Höhere Technische Bundeslehranstalt Wien 3, Rennweg
IT & Mechatronik

HTL Rennweg :: Rennweg 89b
A-1030 Wien :: Tel +43 1 24215-10 :: Fax DW 18

# Diplomarbeit

## Capentory
## Digitalisierung der Schulinventur

ausgeführt an der
Höheren Abteilung für Informationstechnologie/Ausbildungsschwerpunkt
Netzwerktechnik
der Höheren Technischen Lehranstalt Wien 3 Rennweg

im Schuljahr 2019/2020

durch

**Josip Domazet**
**Mathias Möller**
**Hannes Weiss**

unter der Anleitung von

DI Clemens Kussbach
DI August Hörandl

Wien, 15. März 2020

# Kurzfassung

Aktuell ist eine Inventur an der HTL Rennweg überaus mühsam, da es dafür dreier separater Listen bedarf. Die erste Liste stammt direkt aus dem SAP-System und beinhaltet infolgedessen Informationen über alle Gegenstände in der Organisation. Diese Liste wird fortan als „Primäre Liste“ bezeichnet. Das SAP-System ist eine Datenbank, die durch gesetzliche Vorgaben von unserer Schule verwendet werden muss.

Bei der zweiten und dritten Liste handelt es sich um interne Listen, die sich auf die IT-bezogenen Gegenstände in der Organisation beschränken. Diese Listen werden fortan als „Sekundäre Liste“ bzw. „Tertiäre Liste“ bezeichnet. Logisch betrachtet handelt es sich bei der sekundären und tertiären Liste um eine Teilmenge der primären Liste. Allerdings sind diese Listen nicht synchron zueinander und führen daher zahlreiche Komplikationen herbei. Das vorliegende Projekt soll die Schulinventarisierung erheblich erleichtern, indem es die erwähnten Listen sinnvoll vereint.

Außerdem soll der Inventurvorgang selbst durch eine mobile Android-Applikation vereinfacht werden. Ausgedruckte Listen, die bisher dafür zum Einsatz kommen, sollen durch unsere Applikation ersetzt werden. Durch das Scannen von Barcodes können Gegenstände auf der App validiert werden - mit dem angenehmen Nebeneffekt, dass der Inventurvorgang massiv beschleunigt wird. Die Barcodes werden vom SAP-System für jeden Gegenstand generiert und sind in Form eines Aufklebers an den Gegenständen angebracht. Da alle Änderungen protokolliert werden, ist ein genauer Verlauf und damit eine genaue Zuordenbarkeit zu im Serversystem registrierten Benutzern möglich.

# Abstract

The current inventory process at our institution is extremly tedious due to the fact that three independent lists serve as data source for all items. The first list originates from the SAP-System and therefore, contains information about all items in our organization. Henceforth, this list will be labeled as „Primary Source“. The SAP-System is a database that our school is required to use by law.

The second and third lists are unofficial lists for internal usage that are limited to the IT-related items at our organization. Henceforth, these lists will be labeled as „Secondary Source“ and „Tertiary Source“ respectively. From a logical point of view, the secondary and tertiary sources are a subset of the primary source. However, these sources are not in sync with each other and are thus, the cause of countless complications. The aim of this diploma thesis is to simplify the inventory process tremendously by unifying said sources in a reasonable manner.

Furthermore, the current inventory process itself is to be simplified by an mobile Android app. Currently used printed lists are to be replaced by said app. By scanning barcodes one can validate items on the application - with the pleasant side effect of speeding up the inventory process considerably. Mentioned barcodes are provided for each item by the SAP-System and are physically attached to the items as stickers. Due to the fact that all changes are being logged, there is an exact history and accountability to users that are registered in the server system.

# Ehrenwörtliche Erklärung

Ich erkläre an Eides statt, dass ich die individuelle Themenstellung selbstständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Wien, am 15. März 2020

Josip Domazet

Mathias Möller

Hannes Weiss

# Inhaltsverzeichnis

# Tabellenverzeichnis

kann entfallen falls (fast) leer

# Abbildungsverzeichnis

# 1 Ziele

## 1.1 Hauptziele

1. **Online Inventurauflistung**

   Eine zentrale Auflistung aller in der vorliegenden Organisation enthaltenen und registrierten Gegenstände ist per Webbrowser erreichbar und verwaltbar. Dabei kann nach Eigenschaften der Gegenstände gesucht, gefiltert und sortiert werden. Außerdem wird dabei zwischen Einträgen, die aus der primären Liste und jenen, die aus der sekundären Liste stammen, unterschieden. Einträge der beiden Quellen sind miteinander, sowie mit einem zugehörigen Raum verlinkt. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Das "Capentory" Modul"**.

   a) **Verlauf auf Gegenstandsbasis**

      Ein Gegenstand verfügt in der Datenbank über eine Geschichte, die beschreibt, wie sich der Gegenstand durch die verschiedenen Inventuren verändert hat. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Änderungsverlauf"**.

   b) **Inventurverlauf**

      Der Server speichert jede vorgenommene Inventur. Somit ist eine Versionsgeschichte an Inventuren abrufbar. Falls eine Inventur gelöscht wird, wird diese nicht unmittelbar verworfen, sondern vorerst in ein Archiv verschoben. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Das "Stocktaking" Modul"**.

   c) **Entity-Relationship-Modell**

      Die Datenbankstruktur inkl. Verlaufsfunktion ist durch ein Entity-Relationship-Modell definiert. Vor der Implementierung wurde die Datenbank skizziert und vorerst auf Papier modelliert. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Das "Stocktaking" Modul"**.

2. **Datenimport/-export**

Daten von vorhandenen Inventarlisten (Primär und Sekundär) sind in die Datenbank importierbar, wobei Fehler des importierten Datensatzes erkannt und angezeigt werden. Das Importformat ist dabei.csv und .xlsx. Die in der zentralen Datenbank enthaltenen Daten sind in das .csv-, sowie das .xlsx-Format exportierbar. Dabei richtet sich die Formatierung der exportierten Daten an jene der importierten Daten. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Datenimport"**.

3. **Statussystem**

   Ein Statussystem zeigt Diskrepanzen (etwa Gegenstände, die nur in einer der beiden Importquellen existieren, siehe RE-M 2) zwischen durchgeführten Inventuren und den importierten Daten in Form eines speziellen Status-Feldes auf. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Das HTLItem Modell"**.

4. **Infrastruktur**

   Das System ist vorbereitet, aus dem Schulnetz per HTTPS erreichbar zu sein. Die nötigen Schritte zur Inbetriebnahme der vorliegenden Infrastruktur sind dokumentiert. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Technische Umsetzung: Infrastruktur"**.

   a) **Betriebsbereiter Server**

      Ein Server inkl. Betriebssystem ist betriebsbereit. Die dazu benötigten Ressourcen sind erfasst und stehen dem Serversystem – insofern hardwaretechnisch realisierbar – zur Verfügung. Dieses Ziel wird in folgenden Kapiteln näher erläutert: **"Anschaffung des Servers"**, **"Wahl des Betriebsystems"** und **"Installation des Betriebsystems"**.

   b) **Applikationskonfiguration**

      Der einsatzfähige Server ist mit einer Konfiguration für den Produktivbetrieb ausgestattet. Die Konnektivität innerhalb des Schulnetzes ist getestet, sofern das Netzwerk dazu fähig ist. Dieses Ziel wird in folgenden Kapiteln näher erläutert: **"Konfiguration der Netzwerkschnittstellen"** und **"Installation der notwendigen Applikationen"**.

   c) **Virtualisierung**

      Die verschiedenen Komponenten (i.e. Datenbank, Webserver) des Servers sind containervirtualisiert. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Produktivbetrieb der Applikation"**.

   d) **Server-Dokumentation**

Die Maßnahmen zur Installation und Inbetriebnahme des Gesamtsystems ist dokumentiert. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Verfassen einer Serverdokumentation"**.

5. **Inventur per Android-App**

   Ein Administrator ist in der Lage mit einer mobilen Android-Applikation eine Inventur durchzuführen. Dabei scannt dieser den Barcode eines Gegenstandes oder gibt den Code manuell ein und hat dann die Möglichkeit diesen Gegenstand zu validieren. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Die Fragments"**.

   a) **Scanvarianten**

      Gegenstände werden per integriertem Zebra-Scanner – insofern vorhanden – gescannt. Alternativ wird dem Benutzer die Möglichkeit angeboten, den Barcode mit der Handykamera – insofern vorhanden – zu scannen. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Der Scan"**.

   b) **Kommunikationsschnittstelle am Server**

      Die App verfügt über die Fähigkeit, den Server anzusprechen und die benötigten Inventurinformationen zu erhalten. Zusätzlich können auch Inventurveränderungsvorschläge über diese Weise vorgenommen werden. Dieses Ziel wird in folgenden Kapiteln näher erläutert: **"Die Modelle"** und **"Der ValidationEntry"**.

   c) **Inventurverhalten auf Raumbasis**

      Der Benutzer arbeitet im Rahmen einer Inventur eine Liste an Räumen ab, die ihm von der mobilen Applikation angezeigt werden. Hierbei scannt und validiert er die vorhandenen Gegenstände, die ihm angezeigt werden, sowie etwaige unbekannte/unerwartete Gegenstände (denen im Anschluss entsprechende Einträge zugewiesen werden). Dieses Ziel wird in folgenden Kapiteln näher erläutert: **"Die Fragments"** und **"Sonderfälle auf der App"**.

6. **Serversicherung**

   Der Server, der die oben beschriebenen Features zur Verfügung stellt, ist selbst von verschiedenen Angriffsszenarien, die in der Schule realistischerweise stattfinden könnten, geschützt, insofern dies technisch umsetzbar ist. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Absicherung der virtuellen Maschine"**.

## 1.2 Optionale Ziele

1. **Texterkennung** Für den Fall, dass der Barcode beschädigt sein sollte, verfügt die Applikation zusätzlich über die Fähigkeit, Text zu scannen. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Textscan"**.

2. **Bilder am Server** Der Server speichert Bilder zu Gegenständen. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Speichern von Bildern und Anhängen"**.

3. **Bilder in der App**

   Die App zeigt Bilder, die am Server Gegenständen zugeordnet wurden, an. Außerdem kann der Benutzer per App Bilder bestimmten Gegenständen zuordnen und diese Bilder am Server hochladen. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"AttachmentsFragment"**.

4. **Benutzerdefinierte Felder**

   Der Server speichert auf Gegenstandsbasis benutzerdefinierte Felder, die wiederum von der App angezeigt werden können. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"DetailedItemFragment"**.

5. **Anhänge**

   Anhänge können Gegenständen zugewiesen werden. Die Speicherung erfolgt hierbei so, dass ein Anhang mehreren Gegenständen zugewiesen werden kann und somit keine Duplikate gespeichert werden müssen. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Speichern von Bildern und Anhängen"**.

6. **Ralph-Pull-Request**

   Es wird versucht, die entwickelte Inventurfunktion in den offiziell Ralph-Code einzuführen. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Pull-Request"**.

7. **Automatische Importkorrektur**

   Der Benutzer hat die Möglichkeit nach dem Import vordefinierte Änderungen anzuwenden. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Datenimport"**.

8. **Servermonitoring**

   Der Server ist für Monitoring vorbereitet. Dieses Ziel wird in folgendem Kapitel näher erläutert: **"Überwachung des Netzwerks"**.

# 2 Einführung in die App-Architektur

Wie im vorherigen Kapitel angesprochen, ist das Ziel der Diplomarbeit, eine App zu entwickeln, mit der man in der Lage ist, eine Inventur durchzuführen. Doch wieso eine App und wieso überhaupt Android? Um diese Frage zu klären, muss man zwischen zwei Begriffen unterscheiden [41]:

- Native App
- Web-App

## 2.1 Nativ vs. Web

Unter einer nativen App versteht man eine App, die für ein bestimmtes Betriebssystem geschrieben wurde [7]. Die Definition ist allerdings nicht ganz eindeutig, da Frameworks wie Flutter und Xamarin nativer Funktionalität sehr nahekommen, obwohl sie mehrere verschiedene Betriebssysteme unterstützten. Eine Web-App hingegen basiert auf HTML und wird per Browser aufgerufen. Sie stellt nichts anderes als eine für mobile Geräte optimierte Website dar.

Am Markt zeichnet sich in letzter Zeit ein klarer Trend ab – native Apps sterben allmählich aus [5] und werden durch mobile Webseiten ersetzt. Das ist dadurch erklärbar, dass mobile Webseiten immer funktionsreicher werden. Mittlerweile haben Web-Apps nur noch geringfügig weniger Möglichkeiten als native Apps. Aus wirtschaftlicher Betrachtungsweise amortisieren sich Web-Apps de facto um einiges schneller und sind auch dementsprechend lukrativer.

## 2.2 Begründung: Native App

Das Projektteam hat sich dennoch für eine native App entschieden. Um diese Entscheidung nachvollziehen zu können, ist ein tieferer Einblick in den gegebenen Use-Case erforderlich.

Das Ziel ist es nicht, möglichst viele Downloads im Play Store zu erzielen oder etwaige Marketingmaßnahmen zu setzen. Es soll stattdessen mit den gegebenen Ressourcen eine Inventurlösung entwickelt werden, die die bestmögliche Lösung für unsere Schule darstellt. Eine native App wird eine Web-App immer hinsichtlich Qualität und User Experience klar übertreffen. Im vorliegenden Fall wäre es sicherlich möglich, eine Inventur mittels Web-App durchzuführen, allerdings würde diese vor allem in den Bereichen Performanz und Verlässlichkeit Mängel aufweisen. Diese zwei Bereiche stellen genau die zwei Problembereiche dar, die es mit der vorliegenden Gesamtlösung bestmöglich zu optimieren gilt. Des Weiteren bieten sich native Apps ebenfalls für komplexe Projekte an, da Web-Apps aktuell noch nicht in der Lage sind, komplexe Aufgabenstellungen mit vergleichbar geringem Aufwand zu inkorporieren [80]. Die vorliegende Arbeit ist dabei bereits allein aufgrund der in Betracht zu ziehenden Sonderfälle als komplexe Aufgabenstellung einzustufen.

Unter Berücksichtigung dieser Gesichtspunkte wurde also der Entschluss gefasst, eine native Applikation zu entwickeln, da diese ein insgesamt besseres Produkt darstellen wird. Es sei gesagt, dass es auch Hybride Apps gibt. Diese sind jedoch einer nativen App in denselben Aspekten wie eine Web-App klar unterlegen.

### 2.2.1 Auswahl der nativen Technologie

Folgende nativen Alternativen waren zu vergleichen:

- Flutter [42]
- Xamarin [82]
- Native IOS
- Native Android (Java/Kotlin)

Flutter ist ein von Google entwickeltes Framework, dass eine gemeinsame Codebasis für Android und IOS anbietet. Eine gemeinsame Codebasis wird oftmals unter dem Begriff `cross-platform` zusammengefasst und bedeutet, dass man eine mit Flutter entwickelte native App sowohl mit Android-Geräten als auch mit IOS-Geräten verwenden kann. Flutter ist eine relativ neue Plattform – das erste stabile Release wurde erst im Dezember 2018 veröffentlicht [40]. Außerdem verwendet Flutter die Programmiersprache `Dart`, die Java ähnelt. Diese Umstände sind ein Segen und Fluch zugleich. Flutter wird in Zukunft sicherlich weiterhin an Popularität zulegen, allerdings ist die Anzahl an verfügbarer Dokumentation für das junge Flutter im Vergleich zu den anderen Optionen immer noch weitaus geringer.

