
# Entwicklung eines interaktiven Kundeninformationssystems für eine Bäckerei

Studienarbeit

*Gregory Seibert*

21. Mai 2019

**Matrikelnummer, Kurs** 9234269, TINF-16-ITA

**Betreuer** Enrico Keil

**Name, Vorname**: Seibert, Gregory
**Matrikelnummer**: 9234269
**Studiengang/Kurs**: TINF-16-ITA
**Titel der Arbeit**: Entwicklung eines interaktiven Kundeninformationssystems für eine Bäckerei

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Falls sowohl eine gedruckte als auch elektronische Fassung abgegeben wurde, versichere ich zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum

Unterschrift

# Abstract

# Inhaltsverzeichnis

# Glossar

| | |
|---|---|
| **Backend** | Das Backend verwaltet die Daten. |
| **DELETE** | Das Löschen eines bestehenden Datenobjektes. |
| **Frontend** | Das Frontend stellt die notwendigen Informationen dar und ist zur Interaktion mit dem Benutzer. |
| **GET** | Das Abfragen eines bestehenden Datenobjektes. |
| **POST** | Das Erstellen eines neuen Datenobjektes. |
| **PUT** | Das Aktualisieren eines bestehenden Datenobjektes. |
| **Separations of concerns** | Ein Software Engineering Prinzip, bei dem Bereiche mit unterschiedlichen funktionalen Absichten voneinander getrennt werden. |

# Akronyme

| | |
|---|---|
| **ACID** | Atomicity, Consistency, Isolation and Durability. |
| **API** | Application Programming Interface. |
| **CRUD** | Create, Read, Update, Delete. |
| **CSRF** | Cross-Site-Request-Forgery. |
| **DTO** | Data Transfer Object. |
| **HTML** | Hypertext Markup Language. |
| **HTTP** | Hypertext Transfer Protocol. |
| **HTTPS** | Hypertext Transfer Protocol Secure. |
| **JSON** | JavaScript Object Notation. |
| **MITM** | Man In The Middle. |
| **MVC** | Model-View-Controller. |
| **MVCC** | Multi-Version Concurrency Control. |
| **REST** | Representational State Transfer. |
| **SQL** | Structured Query Language. |
| **SSL** | Secure Sockets Layer. |
| **TLS** | Transport Layer Security. |
| **URI** | Uniform Resource Identifier. |
| **URL** | Uniform Resource Locator. |
| **XSS** | Cross-Site-Scripting. |

# Abbildungsverzeichnis

# 1 Einleitung

## 1.1 Stand der Technik

## 1.2 Zielsetzung

# 2 Theoretische Grundlagen

In diesem Kapitel werden die notwendigen Grundlagen der Technologien, die für das Backend und das Frontend notwendig sind, vermittelt.

## 2.1 JSON

JavaScript Object Notation (JSON) ist ein leichtgewichtiges Format zum Austauschen von Daten, typischerweise zwischen Webserver und Client [Abt19, S. 23]. Dieses Format wurde von der JavaScript Object Syntax abgewandelt, weswegen eine bidirektionale Umwandlung im JavaScript Code unkompliziert funktioniert. Doch auch in Programmiersprachen, wie zum Beispiel Java und Swift, gibt es standardmäßig eingebaute Funktionen, um JSON zu parsen. Die Syntax von JSON besteht aus folgenden Regeln [Abt19, S. 23]:

- Arrays bestehen aus eckigen Klammern
- Objekte bestehen aus geschweiften Klammern
- Datenfelder von Objekten werden durch Key-Value Paaren dargestellt
- Datenfelder und Objekte werden durch Kommata getrennt

Im folgenden Codeblock ist ein beispielhaftes Objekt in Form eines JSON Dokuments aufgeführt.

```
{
   "id": 1,
   "name": "Beispielobjekt Eins",
   "objektListe": [
      {
         "id": 2,
         "name": "Beispielobjekt Zwei"
      }
   ]
}
```

## 2.2 REST API

Representational State Transfer (REST) Application Programming Interfaces (APIs) spielen mittlerweile eine große Rolle in der Entwicklung und finden bereits in zahlreichen Projekten namhafter Unternehmen Anwendung. Diese REST Schnittstellen sind sogenannte Programmierschnittstellen über das Hypertext Transfer Protocol (HTTP) [Abt19, S. 259]. Mit deren Hilfe lassen sich Daten beispielsweise im JSON Format (siehe Kapitel 2.1) abrufen und verwalten. Hierbei ist jede Anfrage auf ein spezifisches Objekt oder eine Liste an Objekten bezogen, welche durch eine Uniform Resource Identifier (URI) identifiziert wird [Abt19, S. 259].

Unter anderem sind diese vier Request-Methoden durch REST definiert [Abt19, S. 260f]:

- **GET**: Abfrage eines bestimmten Objekts oder mehrere bestimmte Objekte
- **PUT**: Erstellen oder Bearbeiten eines bestimmten Objekts
- **POST**: Erstellen eines Objekts oder mehrere Objekte
- **DELETE**: Löschen eines bestimmten Objekts

Der Sinn dahinter ist, dass man das Frontend, also das, was der Benutzer sieht, und das Backend, die Verarbeitung und Verwaltung der Daten, trennt. Dieses Vorgehen bietet nicht nur eine erhöhte Sicherheit, sondern ist auch im späteren Verlauf einfacher anzupassen, da nun in modularen Gebieten des Projektes gearbeitet werden kann [Abt19, S. 260]. Deshalb bedienen sich große und bekannte Internetanbieter, wie Google und Amazon, diesem nützlichen Prinzip.

Um REST Schnittstellen komfortabel und zeitsparend testen zu können, bietet sich das Programm Postman[1] an. Die Request-Methode und die Uniform Resource Locator (URL) müssen angegeben werden. Der Request-Body ist nur für Requests zum Anlegen oder Bearbeiten von Daten notwendig. Wie das Ganze in Postman anhand eines Beispiels aussieht, lässt sich der Abbildung 2.1 entnehmen.

[1] https://www.getpostman.com/

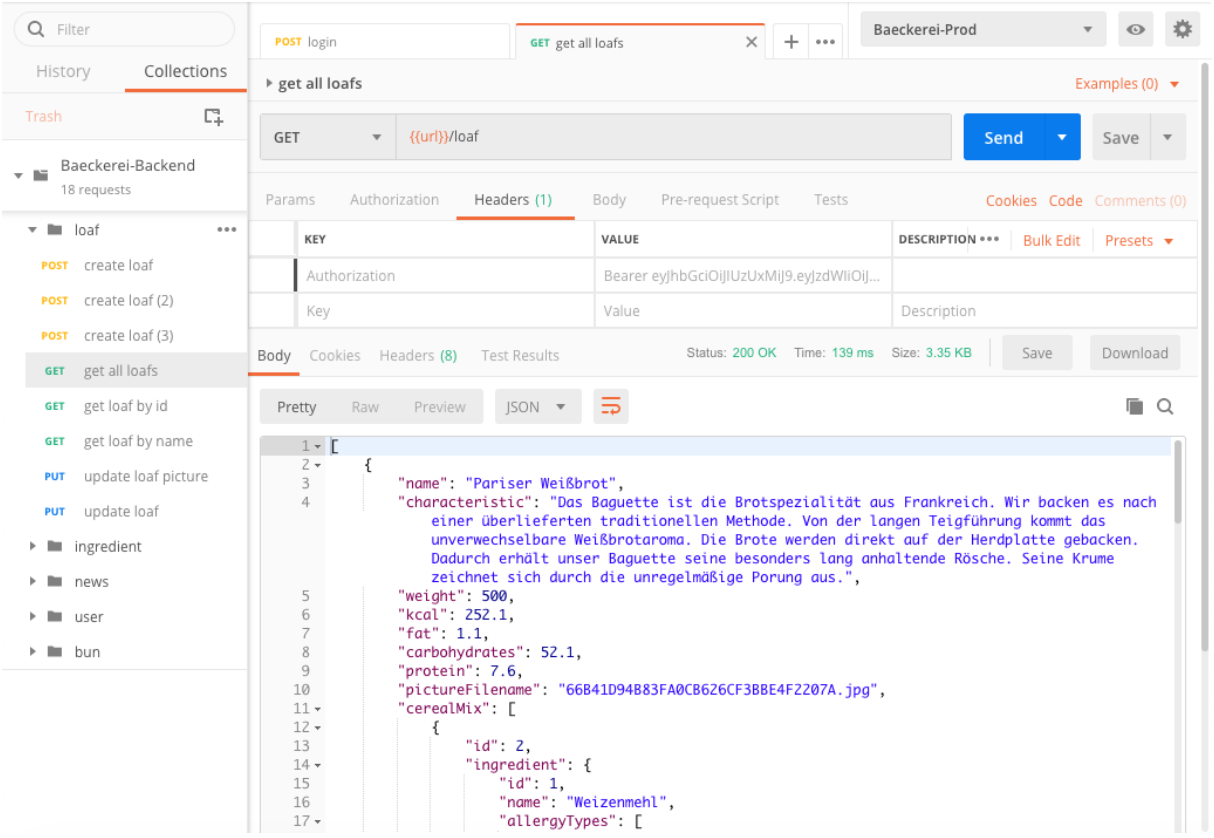
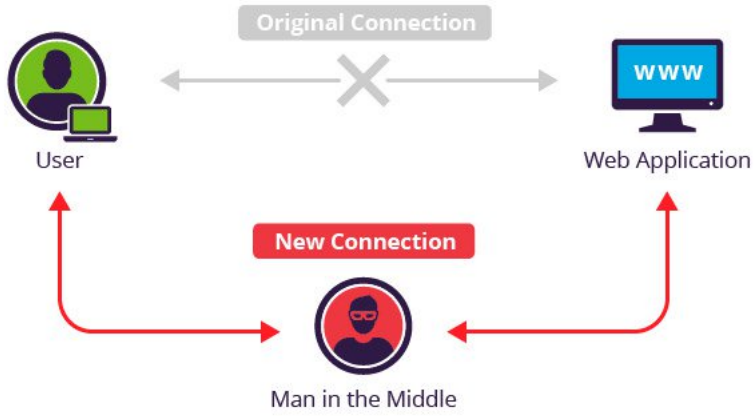



Abbildung 2.1: Das Programm Postman wird genutzt, um REST Schnittstellen zu testen

## 2.3 Sicherheit von Webanwendungen

### 2.3.1 Man In The Middle

Man In The Middle (MITM) ist eine Angriffsform gegen Rechnernetze. Hierbei gibt sich der Angreifer als den Ziel-Server oder einen Router aus. Er greift die Pakete ab, die der Sender an den Empfänger sendet, verfügt über diese nach Wunsch und sendet sie an die vorgesehene Adresse weiter. Dabei bleibt er im schlechtesten Fall sowohl vor dem Sender als auch dem Empfänger versteckt, so dass diese von dem Angriff nichts mitbekommen. Dies wird in der Abbildung 2.2 verdeutlicht. Durch MITM erlangt der Angreifer demnach die vollständige Kontrolle über den Datenverkehr von Dritten, was ihn zum Mitlesen und zur Manipulation des Datenverkehrs befähigt [Roh15, S. 9].




