entfernung des angebotenen Lebensmittels ein Wert ermittelt. Diese ermittelten Werte wurden dann zu dem jeweiligen Abholwert des angebotenen Lebensmittels unter Berücksichtigung der individuellen Einschränkungen des Benutzers zusammengefasst. Die Tabelle für die Punkteverteilung der einzelnen Schritte ist in Meilenstein 2 vorzufinden.
In der Route **Angebote/User/:id/:laengengrad/:breitengrad/:mobilität/:radius** ist das Ergebnis dieser Anwendungslogik für den Benutzer abrufbar.

Was lief beim Server nicht so gut?

Insgesamt konnte bei dem Server weitgehend alles realisiert werden, was auch geplant wurde. Einige Aspekte wurden leider nicht so umgesetzt wie geplant. Hierzu gehört zum Beispiel, dass die Kommunikation durch HTTPS nicht umgesetzt werden konnte und so auf die Kommunikation über HTTP zurückgegriffen werden musste.
Auch die Funktion, dass ein Benutzer automatisch über besonders gut passende Angebote in seiner Nähe benachrichtigt wird, konnte nicht mehr in der vorhandenen Zeit realisiert werden.
Bei den HTTP-Requests konnten keine PUT Requests auf Subressourcen umgesetzt werden, daher sind nur PUT-Requests auf die Primärressourcen User, Angebote und Reservierungen möglich.
Die PUT-Requests auf die Ressource Angebote sind hierbei jedoch nur unter der Voraussetzung, dass keine Einschränkungen bei dem Angebot vorliegen möglich.

Die Fehlerbehandlung und das Versenden der Statuscodes wurde nicht so implementiert wie es ursprünglich geplant wurde. Hierbei wurde sich auf den wichtigsten Aspekt beschränkt, welcher die detaillierte Fehlerausgabe bei einem falsch zugestellten Body ist. Die anderen Fehlermeldungen beschränken sich auf eine kurze Nachricht mit einem Hinweis über den aktuellen Zustand. An dieser Stelle wurde mehr Fokus auf die Fehlerbehandlung auf der Seite des Clients gelegt, um mögliche Abstürze zu vermeiden und tatsächlichen Nutzen aus den Statuscodes zu ziehen.

Die Methode „**getUserNachrichtVerlauf**“ wurde zwar realisiert, aber funktioniert nicht so wie geplant. Aus Zeitmangel so wie der niedrigen Priorität dieser spezifischen Methode konnte dieses Problem jedoch nicht mehr behoben werden.
Auch die Methode **„removeToken“** wurde zwar realisiert, aber funktioniert aufgrund eines nicht identifizierten Fehlers leider nicht. Da auch hier die Priorität ein Token zu löschen eher gering war konnte das Problem nicht mehr behoben werden.

Fazit:

Grundsätzlich sind die wir mit dem Ergebnis des Clients sehr zufrieden, da sowohl die Anwendungslogik als auch die komplexen UI-Elemente implementiert werden konnte. Wir waren bei der Planung des User Interfaces sehr optimistisch, sind jedoch den hohen Ansprüchen zu einem sehr hohen Grad nachgekommen. Besonders freut es uns, dass neben der Anwendungslogik selbst die Standortbasierten Funktionen, die Verarbeitung von Bildern, die Notifications und die persistente Speicherung von Daten auf dem Gerät realisiert werden konnten. Das Projekt hat uns eine sehr gute Grundlage für kommende Projekte gegeben und es wurde eine Menge gelernt.

Auch das Ergebnis des Servers ist unserer Meinung nach zufriedenstellend. Besonders große Sorgen hatte uns die Implementierung der MongoDB Datenbank gemacht, da wir keine Erfahrung in diesem Bereich hatten. Da die Implementierung und Einbindung dieser jedoch

sehr gut funktionierte, sind wir vor allem mit diesem Teil sehr zufrieden. Auch mit der Kommunikation zwischen Client und Server sowie der Umsetzung der verteilten Anwendungslogik sind wir insgesamt zufrieden.

