retten

Retten

Projekt Softwareentwicklung - WiSe 18/19

HTW Berlin

[Max Ruhnau](https://github.com/MaxArne)

[Giovanni Rodriguez](https://github.com/whiterabbit-ce)

[Alain C.Mendy](https://github.com/amendy)

[Paul Senff](https://github.com/PaulKetchup)

Inhaltsverzeichnis

EinfÃ¼hrung

(
)

1. [Abstrakt](#Abstrakt)

2. [Produktidee](#Produktidee)

Unser Produkt soll helfen zu verhindern, dass Lebensmittel weggeschmissen werden, welche noch Verwendung finden kÃ¶nnen. In Deutschland und auch in vielen anderen LÃ¤ndern landen noch intakte Produkte auf dem MÃ¼ll. Auf der ganzen Welt werden rund 1,3 Milliarden Tonnen Lebensmittel jÃ¤hrlich entsorgt siehe https://www.wwf.de/themen-projekte/landwirtschaft/ernaehrung-konsum/verschwendung/ und https://www.bzfe.de/inhalt/lebensmittelverschwendung-1868.html. Um diesen Trend zumindest marginal entgegenzuwirken, haben wir die App â€žrettenâ€œ entwickelt.
Mit unserer App kann ein Supermarkt Produkte verkaufen, die er normalerweise entsorgen mÃ¼sste. Die Kunden kÃ¶nnen mithilfe der App Produkte gÃ¼nstiger erwerben und direkt bei ihrem Supermarkt in der NÃ¤he abholen. Die App versucht hierbei, mÃ¶glichst einfach und schnell zwischen Kunde und Supermarkt zu vermitteln und den Gebrauch fÃ¼r beide Seiten so einfach wie mÃ¶glich zu gestalten.