Xamarin ist ebenfalls ein cross-platform Framework, das jedoch in C# geschrieben wird und älter (und damit bewährter) als Flutter ist. Weiters macht Xamarin von der proprietären .NET-Platform Gebrauch. Infolgedessen haben alle Xamarin-Apps Zugriff auf ein umfassendes Repertoire von .NET-Libraries [83]. Da Xamarin und .NET

beide zu Microsoft gehören, ist eine leichtere Azure-Integration oftmals ein Argument, das von offzielen Quellen verwendet wird. Xamarin wird - anders als die restlichen Optionen - bevorzugterweise in Visual Studio entwickelt [84].

Native IOS wird nur der Vollständigkeit halber aufgelistet, stellte allerdings zu keinem Zeitpunkt eine wirkliche Alternative dar, weil IOS-Geräte einige Eigenschaften besitzen, die für eine Inventur nicht optimal sind (z.B. die Akkukapazität). Außerdem haben in etwa nur 20% aller Geräte [4] IOS als Betriebssystem und die Entwicklung einer IOS-App wird durch strenge Voraussetzungen äußerst unattraktiv gemacht. So kann man beispielsweise nur auf einem Apple-Gerät IOS-Apps entwickeln.

#### 2.2.1.1 Begründung: Natives Android (Java)

Die Entscheidung ist schlussendlich auf natives Android (Java) gefallen. Es mag zwar vielleicht nicht die innovativste Entscheidung sein, stellt aber aus folgenden Gründen die bewährteste und risikoloseste Option dar:

- Natives Android ist eine allbekannte und weit etablierte Lösung. Die Wahrscheinlichkeit, dass die Unterstützung durch Google eingestellt wird, ist also äußerst gering.
- Die App wird in den nächsten Jahren immer noch am Stand der Technik sein.
- Natives Android hat mit großem Abstand die umfassendste Dokumentation.
- An der Schule wird Java unterrichtet. Das macht somit eventuelle Modifikationen nach Projektabschluss durch andere Schüler viel einfacher möglich.
- Dadurch, dass Kotlin erst seit 2019 [50] offiziell die von Google bevorzugte Sprache ist, sind die meisten Tutorials immer noch in Java.
- Sehr viele Unternehmen haben viele aktive Java-Entwickler. Dadurch wird die App attraktiver, da die Unternehmensmitarbeiter (von z.B. allegro) keine neue Sprache lernen müssen, um Anpassungen durchzuführen.
- Das Projektteam hat im Rahmen eines Praktikums bereits Erfahrungen mit nativem Java gesammelt.

Aus den Projektzielen hat sich in Absprache mit den Betreuern ergeben, dass die App nicht auf jedem “Steinzeitgerät” zu funktionieren hat. Das minimale API-Level der App ist daher 21 - auch bekannt als Android 5.0 ‘Lollipop’. Android 4.0 hat sehr viele nützliche Libraries hervorgebracht. So zum Beispiel die `Mobile Vision API` von Google, dank derer man in der Lage ist, Barcodes in akzeptabler Zeit mit der Kamera des Geräts zu scannen. Die Wahl ist auf 5.0 gefallen, da somit ein Puffer zur Verfügung steht und in etwa 90% aller Android-Geräte ohnehin auf 5.0 oder einer neueren Version laufen [3].

## 2.3 Einführung zu nativem Java

Um eine Basis für die folgenden Kapitel zu schaffen, werden hier die Basics der Android-Entwicklung mit nativem Java näher beschrieben. Das Layout einer App wird in XML Files gespeichert, während das wirkliche Programmieren mit Java erfolgt.

### 2.3.1 Single-Activity-App

Als Einstiegspunkt in eine App dient eine sogenannte `Activity`. Eine Activity ist eine normale Java-Klasse, der durch Vererbung UI-Funktionen verleiht werden.

Bis vor kurzem war es üblich, dass eine App mehrere Activities hat. Das wird bei den Benutzern dadurch bemerkbar, dass die App z.B. bei einem Tastendruck ein weiteres Fenster öffnet, das das bisherige überdeckt. Das neue Fenster ist eine eigene Activity. Google hat sich nun offiziell für sogenannte Single-Activities ausgesprochen [46]. Das heißt, dass es nur eine Activity und mehrere `Fragments` gibt. Ein Fragment ist eine Teilmenge des UIs bzw. einer Activity. Anstatt jetzt beim Tastendruck eine neue Activity zu starten, wird einfach das aktuelle Fragment ausgetauscht. Dadurch, dass keine neuen Fenster geöffnet werden, ist die User Expierence (UX) um ein Vielfaches besser – die Performanz leidet nur minimal darunter. Die vorliegende App ist aus diesen Gründen ebenfalls eine Single-Activity-App.

### 2.3.2 Separation of Concerns

In Android ist es eine äußerst schlechte Idee, sämtliche Logik in einer Activity oder einem Fragment zu implementieren. Das softwaretechnische Prinzip `Separation of Concerns (SoC)` hat unter Android einen besonderen Stellenwert. Dieses Prinzip beschreibt im Wesentlichen, dass eine Klasse nur einer Aufgabe dienen sollte. Falls eine Klasse mehrere Aufgaben erfüllt, so muss diese auf mehrere logische Komponenten aufgeteilt werden. Beispiel: Eine Activity hat immer die Verantwortung, die Kommunikation zwischen UI und Benutzer abzuwickeln. Bad Practice wäre es, wenn jene Activity ebenfalls dafür verantwortlich ist, Daten von einem Server abzurufen. Das Prinzip verfolgt das Ziel, die `God Activity Architecture (GAA)` möglichst zu vermeiden [6]. Eine God-Activity ist unter Android eine Activity, die die komplette Business-Logic beinhaltet und SoP in jeglicher Hinsicht widerspricht. God-Activities gilt es dringlichst zu vermeiden, da sie folgende Nachteile mit sich bringen:

- Refactoring wird kompliziert
- Wartung und Dokumentation werden äußerst schwierig
- Automatisiertes Testing (z.B. Unit-Testing) wird nahezu unmöglich gemacht
- Größere Bug-Anfälligkeit

- Im Bezug auf Android gibt es oftmals massive Probleme mit dem `Lifecycle` einer Activity - da eine Activity und ihre Daten schnell vernichtet werden können (z.B. wenn der Benutzer sein Gerät rotiert und das Gerät den Bildschirmmodus wechselt)

God-Activities sind ein typisches Beispiel für Spaghetticode. Es bedarf also einer wohlüberlegten und strukturierten Architektur, um diese Probleme zu unterbinden. Im nächsten Kapitel wird dementsprechend die Architektur der App im Detail erklärt.

# 3 MVVM - Die App-Architektur

Als Reaktion auf eine Vielzahl von Apps, die Probleme mit God-Activites aufwiesen, hat Google Libraries veröffentlicht, die klar auf eine MVVM-Architektur abzielen [56]. Daher fiel die Wahl der App-Architektur auf MVVM.

## 3.1 Designgrundlagen von MVVM

MVVM steht für Model–view–viewmodel [81]. Wie man am Namen bereits erkennt, gilt es zwischen drei Komponenten/Ebenen zu unterscheiden [55].

**Model** Model beschreibt die Ebene der Daten und wird daher oftmals auch als Datenzugriffsschicht bezeichnet. Diese Ebene beinhaltet so viel Anwendungslogik wie möglich.

**View** View beschreibt die graphische Ebene und umfasst daher das GUI. Diese Ebene soll so wenig Logik wie möglich beinhalten.

**ViewModel** Das ViewModel dient als Bindeglied zwischen dem Model und der View. Die Logik der View wird in diese Ebene hinaufverschoben.

## 3.2 MVVM in Android

Mit der Einführung der Architecture Components hat Google Android-Entwicklern eine Vielzahl an Libraries zur Verfügung gestellt, um MVVM leichter in Android implementieren zu können [1]. Die konkrete Implementierung in Android ist in der Abbildung ersichtlich.

In dem vorliegenden Fall ist unser `Fragment` die `View`, das `Repository` das `Model` und das `ViewModel` ist in Android namensgleich.

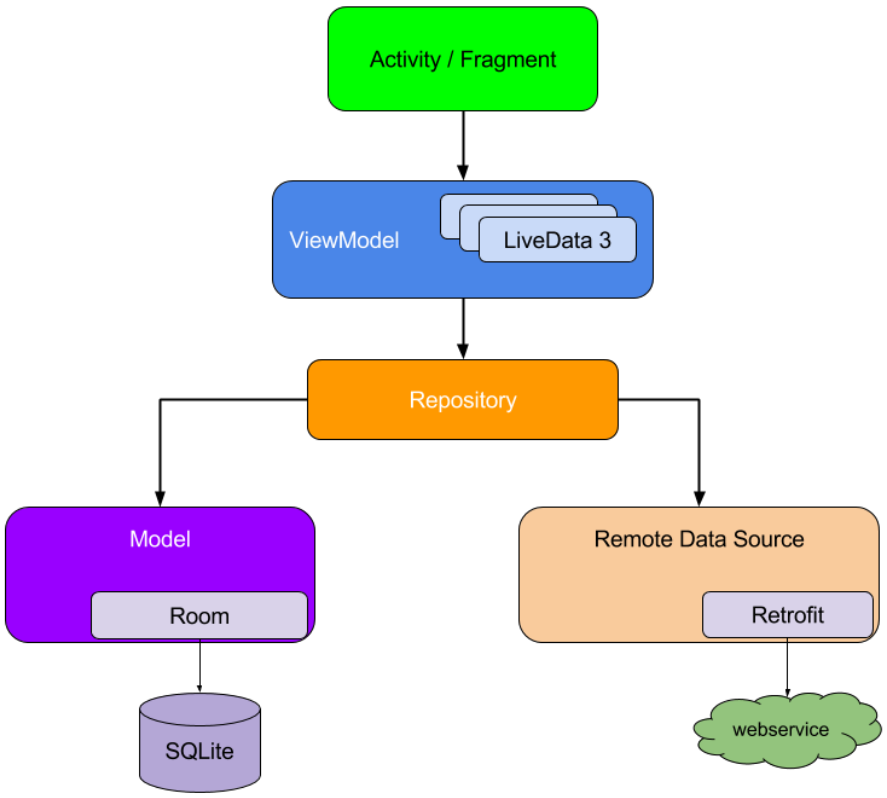



Abbildung 3.1: MVVM in Android nach Google [56]

### 3.2.1 Das Repository im Detail

Wie in der Grafik veranschaulicht, ist das Repository dafür zuständig, Daten vom Server anzufordern. Im vorliegenden Fall besteht kein Grund, eine Datenbank am Client zu führen. Damit fällt dieser Aspekt der Grafik - der in violetter Farbe gehalten ist - für die vorliegende Diplomarbeit weg. Die Aufgabe des Repositories ist es also immer Daten vom Server anzufordern.

#### 3.2.1.1 JsonRequest

Die Kommunikation zwischen dem Client und der App findet ausschließlich im `JSON`-Format statt. JSON ist ein text-basiertes und kompaktes Datenaustauschformat. Android bietet Entwicklern eine Out-of-the-box Netzwerklibrary namens `Volley` an, mithilfe derer man unter anderem JSON-Anfragen verarbeiten kann [2]. Da diese für die vorliegenden Zwecke nicht komplett geeignet war, hat das Diplomarbeitsteam die gegebene Library durch den Einsatz von Vererbung und einer Wrapper-Klasse modifiziert. Die Library wurde in folgenden Punkten angepasst:

- Im Falle eines Fehlers wird die Anfrage wiederholt (Ausnahme: Zeitüberschreitungsfehler). Die maximale Anzahl an Wiederholungen ist limitiert.
- Die maximale Timeout-Dauer wurde erhöht.
- Leere Antworten werden von der App als valide Antwort behandelt und können ohne Fehler verarbeitet werden.
- Im Header der Anfrage wird der Content-Type der Anfrage auf JSON festgelegt.
- Im Header wird das zur Authentikation notwendige API-Token mitgeschickt. Die Authentifizierung ist über einen Parameter deaktivierbar.
- Im Header wird die Systemsprache des Clients als standardisierter ISO-639-Code mitgesendet. Der Server passt seine Antwort auf die verwendete Sprache an [48]. Die Bezeichnung der Felder, die dem Benutzer auf Gegenstandsbasis angezeigt werden, ist beispielsweise abhängig von der Systemsprache.
- Zum Zeitpunkt der Anfrage ist nicht bekannt, ob die Antwort als JSONArray oder als JSONObject erfolgen wird. Da das Backend abhängig von der Anfrage sowohl mit einem JSONArray als auch einem JSONObject antworten kann, ist der Rückgabewert am Client immer ein String. Die Umwandlung erfolgt erst im Repository.

Diese Unterscheidung zwischen JSONArray und JSONObject ist notwendig, da Android zwei Möglichkeiten anbietet, JSON als Objekt zu speichern:

Direkt als JSONArray:

```
JSONArray jsonArray = new JSONArray(payload);
```

Direkt als JSONObject:

```
JSONObject jsonObject = new JSONObject(payload);
```

Beide Optionen haben einen Konstruktor, der einen String akzeptiert (im obigen Beispiel ist dies der String `payload`). Das Backend könnte - natürlich in Abhängigkeit von der ursprünglichen Anfrage - folgende (vereinfachte) Antworten senden:

Ein JSONArray (wenn die Anfrage beispielsweise eine Gegenstandsliste verlangt):

```
[
  {
    "id": 1,
    "item": "PC"
  },
  {
    "id": 2,
    "item": "Schrank"
  }
]
```

Ein JSONObject (wenn die Anfrage beispielsweise einen spezifischen Gegenstand verlangt):

```
{
  "id": 1,
  "item": "PC"
}
```

Wenn das Backend ein JSONArray sendet und man versucht, jenes als JSONObject zu speichern, kommt es zu einem Fehler:

```
// Dieser Code entspricht keiner korrekten Java-Syntax!
// Antwort vom Backend
String payload =   "[
                        {
                            "id": 1,
                            "item": "PC"
                        },
                        {
                            "id": 2,
                            "item": "Schrank"
                        }
                   ]";

// Die Umwandlung eines JSONArray zu einem JSONObjects
```

```
// wirft eine Exception.
JSONObject jsonObject = new JSONObject(payload);
```

Aus diesem Grund kann die Umwandlung der Backend-Antwort erst im Repository erfolgen. Dies führt zu keinen Perfomance-Problemen, da die vorgefertigte Android Library den String zwar zu einem früheren Zeitpunkt aber auf dieselbe Weise umwandeln würde. Die Umwandlung eines Strings zu einem JSONArray oder einem JSONObject entspricht nur dem Bruchteil der Zeitdauer, die die Backend-Antwort benötigt, um am Client anzukommen. Selbst bei größeren Strings bemerkt der Benutzer daher keinerlei Unterschied.