Quelle: https://www.incapsula.com/web-application-security/wp-content/uploads/sites/6/2018/02/man-in-the-middle-mitm.jpg

Abbildung 2.2: Man In The Middle

### 2.3.2 Cross-Site-Request-Forgery

Cross-Site-Request-Forgery (CSRF) ist eine Angriffsform gegen Nutzer einer Webanwendung mit dem Ziel, eine bestehende Sitzung eines Anwenders zu übernehmen oder Anfragen über einen angemeldeten Nutzer auszuführen [Roh15, S. 83]. Hierbei wird eine Anfrage in Form einer URL durch den Angreifer so gewählt, dass die gewünschte Aktion ausgeführt wird. Dies geschieht zum einen, sobald der Nutzer der Webanwendung auf diese URL klickt, zum anderen aber auch durch eingebettete Bilder, Formulare oder Javascript Dateien einer Webseite [Roh15, S. 81]. Solche eingebetteten Weblinks werden standardmäßig automatisch ausgeführt, sobald die Seite geladen wird.

### 2.3.3 Session-Hijacking

Eine Möglichkeit zur Realisierung eines Sitzungsmanagements in Webanwendungen besteht in der Nutzung von Sitzungstoken. Sobald sich der Nutzer über die Webseite anmeldet, wird sein Browser einen Cookie abspeichern, der den Sitzungstoken beinhaltet. Doch auch in mobilen Applikationen ist die Anmeldung und Authentifizierung per Sitzungstoken möglich und wird durch viele Bibliotheken unterstützt. In den meisten Fällen ist dieser Sitzungstoken die alleinige Authentifizierung nach der Anmeldung. Das heißt, dass ein Zugriff auf den Token einen direkten Zugriff auf die Sitzung ermöglicht. Das Abgreifen oder Ermitteln dieses Tokens wird auch als Session-Hijacking bezeichnet [Roh15, S. 89].

### 2.3.4 XSS

Cross-Site-Scripting (XSS) ist eine Angriffsform auf Webanwendungen, welche eine unzureichende Eingabe- oder Ausgabevalidierung der Anwendung ausnutzt [Roh15, S. 79]. Der Angreifer schleust hierbei Schadcode, welcher häufig in Javascript geschrieben wurde, in die Webanwendung ein. Dieser Inhalt wird nun im schlimmsten Fall auf Browsern anderer Nutzer der Anwendung ausgeführt, was unter anderem zum Datendiebstahl oder Infektion des Systems eines Benutzers führen kann [Roh15, S. 77].

## 2.4 Dependency Injection

Dependency Injection ist ein Entwurfsmuster aus der objektorientierten Programmierung, welches dabei hilft, die Abhängigkeiten von Klassen zu organisieren [PP18, S. 27]. Die Abhängigkeiten werden an einem zentralen Ort einmalig instanziiert, identisch zum Singleton Pattern. Dabei werden die jeweiligen Klassen von ihren Abhängigkeiten entkoppelt und es wird verhindert, dass von ressourcen-intensiven Klassen mehrere Instanzen erzeugt werden. Des Weiteren wird die Verwendung und Erstellung von Unit-Tests erleichtert.

Für die Umsetzung der Dependency Injection finden die drei Verfahren

- Constructor Injection
- Setter Injection
- Interface Injection

die meiste Verwendung.

Bei der Constructor Injection werden die Abhängigkeiten einer Klasse durch den Konstruktor übergeben und gesetzt [PP18, S. 119].
Durch jeweilige Methoden können die Abhängigkeiten bei der Setter Injection gesetzt werden [PP18, S. 120].
Die Interface Injection zeichnet sich dadurch aus, dass die abhängige Klasse eine Schnittstelle implementiert, durch die eine Methode vorgegeben wird, über welche die Abhängigkeit zur Verfügung gestellt wird [PP18, S. 120].

## 2.5 Spring Framework

Das Spring Framework[2] ist ein Open-Source Framework für Java basierte Enterprise Projekte. Es wurde entworfen, um die Java EE Entwicklung zu vereinfachen und zu beschleunigen [Sch]. Spring implementiert Funktionalitäten wie

- Dependency Injection (siehe Kapitel 2.4)
- Internationalisierung
- Datenbindung
- Validierung
- Typenkonvertierung

Insbesondere durch das Bereitstellen der Dependency Injection wird ein Einsatz von Softwarekonventionen gefördert, welche für ein leicht wartbares und erweiterbares Projekt sorgen [Wol10, S. 20].

### 2.5.1 Spring Boot

Spring Boot ist eine Erweiterung zum Spring Framework mit dem Fokus auf Konventionen statt Konfiguration [VB15, S. 31]. Dadurch ist es deutlich schneller möglich, eine produktionsfähige Applikation zu erstellen, da bereits durch das Framework die meisten Entscheidungen getroffen wurden. Dennoch lassen sich jederzeit besagte Entscheidungen durch eine eigene Konfiguration überschreiben.

### 2.5.2 Spring Security

Spring Security erweitert das Spring Framework um einige Sicherheitsmechanismen, wie das Authentifizieren von Benutzern oder die Autorisierung in Form eines Rollenschemas mit verschiedenen Berechtigungen.

[2] https://spring.io/

## 2.6 PostgreSQL

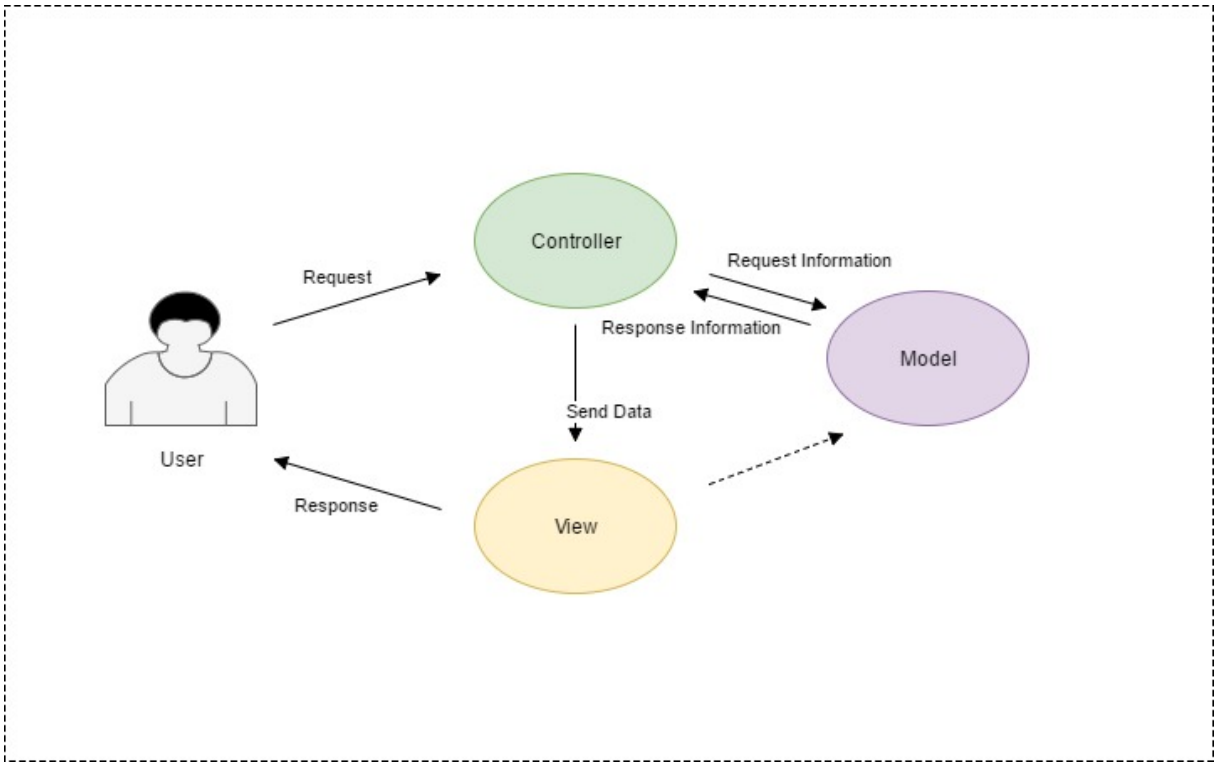
PostgreSQL, oft auch nur Postgres genannt, ist ein objektrelationales Datenbankmanagementsystem, welches als Open-Source Projekt angeboten wird. Es ist ein ehemaliges Projekt der „University of California, Berkeley“, das 1986 gestartet ist [Posa]. Als ein SQL-Interpreter im Jahre 1994 für Postgres geschrieben, woraufhin das gesamte Projekt als Open-Source unter dem Namen Postgres95 freigegeben wurde. Im Jahre 1996 wurde der aktuelle Name PostgreSQL gewählt, um die hinzugefügten SQL Fähigkeiten zu verdeutlichen [Posa].
Postgres ist Atomicity, Consistency, Isolation and Durability (ACID)-konform und fast vollständig mit dem Structured Query Language (SQL) Standard SQL:2011 konform, da mindestens 160 von 179 notwendige Funktionen erfüllt sind [Posb]. Es ist unter anderen mit den Programmiersprachen C, C++, Perl, Python, Java und PHP kompatibel [MS05]. Postgres unterstützt Multi-Version Concurrency Control (MVCC), ein Verfahren, um für eine gleichzeitige und Konsistenz wahrende Ausführung von konkurrierenden Zugriffen auf die Datenbank zu sorgen [Posb]. Die maximale Größe der Datenbanken wird entweder durch 32TB oder durch den tatsächlich verfügbaren Speicher begrenzt [Posb]. Postgres kann zudem durch selbst entworfene Funktionen, Operatoren und Datentypen erweitert werden und es unterstützt einige NoSQL Funktionen [Posb].