3. [Ziele](#Ziele)
Ziele

Zu Beginn des Projektes wurden verschiedene Soll- sowie Kann-Ziele determiniert, welche im Folgenden nÃ¤her erlÃ¤utert werden:
```
  Soll-Ziele:```

1. Die App soll zwei Benutzerprofile haben â€“ den Supermarkt und den Kunden. Beide Benutzertypen haben unterschiedliche Views zur VerfÃ¼gung und kÃ¶nnen somit verschiedene AktivitÃ¤ten ausfÃ¼hren.
2. Die Kunden sollen nach MÃ¤rkten oder direkt nach Produkten suchen kÃ¶nnen.
3. Der Warenkorb soll verschiedene ZahlungsmÃ¶glichkeiten bieten und eine Ãœbersicht sowohl Ã¼ber die Produkte als auch Ã¼ber den Warenwert geben.
4. Alle Produkte mÃ¼ssen nach Kategorien sortiert sein und gewisse Informationen wie beispielsweise Haltbarkeitsdatum und Preis enthalten.
5. SupermÃ¤rkte mÃ¼ssen Produkte hinzufÃ¼gen und lÃ¶schen kÃ¶nnen.
6. Kunden mÃ¼ssen Produkte reservieren kÃ¶nnen.
```

```
  Kann-Ziele:```

1. Es kann eine Karte entwickelt werden, auf welcher der aktuelle Standort angezeigt wird sowie die darum liegenden MÃ¤rkte. Somit wird die Suche nach dem nÃ¤chstgelegenen Markt erleichtert.
2. Es kann einen Barcodescanner geben, mit dessen Hilfe die Produkte leichter der Produktliste hinzugefÃ¼gt werden kÃ¶nnen.
3. Es kann eine MÃ¶glichkeit implementiert werden, mit welcher die Kaufabwicklung beim Supermarkt vereinfacht wird. Ein Beispiel hierfÃ¼r ist ein generierter QR-Code, welcher auf einen bestimmten Warenkorb referenziert.
In der finalen Zusammenfassung wird nÃ¤her auf den Erfolg sowie den Abschluss der Soll- und Kann-Ziele eingegangen.

## Produktmodelle <a name="Produktmodelle"></a>

In dem Punkt Produktmodelle wird nÃ¤her auf die Planung der Datenbank und den Ablauf unserer Screens eingegangen. Als Erstes wird der Punkt â€žDatenbankâ€œ nÃ¤her diskutiert und Ã¼ber der konzeptionelle und relationale Datenbankentwurf erlÃ¤utert.
.....

1. [Datenbank](#db)

![Datenbank]

(<a href="https://imgur.com/eSOT5mS"><img src="https://i.imgur.com/eSOT5mS.jpg" title="source: imgur.com" /></a>)


 2. [Zustandsdiagramm](#ZustandDiagramm)

In folgendem Zustandsdiagramm auf Bild  wird der Ablauf unserer App verdeutlicht.

![Zustandsdiagramm](<a href="https://imgur.com/dbI6BT8"><img src="https://i.imgur.com/dbI6BT8.jpg" title="source: imgur.com" /></a>)

Beim ersten Ã–ffnen wird eine Startseite mit der Option einer Anmeldung oder Registrierung angezeigt. Dort kann sich der Kunde registrieren. Ein neuer Supermarkt kann nur von einem Admin erstellt werden, um zu verhindern, dass sich Privatpersonen als Supermarkt ausgeben. Ist eine Anmeldung erfolgt, wird der jeweilige nutzerspezifische Homescreen aufgerufen.
Als Kunde steht die Auswahl zwischen MÃ¤rkten und Produkten aus. Bei den MÃ¤rkten werden die einzelnen Symbole angezeigt, woraufhin einer dieser ausgewÃ¤hlt werden kann. Jeder Markt besitzt eine eigene Liste mit den angebotenen Produkten. Zeitgleich kann auch direkt nach Produkten gesucht werden. Die Produkte sind in bestimmt Kategorien unterteilt und werden dann in einer Liste angezeigt. In der Produktliste kÃ¶nnen Produkte dem eigenen Warenkorb hinzugefÃ¼gt und mithilfe eines Bezahlverfahrens und eines generierten QR-Codes direkt beim jeweiligen Markt abgeholt werden. In jeder der Schritte kann mithilfe der ZurÃ¼cktaste zur vorherigen Aktion zurÃ¼ckgekehrt werden.
Als Supermarkt kann in der Produktliste ein neues Produkt hinzugefÃ¼gt oder entfernt werden. AuÃŸerdem kÃ¶nnen reservierte Produkte eingesehen und nach ihrem Bezahlstatus Ã¼berprÃ¼ft werden.
Der Admin kann SupermÃ¤rkte hinzufÃ¼gen oder Informationen Ã¼ber die MÃ¤rkte Ã¤ndern und lÃ¶schen.

3. [Widgets](#Widgets)


Der Entwurf der Widgets legt, wie auch in den Zielen festgelegt, zwei verschiedene Sichten fest. Die erste Sicht, in Bild (Nummer einfÃ¼gen!) zu sehen, ist die eines Supermarktes. Dieser hat Links eine Anmeldeseite. Auf der nÃ¤chsten Seite ist eine Auswahl zu erkennen, welche bereits im Zustandsdiagramm gezeigt wurde. Hier kann der Supermarkt eine Liste einsehen und mithilfe dieser neue Produkte hinzufÃ¼gen oder lÃ¶schen. Diese Liste ist exemplarisch auf â€žSicht 3â€œ gezeigt. Das LÃ¶schen ist mit der Checkboxfunktion vereinfacht. Jedes Produkt kann zudem erweitert werden, um nÃ¤here Informationen zu erhalten. Im zweiten Auswahlpunkt, den reservierten Produkten, kann der Supermarkt die Kunden mit den jeweiligen Produktlisten einsehen. Diese Liste kann auf â€žSicht 6â€œ gesehen werden. Unter Zuhilfenahme dieser Listen kÃ¶nnen somit die Produkte verpackt und abholfertig fÃ¼r den Kunden vorbereitet werden. Auch der Bezahlstatus kann dieser Liste entnommen werden.

Bild:Supermarktsicht Widgets
Auf dem Bild (Nummer einfÃ¼gen!) ist die Sicht des Kunden dargestellt. Genauso wie beim Supermarkt gibt es auch hier die Anmeldung in â€žSicht 1â€œ. Auf der zweiten Sicht gibt es wieder eine Auswahl, mit dessen Hilfe zwischen einer Produktsuche nach MÃ¤rkten und nach Produkten unterschieden wird. Auf der MÃ¤rktesicht sind die verschiedenen Symbole der MÃ¤rkte dargestellt. Jedes Symbol kann erweitert werden, um mehr Informationen, wie die Adresse oder Ã–ffnungszeiten, preis zu geben. Unter dem Punkt â€žProkteâ€œ kÃ¶nnen die Waren nach Kategorien sortiert und mithilfe von Checkboxen in den Einkaufskorb gelegt werden. Im Einkaufskorb kann zwischen einer Bar- und einer Kartenzahlung gewÃ¤hlt werden.
Bild:Kundensicht Widgets

4. [Programme](#Programm)

Mithilfe des Programms â€žUMLETâ€œ wurden die Bilder â€¦, â€¦, â€¦ (Nummern einfÃ¼gen!) erstellt. Das Programm bietet die schnelle und einfache MÃ¶glichkeit, konzeptionelle und relationale Datenbankmodelle zu erstellen, sowie den Entwurf von Zustandsdiagrammen. â€žUMLETâ€œ funktioniert mit einem Drag&Drop System und ein paar einfachen Kommandos innerhalb der Vorlagen. Mit jeweils vor und nach dem Wort eingefÃ¼gten Unterstrichen, kann dieses unterstrichen eingetragen werden. Mit dem SchlÃ¼sselwort â€žbg=â€œ kann die Hintergrundfarbe geÃ¤ndert werden, um lediglich einige Beispiele zu nennen.

## Implementierung der Anwendung <a name="#db"></a>


Bei der Wahl der Technologie haben wir uns fÃ¼r die Tool-Suite â€žFirebaseâ€œ von Google entschieden. Diese beinhaltet umfangreiche FunktionalitÃ¤ten. FÃ¼r die Datenbank wird die â€žRealtime Databaseâ€œ genutzt. Damit ist es mÃ¶glich sehr effizient zu entwickeln und es wird dem Programmierer viel Konfiguration abgenommen. AuÃŸerdem ist sichergestellt, dass diese Technologie fehlerfrei funktioniert und auch unter hohen Belastungen standhÃ¤lt. Es muss kein Server initialisiert werden. Daten werden in der Cloud gespeichert. Es handelt sich um eine nicht relationale Datenbank (NoSQL). Einer der HauptgrÃ¼nde fÃ¼r die Wahl dieser Technologie sind die Synchronisationsmechanismen in Echtzeit. Sogenannte Event-Listener Ã¼berprÃ¼fen dauerhaft ob sich Daten geÃ¤ndert haben. Jedes GerÃ¤t welches mit dieser Datenbank verbunden ist aktualisiert sich automatisch. Somit werden keine komplizierten Netzwerkbibliotheken benÃ¶tigt. Ein weiterer Vorteil ist das garantierte Funktionieren der App bei Verbindungsunterbrechungen des GerÃ¤tes mit dem Internet. Der aktuelle Zustand der Datenbank wird lokal auf dem GerÃ¤t gespeichert. Auf diesen Datenbestand wird zugegriffen, wenn die Datenbank in der
```