#### 3.2.1.2 JsonRequest - Beispiel

Eine Anfrage wird nie direkt, sondern immer über einen Wrapper ausgeführt. Der Konstruktor ist wie folgt aufgebaut:

- `Context context`: Ist eine Schnittstelle, die globale Information über die App-Umgebung zur Verfügung stellt [12] und von Android zur Verfügung gestellt wird. Jede UI-Komponente (z.B. Textfelder, Buttons, Fragments, etc.) verfügt über einen Context. Ein besonderer Context ist der globale Application-Context. Dieser ist einzigartig und ist ein `Singleton`. Ein Singleton bedeutet, dass von einer Klasse nur ein (globales) Objekt besteht [71].
- `int method`: Ist die HTTP-Methode. Die App verwendet `GET`, `OPTIONS` und `POST`.
- `String url`: Ist die Seite, die eine JSON-Antwort liefern soll.
- `@Nullable String requestBody`: Eventuelle Parameter, die an den Server gesendet werden sollen. Dieser Paramter ist für einen `POST`-Request wichtig.
- `NetworkSuccessHandler successHandler`: Funktionales Interface
- `NetworkErrorHandler errorHandler`: Funktionales Interface

Ein Request kann wie folgt ausschauen:

```
RobustJsonRequestExecutioner robustJsonRequestExecutioner =
 new RobustJsonRequestExecutioner(context, Request.Method.GET,
  "https://www.beispiel.org/", null,
  payload -> {
         // TODO: Antwort verarbeiten
     },
  error -> {
         // TODO: Fehler verarbeiten
     });

 robustJsonRequestExecutioner.launchRequest();
```

Wie man sehen kann, sind die letzten beiden Parameter funktionale Interfaces, die dazu dienen, Methoden als Parameter übergeben zu können. Ein Interface mit einer einzigen abstrakten Methode ist als funktionales Interface zu bezeichnen [51]. Da Android Studio Java 8 Language Features unterstützt, verwendet die App mehrheitlich Lambda-Ausdrücke [49]. Lambda-Ausdrücke sind im Wesentlichen dazu da, funktionale Interfaces in vereinfachter Schreibweise verwenden zu können. Da es nur eine einzige abstrakte Methode gibt, kann die Schreibweise simplifiziert werden, weil klar ist von welcher Methode die Rede ist. Damit fallen - wie im obigen Beispiel zu sehen - redundante Informationen wie Rückgabetyp und Methodenkörper vollständig weg. Das obige Beispiel würde ohne Lambda-Ausdrücke wie folgt ausschauen:

```
RobustJsonRequestExecutioner robustJsonRequestExecutioner =
 new RobustJsonRequestExecutioner(context, Request.Method.GET,
 "https://www.beispiel.org/", null,
     new NetworkSuccessHandler() {
         @Override
         public void handleSuccess(String payload) {
             // TODO: Antwort verarbeiten
         }
     },
     new NetworkErrorHandler() {
         @Override
         public void handleError(Exception error) {
             // TODO: Fehler verarbeiten
         }
     });

robustJsonRequestExecutioner.launchRequest();
```

Lambdas sind eine Option, um `Callbacks` in Java zu implementieren. Ein Callback ("Rückruffunktion") ist eine Methode, die einer anderen Methode als Parameter übergeben werden kann. Die soeben erwähnten funktionalen Interfaces sind typische Callbacks [10]. Es gibt zwei Arten von Callbacks:

- Synchrone Callbacks: Die Ausführung der übergebenen Methode erfolgt sofort
- Asynchrone Callbacks: Die Ausführung der übergebenen Methode erfolgt später

In diesem Fall wird die Methode `handleSuccess` aufgerufen, sobald der Client die Antwort erhalten hat. Damit handelt es sich um ein asynchrones Callback.

#### 3.2.1.3 Retrofit

`Retrofit` ist eine weitere Netzwerk-Library (bzw. Libraries), die das Projektteam eingesetzt hat. Retrofit wurde nur zum Senden von Dateien eingesetzt, weil dies mit Volley nur erschwert möglich ist. Das Projektteam hat die von Retrofit zur Verfügung gestellten Libraries nicht wesentlich modifiziert. Da die Anhangfunktion das einzige Einsatzgebiet von Retrofit darstellt, nimmt Volley in der vorliegenden Diplomarbeit die weitaus wichtigere Rolle ein. Volley kann grundsätzlich alles was Retrofit kann und vice versa. Aufgrund dessen, dass Volley jedoch mehr Möglichkeiten zur Zuschneidung auf einen spezifischen Use-Case hat, hat sich das Projektteam für Volley entschieden. Ein Vorteil der jedoch damit verloren geht, ist das automatische Parsing von JSON-Antworten, das zwar in Retrofit implementiert ist, jedoch nicht in Volley [70].

Das Projektteam hat bei dem einzigen Einsatz von Retrofit auf das automatische Parsing verzichtet und stattdessen die bereits bekannte Callback-Logik verwendet.

```
Call<String> call = prepareCall(args);
call.enqueue(new Callback<String>() {
    @Override
    public void onResponse(Call<String> call, Response<String> response) {
        // TODO: Antwort verarbeiten
    }
    @Override
    public void onFailure(Call<String> call, Throwable t) {
        // TODO: Fehler verarbeiten
    }
});
```

In Retrofit werden API-Endpunkte über Interfaces definiert:

```
public interface AttachmentAPI {
    @Multipart
    @POST(SerializerEntry.attachmentUrl)
    Call<String> addFile(@Header("authorization") String auth,
                         @Part MultipartBody.Part file,
                         @Part("description") String description);
}
```

Dies führt zu einem besseren Überblick als bei Volley, da man pro API-Endpunkt des Backends ein Interface hat. Damit ist sofort ersichtlich, mit welchen Backend-Endpunkten ein Repository kommuniziert.

### 3.2.2 Das ViewModel und das Fragment im Detail

Das ViewModel ist eine Klasse, die dafür ausgelegt ist, UI-bezogene Daten lifecycle-aware zu speichern. Ein Fragment durchlebt im Laufe seines Daseins eine Vielzahl an Zuständen/Phasen - man spricht von einem `Lifecycle`. Wenn der Benutzer zum Beispiel sein Gerät rotiert, führt dies dazu, dass das Fragment *zerstört* wird und das Fragment durch erneutes Durchleben alter Zustände wiederaufgebaut wird - dies führt zu einer Zerstörung des aktuellen UIs des Fragments sowie sämtlicher Referenzen, die das Fragment besitzt. Eine Gerätrotierung gehört zur Kategorie der `Configuration Changes` [77]. Der Grund hierfür liegt darin, dass Android das aktuelle Layout ändert, da beispielsweise andere Layouts (XML-Files) für den Landscape-Modus zur Verfügung stehen [11]. In der Literatur wird der Begriff *zerstören* verwendet, da dabei das Callback `onDestroy` in einer Activity aufgerufen wird.

Folgende Probleme können dadurch auftreten:

- Die App stürzt App. Wenn eine Methode ausgeführt wird, die eine Referenz auf ein zerstörtes Objekt hat, kann dies zum Absturz der gesamten App führen.
- Memory Leaks entstehen, da Referenzen auf zerstörte Objekte vom Gargabe Collector nicht freigegeben werden können. In Android wird die Minimierung des Speicherbedarfs der App einzig und allein vom Garbage Collector übernommen. Falls dieser Objekte nicht freigeben kann, führt dies dazu, dass die App immer mehr und mehr Arbeitsspeicher benötigt. Je nach Größe des Memory Leaks kann dies zu kleineren Lags bis zu einem Absturz der App führen.
- Nach einem Configuration Change gehen die aktuellen Daten verloren und der Benutzer muss das Problem selbst lösen.
- Wenn die aktuellen Daten verloren gehen, verhält sich eine App oftmals unvorhersehbar.

#### 3.2.2.1 ViewModel als Lösung

Bei genauerer Betrachtung der Grafik wird ersichtlich, welche Phasen eine Activity bei einer Gerätrotierung durchlebt:

- Activity wird zerstört:
  - `onPause`
  - `onStop`
  - `onDestroy`
- Activity wird wieder aufgebaut:
  - `onCreate`
  - `onStart`

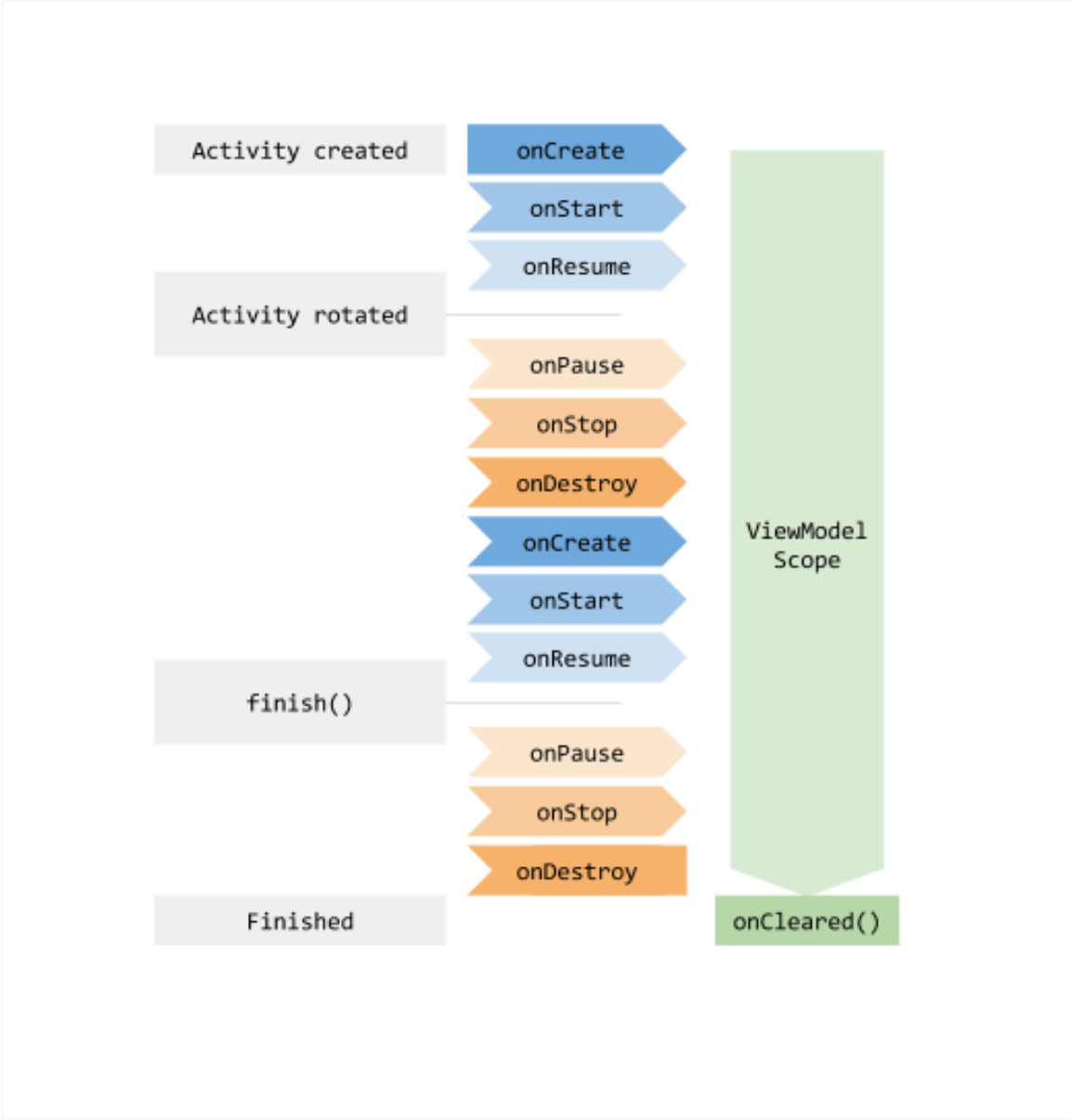



Abbildung 3.2: Zustände einer Activity im Vergleich zu den Zuständen eines ViewModels, Fragments haben einen ähnlichen Lifecycle [43] [77]

– `onResume`

Wie in der Abbildung zu sehen ist, stellt ein ViewModel eine Lösung für diese Probleme dar. Ein ViewModel ist von einem Configuration Changes nicht betroffen und kann dem UI damit stets die aktuellen Daten zur Verfügung stellen. Der gegebene Sachverhalt trifft genauso auf Fragments zu. Diese haben einen leicht veränderten Lifecycle, sind allerdings genauso von Configuration Changes betroffen wie Activities.

Anmerkung: Man kann das Zerstören & Wiederaufbauen von Activities/Fragments manuell blockieren. Dies ist jedoch kein Ersatz für eine wohlüberlegte App-Architektur und führt in den meisten Fällen zu unerwünschten Nebenwirkungen, da man sich nun auch manuell um das Wechseln der Konfiguration (Layouts etc.) kümmern muss und dies weitaus komplizierter ist, als auf ViewModels zu setzen [37].

Folgende Aspekte hat man bei der Verwendung eines ViewModels zu beachten [75]:

- ViewModel sollte bei einem Configuration Changes keinen neuen Request starten, da es bereits über die Daten verfügt. Dies lässt sich mit einer If-Anweisung beheben.
- Referenzen zu Objekten, die an einen Lifecycle gebunden sind, sind ein absolutes NO-GO. Objekte mit Lifecycle haben ein klares Schicksal - wenn ihr Host vernichtet wird, müssen sie ebenfalls vernichtet werden. Folgendes Szenario: Ein ViewModel hat eine `TextView`-Variable. Dreht der Benutzer sein Gerät wird das aktuelle Fragment inklusive TextView vernichtet. Das ViewModel überlebt den Configuration Change und hat nun eine Referenz auf eine invalide TextView. Dies ist ein Memory Leak.
- ViewModel überleben ein Beenden des Prozesses nicht. Wenn das BS aktuell wenig Ressourcen zur Verfügung hat, kann es sein, dass Prozesse beendet werden. Falls man diesen Sonderfall behandeln will, ist dies mit Extra-Aufwand verbunden [76].
- ViewModels sollen nicht zu “God-ViewModels” werden. Das SoC-Prinzip ist anzuwenden.

Wie gelangen die Daten ins UI, wenn das ViewModel keine Referenzen darauf haben darf? Die Antwort lautet `LiveData`.

#### 3.2.2.2 LiveData

LiveData ist eine observierbare Container-Klasse. Observierbar heißt, dass bei Änderungen des enkapsulierten Objektes ein Callback aufgerufen wird. LiveData ist ebenfalls lifecycle-aware. Daher wird LiveData immer nur aktive Komponenten mit Daten versorgen. Eine TextView, die bereits zerstört wurde, erhält dementsprechend auch keine Updates mehr.