## 2.7 MVC Pattern

Das Model-View-Controller (MVC) Pattern ist ein Architekturmuster beziehungsweise Entwurfsmuster im Bereich der Software-Entwicklung. Hierbei wird das Programm in die drei Komponenten

- Model
- View
- Controller

aufgeteilt [VB15, S. 18]. Die Model Schicht enthält alle Klassen, die reale oder irreale Objekte modellieren, also jene, die nach der objektorientierten Programmierung entworfen wurden. Die View Schicht enthält die Oberfläche mit den Elementen zur Interaktion mit der Software. Da die Model Schicht und die View Schicht miteinander kommunizieren müssen und die Eingaben des Benutzers verarbeitet werden müssen, ist die Controller Schicht gleichermaßen essenziell. Dieser Aufbau ist, wie beschrieben, in der Abbildung 2.3 ersichtlich.




Quelle: https://csharpcorner-mindcrackerinc.netdna-ssl.com/article/difference-between-mvc-and-web-forms/Images/MVC.jpg

Abbildung 2.3: Der Aufbau des MVC Patterns

Der Sinn dieses Architekturmusters ist eine lockere Kopplung der einzelnen Software-Module, um die Abhängigkeiten zu verringern und den Wartungsprozess, Erweiterungsprozess sowie Aktualisierungsprozess zu verbessern [VB15, S. 18].

## 2.8 Swift

Swift ist eine Open Source Programmiersprache, die im Jahre 2014 von Apple veröffentlicht wurde. Sie wurde zum Entwickeln von iOS, MacOS und Linux Applikationen entworfen und ist eine nach Objective-C optimierte Programmiersprache [Inc]. Hierbei wurden viele Syntax Besonderheiten anderer Sprachen, wie beispielsweise das Semikolon am Ende einer Zeile oder die runden Klammern bei Verzweigungen oder Schleifen, als optional deklariert. Zudem gibt es in Swift keine Header Datei mehr. Diese Änderungen sollen Swift zu einer einfacheren und übersichtlicheren Programmiersprache gegenüber Objective-C machen [Inc]. Auch werden einige der neuen Programmierparadigmen, wie zum Beispiel Closures, Tupel, mehrere Rückgabewerte, Generics oder Optionals, unterstützt.

# 3 Anforderungsanalyse

## 3.1 Begriffsdefinitionen

Im Folgenden werden die Begriffe, die für die funktionalen und nichtfunktionalen Anforderungen notwendig sind, definiert.

### 3.1.1 Daten

Bei dem Begriff „Daten“ handelt es sich um Informationen zu Backprodukten oder Neuigkeiten.

### 3.1.2 System

Mit dem Begriff „System“ ist die Verwaltung für die hinterlegten Daten gemeint.

### 3.1.3 Administrator

Mit dem Begriff „Administrator“ ist die Person, die für das Pflegen der Daten zulässig ist, gemeint.

### 3.1.4 Benutzer

Mit dem Begriff „Benutzer“ ist eine Technologie zur Anzeige der angelegten Daten gemeint.

## 3.2 Funktionale Anforderungen

### 3.2.1 [FA10] Zentraler Speicherort

Das System muss die Daten zentral in einer Datenbank speichern.

### 3.2.2 [FA20] Zentrale Administration

Das System muss dem Administrator über eine zentrale Schnittstelle die Möglichkeit bieten, die Daten pflegen zu können.

### 3.2.3 [FA30] Alle Backprodukte anzeigen

Das System muss dem Benutzer und dem Administrator die Möglichkeit bieten, die Daten zu allen Backprodukten, sofern bereits welche angelegt wurden, anzeigen lassen zu können.

### 3.2.4 [FA40] Backprodukt anzeigen

Das System muss dem Benutzer und dem Administrator die Möglichkeit bieten, die Daten zu einem ausgewählten Backprodukt, sofern dieses bereits angelegt wurde, anzeigen lassen zu können.

### 3.2.5 [FA50] Backprodukt erstellen

Das System muss dem Administrator die Möglichkeit bieten, ein neues Backprodukt anlegen zu können.

### 3.2.6 [FA60] Backprodukt bearbeiten

Das System muss dem Administrator die Möglichkeit bieten, ein bestehendes Backprodukt bearbeiten zu können.

### 3.2.7 [FA70] Backprodukt löschen

Das System muss dem Administrator die Möglichkeit bieten, ein bestehendes Backprodukt löschen zu können.

### 3.2.8 [FA80] Alle Neuigkeiten anzeigen

Das System muss dem Benutzer und dem Administrator die Möglichkeit bieten, die Daten zu allen Neuigkeiten, sofern bereits welche angelegt wurden, anzeigen lassen zu können.

### 3.2.9 [FA90] Neuigkeit anzeigen

Das System muss dem Benutzer und dem Administrator die Möglichkeit bieten, die Daten zu einer ausgewählten Neuigkeit, sofern diese bereits angelegt wurde, anzeigen lassen zu können.

### 3.2.10 [FA100] Neuigkeit erstellen

Das System muss dem Administrator die Möglichkeit bieten, eine neue Neuigkeit anlegen zu können.

### 3.2.11 [FA110] Neuigkeit bearbeiten

Das System muss dem Administrator die Möglichkeit bieten, eine bestehende Neuigkeit bearbeiten zu können.

### 3.2.12 [FA120] Neuigkeit löschen

Das System muss dem Administrator die Möglichkeit bieten, eine bestehende Neuigkeit löschen zu können.

## 3.3 Nichtfunktionale Anforderungen

### 3.3.1 [NFA10] Authentifizierung zur Administration

Das System muss die Administration von Daten bei einem nicht autorisierten Zugriff verweigern, sofern es sich nicht um den Administrator handelt.

### 3.3.2 [NFA20] Authentifizierung zur Anzeige von Daten

Das System muss die Anzeige von Daten bei einem nicht autorisierten Zugriff verweigern, sofern es sich nicht um den Benutzer handelt.

### 3.3.3 [NFA30] iOS App mit iPad Kompatibilität

Das System muss eine iOS App mit iPad Kompatibilität bereitstellen, um die Daten für Endnutzer anzeigen lassen zu können.

# 4 Konzept

## 4.1 Systemarchitektur

Die funktionalen Anforderungen [FA10] Zentraler Speicherort sowie [FA20] Zentrale Administration fordern eine zentrale Datenverwaltung. Um dies gewährleisten zu können, wird das System in zwei Komponentengruppen, dem Backend und Frontend, aufgeteilt. Das Backend soll eine Anbindung zu einer Postgres Datenbank (siehe Kapitel 2.6) besitzen, um neue sowie bereits vorhandene Daten speichern zu können. Damit der Administrator die Daten möglichst komfortabel pflegen kann und es zusätzlich dem Administrator und dem Benutzer möglich sein soll, die Daten einzusehen, ist ein Frontend notwendig. Das Frontend muss insgesamt zwei Hauptfunktionen erfüllen.

1. Einsehen der Backprodukte für Endkunden (Funktionale Anforderungen [FA30] Alle Backprodukte anzeigen, [FA40] Backprodukt anzeigen, [FA80] Alle Neuigkeiten anzeigen und [FA90] Neuigkeit anzeigen)

2. Pflegen der Daten durch den Administrator (Funktionale Anforderungen [FA40] Backprodukt anzeigen bis [FA70] Backprodukt löschen und [FA100] Neuigkeit erstellen bis [FA120] Neuigkeit löschen)

Die erste Option zur Entwicklung des Frontends besteht darin, dass eine zusammenfassende Anwendung entwickelt wird, welche die beiden Hauptfunktionen vereint. Hierbei können sich die Endkunden die Backprodukte anzeigen lassen und der Administrator kann, nachdem er sich gemäß der nichtfunktionalen Anforderung [NFA10] Authentifizierung zur Administration authentifiziert hat, die Daten pflegen. Dies ließe sich mittels einer nativen iOS Applikation realisieren.
Die zweite Option ist die getrennte Entwicklung zweier Frontend Anwendungen, welche je eine der Hauptfunktionen anbieten. Hierfür eignet sich eine native iOS Applikation zum Einsehen der Backprodukte sowie eine Webseite zur Administration.
Eine einzelne Applikation für das Einsehen und Pflegen der Daten zu entwickeln, würde den kleineren Entwicklungsaufwand bedeuten, da grundlegende, abstrahierte Klassen zur Kommunikation mit dem Backend für beide abgetrennte Bereiche benutzt werden

können. Jedoch kommt mit diesem Vorteil auch ein direkter Nachteil einher. Die Fusion des Administrationsbereichs mit dem Bereich, welcher für Endkunden vorgesehen ist, würde eine Verletzung des Prinzips Separations of concerns bedeuten, was zu einer verringerten Sicherheit des Gesamtsystems führt. Des Weiteren ist das Pflegen von größeren Datensätzen auf einem Tablet im Vergleich zu einem PC unkomfortabel.
Die getrennte Entwicklung zweier Frontend Anwendungen bedeutet einen Mehraufwand zur Entwicklung. Jedoch wird das Prinzip Separations of concerns eingehalten und der Administrator des Systems hat die Entscheidungsfreiheit zur Wahl der Hardware, über die er die Webseite zur Administration aufruft.
Trotz des erhöhten Aufwands zur Entwicklung fällt die Entscheidung auf die zweite Option, die Entwicklung einer iOS Applikation für die Endkunden und einer Webseite zur Administration der Daten. Der bessere Komfort beim Pflegen der Daten sowie die erhöhte Sicherheit sind für das System von größerer Bedeutung als dieser Nachteil. Die Verwaltung der Daten durch den Administrator soll hierbei über ein Web Admin-Dashboard geschehen. Diese Softwarearchitektur ist in der folgenden Abbildung 4.1 als UML Komponentendiagramm dargestellt.