Cloud nicht erreichbar ist. Sobald die Internetverbindung wieder aufgebaut wird erhÃ¤lt der Klient den aktuellen Datenbestand.
Im Gegensatz zu SQL-Datenbanken werden keine Tabellen und Relationen verwendet. Alle Daten sind JSON-Objekte. Beim HinzufÃ¼gen eines Objektes wird ein neuer Knoten in diesem Baum erzeugt. Jeder Knoten ist eindeutig durch einen SchlÃ¼ssel identifizierbar. Das Datenbankmodell mit seinen EntitÃ¤ten wird in Java-Klassen umgesetzt. Jede EntitÃ¤t entspricht eine Klasse mit Attributen. Attribute ergeben Kind-Knoten mit einem SchlÃ¼ssel â€“ Werte Paar. So ergibt sich eine Verschachtelung mit mehreren Ebenen. Beim Entwurf wurde darauf geachtet, dass die Verschachtelung eine geringe Anzahl von Ebenen, eine geringe Tiefe aufweist. Die Methoden sind nicht von Relevanz. Dazu wird im Projektverzeichnis ein neuer Ordner â€šdatabaseâ€˜ angelegt. Dort entspricht jede Datei einer Java Klasse. Es wurden die Klassen â€šSupermarktâ€˜ und â€šProduktâ€˜ angelegt. Die Attribute der Klassen wie z.B. â€šMarktnameâ€˜ oder â€šÃ–ffnungszeitâ€˜ entsprechen EintrÃ¤ge in den JSON-Objekten. Beim Erzeugen eines Objektes wird ein Knoten in der JSON-Baumstruktur angelegt.