Dieses (angepasste) offizielle Beispiel veranschaulicht die Funktionsweise sehr gut [52]. Im Beispiel soll ein Benutzername angezeigt werden:

```
public class NameViewModel extends ViewModel {

    // LiveData-Objekt, das einen String beinhaltet
    private MutableLiveData<String> currentName;

    public LiveData<String> getCurrentName() {
        if (currentName == null) {
            currentName = new MutableLiveData<>();
        }
        // Benutzernamen bekannt geben
        currentName.postValue("Max Mustermann");

        return currentName;
    }

    // Rest des ViewModels...
}
```

Der Unterschied zwischen `MutableLiveData` und `LiveData` besteht darin, dass letzteres nicht veränderbar ist. Mit der `postValue`-Methode kann einer MutableLiveData-Instanz, die ja als Container-Objekt dient, ein neuer Wert zugewiesen werden. Dadurch werden etwaige Callbacks aufgerufen (siehe nächster Code-Ausschnitt). Man sollte bei öffentlichen Methoden immer nur LiveData als Rückgabewert verwenden, damit auf der Ebene der View keine Modifikationen der Daten des ViewModels vorgenommen werden können. Im Realfall stammt LiveData ursprünglich aus dem Repository.

```
public class NameFragment extends Fragment {
 @Override
    public void onViewCreated(@NonNull View view,
    @Nullable Bundle savedInstanceState) {
        ...

        // Mit dieser Anweisung wird ein ViewModel erstellt
        model = new ViewModelProvider(this).get(NameViewModel.class);

        model.getCurrentName().observe(getViewLifecycleOwner(),
         currentName -> {
            // Diese Methode wird bei Änderungen aufgerufen.
            // CurrentName ist ein String.
            // nameTextView ist ein TextFeld,
            // das den aktuellen Benutzernamen anzeigt.
            // Mit .setText(String) kann der angezeigte Text
            // geändert werden.
            nameTextView.setText(currentName);
         });
```

```
}
```

Hier kommt wieder die vorher angesprochene Lambda-Syntax zum Einsatz. Die Methode wird einmal beim erstmaligen Registrieren aufgerufen und wird danach bei jeder weiteren Änderung aufgerufen. Im Callback arbeitet man direkt mit dem eigentlichem Datentypen - in diesem Fall mit einem String -, da LiveData nur ein Container-Objekt ist. Im Fragment befindet sich damit relativ wenig Logik. Das Fragment hört nur auf eventuelle Änderungen und aktualisiert das UI in Abhängigkeit von den Änderungen. Das LiveData-Objekt ist an `getViewLifeycleOwner()` gebunden. Wenn der LifecycleOwner inaktiv wird, werden keine Änderungen mehr entsandt. Man könnte auch `this` als Argument übergeben (`this` wäre in diesem Fall das Fragment, das ebenfalls über einen Lifecycle verfügt). `getViewLifeycleOwner()` hat jedoch den Vorteil, dass der Observer automatisch entfernt wird, sobald der LifecycleOwner zerstört wird.

#### 3.2.2.3 Angepasste LiveData-Klasse

Für den vorliegenden Usecase reichen jedoch die Nutzdaten allein nicht. Es sind weitere Informationen über den Status der Nutzdaten erforderlich. Beispiel: Wenn eine Anfrage zehn Sekunden benötigt, um am Client anzukommen, muss dem Benutzer signalisiert werden, dass er auf das Backend zu warten hat. Das wird mit einer `ProgressBar` realisiert. Man könnte jetzt im ViewModel `LiveData<Boolean> isFetching` verwenden und im Fragment dieses LiveData-Objekt observieren. Falls der aktuelle Wert `true` ist, wird die ProgressBar angezeigt. Falls der Wert auf `false` geändert wird, werden stattdessen die nun zur Verfügung stehenden Nutzdaten angezeigt. Bei mehreren Netzwerkanfragen wird dies bald unübersichtlich, da mehrere LiveData-Objekte vonnöten sind, die aus logischer Perspektive zu einem bereits bestehenden LiveData-Objekt gehören - den Nutzdaten. Daher hat das Diplomarbeitsteam - wie von Google [45] und von einem StackOverflow-Thread [73] empfohlen - die Nutzdaten in einer Wrapper-Klasse enkapsuliert.

```
public class StatusAwareData<T> {

    // Status der Nutzdaten
    @NonNull
    private State status;

    // Nutzdaten
    @Nullable
    private T data;

    // Eventueller Fehler
    @Nullable
    private Throwable error;
```

```
    ...

    @NonNull
    public State getStatus() {
        return status;
    }

    @Nullable
    public T getData() {
        return data;
    }

    @Nullable
    public Throwable getError() {
        return error;
    }

    // Enum, das die legalen Status definiert
    public enum State {
        INITIALIZED,
        SUCCESS,
        ERROR,
        FETCHING
    }
}
```

Diese Wrapper-Klasse speichert Nutzdaten eines generischen Typen. Der Status wird durch eine Finite-state machine in Form eines `Enums` implementiert. In Android sollte man Enums vermeiden, da diese um ein Vielfaches mehr Arbeitsspeicher und persistenten Speicher benötigen als ihre Alternativen. Als Alternative kann man auf Konstanten zurückgreifen. Dieser Code-Ausschnitt stellt das einzige Enum des gesamten Projektes dar. [38]

- `INITIALIZED`: Dieser Status bedeutet, dass das Objekt soeben erstellt wurde. Wird nur bei der erstmaligen Instanziierung verwendet.
- `SUCCESS`: Dieser Status bedeutet, dass die Nutzdaten `data` bereit sind.
- `ERROR`: Dieser Status bedeutet, dass eine Netzwerkanfrage fehlgeschlagen ist (Ob am Client oder am Server spielt keine Rolle). In diesem Fall ist die `error`-Variable gesetzt.
- `FETCHING`: Dieser Status bedeutet, dass auf eine Netzwerkanfrage gewartet wird.

Um diese Wrapper-Klasse elegant benutzen zu können, hat das Projektteam `MutableLiveData` durch Vererbung angepasst:

```
public class StatusAwareLiveData<T>
```

```
    extends MutableLiveData<StatusAwareData<T>> {

    public void postFetching() {
        // Instanziiert StatusAwareData-Objekt mit FECHTING als
        // aktuellen Status.
        // Setzt das StatusAwareData-Objekt anschließend
        // als Wert der LiveData-Instanz.
        postValue(new StatusAwareData<T>().fetching());
    }

    public void postError(Exception exception) {
        // Instanziiert StatusAwareData-Objekt mit ERROR als
        // aktuellen Status.
        // Mit der übergebenenen Exception wird die error-Variable
        // initialisiert.
        // Setzt das StatusAwareData-Objekt anschließend
        // als Wert der LiveData-Instanz.
        postValue(new StatusAwareData<T>().error(exception));
    }

    public void postSuccess(T data) {
        // Instanziiert StatusAwareData-Objekt mit SUCCESS als
        // aktuellen Status.
        // Mit dem übergebenenen Objekt wird die data-Variable
        // initialisiert.
        // Setzt das StatusAwareData-Objekt anschließend
        // als Wert der LiveData-Instanz.
        postValue(new StatusAwareData<T>().success(data));
    }

}
```

Damit entfällt der Bedarf selbst neue StatusAwareData-Objekte zu instanziieren, da diese bereits über die `postFetching`-, `postError`- und `postSuccess`-Methoden - mit korrektem Status - instanziiert werden. Infolgedessen ist StatusAwareData abstrahiert und im ViewModel genügt es, mit den modifizierten LiveData-Instanzen zu arbeiten. Damit ändert sich das vorherige "Beispiel" wie folgt:

```
// Im ViewModel:

// StatusAwareLiveData-Objekt, das einen String mit Status beinhaltet
private StatusAwareLiveData<String> currentName;

public LiveData<StatusAwareData<String>> getCurrentName() {
```

```
    if (currentName == null) {
        currentName = new StatusAwareLiveData<>();
    }
    // Benutzernamen bekannt geben, hier wird eine
    // erfolgreiche Netzwerkanfrage simuliert.
    currentName.postSuccess("Max Mustermann");

    return currentName;
}


// Im Fragment:

model.getCurrentName().observe(getViewLifecycleOwner(),
 statusAwareData -> {
      switch (statusAwareData.getStatus()) {
          case SUCCESS:
              // TODO: Nutzdaten anzeigen
              nameTextView.setText(statusAwareData.getData());
              break;
          case ERROR:
              // TODO: Fehlermeldung anzeigen
              ...
              break;
          case FETCHING:
              // TODO: Ladebalken anzeigen
              ...
              break;
      }
   });
```

### 3.2.3 Konkrete MVVM-Implementierung

Im vorliegenden Anwendungsfall hat ein Fragment immmer mindestens einen Datensatz, der für das Fragment namensgebend ist. Der Room-Screen (also die Anzeige mit einer DropDown zur Raumauswahl) setzt sich beispielsweise aus folgenden Komponenten zusammen:

- Der `RoomsFragment`-Klasse
- Der `fragment_rooms.xml`-Datei, die das UI-Layout definiert
- Der `RoomsViewModel`-Klasse
- Der `RoomsRepository`-Klasse

Das `RoomsRepository` ist dafür verantwortlich, die Raumliste vom Backend anzufordern und in Java-Objekte umzuwandeln. Das `RoomsViewModel` ist dafür verantwortlich, dem Fragment LiveData-Objekte zur Verfügung zu stellen. Das `RoomsFragment` ist dafür verantwortlich, dem Benutzer die Räume anzuzeigen, indem es LiveData observiert. Alternativ zeigt es Fehlermeldungen bzw. einen Ladebalken an.

Bei genauerer Betrachtung wird klar, dass fast jeder Screen dieselbe Aufgabe hat:

- Das Repository fordert Daten vom Backend an und wandelt die JSON-Antwort um.
- Das ViewModel abstrahiert Logik und stellt der View LiveData zur Verfügung.
- Das Fragment zeigt entweder Nutzdaten, eine Fehlermeldung oder einen Ladebalken an.

Hier greift das softwaretechnische Prinzip `Do not repeat yourself (DRY)` [32]. Anstatt `Boilerplate-Code` für jeden einzelnen Screen kopieren zu müssen, hat das Projektteam diese sich wiederholende Logik abstrahiert. Boilerplate-Code sind Code-Abschnitte, die sich immer wieder wiederholen [72]. Wiederholende Logik sollte immer in eine Superklasse abstrahiert werden. Das Projektteam hat demnach drei abstrakte Klassen definiert, die die Menge an Boilerplate-Code signifikant reduzieren:

- `NetworkRepository`
- `NetworkViewModel`
- `NetworkFragment`

Alle Komponenten von Screens, die Netzwerkanforderungen durchführen, erben von diesen drei Klassen. Damit hat das Projektteam folgende Vorteile aggregiert:

- Abstrahierte Fehlerbehandlung
- Abstrahiertes Refreshverhalten
- Abstrahierte Ladeanzeige (durch eine `ProgressBar`)
- Abstrahierter Netzwerkzugriff

Der Refactor, der dies realisierte, war zwar zeitintensiv, hat sich jedoch mittlerweile mehr als rentiert. Das Hinzufügen von neuen Screens benötigt nur mehr einen Bruchteil des ursprünglichen Codes. Infolgedessen wird das Erweitern der App um neue Features enorm erleichtert.

# 4 Das Scannen

Das Scannen ist ein vitaler Aspekt dieser Diplomarbeit. Gegenstände an unserer Organisation sind mit Barcodes ausgestattet. Um die Produktivität maximal zu steigern, muss die App in der Lage sein, diese Barcodes zu erfassen. Wir bieten folgende Varianten zum Scannen an:

- Zebra-Scan
- Kamerascan
- Manuelle Eingabe (für den Fall, dass ein Scan fehlschlägt)

## 4.1 Der Zebra-Scan

Der Sponsor dieser Diplomarbeit - Zebra - hat dem Diplomarbeitsteam einen TC56 ("Touch Computer") zur Verfügung gestellt. Dieser verfügt über einige Eigenschaften, die für eine Inventur vom Vorteil sind [74]:

- Ein Akku mit über 4000 mAh ermöglicht mehrstündige Inventuren.
- Viel Arbeitsspeicher und ein leistungsstarker Prozessor ermöglichen ein flüssiges App-Verhalten.
- Dank robuster Bauart hält das Gerät auch physisch anspruchsvollere Phasen einer Inventur aus.
- Ein in das Smartphone integrierter Barcodescanner reduziert die Scanzeiten drastisch.

## 4.2 Zebra-Scan: Funktionsweise

Die App kommuniziert nicht direkt mit dem Scanner. Stattdessen wickelt das Zebra-Gerät den Scan ab und sendet das Resultat als `Broadcast` aus [9]. Auf dem Zebra-Gerät läuft im Hintergrund immer die DataWege-Applikation. Dies ist eine App, die die Behandlung des tatsächlichen Scans abwickelt und das Ergebnis auf mehrere Arten aussendet [14]. Beispielsweise wird das Ergebnis an die Tastatur geschickt, aber eben auch als Broadcast an das Betriebssystem. Die App registriert sich beim Betriebssystem und hört auf den Broadcast, der den Barcode enthält und automatisch

von DataWege entsandt wird. Broadcast werden durch eine String-ID unterschieden, die über DataWedge konfiguriert wird. Derartige Ansätze werden als Publish–subscribe-Model bezeichnet \cite{publish–subscribe}.

Dies bietet folgende Vorteile:

- Die Hardware wird komplett abstrahiert. Die vorliegende App "sieht" den Scanner zu keinem Zeitpunkt.
- Durch die Abstrahierung des Scanners wird die eigene Codebasis kleiner und weniger kompliziert.
- DataWedge wird von Zebra entwickelt. Damit hat man eine gewisse Sicherheit, dass der Scanner verlässlich funktioniert.

Bei weiterer Überlegung kommt man außerdem zur Erkenntnis, dass der Broadcast nicht von DataWege stammen muss. Wenn eine beliebige andere App, einen Broadcast mit derselben ID ausschickt, wird die vorliegende App dies als Scan werten. In weiterer Folge ist es zumindest theoretisch möglich, beispielsweise einen Bluetooth-Scanner mit einem regulären Android-Gerät zu verwenden und bei einem Resultat, einen Broadcast mit derselben ID auszuschicken. Die vorliegende App würde keinen Unterschied bemerken und somit auch mit dem Bluetooth-Scanner funktionieren. Dieses Einsatzgebiet wurde vom Diplomarbeitsteam jedoch nicht getestet.