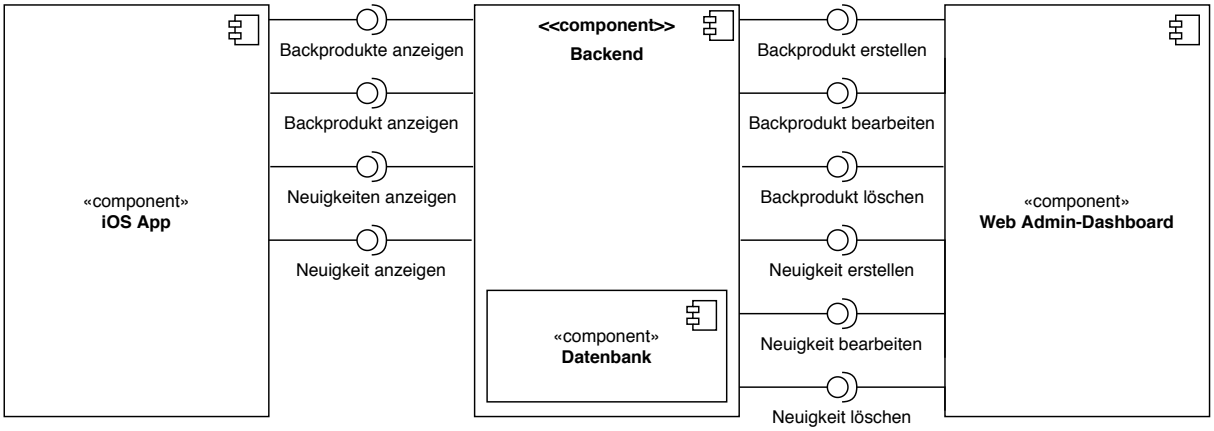



Abbildung 4.1: Das UML Komponentendiagramm für das System, bestehend aus Backend, iOS Applikation und Web Admin-Dashboard

## 4.2 Rollenmodell

Die funktionale Anforderung [FA20] Zentrale Administration setzt voraus, dass der Administrator die Daten über eine zentrale Schnittstelle pflegen kann. Dabei muss dem Administrator die Möglichkeit gegeben werden, nach den funktionalen Anforderungen [FA50] Backprodukt erstellen, [FA60] Backprodukt bearbeiten, [FA70] Backprodukt löschen, [FA100] Neuigkeit erstellen, [FA110] Neuigkeit bearbeiten und [FA120] Neuigkeit löschen, Backprodukte und Neuigkeiten erstellen, bearbeiten und löschen zu können.
Des Weiteren muss es sowohl dem Administrator als auch dem Benutzer des Systems, gemäß den funktionalen Anforderungen [FA30] Alle Backprodukte anzeigen, [FA40] Backprodukt anzeigen, [FA80] Alle Neuigkeiten anzeigen und [FA90] Neuigkeit anzeigen, möglich sein, sich die Daten zu den Backprodukten und Neuigkeiten anzeigen zu lassen.
Hierbei ist nach den nichtfunktionalen Anforderungen [NFA10] Authentifizierung zur Administration und [NFA20] Authentifizierung zur Anzeige von Daten zu beachten, dass die Administration der Daten lediglich bei Existenz einer autorisierten Sitzung des Administrators und das Abfragen der Daten lediglich bei Existenz einer autorisierten Sitzung des Benutzers erlaubt sein darf.
Hieraus folgt, dass es analog zwei Rollen geben soll. Zum einen der Benutzer und zum anderen der Administrator, welcher die für den Benutzer verfügbaren Funktionen erbt. Beiden soll es möglich sein, sich am System anzumelden. Dies ist in der folgenden Abbildung 4.2 als UML Usecase Diagramm visualisiert.

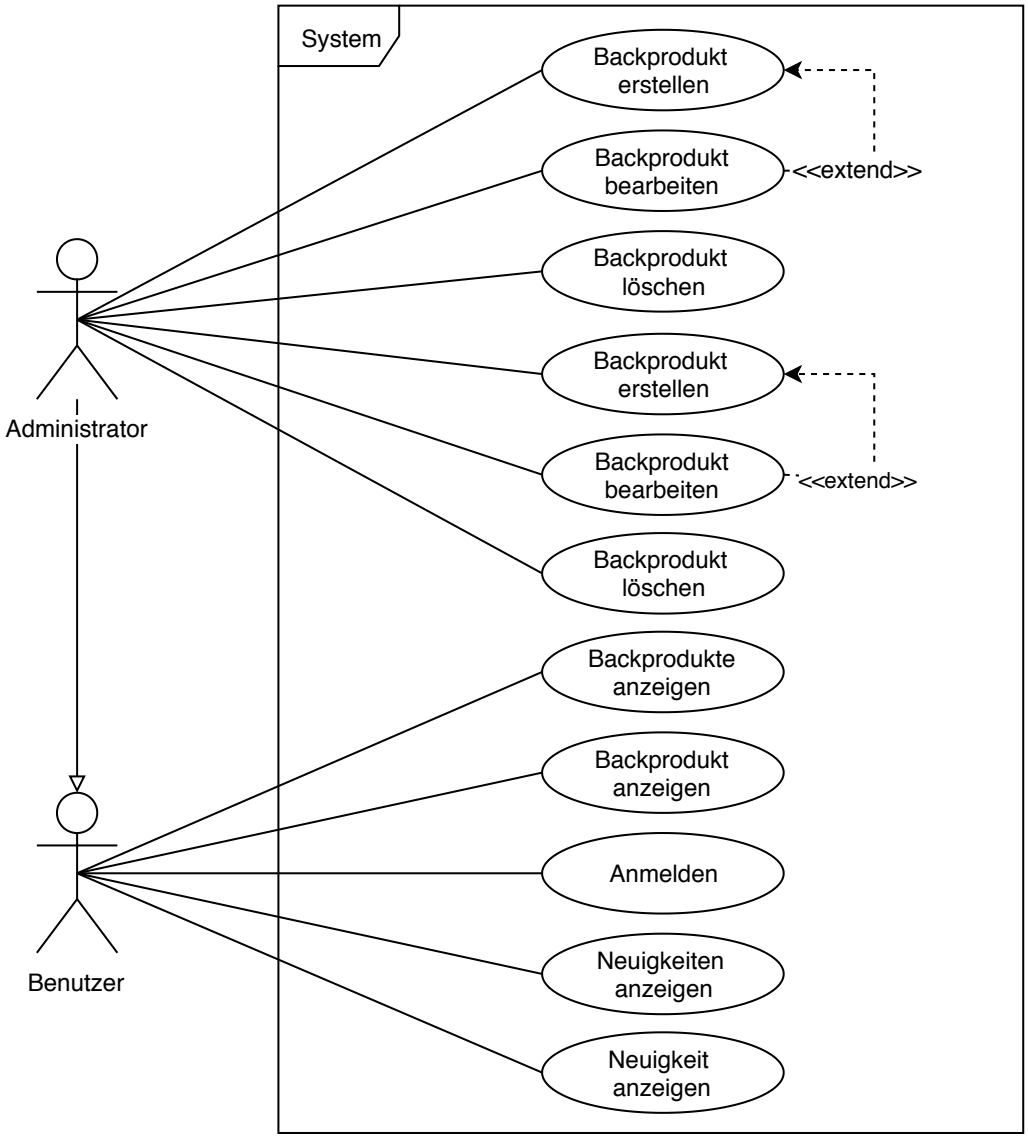



Abbildung 4.2: Das UML Usecase Diagramm für das System und als Aktor der Administrator sowie Benutzer zur Verdeutlichung des Rollenmodells

## 4.3 Datenmodell

Im Kapitel 4.2 wird beschrieben, wie das Rollenmodell des Systems als Entwurf aussehen soll. Im Rahmen des Datenmodells ist hierbei eine Entität, der „ApplicationUser“, notwendig, siehe Abbildung 4.3. Diese Entität besitzt die Attribute „username“ für den Benutzernamen sowie „password“ für das Passwort zur Anmeldung, „name“ für den Namen der jeweiligen Person, „isActive“ für die Möglichkeit zur Deaktivierung und „isArchived“ für die Archivierung des dahinterstehenden Accounts. Des Weiteren besitzt die Entität „ApplicationUser“ eine Liste an „ApplicationUserRole“, wodurch das Rollenmodell vollständig implementiert ist.

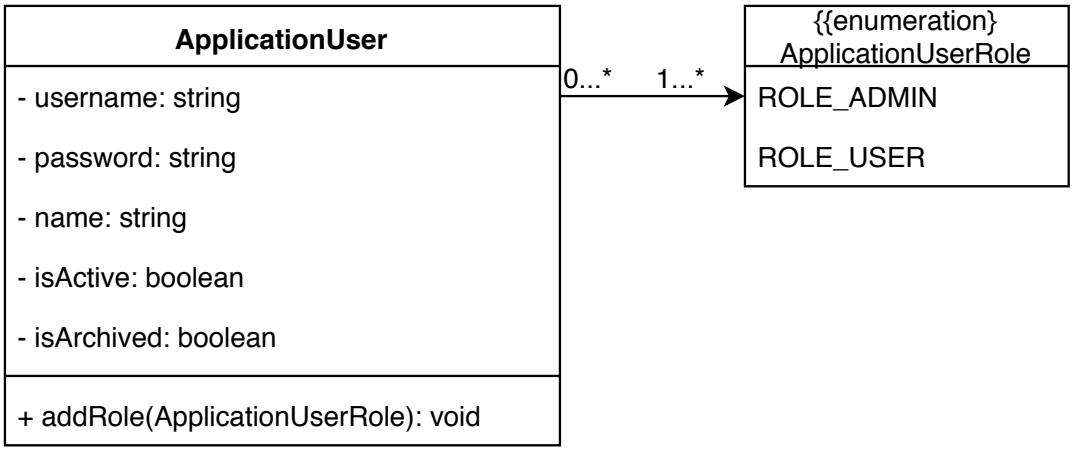



Abbildung 4.3: Das UML Klassendiagramm zur ApplicationUser Entität

Damit das System den funktionalen Anforderungen [FA30] Alle Backprodukte anzeigen, [FA40] Backprodukt anzeigen, [FA50] Backprodukt erstellen, [FA60] Backprodukt bearbeiten, [FA70] Backprodukt löschen und [FA80] Alle Neuigkeiten anzeigen entspricht, ist ein passendes Datenmodell für die abstrakte Klasse „BakedGood“ im Rahmen der objektorientierten Programmierung notwendig. Die Klasse „BakedGood“ wurde als eine abstrakte Klasse entworfen, um diese mit geringem Aufwand um die Kindklassen „Loaf“ und „Bun“ und im Nachhinein um weitere Kindklassen erweitern zu können. Des Weiteren wurden Attribute des Modells „BakedGood“ in die Klassen „Ingredient“ sowie „CerealMixPercentage“ und in die Enumerationen „AllergyType“ sowie „WeekDay“ aufgeteilt. Das Gesamtmodell zur Klasse „BakedGood“ ist in der Abbildung 4.4 als UML Klassendiagramm visualisiert.