1. [Implementierung der Datenbank in Firebase](#dbFirebase)

Im Android-Studio ist Firebase bereits installiert. Es muss dem Projekt als AbhÃ¤ngigkeit hinzugefÃ¼gt werden. Dazu wird in der build.gradle Datei ein entsprechender Eintrag im â€šdependencyâ€˜ Codeabschnitt eingefÃ¼gt.
```java

implementation 'com.google.firebase:firebase-database:16.0.5'
```

Dabei ist die Versionsnummer zu beachten. Diese muss mit den anderen Ã¼bereinstimmen, sonst ist das Projekt nicht kompilierbar. Beim Start der App wird die Datenbank initialisiert. Dazu wird eine Instanz erzeugt und dann die Referenz in einer Variablen gespeichert.
```java

private DatabaseReference mDatabase;

mDatabase = FirebaseDatabase.getInstance().getReference();
```java

Diese Referenz wird verwendet, wenn Daten in die Datenbank geschrieben werden. Um Daten auf VerÃ¤nderung zu Ã¼berprÃ¼fen und daraufhin den Zustand der App entsprechend zu aktualisieren werden asynchrone Listener an die Referenz Ã¼bergeben. Das ist zum Beispiel die periodische Dekrementierung des Ablaufdatums.  Der Listener wird am Anfang und bei VerÃ¤nderung der zu Ã¼berwachenden Daten aufgerufen. Wegen der VerÃ¤nderung des Ablaufdatums Ã¤ndert sich die Anzeige auf der OberflÃ¤che, dem TextView. Der Benutzer weiÃŸ sofort Bescheid. FÃ¼r das HinzufÃ¼gen von â€šSupermarktâ€˜ und â€šProduktâ€˜ wird jeweils eine eigene Activity angelegt. In diesen Activities befinden sich lokale Referenzen auf die Firebase Datenbank. AuÃŸerdem wird eine Liste mit â€šSupermarktâ€˜ bzw. â€šProduktâ€˜ Objekten angelegt. Beim HinzufÃ¼gen von Produkt Objekten werden diese in eine in eine â€šShoppingItemâ€˜ Wrapper Klasse gespeichert. EditText Boxen erhalten Input vom Benutzer und beim Klick auf â€šHinzufÃ¼genâ€˜ wird ein neues Objekt an diese Liste angehÃ¤ngt.

```java
ArrayList<ShoppingItem> productList = new ArrayList<>();

productList.add(new ShoppingItem(productid.getText().toString()â€¦));

```

Dann wird die Liste von â€šProduktâ€˜ Objekten in eine HashMap eingelesen um damit in der Datenbank eine Struktur zu erzeugen. Die Datenbank wird mit updateChildren() aktualisiert. Mit dieser Funktion wird auch sichergestellt, dass keine bereits existierenden EintrÃ¤ge Ã¼berschrieben werden. Es wird ein Wurzel(root)-Knoten mit der Bezeichnung â€šproductsâ€˜ erstellt. Die â€šproductListâ€˜ wird in einzelne EintrÃ¤ge unter diesem Wurzel-Knoten Ã¼bersetzt. â€šProduktâ€˜ JSON Objekte sind SchlÃ¼ÃŸel-Werte Paaren, die die Attribute mit aktuellen Werten enthalten.

```java

Map<String, Object> cartItemsMap = new HashMap<>();
cartItemsMap.put("products", productList);
mDatabase.updateChildren(cartItemsMap);

```

Um das Lesen zu implementieren wird ein â€šValueEventListenerâ€˜ fÃ¼r die Datenbank Referenz erzeugt. Wenn sich ein Wert Ã¤ndert wird eine Callback-Methode OnDataChange() aufgerufen. Es wird ein Snapshot erstellt, der den Zustand der Datenbank zum Zeitpunkt der WertÃ¤nderung enthÃ¤lt.

```java

mDatabase.addListenerForSingleValueEvent(new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        IndividualProduct.this.dataSnapshot = dataSnapshot;
    }
});