### 4.2.1 Zebra-Scan: Codeausschnitt

Broadcast können in Android durch `BroadcastReceiver` ausgelsen werden. Auch hier wurde das DRY-Prinzip angewandt. Jedes Fragment, das Scanergebnisse braucht, verwendet nahezu denselben BroadcastReceiver. Daher hat das Team einen eigenen BroadcastReceiver erstellt, der die gemeinsamen Eigenschaften zusammenführt. Der gesamte Code für den Zebra-Scan konnte damit relativ kompakt in einer Klasse eingebunden werden:

```
public class ZebraBroadcastReceiver extends BroadcastReceiver {
    ...
    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();

        // R.string.activity_intent_filter_action definiert
        // die konfigurierte ID des Broadcasts
        if (action.equals(context.getResources()
           .getString(R.string.activity_intent_filter_action))) {
            // R.string.datawedge_intent_key_data
            // definiert die ID des Scanergebnis selbst.
```

```
            // Der Scan liefert auch andere Ergebnisse,
            // wie z.B. das Barcodeformat.
            // Relevant ist nur der Barcode.
            String barcode = intent.getStringExtra(
                context.getResources()
                .getString(R.string.datawedge_intent_key_data));


            // Das Scanergebnis wird nun per
            // funktionalem Interface an die Fragments weitergegeben.
            scanListener.handleZebraScan(barcode);
        }
    }


    public static void registerZebraReceiver(
                            Context context,
                            ZebraBroadcastReceiver zebraBroadcastReceiver,
                            ErrorHandler errorHandler) {
        ...
        IntentFilter filter = new IntentFilter();
        filter.addCategory(Intent.CATEGORY_DEFAULT);
        // R.string.activity_intent_filter_action definiert
        // die konfigurierte ID des Broadcasts
        filter.addAction(
            context.getResources()
            .getString(R.string.activity_intent_filter_action));
        // Hier wird der BroadcastReceiver mit der ID,
        // auf die er zu hören hat, registriert.
        context.registerReceiver(zebraBroadcastReceiver, filter);
    }


    public static void unregisterZebraReceiver(
                    Context context,
                    ZebraBroadcastReceiver zebraBroadcastReceiver) {
         // Falls ein BrodcastReceiver nicht mehr gebraucht wird
         // Sollte er immer abgemeldet werden.
        context.unregisterReceiver(zebraBroadcastReceiver);
    }
}
```

## 4.3 Der Kamerascan

Für Geräte, die nicht dem Hause Zebra entstammen, bietet die App die Möglichkeit eines Kamerascans an. Im Hintergrund wird dafür die Google Mobile Vision API verwendet, die unter anderem auch Texterkennung oder Gesichtserkennung anbietet [53]. Hierbei wird ein Barcode mittels der Gerätekamera erfasst, ohne dass zuvor ein Bild gemacht werden muss. Dem Benutzer wird eine Preview angezeigt und die Kamera schließt sich, sobald ein Barcode erfasst wurde. Um die Performanz zu maximieren, hat das Diplomarbeitsteam folgende Optimierungen vorgenommen:

- Die API wurde um Blitzfunktionalitäten ergänzt. Dazu wurde eine OpenSource-Variante der Library modifiziert, da die offizielle proprietäre Version keinen Blitz unterstützt. Der Blitz ist über einen ToggleButton sofort deaktivierbar oder aktivierbar. Um einen Klick einzusparen, kann der Benutzer über die Einstellungen den Blitz gleich beim Start des Kamerascans aktivieren lassen. Für eine optimale Erkennung darf man den Blitz jedoch nicht direkt in Richtung des Barcode anvisieren, sondern sollte den Blitz entweder höher oder niedriger als den Barcode halten, um optische Reflexionen zu vermeiden.
- Über die Einstellungen kann der Benutzer die Barcodeformate einschränken. Dies führt ebenfalls zur schnelleren Barcodeerkennung und vermeidet zudem noch das Auftreten von false positives. Wenn der Benutzer die Kamera nicht auf den gesamten Barcode hält, kann es unter Umständen dazu kommen, dass der abgeschnittene Barcode fälschlicherweise als anderes Format interpretiert wird und der Scan daher einen Barcode liefert, der nicht existiert. Offiziell wird in der vorliegenden Organisation nur das `Code_93`-Format eingesetzt.
- Interne Test haben ergeben, dass ein Aspect Ratio von 16:9 das Schnellste ist. Daher wird die Preview-Größe statisch auf 1920x1080 Pixel festgelegt. Die Preview verwendet jedoch tatsächlich die Größe, die am nähesten zu 1920x1080 ist und vom Gerät unterstützt wird.
- Falls ein Scan erfolgreich ist, wird ein Piepston abgespielt, der als akustisches Feedback fungiert.

### 4.3.1 Kamerascan: Codeausschnitt

Für die Scanfeatures wird bezüglich der Single-Activity-App eine Ausnahme gemacht. Da diese Screens navigationstechnisch unabhängig sind und im Vollbildmodus gestartet werden, macht es mehr Sinn, sie als Activities anstatt als Fragments zu implementieren. Die Fragments, die einen Kamerascan verwenden, können ihn mit `startActivityForResult` aufrufen. Sie starten also eine neue Activity um ein Ergebnis zu erhalten:

```
Intent intent = new Intent(getContext(), ScanBarcodeActivity.class);
// 0 ist der Request Code, der die Aktivität identifiziert.
startActivityForResult(intent, 0);
```

Mit der Vision API erstellt man einen `BarcodeDetector` dem ein `Processor` zugewiesen wird. Falls ein Barcode erkannt wird, wird `receiveDetections` automatisch aufgerufen. Die App nimmt sich den ersten erkannten Barcode heraus, setzt ihn als Ergebnis dieser Aktivität und beendet die Aktivität.

```
barcodeDetector.setProcessor(new Detector.Processor<Barcode>() {
        ...

     @Override
    public void receiveDetections(Detector.Detections<Barcode> detections) {
            ...
            // Barcode einlesen
            String barcode = barcodeSparseArray.valueAt(0).rawValue;

            //Barcode bekanntgeben
            Intent intent = new Intent();
            intent.putExtra("barcode", barcode);
            setResult(CommonStatusCodes.SUCCESS, intent);
            finish();
            ...
    }
```

Das Ergebnis kann dann wiederum im Fragment wie folgt abgefangen und weiter verarbeitet werden:

```
@Override
public void onActivityResult(int requestCode, int resultCode,
                             @Nullable Intent data) {
    if (requestCode == 0) {
        if (resultCode == CommonStatusCodes.SUCCESS) {
            ...
            // Barcode einlesen
            String barcode = data.getStringExtra("barcode");
            // Anhand des Barcodes werden dann
            // weitere Aktionen gesetzt
            launchItemDetailFragmentFromBarcode(barcode);
        }
        ...
}
```

## 4.4 Die manuelle Eingabe

Gegenstände können durch Scannen, aber auch durch manuelles Klicken auf ihre GUI-Darstellung validiert werden. Es kann durchaus Sinn machen, auf einen Scan zu verzichten, wenn:

- der Scan langsam oder unmöglich ist (z.B. bei Barcodes in unerreichbarer Höhe oder beschädigten Barcodes).
- man einen Gegenstand auch ohne Barcode identifizieren kann.
- man mit einer manuellen Vorgehensweise schneller ist, als mit dem Kamerascan.

### 4.4.1 Suche

Die Gegenstandsliste, die dem Benutzer angezeigt wird, ist durch eine Suchleiste (`SearchBar`) filterbar. Der Benutzer kann nach Barcode und Gegenstandsbeschreibung filtern. Dadurch ergibt sich auch die Möglichkeit, eine Voice-Tastatur - insofern das Gerät eine hat - einzusetzen.

### 4.4.2 Textscan

Falls der Benutzer weder Voice noch Tastatur verwenden will, kann er den Textscan verwenden. Hierbei wird per Google Mobile Vision API der aufgedruckte Text auf einem Barcode - jedoch NICHT der Barcode selbst - gescannt [54]. Der Scan ist für Sprachen, die ein lateinisches Schriftsystem verwenden, optimiert. Dem Benutzer wird das aktuelle Scanergebnis laufend angezeigt. Da der Out-Of-The-Box-Scan für die vorliegenden Zwecke nicht unbedingt geeignet ist, wurden folgende Modifikationen vorgenommen (Der Textscan wurde als `TextScanActivity` realisiert):

- Der Scanner gibt grundsätzlich alles was er sieht wieder. Das Projektteam hat dies mit Regex-Ausdrücken kombiniert um sinnvolle Ergebnisse zu erlangen. Es gibt daher verschiedene Texterkennungsmodi:
  - Kein Filter. Hier wird keine Regex verwendet und der Benutzer sieht den ungefilterten Text, den der Scanner erkannt hat.
  - Alphanumerisch.
  - Alphabetisch.
  - Numerisch (Barcodes).
  - IP-Adressen. Jeder Modus verwendet die Regex, die am besten für seine Kategorie geeignet ist. Die Library bietet nur geringfügige Möglichkeiten an, den Scan selbst zu beeinflussen. Man kann einen Fokus auf ein Scanergebnis setzen. Daher wird der Fokus auf ein Scanergebnis gesetzt, sobald dieses durch die aktuelle

Regex ausgedrückt werden kann. Die API beschränkt sich allerdings nicht auf das Ergebnis, auf das der Fokus gesetzt wurde, sondern kann jederzeit den Fokus selbst wieder aufheben. Unabhängig davon werden alle Zeichen des Ergebnisses entfernt, die nicht durch die Regex erfasst wurden.

- Die API wurde um Blitzfunktionalitäten ergänzt.

Die Erkennung von Barcodes (also der Numerische Modus) hat zusätzlich folgende Optimierungen erhalten:

- Gescannte Ergebnisse werden durch die Regex optimiert und gespeichert. Das Ergebnis, das zuerst dreimal vorkam, wird eingelockt. Das heißt, dass alle restlichen Scans ignoriert werden.
- Zeichen die häufig vom Scanner verwechselt werden (z.B. wird die Ziffer “0” häufig als Buchstabe “O” erkannt), werden durch ihr numerisches Gegenstück ersetzt.
- Längere Zeichenketten erhalten eine höhere Priorität und werden dem Benutzer vorzugsweise angezeigt. Damit werden abgeschnittene Ergebnisse benachteiligt. Falls ein kürzeres Ergebnis jedoch dreimal vorkommt, wird nicht mehr die längste Zeichenkette, sondern die häufigste bevorzugt und dem Benutzer angezeigt.
- Ergebnisse, die weniger als sieben Zeichen enthalten, werden nicht gespeichert und können damit auch nicht eingelockt werden.

Der Benutzer kann das aktuell angezeigte Ergebnis in seine Zwischenablage kopieren und anschließend die Suchleiste nutzen. Die Kommunikation zwischen den restlichen Screens und dem Textscan erfolgt analog zum Kamerascan, da der Textscan aus denselben Gründen als Activity implementiert wurde.

# 5 Die Inventurlogik auf der App

## 5.1 Die Modelle

Um die Antworten des Servers abzubilden, wurden mehrere Modell-Klassen erstellt. Deren Konstruktur hat ein JSONObject als Parameter. Die Werte der einzelnen Attribute werden aus diesem JSONObject ausgelesen. Folgende Modell-Klassen wurden erstellt.

- `SerializerEntry`: Stelt eine Datenbanksicht dar.
- `Stocktaking`: Stellt eine Inventur dar.
- `Room`: Stellt einen Raum dar.
- `MergedItem`: Stellt einen Gegenstand dar.
- `MergedItemField`: Stellt ein dynamisches Feld dar.
- `Attachment`: Stellt einen Anhang dar.

### 5.1.1 Grundsätze

Eine Modell-Klasse verwaltet etwaige Statusinformationen immer selbst. So weiß beispielsweise nur ein Gegenstand selbst, dass er ursprünglich aus einem anderen Raum stammt. Dies verbessert die Lesbarkeit und Wartbarkeit des Codes massiv, da diese Informationen abstrahiert sind und nicht mehrmals an verschiendenen Stellen im Quellcode schlummern.

## 5.2 Die Fragments

Dieses Kapitel basiert auf den Erklärungen der vorhergehenden Artikel. Eine Inventur wird auf der App durch folgenden Screens (Fragments) abgewickelt, die gleichzeitig als Phasen verstanden werden können:

- `StocktakingFragment`
- `RoomsFragment`

- `ViewPagerFragment`
  - `MergedItemsFragment`
  - `ValidatedMergedItemsFragment`
- `DetailedItemFragment`
- `AttachmentsFragment`

**StocktakingFragment** ist das Fragment, in dem der Benutzer die aktuelle Inventur auswählt. Die Inventur kann nur vom Administrator am Server angelegt werden. Außerdem wählt der Benutzer hier die Datenbanksicht ("Serializer") aus. Die App kommuniziert ausschließlich mittels REST API mit dem Server. Diese Schnittstelle kann verschiedene Quellen haben. Für eine Inventur an unserer Schule ist die "HTL"-Datenbanksicht auszuwählen. Durch das Bestätigen eines langegezogenen blauen Buttons gelangt man immer zur nächsten Inventurphase. In diesem Fall gelangt man zum `RoomsFragment`.

**RoomsFragment** ist das Fragment, in dem der Benutzer den aktuellen Raum über eine DropDown auswählt. Anstatt die DropDown zu verwenden, kann er alternativ auch die Suchleiste verwenden. Als zusätzliche Alternative hat der Benutzer die Möglichkeit den Barcode eines Raumes zu scannen. Nach der Auswahl eines Raumes gelangt er zum `ViewPagerFragment`.

**ViewPagerFragment** ist das Fragment, das als Wrapper für das `MergedItemsFragment` und das `ValidatedMergedItemsFragment` dient. Die einzige Aufgabe dieses Fragments ist es, die zwei vorher genannten Fragments als Tabs anzuzeigen. Dies wurde mit der neuen ViewPager2-Library realisiert [78]. Die Tabs kommunizieren über ein geteiltes ViewModel miteinander.

**MergedItemsFragment & ValidatedMergedItemsFragment** sind die Fragments, die die Gegenstandsliste eines Raumes verwalten und sie dem Benutzer anzeigen. Die Gegenstände werden einzeln validiert. Durch das Scannen des Barcodes eines Gegenstands (beziehungsweise das Klicken auf seine GUI-Repräsentation) gelangt er zum `DetailedItemFragment`. `ValidatedMergedItemsFragment` erfüllt nur den Zweck, dem Benutzer bereits validierte Gegenstände anzuzeigen und ihm die Möglichkeit zu geben, validierte Gegenstände zurück zu den nicht-validierten Gegenständen in `MergedItemsFragment` zu verschieben. Daher trägt der Tab für das `MergedItemsFragment` die Beschriftung "TODO", währenddessen der Tab für das `ValidatedMergedItemsFragment` die Beschriftung "DONE" trägt.

Beide Fragments verwenden eine `RecyclerView`, um dem Benutzer die Gegenstände anzuzeigen [69]. Eine RecyclerView generiert pro Eintrag ein Layout, hält aber nur die aktuell angezeigten Einträge inkl. Layout im RAM.

**DetailedItemFragment** ist das Fragment, das zur Validierung eines einzelnen Gegenstands dient. Der Benutzer hat hier die Möglichkeit, etwaige Eigenschaften des Gegenstandes (beispielsweise den Anzeigenamen) zu ändern. Ein Formular, dass die Felder eines Gegenstandes beinhaltet, wird einmal angefordert, und anschließend für die gesamte Lebensdauer der App gespeichert. Anhand dieses Formulars wird dann eine GUI-Repräsentation dynamisch erstellt. Das Formular kann `ExtraFields` beinhalten. Das sind Felder, die als nicht essentiel angesehen werden und infolgedessen standardmäßig eingeklappt sind. Dazu gehören auch benutzerdefinierte Felder - sognannte `CustomFields`.

Dadurch braucht man für die gesamte Validierung eines Raumes - insofern keine Sonderfälle auftreten - keine Verbindung zum Server. Der Benutzer kann die Liste an einer Lokalität mit einer guten Verbindung anfordern, den Raum mit schlechter Verbindung betreten und validieren. Anschließend kann er den Raum verlassen und seine Validierungen an den Server senden. Damit wird der Bedarf an Netzwerkanfragen in Räumen mit schlechter Netzwerkverbindung minimiert. Der Benutzer kann zusätzlich zur Validierung auch Anhänge für einen Gegenstand definieren, dazu landet er beim `AttachmentsFragment`.