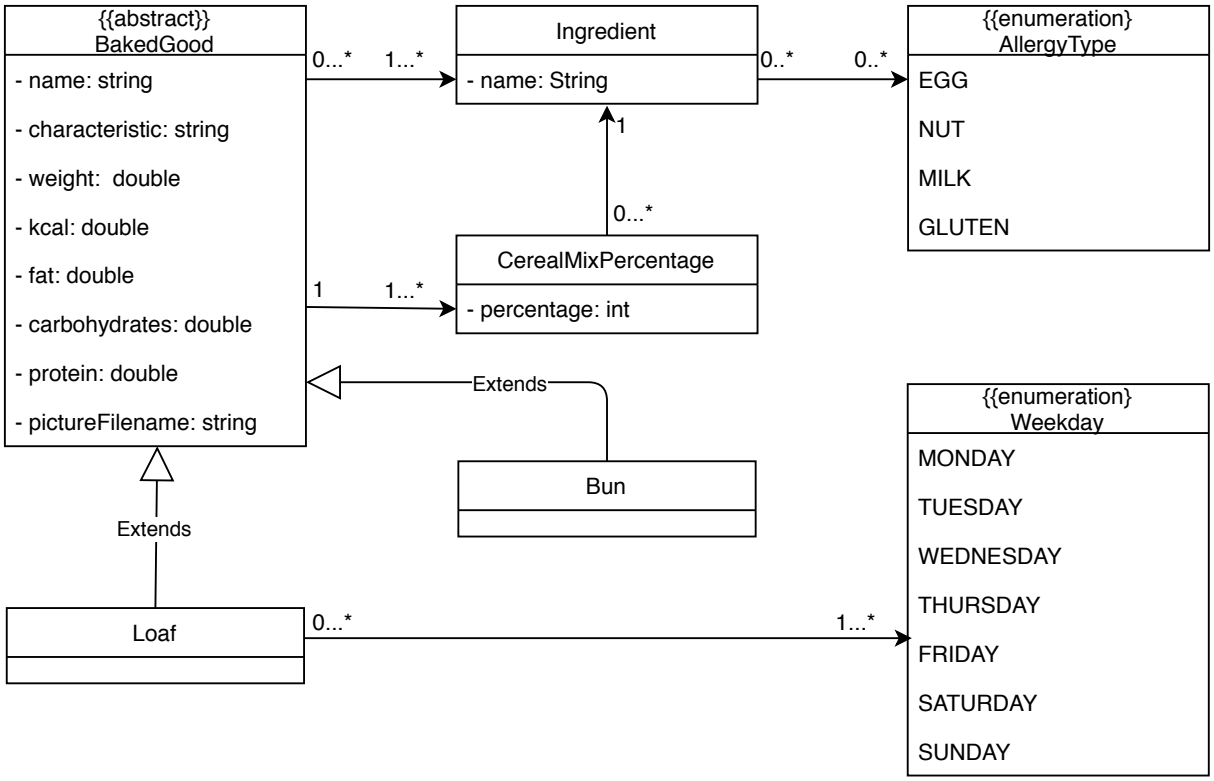



Abbildung 4.4: Das UML Klassendiagramm zur BakedGood Entität

Gemäß den funktionalen Anforderungen [FA90] Neuigkeit anzeigen, [FA100] Neuigkeit erstellen, [FA110] Neuigkeit bearbeiten, und [FA120] Neuigkeit löschen wurde die Klasse „NewsItem“ entworfen. Diese ist in der folgenden Abbildung 4.5 als UML Klassendiagramm dargestellt.

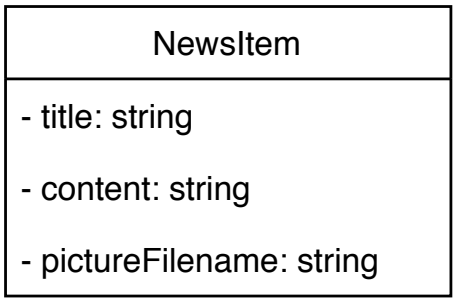



Abbildung 4.5: Das UML Klassendiagramm zur NewsItem Entität

## 4.4 Technologie des Backends

Wie bereits beschrieben, soll das Backend als eine zentrale Anlaufstelle für Create, Read, Update, Delete (CRUD) Operationen zu den Backprodukten dienen. Da es sich hierbei um eine Client-Server Struktur handelt, eignet sich eine REST-API an, um unabhängig des Clients und der jeweilig gewählten Programmiersprache ein Datenverwaltungssystem zu schaffen. Weitere Vorteile und Besonderheiten des Konzepts REST wurden im Kapitel 2.2 beschrieben. Drei beliebte Optionen zum Entwickeln eines Backends sind PHP, Django sowie Spring Boot. An Hand der Kriterien Quellcode, Community, verfügbare Bibliotheken, Aufwand zur Programmierung einer REST-API sowie der verwendeten Programmiersprache soll eine Auswahl zur zu verwendenden Technologie getroffen werden. Da Spring Boot bereits im Kapitel 2.5.1 erläutert wurde, wird im Folgenden lediglich PHP und Django erklärt.
PHP ist eine weit verbreitete Open-Source Skriptsprache, welche in der Webprogrammierung bereits seit Jahren ihre Verwendung findet. Im Gegensatz zu Javascript wird PHP durch den Server ausgeführt, so dass HTML für den Benutzer des Webdienstes generiert wird. Hierbei kann der Benutzer den Code des Webdienstes in der Regel, das heißt bei einem korrekt konfigurierten Webserver, nicht einsehen. PHP wird regelmäßig verbessert, es gibt mittlerweile einige robuste Bibliotheken und die Programmiersprache ist für Anfänger leicht zu lernen.
Django ist ein Web Framework, welches ebenfalls als Open-Source Software verfügbar ist und auf der Programmiersprache Python basiert. Es verfügt über eine vorkonfigurierte Anbindung an Datenbanksysteme, wie es auch bei Spring Boot der Fall ist, und über einige Module, welche beispielsweise ein Authentifizierungssystem bereitstellen.

| Technologie | Quellcode | Community | Bibliotheken | Aufwand | Programmiersprache |
|---|---|---|---|---|---|
| PHP | 5/5 | 5/5 | 4/5 | 1/5 | 3/5 |
| Django | 5/5 | 4/5 | 5/5 | 5/5 | 4/5 |
| Spring Boot | 5/5 | 4/5 | 5/5 | 5/5 | 5/5 |

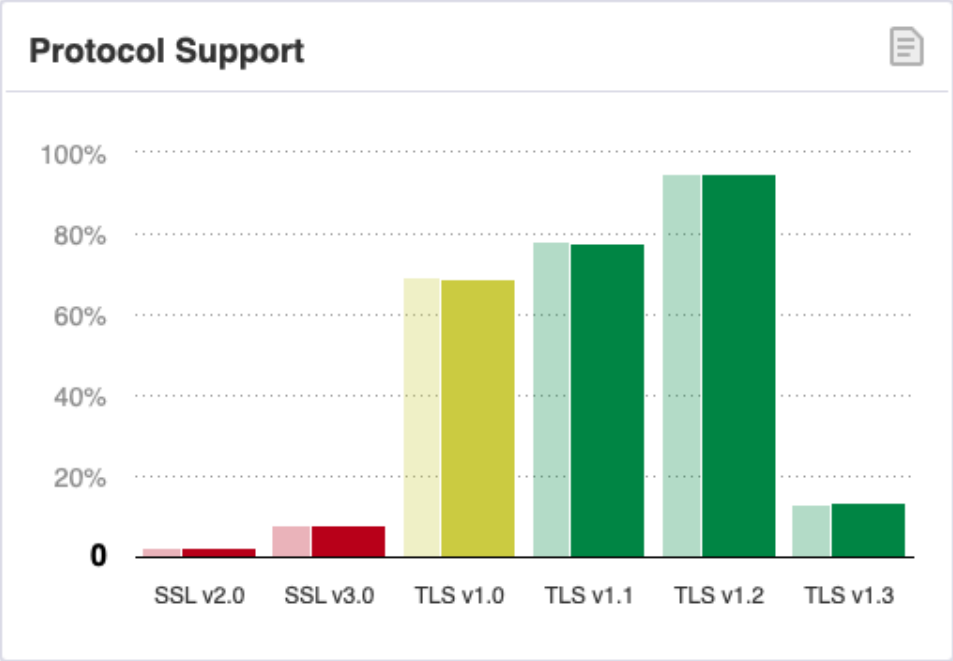
Tabelle 4.1: Vergleich der Backend Technologien PHP, Django und Spring Boot

Zur Implementierung des Backends mit der REST Schnittstelle eignet sich demnach das Spring Framework in Verbund mit Spring Boot (siehe Kapitel 2.5), da es, insbesondere durch voreingestellte Konfigurationen und verfügbare Bibliotheken, die Entwicklung für diesen Anwendungszweck erleichtert.
Die Datenbank wird durch das objektrelationale Datenbankmanagementsystem Postgres (siehe Kapitel 2.6) bereitgestellt.