```

Aus diesem Snapshot kÃ¶nnen Werte gelesen werden. Dazu wird der entsprechende SchlÃ¼ÃŸel angegeben. HauptsÃ¤chlich werden Produkte gelesen, die von SupermÃ¤rkten hinzugefÃ¼gt wurden. Diese Produkte werden den Benutzern in der App angezeigt. Es ist oft notwendig gelesene Werte in geeignete Variablen-Typen umzuwandeln. Zahlen werden in Integer gecastet und Zeichenketten in Strings. Mit getValue() wird der Wert gelesen, der zum jeweiligem SchlÃ¼ssel gehÃ¶rt. Es kÃ¶nnen auch alle EintrÃ¤ge eines Wurzel-Knotens mit getChildren() ausgegeben werden. Das ist nÃ¼tzlich um zum Beispiel alle Produkte zu lesen.

```java

String productID = snapShot.child("productID").getValue().toString();
dataSnapshot.child("products").getChildren())

```

Implementierung in Android Studio

Automatisiertes Testen
Um die FunktionalitÃ¤t der App wÃ¤hrend der Entwicklung zu gewÃ¤hrleisten werden Tests durchgefÃ¼hrt. Dies geschieht manuell und auch automatisch. Manuelles Testen erfordert einen hohen Zeitaufwand. Es mÃ¼ssen Ergebnisse und das Verhalten der App Ã¼berprÃ¼ft und die Resultate Ã¼bersichtlich aufgeschrieben werden. DafÃ¼r hat der Entwickler die MÃ¶glichkeit Einsichten durch Beobachten und Ausprobieren zu erhalten. Fehler fallen dem menschlichen Tester schnell auf. DafÃ¼r kann man nicht bei jeder Weiterentwicklung des Codes die komplette App manuell testen. Deswegen verlassen wir uns auf automatisierte Tests. Das AusfÃ¼hren erfolgt automatisch. Entweder ausgelÃ¶st durch das Einbinden in eine Continous-Integration Umgebung wie Jenkins oder auf Wunsch des Entwicklers. Es gibt zwei Arten um Android-Applikation zu testen. Entweder lokal auf der JavaVirtualMachine oder auf einem echten Android-GerÃ¤t. Wir haben uns auf lokale Tests mit jUnit konzentriert. Damit werden die elementarsten Komponenten unserer App Ã¼berprÃ¼ft.

Unit Testing
Mit Unit-Tests werden einzelne Klassen isoliert voneinander getestet. Wir Ã¼berprÃ¼fen ob Klassen nach Initialisierung den gewÃ¼nschten Zustand haben und ob Methoden richtige RÃ¼ckgabewerte liefern. Dazu muss in der app/build.gradle die jUnit AbhÃ¤ngigkeit eingetragen werden:

```java

dependencies {
        testImplementation 'junit:junit:4.12'
}
```

Beim Anlegen des Projektes wurde ein Ordner angelegt, der die Unit-Test Klassen enthÃ¤lt. Dieser befindet sich unter â€šapp/src/test/javaâ€˜ in der Projekt Verzeichnisstruktur. Die zu testenden Klassen mÃ¼ssen in der Unit-Test Klasse importiert werden.

```java

import com.example.retten.retten.model.Addresse;

```

In der Test-Klasse muss zuerst eine Instanz des Objektes erzeugt werden. Dann werden Annahmen Ã¼ber die Resultate von Methodenaufrufen der zu testenden Klasse getroffen. Stimmen diese mit den RÃ¼ckgabewerten Ã¼berein ist der Test erfolgreich. Es wird Ã¼berprÃ¼ft ob Objekte Daten korrekt speichern und zurÃ¼ckgeben.

```java

test_addresse = new Addresse("StraÃŸe","Hausnummer",â€¦);
String result = test_addresse.get_streetName();
assertEquals(result, "StraÃŸe");

```

Im Android Studio lassen sich mit einem Rechtsklick auf die Test-Klasse mit â€šRun UnitTestâ€˜ die Tests ausfÃ¼hren. In der Console werden die Ergebnisse angezeigt.