Folgende GUI-Komponenten wurden statisch implementiert, da sie Felder repräsentieren, die unabhängig von der ausgewählten Datenbanksicht immer vorhanden sind und daher nicht dynamisch sind:

- Ein read-only Textfeld für die Gegendstandsbeschreibung
- Ein read-only Textfeld für den Barcode
- Eine Checkbox “Erst später entscheiden”
- Eine DropDown für Subrooms

**AttachmentsFragment** ist das Fragment, das die Anhänge eines Gegenstandes verwaltet. Der Benutzer sieht hier die bereits vorhandenen Anhänge mit Beschriftungen und kann weitere Anhänge hinzufügen. Bilder werden direkt angezeigt. Andere Dateien werden hingegen als Hyperlink dargestellt. Der Benutzer kann diese per Browser runterladen. Zum Hochladen eigener Anhänge greift die App auf den Standard-Dateibrowser des Systems zurück. Das Outsourcen auf Webbrowser und Dateibrowser bietet den massiven Vorteil, dass man sich die Entwicklung eigener Download-Manager bzw. File-Manager erspart und auf Apps setzen kann, die von namenhaften Herstellern entwickelt werden. Fast jedes System hat bereits beide Komponenten vorinstalliert, daher treten bezüglich der Verfügbarkeit keine Probleme auf.

Beim dem Hochladen von Bildern komprimiert die App jene zuvor. Die Qualität des Bildes ist einstellbar:

- 100 % (keine Kompression)
- 95 %
- 85 %
- 75 %

Zur Kompression wird das WEBP-Format verwendet, das dem mittlerweile veralteten JPG-Standard überlegen ist [79]. Der Server speichert den gesendeten Anhang nur einmal (Dateien werden anhand von Hashes unterschieden). Wenn ein Benutzer allen PCs in einem EDV-Saal dasselbe Bild zuweist, wird es nur einmal am Server hinterlegt. Die Anzahl der Anhänge ist nicht limitiert.

## 5.3 Validierungslogik

`MergedItemsFragment` & `DetailedItemFragment` sind die Fragments, die den Großteil einer Inventur ausmachen. Der Benutzer scannt alle SAP-Barcodes, die sich in einem Raum befinden. Im Idealfall entspricht diese Menge exakt der Menge der Gegenstände, die dem Benutzer im MergedItemsFragment angezeigt wird. Im Normalfall wird dies durch etwaige Sonderfälle jedoch nicht gegeben sein.

Nach dem Scannen eines Gegenstandes werden die Felder des Gegenstandes dem Benutzer im DetailItemFragment angezeigt. In diesem Fragment hat der Benutzer zwei Buttons, mit denen er den Gegenstand validieren kann:

- Grüner Button: Mit diesem Button wird signalisiert, dass sich der Gegenstand im Raum befindet und der Gegenstand wird mitsamt etwaigen Änderungen an seinen Attributen/Feldern übernommen. Dazu wird ein `ValidationEntry` erstellt. Alternativ kann der Benutzer das Klicken dieses Buttons mit dem Schütteln des Gerätes ersetzen. Die Schüttel-Sensibilität ist über die Einstellungen konfigurierbar (und auch deaktivierbar).
- Roter Button: Mit diesem Button wird signalisiert, dass sich der Gegenstand nicht im Raum befindet. Dieser Button wird im Normalfall nie betätigt werden, da ein Gegenstand, der sich nicht in diesem Raum befindet, nicht gescannt werden kann und daher dieses Fenster nie geöffnet werden wird. Der Button hat trotzdem einen Sinn, da der Benutzer damit die “TODO”-Liste über das GUI verkleinern kann, um sich einen besseren Überblick zu verschaffen.

### 5.3.1 Der ValidationEntry

Das MergedItemsFragment (bzw. das MergedItemsViewModel) verfügt über eine HashMap (`Map<MergedItem, List<ValidationEntry>>`), die alle ValidationEntries beinhaltet. Ein ValidationEntry beinhaltet sämtliche Informationen, die der Server benötigt, um die Datensätze eines Gengenstandes entsprechend anzupassen. Einem Gegenstand ist eine Liste an ValidationEntries zugeordnet, da Subitems eigene ValidtionEntries bekommen können (siehe "Sonderfälle").

Ein ValidationEntry beinhaltet immer den Primary Key eines Gegenstandes und die Felder, die sich geändert haben. Ein ValidationEntry hat daher eine Liste an Feldern `List<Field>`. Wenn sich der Wert eines Feldes geändert hat, wird diese Liste um einen Eintrag erweitert. Da die Felder wie erwähnt dynamisch sind, wurden Java Generics eingesetzt [44], um diese abbilden zu können. `Field` ist eine innere Klasse in `ValidationEntry`:

```
public static class Field<T> {
    private String fieldName;
    private T fieldValue;
    ...
```

Ein Feld besteht also immer aus einem Feldnamen und einem generischen Feldwert. Selbiges gilt für ExtraFields.

#### 5.3.1.1 Sendeformat

Wenn ein Raum abgeschlossen ist, werden alle ValidationEntries in einer Liste vereint und anschließend in eine JSON-Darstellung transformiert. Dieses JSON wird dem Server gesandt und damit ist der aktuelle Raum abgeschlossen und der Benutzer kann sich den restlichen Räumen annehmen. Das POST-Format wird im Kapitel "JSON-Schema" beschrieben.

### 5.3.2 Quickscan

Der häufigste Fall einer Inventur wird der sein, dass ein Gegenstand im richtigen Raum ist und der Benutzer ohne weiteren Input auf den grünen Button drückt. Da dies einen unnötigen Overhead darstellt, wurde die App um den `QuickScan`-Modus erweitert. Hierbei wird sofort nach dem Scannen ein `ValidationEntry` erstellt, ohne dass zuvor das DetailedItemFragment geöffnet wird. Dieser Modus ist durch einen weiteren Button im MergedItemsFragment aktivierbar/deaktivierbar. Falls ein Sonderfall auftreten sollte, vibriert das Gerät zweimal und öffnet das DetailedItemFragment. Damit wird

gewährleistet, dass der Benutzer nicht irrtümlich mit dem Scannen weitermacht. Er muss diesen Sonderfall händisch validieren. Haptisches Feedback ist für Sonderfälle reserviert.

### 5.3.3 Sonderfälle auf der App

Ein zentrales Thema der vorliegenden Diplomarbeit ist die Behandlung der Sonderfälle.

#### 5.3.3.1 Subitems

Falls ein Gegenstand mehrmals vorhanden sein sollte, ist der `times_found_last`-Zähler in der Antwort des Servers größer als 1 und der Gegenstand gilt als Subitem. Dieser Counter wird dem Benutzer in folgender Form angezeigt: `[Anzahl aktuell gefunden] / [Anzahl zuletzt gefunden]`.

Pro Subitem wird ein eigener ValidationEntry erstellt. An unserer Schule haben Subitems jedoch keinen Barcode sondern sind beispielsweise Teil eines Bundles, was dazu führt, dass Subitems auch in der Datenbank keine selbstständigen Gegenstände sind. CPs die auf Basis dieser ValidationEntries erstellt werden, werden dem echten "Parent"-Gegenstand zugeordnet und können wahlweise angewandt werden. Falls ein Gegenstand mehrmals gescannt wird, wird - nach Bestätigung durch den Benutzer - die Anzahl der aktuellen Funde erhöht und wiederum ein ValidationEntry erstellt, selbst wenn es sich bei dem Gegenstand aktuell nicht um ein Subitem handelt.

#### 5.3.3.2 Subrooms

Subrooms sind logische Räume in einem Raum. Subrooms werden dem Benutzer im MergedItemsFragment als einklappbare Teilmenge aller Gegenstände des Raumes angezeigt. Die Subroom-Zugehörigkeit kann auf Gegenstandsbasis über eine DropDown geändert werden. Die Subroom-Zugehörigkeit wird in einem ValidationEntry immer gesetzt, auch wenn sie sich nicht geändert hat.

Die Gegenstandsliste des MergedItemsFragment beinhaltet in Wahrheit auch die Subrooms. Dies ist notwendig, da die `RecyclerView`, die dazu genutzt wird dem Benutzer die Gegenstände anzuzeigen, keine Möglichkeit bietet, eine Hierarchie bzw. Zwischenebenen darzustellen. Daher implementieren das Room-Model und das MergedItem-Model das Interface `RecyclerViewItem` und die RecyclerView erhält eine Liste an RecyclerViewItems - dies ist ein typisch polymorpher Ansatz. Abhängig vom Typen des

aktuellen Listenelements baut die RecyclerView entweder ein Gegenstandslayout oder ein Raumlayout auf.

#### 5.3.3.3 Unbekannte Gegenstände

Falls ein Gegenstand gescannt wird, der sich nicht in der aktuellen Gegenstandsliste befindet, muss der Server befragt werden. Es gibt zwei mögliche Antwortszenarien. Die ValidationEntries für diese Sonderfälle unterscheiden sich nicht von den bisherigen.

**Neuer Gegenstand** Der Gegenstand befindet sich überhaupt nicht in der Datenbank. Im “DONE”-Tab haben solche Gegenstände eine blaue Hervorhebung.

**Gegenstand aus anderem Raum** Der Gegenstand befindet in der Datenbank und stammt ursprünglich aus einem anderen Raum. Im “DONE”-Tab haben solche Gegenstände eine orange Hervorhebung.

# 6 Einführung in die Server-Architektur

Die Inventur- sowie Gegenstandsdaten der HTL Rennweg sollen an einem zentralen Ort verwaltet und geführt werden. Der für diesen Zweck entwickelte Server muss also folgende Anforderungen erfüllen:

- eine einfache Datenbankverwaltung und -verbindung
- das Führen einer Historie aller Zustände der Inventar-Gegenstände
- eine Grundlage für eine Web-Administrationsoberfläche
- die Möglichkeit für Datenimport und -export, etwa als *.xlsx* Datei
- eine Grundlage für die Kommunikation mit der Client-Applikation
- hohe Stabilität und Verfügbarkeit

Angesichts der Programmiersprachen, auf die das Projektteam spezialisiert ist, stehen als Backend-Lösung vier *Frameworks* zur öffentlichen Verfügung, zwischen denen gewählt wurde:

- Django [58]
- Pyramid [65]
- Web2Py [66]
- Flask [63]

Alle genannten Alternativen sind *Frameworks* der Programmiersprache Python. Gewählt wurde "Django" aufgrund einer bestehenden und frei verfügbaren Inventarverwaltungsplattform "Ralph" [67], die auf Django aufbaut und durch die vorliegende Diplomarbeit hinreichend erweitert wird.

## 6.1 Django und Ralph

Django ist ein in der Programmiersprache Python geschriebenes Webserver-*Framework*. Ralph ist eine auf dem Django-*Framework* basierende *DCIM* und *CMDB* Softwarelösung. Haupteinsatzgebiet dieser Software sind vor allem Rechenzentren mit hoher Komplexität, die externe Verwaltungsplattformen benötigen. Zusätzlich können aber

auch herkömmliche Inventardaten von IT-spezifischen Gegenständen in die Ralph-Plattform aufgenommen werden.

Ralph wurde von der polnischen Softwarefirma “Allegro” entwickelt und ist unter der Apache 2.0 Lizenz öffentlich verfügbar. Dies ermöglicht auch Veränderungen und Erweiterungen. Zu Demonstrationszwecken bietet Allegro eine öffentlich nutzbare Demo-Version [16] von Ralph an.

Die vorliegende Diplomarbeit bietet eine Erweiterung des Ralph-Systems.

### 6.1.1 Begründung der Wahl von Django und Ralph

Django bietet eine weit verbreitete Open-Source Lösung für die Entwicklung von Web-Diensten. Bekannte Webseiten, die auf Django basieren sind u.a. Instagram, Mozilla, Pinterest und Open Stack. [8] Django zeichnet sich besonders durch die sog. *"Batteries included"* Mentalität aus. Das heißt, dass Django bereits die gängigsten *Features* eines Webserver-Backends standardmäßig innehat. Diese sind (im Vergleich zu Alternativen wie etwa “Flask”) u.a.:

- Authentifikation und Autorisierung, sowie eine damit verbundene Benutzerverwaltung
- Schutz vor gängigen Attacken (wie *SQL-Injections* oder *CSRF* [68]), siehe Abschnitt “Views”

Zusätzlich bietet Ralph bereits einige *Features*, die die grundlegende Führung und Verwaltung eines herkömmlichen Inventars unterstützen (beispielsweise eine Suchfunktion mit automatischer Textvervollständigung).

## 6.2 Kurzfassung der Funktionsweise von Django und RIaph

Im folgenden Kapitel wird die Funktionsweise des Django-*Frameworks*, sowie Ralph beschrieben. Ziel dieses Kapitels ist es, eine Wissensbasis für die darauffolgenden Kapitel zu schaffen.

### 6.2.1 Datenbank-Verbindung, Pakete und Tabellen-Definition

Folgende Datenbank-Typen werden von Django unterstützt:

- PostgreSQL
- MariaDB
- MySQL
- Oracle
- SQLite

Die Konfiguration der Datenbank-Verbindung geschieht unter Standard-Django in der Datei `settings.py` , unter Ralph in der jeweiligen Datei im Verzeichnis `settings`. Eine detaillierte Anleitung zur Verbindung mit einer Datenbank ist in der offiziellen Django-Dokumentation [20] zu finden.

Die verschiedenen Funktionsbereiche des Servers sind in Pakete bzw. Module gegliedert. Jedes Paket ist ein Ordner, der verschiedene Dateien und Unterordner beinhalten kann. Die Dateinamen-Nomenklatur eines Packets ist normiert.[19] Der Name eines Pakets wird fortan "App-Label" genannt. Standardmäßig ist dieser Name erster Bestandteil einer *URL* zu einer beliebigen graphischen Administrationsoberfläche des Pakets. Pakete werden durch einen Eintrag in die Variable `INSTALLED_APPS` innerhalb der o.a. Einstellungsdatei registriert. Beispiele sind die beiden durch die vorliegende Diplomarbeit registrierten Pakete `"ralph.capentory"` und `"ralph.stocktaking"`

Ist ein Python-Paket erfolgreich registriert, können in der Datei `models.py` Datenbank-Tabellen als python Klassen[1] definiert werden. Diese Klassen werden fortan als "Modell" bezeichnet. Tabellenattribute werden als Attribute dieser Klassen definiert und sind jeweils Instanzen der Klasse `Field`[21][2]. Datenbankeinträge können demnach als Instanzen der Modellklassen betrachtet und behandelt werden. Standardmäßig besitzt jedes Modell ein Attribut `id`, welches als *primärer Schlüssel* dient. Der Wert des `id` Attributs ist unter allen Instanzen eines Modells einzigartig. Die Anpassung dieses Attributs wird in der offiziellen Django-Dokumentation genauer behandelt. [21]

Jedes Modell benötigt eine innere Klasse `Meta`. Sie beschreibt die *Metadaten* der Modellklasse. Dazu gehört vor allem der von Benutzern lesbare Name des Modells `verbose_name`. [24]

### 6.2.2 Administration über das Webinterface

Um die Administration von Modelldaten über das Webinterface des Servers zu ermöglichen, werden grundsätzlich zwei Ansichten der Daten benötigt: Eine Listenansicht aller Datensätze und eine Detailansicht einzelner Datensätze.