## 4.5 Sicherheit

### 4.5.1 Verschlüsselung der Kommunikation

Die Kommunikation über HTTP erfolgt lediglich unverschlüsselt. Daher ist es jederzeit möglich, Inhalte von versendeten Paketen mittels MITM mitzulesen oder gar zu manipulieren. Um dies zu verhindern, wurde HTTP um einen Secure Sockets Layer (SSL)/Transport Layer Security (TLS)-Layer erweitert, was als Hypertext Transfer Protocol Secure (HTTPS) bezeichnet wird. Hierbei werden alle übertragenen Daten verschlüsselt und es erfolgt eine gegenseitige Authentifizierung durch SSL-Zertifikate. Mittlerweile wurde jedoch die Versionen SSL1.0 bis SSL3.0 sowie TLS1.0 als unsicher deklariert. Dennoch unterstützen viele Webserver diese unsicheren SSL und TLS Versionen, wie man der Abbildung 4.6 entnehmen kann. Da zur Nutzung des Bäckerei Systems eine Authentifizierung über das Internet vorausgesetzt wird, sollte der Webserver, auf dem das Backend ausgeführt wird, ausschließlich eine Verbindung per HTTPS akzeptieren, da sonst Angreifer die Anmeldedaten abgreifen können.




Quelle: https://www.ssllabs.com/ssl-pulse/

Abbildung 4.6: SSL und TLS Unterstützung von Webservern

### 4.5.2 Eingabevalidierung und Ausgabevalidierung

Webanwendungen, wie beispielsweise Foren, müssen alle Daten validieren, die durch Nutzereingaben übermittelt werden, da es sonst zur Speicherung von unzulässigen Daten oder Werten kommen kann. Eine direkte Folge hiervon äußert sich in XSS, aber auch in einem Einbruch in das System oder generellen Datenverlust. Daher sollten alle Daten, die eingegeben oder ausgegeben werden, stets validiert und gegebenenfalls gesäubert werden. Da über die Administrationsschnittstelle benutzerdefinierte Texte zur Erstellung und Bearbeitung von Datensätzen übertragen werden können, ist es von Vorteil, wenn diese vorher validiert werden. Das System würde jedoch durch eine Validierung der Eingabedaten und Ausgabedaten deutlich an Komplexität gewinnen. Dieser Faktor ist also zu vernachlässigen, da die Erstellung und Bearbeitung von Datensätzen lediglich dem Administrator möglich ist und eine iOS App keine Anfälligkeit gegenüber „xss“ Angriffe besitzt, da diese nicht auf Hypertext Markup Language (HTML) und JavaScript basiert.

### 4.5.3 Sichtbarkeit von privilegierten Schnittstellen

Webanwendungen verfügen meist über Schnittstellen zur Administration, um Daten zu pflegen und zu verwalten. In den meisten Fällen sind diese jedoch ausschließlich intern notwendig und müssen nicht öffentlich erreichbar sein. Dennoch sind viele Administrationsschnittstellen von außen erreichbar, was beispielsweise der Fund des Systems der Schließanlage einer JVA durch das c’t Magazin zeigt [Sta13, S. 78]. Die unmittelbare Folge hiervon ist ein erhöhtes Risiko, insbesondere durch mehr Transparenz für den Angreifer. Daher sollte man solche Schnittstellen zur Administration „verstecken“ oder extern verlagern. Generell sollte man demnach privilegierte Schnittstellen nicht öffentlich zugänglich machen. Da hierdurch jedoch ein Mehraufwand entsteht und das System nicht über überdurchschnittliche Sicherheitsanforderungen verfügt, ist dieser Faktor optional.

### 4.5.4 Sicherheitskopplung durch verbundene Systeme

Mittlerweile bestehen viele Webanwendungen aus einem Backend inklusive einer Datenbank und mindestens einem Frontend, was zu einer Kopplung dieser verbundenen Systeme führt. Kann ein Angreifer die Sicherheitsmechanismen eins dieser Systeme überwinden, so erfolgt eine stark verringerte Sicherheit der anderen Systeme. Dies wird auch als „Pivot-Angriff“ bezeichnet. Daher sollte man als Gegenmaßnahme die Systeme einzeln absichern und kein „blindes Vertrauen“ zwischen den Systemen implementieren.

### 4.5.5 Frameworks oder Bibliotheken

Frameworks und Bibliotheken müssen regelmäßig aktualisiert werden, um vorhandene, starke Schwachstellen auszubessern. Webanwendungen, die Frameworks und Bibliotheken verwenden, müssen daher regelmäßig ihre Abhängigkeiten auf den aktuellen Stand bringen. Ebenfalls sollten nur vertrauenswürdige Bibliotheken und Frameworks verwendet werden. Doch auch die standardmäßigen Sicherheitseinstellungen „Security by Default“ solcher Programme sind meist unzureichend und verlangen einer Maßnahmenüberprüfung. Ansonsten drohen Folgen wie beispielsweise eine Systemkompromittierung, bei der ein Angreifer vollen Zugriff auf das System erlangt. Ein Beispiel hierzu ist die Schwachstelle „Heartbleed“ in OpenSSL.

# 5 Implementierung

In diesem Kapitel wird beschrieben, wie die Backend Komponente und die zwei Frontend Komponenten, eine iOS App sowie ein Web Admin-Dashboard, als Referenz Implementierung entwickelt wird.

## 5.1 Backend

Das Backend des Bäckerei Systems ist die zentrale Datenverwaltung, durch die neue Datensätze angelegt oder bestehende Datensätze abgerufen, bearbeitet oder gelöscht werden können. Um neue Datensätze nachhaltig speichern zu können, besitzt das Backend eine Datenbank.
Im Folgenden wird beschrieben, aus welchen Hauptbestandteilen sich das Spring Backend zusammensetzt. Diese Hauptbestandteile sind die Repository Klassen, die Service Klassen und die Controller Klassen. Die Controller Klassen nutzen die Service Klassen und die Service Klassen nutzen die Repository Klassen als Abstraktionsschicht über der Datenbank.

### Repository Ebene

Wie bereits beschrieben, stellt die Repository Ebene eine direkte Abstraktionsschicht der Datenbank dar. Um eine Repository Klasse erstellen zu können, muss einem Java Interface die Annotation *@Transactional* zugewiesen werden. Das Interface muss außerdem das Interface *CrudRepository* erweitern und den zu behandelnden Objekttyp sowie den Datentyp des Attributs zur Identifikation angeben. Dies ist im folgenden Codeblock als Beispiel aufgeführt.

```
@Transactional
public interface ExampleRepository extends CrudRepository<Example, Long> {
   //optional repository methods
}
```

Hierbei sind Methoden zum Speichern, Aktualisieren und Löschen. von Objekten bereits standardmäßig definiert. Da nach Kapitel 4.3 die Objekte der Klassen *Loaf*, *Bun*, *Ingredient*, *CerealMixPercentage*, *NewsItem* und *ApplicationUser* verwaltet werden müssen, werden entsprechende Repository Klassen entwickelt. Um später im Objektbestand der Klassen *Loaf*, *Bun* und *Ingredient* nach dem Namen suchen zu können, ist eine Methode „findByName“ (siehe folgenden Codeblock) notwendig.

```
Example findByName(String name);
```

Des Weiteren soll es möglich sein. einzelne Objekte oder alle Objekte der Klassen *Loaf*, *Bun*, *Ingredient*, *CerealMixPercentage*, *NewsItem* und *ApplicationUser* abzurufen. Hierfür sind die Methoden *findById* und *findAll* (siehe folgenden Codeblock) notwendig.

```
Example findById(long id);
List<Example> findAll();
```

Für die Klasse „ApplicationUser“ soll es zudem möglich sein, ein Objekt über den Benutzernamen mit der Methode *findByUsername* zu finden sowie mit der Methode „existsByUsername“ zu überprüfen, ob bereits ein Benutzer mit einem spezifischen Benutzernamen existiert (siehe folgenden Codeblock).

```
User findByUsername(String username);
boolean existsByUsername(String username);
```

Die nach diesem Muster implementierten Repository Klassen mit den jeweiligen Methoden für *Loaf*, *Bun*, *Ingredient*, *CerealMixPercentage*, *NewsItem* und *ApplicationUser* sind in der Abbildung 5.1 als UML Klassendiagramm ersichtlich.