Installationsdokumentation:

Client:

1.) Das GitHub Repository https://github.com/kristianczepluch/EISWS1819CzepluchKuebler herunterladen oder alternativ über die Kommandoeingabe Git Bash mit dem Befehl „clone“ klonen.
2.) Android Studio 3.2.1 herunterladen.
3.) Android Studio starten und das Projekt im Ordner MS3/Client/FoodUse öffnen.
4.) Projekt Navigator auf Android stellen und im Ordner res/values/strings.xml die Domainadress und Port des Servers angeben.
5.) Das Projekt über Build -> Make Project erstellen (falls dies nicht schon automatisch geschehen ist) und InstantRun in den Settings ausschalten.
6.) Zielgerät per USB-Kabel und im Entwicklermodus an den Computer anschließen.
7.) Über Run -> Run Aoo das Projekt ausführen und als Zielgerät das angeschlossene Gerät angeben (Alternativ auf den grünen Pfeil oben rechts drücken)
8.) In den Einstellungen des Zielgeräts alle Berechtigungen, insbesondere Standortberechtigungen und Kameraberechtigungen freigeben
9.) Über die Zugangsdaten: Email: MaxMustermann@Mail.de, Passwort: Passwort123 einloggen oder alternativ (empfohlen) sich ein neues Benutzerkonto erstellen. Dies wird empfohlen, da so die Anwendung erst konkret getestet werden kann. Drei BeispielQR-Codes für das Testen der Lebensmittelverwaltung sind ebenfalls im GitHub Repository unter MS3/Beispiel QR-Codes zu finden.

    Hinweis: Die Applikation konnte während der Entwicklung aufgrund mangelnder Ressourcen nur auf einem Endgerät (Samsung Galaxy S9+) getestet werden.

Server:

1) Zuerst muss das folgende Repository entweder gedownloaded oder geklont werden. https://github.com/kristianczepluch/EISWS1819CzepluchKuebler
2) Command Prompt öffnen und in den Ordner /EISWS1819CzepluchKuebler/MS3/Server navigieren.
3) Sicherstellen, dass Node.js auf dem Computer installiert ist. Danach den Befehl „npm install“ ausführen und so alle notwendigen Module installieren.

4) Nachdem die Module installiert wurden kann der Server mit dem Befehl „node index.js“ gestartet werden.
5) Der Server sollte nun unter der eigenen IP Adresse und dem Port 3002 erreichbar sein.
6) Hinweis: Bei dem Testen der verschiedenen Anfragen über Postman ist es notwendig die Ressourcen Einträge und Einschränkungen mit „ae“ zu schreiben (Einschraenkungen, Eintraege).

Server Datenbank:

Für Mac:

1. Auf der Website https://brew.sh den Package Manager Homebrew herunterladen.
2. Im Command Prompt mit dem Befehl „brew Install mongodb“ MongoDB installieren.
3. Directory für MongoDB erstellen mit dem Befehl „sudo mkdir -p /data/db“ und mit dem Befehl „sudo chown -R `id -un` /data/db“ sicherstellen, dass das Directory über die benötigten Berechtigungen verfügt.
4. Mit Befehl „mongod“ den Datenbank-Server starten, damit dieser auf Verbindungen hören kann. -> Die Ausgabe sollte in der letzen Zeile „listening to connections on Port 27017“ anzeigen.
5. Nun unter dem Link https://www.mongodb.com/download-center/compass das Programm MongoDB Compass herunterladen.
6. Compass starten und mit Hostname: localhost und Port: 27017 verbinden .

Für Windows:

1. Unter dem Link https://www.mongodb.com/download-center/community den benötigten MongoDB Server herunterladen und Installation durchführen.
2. Zusätzlich unter dem Link https://www.mongodb.com/download-center/compass das Programm MongoDB Compass herunterladen.
3. Im File Explorer zu Folgendem Ordner navigieren: Programm Files/MongoDB/Server/(Nummer der Version)/bin. In diesem Ordner dann die Datei mongod auswählen und den Pfad kopieren.
4. Unter der Windows Suche die Erweiterten Systemeinstellungen öffnen und dort den Reiter Umgebungsvariablen öffnen. Dort dann Path wählen und unter Neu den zuvor kopierten Pfad einfügen.
5. Command Prompt öffnen und den Befehl „md c:\data\db“ ausführen um das Directory für MongoDB zu erstellen. Dann den Befehl „mongod“ ausführen um so den Datenbank-Server zu starten. -> Die Ausgabe sollte in der letzen Zeile „listening to connections on Port 27017“ anzeigen.
6. Compass starten und mit Hostname: localhost und Port: 27017 verbinden .