Die Listenansicht aller Datensätze eines Modells wird in der Datei `admin.py` als

[1] erbend von der Superklasse `Model`[21]
[2] oder davon erbende Klassen

*Subklasse* von `ModelAdmin`[3] definiert. Attribute dieser Klasse beeinflussen das Aussehen und die Funktionsweise der Weboberfläche. Durch das Setzen von `list_display` werden beispielsweise die in der Liste anzuzeigenden Attribute definiert.

Die Detailansicht einzelner Datensätze wird grundsätzlich durch die `ModelAdmin` Klasse automatisch generiert, kann aber durch Setzen dessen `form` Attributs auf eine eigens definierte *Subklasse* von `ModelForm`[4] angepasst werden. Diese Klassen werden in der Datei `forms.py` definiert und besitzen, ähnlich der `Model` Klasse, auch eine innere Klasse `Meta`.

Um die `ModelAdmin` *Subklassen* über eine *URL* erreichbar zu machen, müssen diese registriert werden. Dies geschieht durch den `register` *Dekorator*. Dieser Dekorator akzeptiert die zu registrierende Modellklasse, die zu dem `ModelAdmin` gehört, als Parameter. Die Listenansicht einer registrierten `ModelAdmin` Subklasse ist standardmäßig unter der *URL*

```
/<App-Label>/<Modell-Name>/
```

erreichbar, die Detailansicht einer Modellinstanz unter der *URL*

```
/<App-Label>/<Modell-Name>/<Modellinstanz-ID>/
```

. Die Dokumentation der Administrationsfeatures von Django ist auf der offiziellen Dokumentationswebseite von Django [17] zu finden.

### 6.2.3 API und DRF

Um Daten außerhalb der graphischen Administrationsoberfläche zu bearbeiten, wird eine *API* benötigt. Eine besondere und weit verbreitete Form einer API ist eine *REST-API* [47], die unter Django durch das integrierte *DRF* implementiert wird.[64] API Definitionen werden unter Django in einem Paket in der Datei `api.py` getätigt.

Um den API-Zugriff auf ein Modell zu ermöglichen werden üblicherweise eine `APIView`[5] *Subklasse* und eine `Serializer`[6] *Subklasse* definiert. `APIView` Klassen sind zuständig für das Abarbeiten von Anfragen mithilfe einer `Serializer` Klasse, die die Daten aus der Datenbank repräsentiert und in das gewünschte Format konvertiert. Durch `APIView` Klassen werden Berechtigungen und sonstige Attribute definiert, die sich

[3] Unter Ralph steht hierfür die Klasse `RalphAdmin` zur Verfügung.[59]
[4] Unter Ralph steht hierfür die Klasse `RalphAdminForm` zur Verfügung.
[5] Unter Ralph steht hierfür die Klasse `RalphAPISerializer` zur Verfügung. [60]
[6] Unter Ralph steht hierfür die Klasse `RalphAPIViewSet` zur Verfügung. [60]

auf das wahrgenommene Erscheinungsbild des Servers auf einen Client auswirken. Beispiel dafür ist die Art der *Paginierung* [64]. Die erstellten `APIView` Klassen können dann mithilfe einer `Router`[7] Instanz registriert werden. Anleitungen zur Erstellung dieser API-Klassen sind auf der offiziellen Webseite des DRF [64] und der offiziellen Ralph-Dokumentationsseite [60] zu finden.

### 6.2.4 Views

Schnittstellen, die keiner der beiden o.a. Kategorien zugeordnet werden können, werden in der Datei `views.py` definiert. Bei diesen *generischen* Schnittstellen handelt es sich entweder um *Subklassen* der Klasse `View`[8] [22] oder vereinzelte Methoden mit einem `request`[9] Parameter. [31] Diese Schnittstellen werden fortan Views genannt.

Soll ein View als Antwort auf eine Anfrage HTML-Daten liefern, so sollte dazu ein *Template* verwendet werden. Mithilfe von Templates können Daten, die etwa durch Datenbankabfrage entstehen, zu einer HTML Antwort aufbereitet werden. Besonders ist hierbei die zusätzlich zu HTML verfügbare Django-Template-*Syntax* [29]. Damit können HTML Elemente auf den Input-Daten basierend dynamisch generiert werden. So stehen beispielsweise `if` Statements direkt in der Definition des Templates zur Verfügung. Die Benutzung von Templates schützt standardmäßig gegen Attacken, wie *SQL-Injections* oder *CSRF* [68] und gilt daher als besonders sicher. Durch das Diplomarbeitsteam wurden weitere Möglichkeiten zur Sicherung des Serversystems [27] implementiert und alle Sicherheitsempfehlungen der Entwickler von Django [27] eingehalten.

Da reguläre Views nicht automatisch registriert werden, müssen sie manuell bekanntgegeben werden. Dies geschieht durch einen Eintrag in die Variable `urlpatterns` in der Datei `urls.py`. [30]

### 6.2.5 Datenbankabfragen

Datenbankabfragen werden in Django durch `Queryset`-Objekte getätigt. Das Definieren eines `Queryset`-Objekts löst nicht sofort eine Datenbankabfrage aus. Erst, wenn Werte aus einem `Queryset`-Objekt gelesen werden, wird eine Datenbankabfrage ausgelöst. So kann ein `Queryset`-Objekt beliebig oft verändert werden, bevor davon ausgelesen wird. Ein Beispiel hierfür ist das Anwenden der `filter()` Methode.

[7] Unter Ralph steht hierfür die globale `RalphRouter` Instanz `router` zur Verfügung. [60]
[8] die ebenfalls Superklasse der Klasse `ApiView` ist
[9] zu Deutch: Anfrage; entspricht den empfangenen Daten

In dem folgenden Code-Auszug[10] werden aus dem Modell `Entry` bestimmte Einträge gefiltert:

```
# Erstelle ein Queryset aller Entry-Objekte,
# dessen Attribut "headline" mit "What" beginnt.
q = Entry.objects.filter(headline__startswith="What")

# Filtere aus dem erstellten Queryset alle Entry-Objekte,
# dessen Attribut "pub_date" kleiner oder gleich
# dem aktuellen Datum ist.
q = q.filter(pub_date__lte=datetime.date.today())

# Schließe aus dem erstellten Queryset alle Entry-Objekte,
# dessen Attribut "body_text" den Text "food" beinhaltet, aus.
q = q.exclude(body_text__icontains="food")

# Ausgabe des erstellten Querysets.
# Erst hier kommt es zu der ersten Datenbankabfrage!
print(q)
```

Weitere Beispiele und Methoden sind der offiziellen Django-Dokumentation zu entnehmen. [25]

## 6.3 Designgrundlagen

Designgrundlagen für Django-Entwickler sind auf der offiziellen Dokumentationsseite von Django [57] abrufbar. Die Erweiterung von Django durch die vorliegende Diplomarbeit wurde anhand dieser Grundlagen entwickelt.

Das Konzept des `Mixins` wird von der Ralph-Plattform besonders häufig genutzt. `Mixins` sind Klassen, die anderen von ihnen erbenden Klassen bestimmte Attribute und Methoden hinzufügen. Manche `Mixins` setzen implizit voraus, dass die davon erbenden Klassen ebenfalls von bestimmten anderen Klassen erben. Beispiel ist die Klasse `AdminAbsoluteUrlMixin`, die eine Methode `get_absolute_url` zur Verfügung stellt. Diese Methode liefert die *URL*, die zu der Detailansicht der Modellinstanz führt, die die Methode aufruft. Voraussetzung für das Erben einer Klasse von `AdminAbsoluteUrlMixin` ist daher, dass sie ebenfalls von der Klasse `Model` erbt.

[10] entnommen aus der offiziellen Django Dokumentation [25]

# 7 Die 2 Erweiterungsmodule des Serversystems

Die vorliegende Diplomarbeit erweitert das “Ralph” System um 2 Module. Dabei handelt es sich um die beiden Pakete “Capentory” und “Stocktaking”. Das Paket “Capentory” behandelt die Führung der Inventardaten und wurde speziell an die Inventardaten der HTL Rennweg angepasst. Das Paket “Stocktaking” ermöglicht die Verwaltung der durch die mobile Applikation durchgeführten Inventuren. Dazu zählen Aufgaben wie das Erstellen der Inventuren, das Einsehen von Inventurberichten oder das Anwenden der aufgetretenen Änderungen.

Dieses Kapitel beschreibt die grundlegende Funktionsweise der beiden Module. Eine Anleitung zur Bedienung der *Weboberfläche* ist dem Handbuch zum Server zu entnehmen.

## 7.1 Das “Capentory” Modul

Das “Capentory” Modul beherbergt 3 wichtige Modelle:

1. HTLItem
2. HTLRoom
3. HTLItemType

Die wichtigsten Eigenschaften der Modelle und damit verbundenen Funktionsweisen werden in diesem Unterkapitel beschrieben.

### 7.1.1 Das HTLItem Modell

Das `HTLItem`-Modell repräsentiert die Gegenstandsdaten des Inventars der HTL Rennweg. Es sind typische Merkmale aus dem *SAP ERP* System vertreten. Die Attribute `anlage`, `asset_subnumber` und `company_code` werden direkt aus dem *SAP ERP* System übernommen.

#### 7.1.1.1 Die wichtigsten Attribute

Zu den wichtigsten Attributen des `HTLItem` Modells zählen u.a.:

- `anlage` und `asset_subnumber`: Diese Attribute bilden den Barcode eines Gegenstandes.
- `barcode_prio`: Wenn dieses Attribut gesetzt ist, überschreibt es den durch die Attribute `anlage` und `asset_subnumber` entstandenen Barcode.
- `anlagenbeschreibung`: Dieses Attribut repräsentiert die aus dem *SAP ERP* System entnommene Gegenstandsbeschreibung und kann nur durch den Import von Daten direkt aus dem *SAP ERP* System geändert werden.
- `anlagenbeschreibung_prio`: Dieses Attribut dient als interne Gegenstandsbeschreibung, die auch ohne einen Import aus dem *SAP ERP* System geändert werden kann.
- `room`: Dieses Attribut referenziert auf ein `HTLRoom` Objekt, in dem sich ein `HTLItem` Objekt befindet.
- `is_in_sap`: Der Wert dieses *Boolean*-Attributs ist `Wahr`, wenn der `HTLItem`-Datensatz aus dem *SAP ERP* System importiert wurde. Umgekehrt ist der Wert dieses Attributes `Falsch`, wenn der `HTLItem`-Datensatz aus einer anderen Quelle entstanden ist. Ein manuell hinzugefügter `HTLItem`-Datensatz hat für dieses Attribut den Wert `Falsch`.
- `item_type`: Dieses Attribut referenziert auf das `HTLItemType` Objekt, das einem `HTLItem` Objekts zugeordnet ist. Es repräsentiert die Gegenstandskategorie eines `HTLItem` Objekts.

#### 7.1.1.2 Einzigartigkeit von HTLItem Objekten

Bezüglich der Einzigartigkeit von `HTLItem` Objekten gelten einige Bestimmungen.

Sind die Attribute `anlage`, `asset_subnumber` und `company_code` je mit einem nichtleeren Wert befüllt, so repräsentieren sie ein `HTLItem` Objekt eindeutig. Es dürfen keine 2 `HTLItem` Objekte denselben Wert dieser Attribute haben. Um diese Bedingung erfüllen zu können muss eine eigens angepasste Validierungslogik implementiert werden. Standardverfahren wäre in diesem Anwendungsfall, die Metavariable `unique_together` [24] anzupassen:

```
unique_together = [["anlage",  "asset_subnumber", "company_code"]]
```

Dieses Verfahren erfüllt nicht die geforderte Bedingung nur in der Theorie. Praktisch werden leere Werte von Attributen dieser Art nicht als `None` (Python) bzw. `null` (MySQL), sondern als Leerstrings `""` gespeichert. Um diese Werte ebenfalls von der

Regel auszuschließen, muss die `validate_unique()`[1] Methode [23] überschrieben werden.

Ist das `barcode_prio` Attribut eines `HTLItem` Objekts gesetzt, darf dessen Wert nicht mit jenem eines anderen `HTLItem` Objekts übereinstimmen. Standardverfahren wäre in diesem Anwendungsfall das Setzen des `unique` Parameters des Attributes auf den Wert `True`. Da dieses Verfahren ebenfalls das o.a. Problem aufwirft, muss die Logik stattdessen in die `validate_unique()` Methode aufgenommen werden. Zusätzlich darf der Wert des `barcode_prio` Attributs nicht mit dem aus den beiden Attributen `anlage` und `asset_subnumber` generierten Barcode übereinstimmen. Um diese Bedingung zu erfüllen kann nur die `validate_unique()` Methode herbeigezogen werden.

#### 7.1.1.3 Änderungsverlauf

Besonders für das `HTLItem` Modell ist es von besonderer Wichtigkeit, ein Objekt auf den Zustand vor einer unbeabsichtigten Änderung zurücksetzen und gelöschte Objekte wiederherstellen zu können. Durch das bereits in Ralph inkludierte Paket `django-reversion` können die aktuellen Zustände von Datenbankobjekten gesichert werden, um später darauf zugreifen zu können. [35] Das Paket bietet die Funktion, der graphischen Administrationsoberfläche entsprechende Funktionen zur Wiederherstellung oder Zurücksetzung einzelner Objekte hinzuzufügen.

Um einen Gegenstand zu entinventarisieren, kann er gelöscht werden. Referenzen auf den gelöschten Gegenstand, die durch eine Inventur entstehen, bleiben in einem ungültigen Zustand erhalten. Der gelöschte Gegenstand kann zu einem späteren Zeitpunkt vollständig wiederhergestellt werden. Die Referenzen auf den wiederhergestellten Gegenstand werden damit wieder gültig.

#### 7.1.1.4 Speichern von Bildern und Anhängen

Die in Ralph verfügbare Klasse `AttachmentsMixin` wird verwendet, um Instanzen der Modellklasse `HTLItem` über dessen graphische Administrationsoberfläche diverse Anhänge zuzuordnen. Ein Anhang ist eine ordinäre Datei, die auf den Server hochgeladen werden kann. Alle hochgeladenen Dateien werden vor dem Speichern anhand deren *Hash-Werten* verglichen. Ist eine Datei mit dem *Hash-Wert* einer hochgeladenen Datei bereits vorhanden, wird die Datei nicht erneut gespeichert und ein Verweis auf die vorhandene Datei wird erstellt.

[1] Eine Methode einer Modell-Klasse, die unter Normalzuständen immer vor dem Speichern eines Objekts des Modells aufgerufen wird. Wirft sie einen Fehler auf, kann das Objekt nicht gespeichert werden.

### 7.1.2 Das HTLRoom Modell

Das `HTLRoom`-Modell repräsentiert die Raumdaten der HTL Rennweg. Es sind typische Merkmale aus dem *SAP ERP* System vertreten. Die Attribute `room_number`, `main_inv` und `location` werden direkt aus dem *SAP ERP* System übernommen.