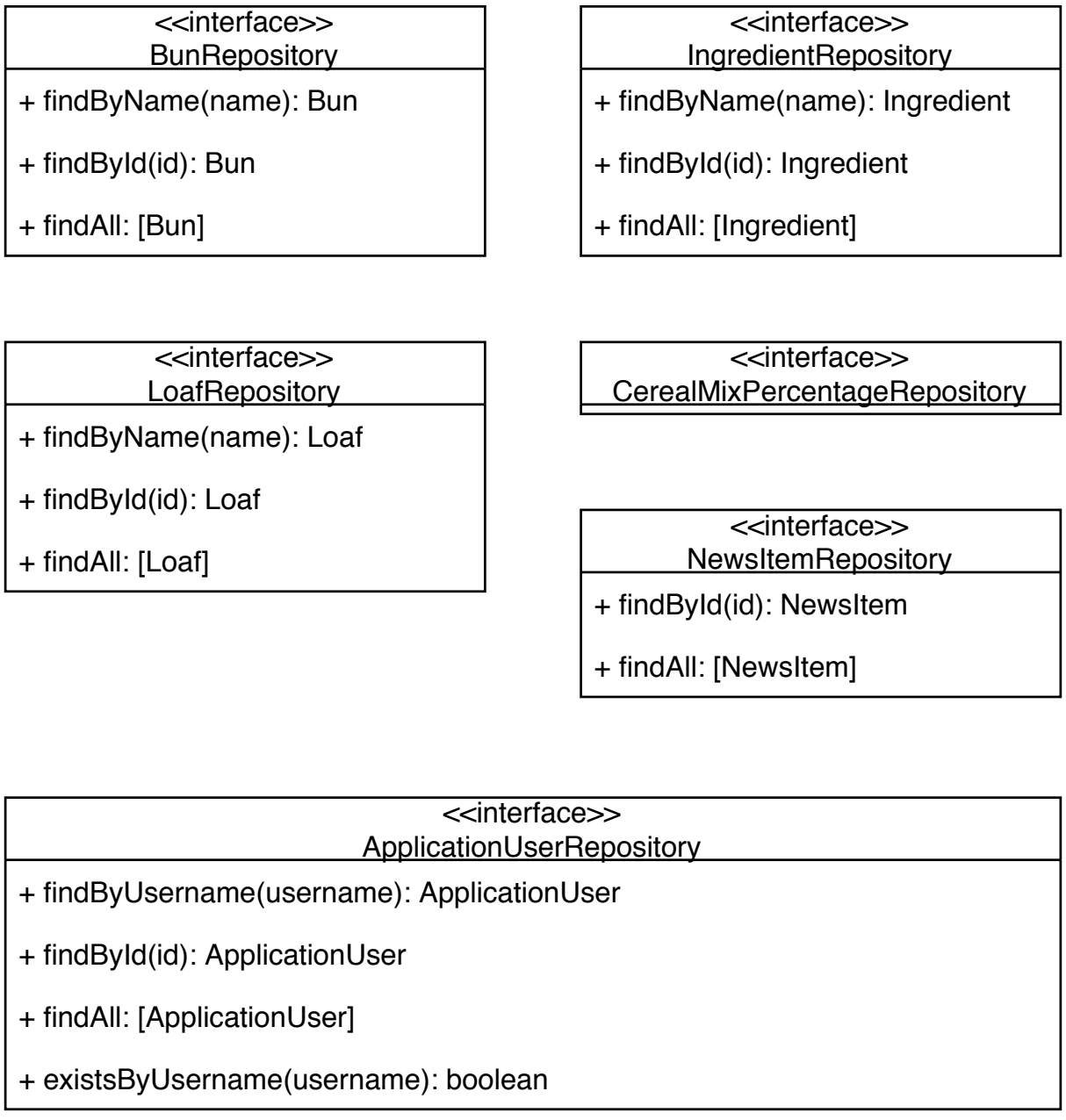



Abbildung 5.1: Die Repository Ebene des Backends

### Service Ebene

Die Service Ebene verwendet die Repository Ebene als Abstraktionsschicht über der Datenbank, um vordefinierte Methoden zum Erstellen, Anzeigen, Bearbeiten, und Löschen von Datensätzen zu ermöglichen. Diese Service Klassen werden anschließend von Controller Klassen verwendet und stellen somit selbst eine Abstraktionsschicht dar. Um eine Klasse als einen Spring Service deklarieren zu können, ist die Verwendung der Annotation *@Service* nowendig.

```
1 @Service
2 public class ExampleService {
3    //service methods
4 }
```

Da eine Referenz zur jeweiligen Instanz einer Repository Klasse notwendig ist, wird diese durch die Annotation *@Autowired* per Dependency Injection bereitgestellt. Analog gilt des auch für die ModelMapper Klasse, um das Architekturmuster Data Transfer Object (DTO) zu unterstützen.

```
1 @Autowired
2 private ModelMapper modelMapper;
3
4 @Autowired
5 private ExampleRepository exampleRepository;
```

Hiermit ist die Implementierung neuer Funktionen in den Service Klassen mit Minimalaufwand möglich, wie in dem folgenden Listing zu sehen ist.

```
1 public Example updateExample(long id, ExampleDTO exampleDTO) {
2        Example existingExample = getExampleById(id);
3
4        modelMapper.map(exampleDTO, existingExample);
5
6        return exampleRepository.save(existingExample);
7    }
```

Nach diesem Schema werden die Service Klassen für *Loaf*, *Bun*, *Ingredient*, *CerealMixPercentage*, *NewsItem* und *ApplicationUser* implementiert. Des Weiteren ist eine Service Klasse mit dem Namen *FileSystemStorageService* notwendig, welche die Speicherung von Daten auf dem Dateisystem ermöglicht (siehe Abbildung 5.2).

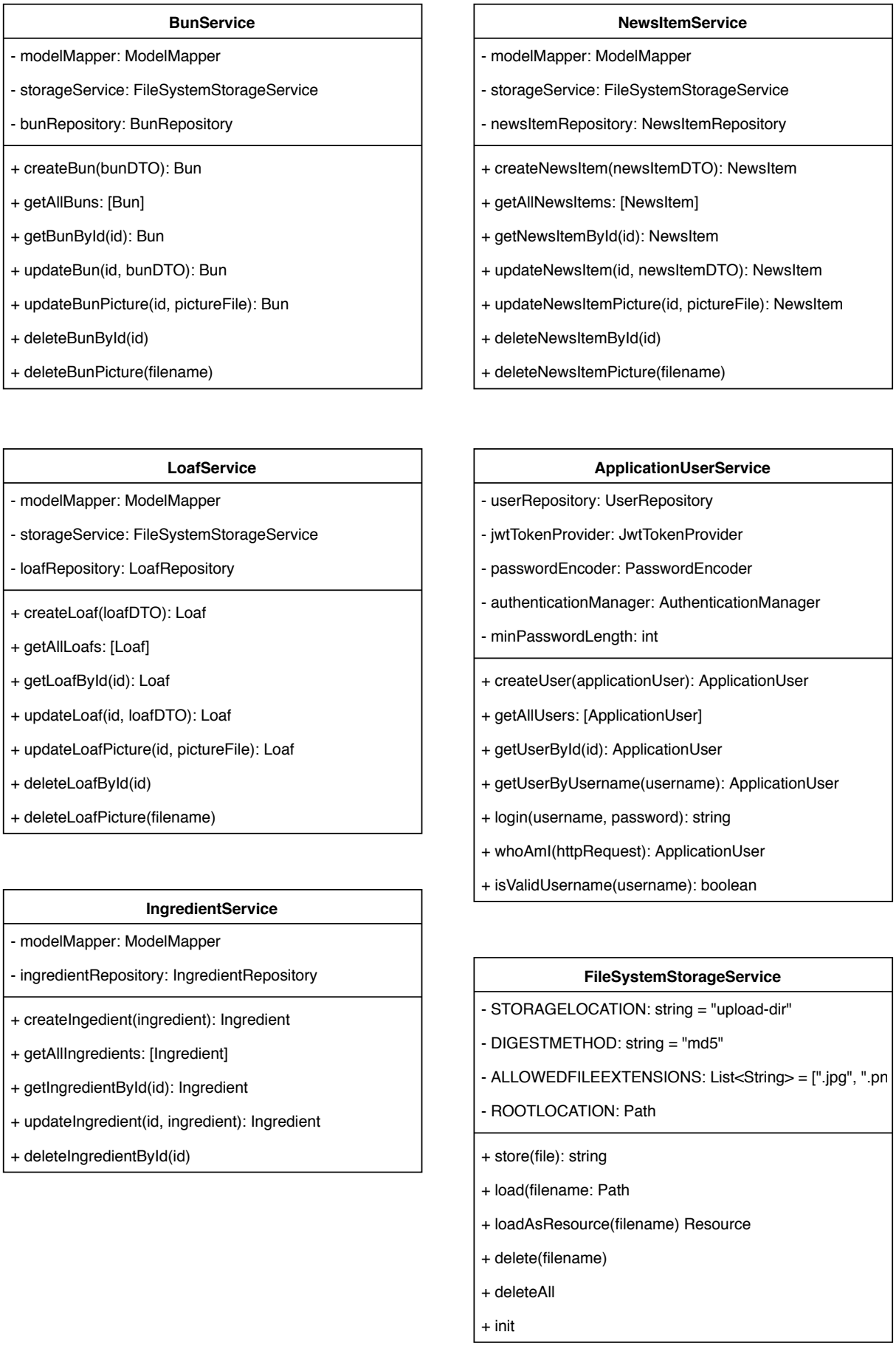



Abbildung 5.2: Die Service Ebene des Backends

### Controller Ebene

Mit der Repository Ebene ist bereits eine Abstraktionsschicht über der Datenbank und mit der Service Ebene ist eine Abstraktionsschicht für bereitgestellte Funktionalitäten vorhanden. Um nun mit deren Hilfe eine REST Schnittstelle anbieten zu können, ist die Controller Ebene notwendig. Eine Controller Klasse kann mittels der Annotation *@RestController* erstellt werden, wobei der Methoden übergreifende Basis-Endpunkt durch die Annotation *@RequestMapping* deklariert werden kann.

```
@RestController
@RequestMapping("/example")
public class ExampleController {
   //controller methods
}
```

Eine Methode kann nun mit Minimalaufwand nach dem folgenden Beispiel implementiert werden.

```
@GetMapping
public List<Example> getAllExamples() {
       return exampleService.getAllExamples();
}
```

Damit eine Methode lediglich durch Administratoren aufgerufen werden können, ist die Annotation *PreAuthorize* notwendig.

```
@PostMapping
@PreAuthorize("hasRole('ROLE_ADMIN')")
public Example createExample(@RequestBody ExampleDTO exampleDTO) {
   return exampleService.createExample(exampleDTO);
}
```

Nach diesem Schema werden die Controller Klassen für *Loaf*, *Bun*, *Ingredient*, *CerealMixPercentage*, *NewsItem* und *ApplicationUser* implementiert (siehe Abbildung 5.3).


Abbildung 5.3: Die Controller Ebene des Backends

## 5.2 iOS Applikation

Damit die Kunden der Bäckerei die im System gespeicherten Daten zu den Produkten anschaulich durchsuchen können, wird eine iOS Applikation entwickelt. Die Referenz-Implementierung hierzu wird in diesem Kapitel erklärt.

### 5.2.1 Architektur

Die Architektur der Applikation ist nach dem MVC Muster, mit einer Modell Ebene, einer View Ebene und einer Controller Ebene, aufgebaut. Dieses Muster wurde bereits im Kapitel 2.7 ausführlich erläutert und dessen Vorteile aufgezählt.

#### Modell Ebene

Die Modellklassen werden analog zum Datenmodell im Kapitel 4.3 implementiert. Im folgenden Codeblock ist eine beispielhafte, gekürzte Umsetzung in der Programmiersprache Swift für ein Backprodukt ersichtlich. Die Klasse für das Backprodukt implementiert das Protokoll *Equatable*, um einen Vergleich zwischen zwei Objekten dieser Klasse nach der Id zu ermöglichen, und das Protokoll „Codable“, damit Swift die JSON Daten in Objekte automatisch konvertieren kann.

```
1  import Foundation
2
3  class BakedGood: Codable, Equatable {
4      let id: Int
5      var name, characteristic, pictureFilename: String
6      var weight, kcal, fat, carbohydrates, protein: Double
7      var cerealMix: [CerealMixPercentage]
8      var ingredients: [Ingredient]
9
10     [...]
11
12     static func == (lhs: BakedGood, rhs: BakedGood) -> Bool {
13         return lhs.id == rhs.id
14     }
15 }
```

### Datasource Ebene

Zur Entkopplung der Abhängigkeiten wird die Logik zur Kommunikation mit dem Backend und zur lokalen Speicherung der Daten als eigene Schicht abstrahiert. Dies bietet den Vorteil, im späteren Verlauf die Datenquelle auszutauschen, ohne alle Klassen der ViewController Ebene abzuändern.
Zur lokalen Speicherung der Daten ist die Klasse *Disk* zuständig, welche im folgenden Codeblock in gekürzter Variante ohne Methodenrümpfe dargestellt ist. Die private Methode *getURL* gibt die URL zurück, welche durch iOS für das angegebene Verzeichnis, *documents* oder *caches*, zur Verfügung gestellt wird und die den spezifischen Speicherort repräsentiert. Die private Methode *storeData* speichert die übergebenen Daten unter dem angegebenen Verzeichnis, indem sie die Funktion *getURL* verwendet, ab. Die Funktion *storeCodable* sorgt für die Umwandlung eines Objekts, dessen Klassen dem *Codable* Protokoll entspricht, in Daten und speichert diese Daten durch Verwendung der privaten Funktion *storeData* und unter der Angabe des Verzeichnisses ab. Damit bereits gespeicherte Daten abgerufen werden können, ist die private Funktion *fetchData* vorhanden. Diese wird von der Funktion *fetchCodable* verwendet, um aus den ausgelesenen Daten wiederum Objekte der angegebenen Klasse zu generieren. Des Weiteren sind Methoden zum Löschen aller Inhalte eines Verzeichnisses mittels *clear*, zum Löschen einer Datei mittels *remove* und zum Überprüfen der Existenz einer Datei mittels *fileExist* implementiert.

```
1  import Foundation
2
3  public class Disk {
4      fileprivate init() { }
5
6      enum Directory { case documents, caches }
7
8      static private func getURL(for directory: Directory) -> URL { ... }
9
10     static private func storeData(data: Data, to directory: Directory, as fileName:
           String) { ... }
11
12     static func storeCodable<T: Codable>(_ objects: [T], to directory: Directory,
           as fileName: String) { ... }
13
14     static private func fetchData(fileName: String, from directory: Directory) ->
           Data { ... }
15
16     static func fetchCodable<DataType: Codable>(_ dump: DataType.Type, fileName:
           String, from directory: Directory) -> [DataType]? { ... }
17
18     static func clear(_ directory: Directory) { }
19
```

```
20      static func remove(_ fileName: String, from directory: Directory) { ... }
21
22      static func fileExists(_ fileName: String, in directory: Directory) -> Bool {
            ... }
23  }
```

Die Klasse *RestApiHandler* ist für die direkte Kommunikation mit einem REST Backend zuständig und nutzt hierfür die Swift internen Bibliotheken. Über die Methode *init*, welche in Swift den Konstruktor repräsentiert, wird die Basis URL gesetzt, welche für alle Anfragen verwendet wird. Durch die private Funktion *createRequest* (nicht authentifiziert) oder *createAuthenticatedRequest* (authentifiziert) an das Backend gesendet werden. Damit die authentifizierten Anfragen jedoch durch das Backend auch als solche erkannt werden, muss vorher der Anmeldeprozess mittels der Funktion *login* durchlaufen sein. Die beiden Methoden *getDecodableArrayFromEndpoint*, zum Abfragen von Objekten, und *getDataFromEndpoint*, zum Abfragen von Ressourcen wie zum Beispiel Bilder, können zum Erstellen und Senden von Anfragen verwendet werden.

```
1   class RestApiHandler {
2       private let apiBaseURL: URL
3
4       init(urlScheme: String, urlHost: String) { ... }
5
6       private func createRequest(endpoint: String, method: String) -> URLRequest {
            ... }
7
8       private func createAuthenticatedRequest(endpoint: String, method: String,
            token: String) -> URLRequest { ... }
9
10      func login(endpoint: String, credentials: ApiCredentials, successCallback:
            @escaping (_ result: String)->(), errorCallback: @escaping (_ result:
            String)->()) { ... }
11
12      func getDecodableArrayFromEndpoint<DataType: Decodable>(_ dump: DataType.Type,
            token: String, endpoint: String, method: String, successCallback: @escaping
            (_ result: [DataType])->(), errorCallback: @escaping (_ result:
            String)->()) { ... }
13
14      func getDataFromEndpoint(endpoint: String) -> Data { ... }
15  }
```

Die Klasse *BakeryDatasource* verwendet die Klasse *RestApiHandler*, um spezialisierte Methoden bereitzustellen. Hierzu gehört *login* zum Anmelden sowie *existingBakedGoodsOnDisk* zum Überprüfen, ob auf dem lokalen Speicher Objekte der Klasse *BakedGood* vorhanden sind. Des Weiteren werden die Funktionen *getBakedGoodsFromAPI* und *getBakedGoodsFromDisk*, zum Abrufen von Objekten der Klasse *BakedGood* vom Backend oder dem lokalen Speicher, bereitgestellt. Analog dazu werden die Funktionen *getNewsItemsFromAPI* und *getNewsItemsFromDisk* für die Klasse *NewsItem* bereitgestellt.

```
1  class BakeryDatasource {
2      public enum Endpoint: String {
3          case login = "/user/login", loaf = "/loaf", bun = "/bun", news = "/news",
               picture = "/picture"
4          func value() -> String { return self.rawValue }
5      }
6
7      public enum DiskFileName: String {
8          case loaf = "loafs.json", bun = "buns.json", news = "news.json"
9          func value() -> String { return self.rawValue }
10     }
11
12     private var authToken: String?
13
14     private lazy var credentials: ApiCredentials = { ... }()
15
16     private lazy var restApiHandler: RestApiHandler = { ... }()
17
18     func login(successCallback: @escaping () -> ()) { ... }
19
20     func getBakedGoodsFromAPI<T: BakedGood>(endpoint: Endpoint, diskFileName:
           DiskFileName, successCallback: @escaping (_ loafArray: [T], _
           imageDataArray: [Data])->()) { ... }
21
22     func getBakedGoodsFromDisk<T: BakedGood>(diskFileName: DiskFileName,
           successCallback: @escaping (_ loafArray: [T], _ imageDataArray:
           [Data])->()) { ... }
23
24     func existingBakedGoodsOnDisk(diskFileName: DiskFileName) -> Bool { ... }
25
26     func getNewsItemsFromAPI(successCallback: @escaping (_ newsArray: [NewsItem], _
           imageDataArray: [Data])->()) { ... }
27
28     func getNewsItemsFromDisk(successCallback: @escaping (_ newsArray: [NewsItem],
           _ imageDataArray: [Data])->()) { ... }
29 }
```

### View Ebene

### Controller Ebene

### 5.2.2 Anzeige von Broten

### 5.2.3 Anzeige von Brötchen

### 5.2.4 Anzeige von Neuigkeiten

## 5.3 Web Admin-Dashboard

### 5.3.1 Architektur

#### Component Ebene

### Page Ebene

### 5.3.2 Erstellen und bearbeiten von Zutaten

### 5.3.3 Erstellen und bearbeiten von Broten

### 5.3.4 Erstellen und bearbeiten von Brötchen

### 5.3.5 Erstellen und bearbeiten von Neuigkeiten

# 6 Zusammenfassung und Ausblick

## 6.1 Zusammenfassung

## 6.2 Ausblick

# Literatur


[Abt19] Prof. Dr. Dietmar Abts. *Masterkurs Client/Server-Programmierung mit Java*. Springer Fachmedien Wiesbaden, 2019. ISBN: 978-3-658-25924-2.

[Inc] Apple Inc. *Swift*. URL: https://www.apple.com/de/swift/ (besucht am 20. 05. 2019).

[MS05] Neil Matthew und Richard Stones. *Beginning Databases with PostgreSQL*. Apress, 2005. ISBN: 978-1-59059-478-0.

[Posa] PostgreSQL. *A Brief History of PostgreSQL*. URL: https://www.postgresql.org/docs/current/static/history.html (besucht am 27. 03. 2019).

[Posb] PostgreSQL. *About PostgreSQL*. URL: https://www.postgresql.org/about/ (besucht am 27. 03. 2019).

[PP18] Nilang Patel und Krunal Patel. *Java 9 Dependency Injection*. Packt Publishing Ltd, 2018. ISBN: 9781788296250.

[Roh15] Matthias Rohr. *Sicherheit von Webanwendungen in der Praxis*. Springer Vieweg, 2015. ISBN: 9783658038519.

[Sch] Sebastian Schelter. *Das Spring Framework – eine Einführung*. URL: http://www.inf.fu-berlin.de/inst/ag-se/teaching/S-BSE/134_schelter_spring.pdf (besucht am 27. 03. 2019).

[Sta13] Louis-F. Stahl. *Industrieanlagen gehackt*. c't-Magazin, November 2013.

[VB15] Balaji Varanasi und Sudha Belida. *Spring REST*. Apress, 2015. ISBN: 978-1-4842-0824-3.

[Wol10] Eberhard Wolff. *Spring 3 - Framework für die Java-Entwicklung*. dpunkt.verlag, 2010. ISBN: 978-3-89864-572-0.