#### 7.1.2.1 Die wichtigsten Attribute

Zu den wichtigsten Attributen des `HTLRoom` Modells zählen u.a.:

- `room_number`: Dieses Attribut bildet den Barcode eines Raumes.
- `barcode_override`: Wenn dieses Attribut gesetzt ist, überschreibt es den durch das Attribut `room_number` gebildeten Barcode.
- `internal_room_number`: Dieses Attribut repräsentiert die schulinterne Raumnummer und ist somit von den Daten aus dem *SAP ERP* System unabhängig.
- `is_in_sap`: Der Wert dieses *Boolean*-Attributs ist `Wahr`, wenn der `HTLRoom`-Datensatz einen aus dem *SAP ERP* System vorhandenen Raum repräsentiert.
- `children`: Mit dieser Beziehung können einem übergeordneten `HTLRoom` Objekt mehrere `HTLRoom` Objekte untergeordnet werden. Anwendungsfall für dieses Attribut ist die Definition von Schränken oder Serverracks, die je einem übergeordneten Raum zugeteilt sind, selbst aber eigenständige Räume repräsentieren.
- `type`: Dieses Attribut spezifiziert die Art eines `HTLRoom` Objekts. So kann ein `HTLRoom` Objekt einen ganzen Raum oder auch nur einen Kasten in einem übergeordneten Raum repräsentieren.
- `item`: Dieses Attribut kann gesetzt werden, um ein `HTLRoom` Objekt durch ein `HTLItem` Objekt zu repräsentieren. Anwendungsbeispiel ist ein Schrank, der sowohl als `HTLRoom` Objekt als auch als `HTLItem` Objekt definiert ist. Sind die beiden Objekte durch das `item` Attribut verbunden, ist der Barcode des `HTLRoom` Objekts automatisch jener des `HTLItem` Objekts.

#### 7.1.2.2 Einzigartigkeit von HTLRoom Objekten

Bezüglich der Einzigertigkeit von `HTLRoom` Objekten gelten ähnliche Bestimmungen wie zu jener von `HTLItem` Objekten.

Die Attribute `room_number`, `main_inv` und `location` sind gemeinsam einzigartig. Leere Werte sind von dieser Regel ausgeschlossen. Gleichzeitig darf der Wert des `barcode_override` Attribut nicht mit dem Wert des `room_number` Attributes eines anderen `HTLRoom` Objekts übereinstimmen und vice versa. Beide Bedingungen müssen

wie im Falle des `HTLItem` Modells durch Überschreiben der `validate_unique()`[2] Methode [23] implementiert werden.

#### 7.1.2.3 Subräume

Wie im Abschnitt “Die wichtigsten Attribute” festgehalten, können einem `HTLRoom` Objekt mehrere `HTLRoom` Objekte untergeordnet werden. Diese untergeordneten Räume werden “Subräume” genannt. Bei “Subräumen” handelt es sich beispielsweise um einen Kasten, der als eigenständiger Raum in einem ihm übergeordneten Raum steht. Logisch betrachtet ist der Kasten auch nur ein Raum, in dem sich Gegenstände befinden. Ob der Raum ein Klassenraum oder ein Kasten in einem Klassenraum ist, hat keine logischen Auswirkungen auf seine Eigenschaften als “Standort von Gegenständen”.

Um eine valide Hierarchie beizubehalten, muss diese Beziehung bei jedem Speicherprozess eines `HTLRoom` Objekts überprüft werden. Das geschieht durch die Methode `clean_children()`, die beim Speichern durch die graphische Administrationsoberfläche automatisch aufgerufen wird. Bei Speichervorgängen, die nicht direkt durch die Administrationsoberfläche initiiert werden [3], muss `clean_children()` manuell aufgerufen werden.

### 7.1.3 Das HTLItemType Modell

Das `HTLItemType` Modell repräsentiert Kategorien von Gegenständen. Das Modell besteht aus 2 Eigenschaften. Die Eigenschaft `item_type` beschreibt ein `HTLItemType` kurz, die Eigenschaft `description` bietet Platz für Kommentare.

Durch das Setzen eines `HTLItemType` Objekts für ein `HTLItem` Objekt durch seine Eigenschaft `item_type` werden dem `HTLItem` Objekt alle *Custom-Fields* des `HTLItemType` Objekts zugewiesen. Anwendungsbeispiel ist das Setzen eines *Custom-Fields* namens “Anzahl Ports” für den `HTLItemType` namens “Switch”. Jedes `HTLItem` Objekt mit dem `item_type` “Switch” hat nun ein *Custom-Field* namens “Anzahl Ports”.

Das für *Custom-Fields* erforderliche Mixin (siehe Abschnitt “Designgrundlagen”) `WithCustomFieldsMixin` bietet die Funktionalität der `custom_fields_inheritance`. Sie ermöglicht das Erben von allen *Custom-Field* Werten eines bestimmten Objekts an ein anderes. Diese Funktion macht sich das `HTLItem` Modell zunutze. Um beim Speichern automatisch vom `HTLItemType` Objekt unabhängige *Custom-Field* Werte zu erstellen, die sofort vom Benutzer bearbeitet werden können, muss der Speicherlogik

[2] Eine Methode einer Modell-Klasse, die unter Normalzuständen immer vor dem Speichern eines Objekts des Modells aufgerufen wird. Wirft sie einen Fehler auf, kann das Objekt nicht gespeichert werden.

[3] etwa das automatisierte Speichern beim Datenimport

eine Funktion hinzugefügt werden. Dazu wird eine `@receiver` Funktion genutzt, die automatisch bei jedem Speichervorgang eines `HTLItemType` oder `HTLItem` Objekts aufgerufen wird:

```
@receiver(post_save, sender=HTLItemType)
def populate_htlitem_custom_field_values(sender, instance, **kwargs):
    populate_inheritants_custom_field_values(instance)

@receiver(post_save, sender=HTLItem)
def populate_htlitemtype_custom_field_values(sender, instance, **kwargs):
    populate_with_parents_custom_field_values(instance)
```

Die beiden angeführten Funktionen werden je beim Aufkommen eines `post_save` Signals [28] ausgeführt, dessen `sender` ein `HTLItemType` oder `HTLItem` ist. Die Funktionen rufen jeweils eine weitere Funktion auf, welche die *Custom-Fields* entsprechend aggregiert.

### 7.1.4 Datenimport

Um die Gegenstands- und Raumdaten des Inventars der HTL Rennweg in das erstellte System importieren zu können, muss dessen standardmäßig verfügbare Importfunktion entsprechend erweitert werden. Dazu sind 4 spezielle Importverhalten notwendig.

Implementiert wird das Importverhalten nicht innerhalb der entsprechenden Modellklasse, sondern in dessen verknüpften `ModelAdmin` Klasse. Durch das Attribut `resource_class` wird spezifiziert, durch welche Python-Klasse die Daten importiert werden. Um für ein einziges Modell mehrere `resource_class` Einträge zu setzen, müssen mehrere `ModelAdmin` Klassen für Proxy-Modelle [21] des eigentlichen Modells definiert werden. Ein Proxy-Modell eines Modells verweist auf dieselbe Tabelle in der Datenbank, kann aber programmiertechnisch als unabhängiges Modell betrachtet werden. Die Daten, die durch das Proxy-Modell ausgelesen oder eingefügt werden entsprechen exakt jenen des eigentlichen Modells.

```
# Definition eines Proxy-Modells zu dem Modell "HTLItem"
class HTLItemSecondaryImportProxy(HTLItem):
    class Meta:
        proxy = True

# Diese Datenbankabfragen liefern beide dasselbe Ergebnis:
print(HTLItem.objects.all())
print(HTLItemSecondaryImportProxy.objects.all())

# Für das Proxy-Modell kann eine ModelAdmin-Klasse definiert werden.
```

```
# Diese bekommt eine eigene "resource_class".
@register(HTLItemSecondaryImportProxy)
class HTLItemSecondaryImportProxyAdmin(HTLItemSecondaryImportMixin, RalphAdmin):
    resource_class = HTLItemSecodaryResource
```

Das Importverhalten wird in der Klasse, die in das Attribut `resource_class` eingetragen wird, programmiertechnisch festgelegt. Es werden alle Zeilen der zu importierenden Datei nacheinander abgearbeitet. Bei sehr großen Datenmengen oder aufwändigem Importverhalten kann es zu Performanceverlusten kommen. Da nahezu jedes durch das Diplomarbeitsteam erstellte Importverhalten die zu importierenden Daten überprüfen oder anderweitig speziell behandeln muss, kann es hier besonders zu Performanceengpässen kommen. Beispielsweise muss beim Import von Daten aus dem *SAP ERP* System auch geprüft werden, ob der Raum eines Gegenstandes existiert. Die oft sehr großen Datenmengen, die aus dem *SAP ERP* System importiert werden müssen, sorgen ebenfalls für Performanceverluste.

Die in den zu importierenden Dateien vorhandenen Überschriften werden vor dem Importprozess auf Modelleigenschaften abgebildet. Manche Werte der zu importierenden Datensätze müssen in Werte gewandelt werden, die in der Datenbank gespeichert werden können. In der Datei `import_settings.py` sind diese Abbildungen bzw. Umwandlungen als “*Aliases* ” deklariert. In dem folgenden Beispiel werden die Überschriften “Erstes Attribut” und “Zweites Attribut” auf die zwei Modelleigenschaften `field_1` und `field_2` abgebildet. Importierte Werte für “Zweites Attribut” werden von den Zeichenketten “Ja” und “Nein” auf die *Boolean*-Werte `True` und `False` übersetzt.

```
# Beispielhafte Definition von Aliases für einen Import
ALIASES_HTLITEM = {
    "field_1": (["Erstes Attribut"], {

    }),
    "field_2": (["Zweites Attribut"], {
      "Ja": True,
      "Nein": False
    }),
```

Der Datenimport wird immer zweimal durchlaufen. Zuerst werden die importierten Daten zwar generiert, aber nicht gespeichert und dem Benutzer nur zur Validierung vorgelegt. Nach einer Bestätigung des Benutzers werden die Daten ein weiteres Mal von Neuem generiert und gespeichert.

Details über das Format einer zu importierenden Quelldatei sind dem Handbuch zum Server zu entnehmen.

#### 7.1.4.1 Import aus dem SAP ERP System

Die Daten aus dem *SAP ERP* System der Schule können in ein gängiges Datenformat exportiert werden. Das üblich gewählte Format ist eine Excel-Tabelle (Dateiendung "`.xlsx`").

Die aus dem *SAP ERP* System exportierten Daten haben grundsätzlich immer volle Gültigkeit. Bereits im "Capentory" System vorhandene Daten werden überschrieben.

Zu importierende Datensätze werden anhand der Werte "BuKr", "Anlage" und "UNr." aus der Quelldatei verglichen. Diese Werte werden auf die `HTLItem` Modellattribute `company_code`, `anlage` und `asset_subnumber` abgebildet. Zusätzlich wird auch das `barcode_prio` Attribut mit dem zusammengefügten Wert der "Anlage" und "UNr." Felder [4] der Quelldatei verglichen. Stimmt ein `HTLItem` Datensatz aus dem "Capentory" System mit einem zu importierenden Datensatz aufgrund einer der beiden verglichenen Wertepaare überein, wird dieser damit assoziiert und überschrieben.

Vor dem Verarbeiten der Daten des `HTLItem` Objekts werden die Daten des zugehörigen `HTLRoom` Objekts verarbeitet. Es wird nach einem existierenden `HTLRoom` Objekt mit einem übereinstimmenden `room_number` Attribut[5] gesucht. Sollten mehrere `HTLRoom` Objekte übereinstimmen[6], wird jener mit übereinstimmenden `main_inv` und `location` Attributen [7] ausgewählt. Ein gefundenes `HTLRoom` Objekt wird mit dem importierten `HTLItem` Objekt verknüpft. Sollte kein `HTLRoom` Objekt gefunden werden, wird es mit den entsprechenden Werten erstellt. In beiden Fällen wird das `is_in_sap` *Boolean*-Attribut des `HTLRoom` Objekts gesetzt.

Es gibt eine Ausnahme der absoluten Gültigkeit der Daten aus dem *SAP ERP* System. Stimmt das gefundene `HTLRoom` Objekt nicht mit jenem aktuell verknüpften `HTLRoom` Objekt eines `HTLItem` Objekts überein, wird es grundsätzlich aktualisiert. Sollte das aktuell verknüpfte `HTLRoom` Objekt ein "Subraum" des gefundenen Objekts sein, wird dieses nicht aktualisiert. So wird verhindert, dass Gegenstände durch den Import aus einem Subraum in den übergeordneten Raum wandern. In dem *SAP ERP* System existieren die "Subräume" grundsätzlich nicht.

#### 7.1.4.2 Import aus sekundärer und tertiärer Quelle

Bei der sekundären und tertiären Quelle handelt es sich um schulinterne Inventarlisten. Beide Quellen enthalten Informationen über den Raum, in dem sich ein bestimmter Gegenstand befindet. Die sekundäre Quelle enthält Informationen über die Kategorie

[4] Diese Felder repräsentieren den Barcode eines `HTLItem` Objekts.
[5] Das "Raum" Feld der Quelldatei wird auf das `room_number` Attribut abgebildet.
[6] Dieser Fall sollte bei einem einzigen Schulstandort nicht auftreten.
[7] Die "Hauptinven" und "Standort" Felder der Quelldatei werden auf die `main_inv` und `location` Attribute abgebildet.

eines EDV-spezifischen Gegenstands. Die tertiäre Quelle enthält Informationen über “Subräume” und welche Gegenstände sich darin befinden.

Beide Quellen besitzen allerdings keine absolute Gültigkeit wie der Import aus dem *SAP ERP* System. Aus diesem Grund werden alle Änderungen von Eigenschaften eines `HTLItem` Objekts in Änderungsvorschläge einer Inventur ausgelagert und nicht direkt angewendet. Die Änderungen können zu einem späteren Zeitpunkt eingesehen, bearbeitet und schlussendlich angewendet werden. Grund für dieses spezielle Importverhalten ist die Vertrauenswürdigkeit der Informationen, die der sekundären bzw. tertiären Quelle entnommen werden. Die Listen haben offiziell kein einheitliches Format und können daher bei sofortigem Übernehmen der Änderungen zu ungewollten Fehlinformationen führen. Der Import einer sekundären oder tertiären Quelle kann wie eine eigenständige Inventur angesehen werden. Es können durch den Import auch neue `HTLItem` Objekte hinzugefügt werden, wenn ein Datensatz mit keinem bestehenden Objekt assoziiert werden kann.

Das Importverhalten für die sekundäre Quelle erstellt zusätzlich durch die Methode `get_or_create_item_type()` der Klasse `HTLItemSecodaryResource` definierte `HTLItemType` Objekte. Die erstellten `HTLItemType` Objekte werden den `HTLItem` Objekten indirekt über Änderungsvorschläge zugewiesen.

Die bereits erwähnten Informationen über “Subräume” der tertiären Quelle werden sofort auf die entsprechenden `HTLRoom` Objekte angewendet. Es bedarf keiner weiteren Bestätigung, um die “Subräume” zu erstellen und den übergeordneten `HTLRoom` Objekten zuzuweisen.

#### 7.1.4.3 Import der Raumliste

Die Importfunktion der Raumliste dient zur Verlinkung von interner Raumnummer (`HTLRoom`-Attribut `internal_room_number`) und der Raumnummer im *SAP ERP* System (`HTLRoom`-Attribut `room_number`). Es werden dadurch bestehenden `HTLRoom` Objekten eine interne Raumnummer und eine Beschreibung zugewiesen, oder gänzlich neue `HTLRoom` Objekte anhand aller erhaltenen Informationen erstellt. Der Import geschieht direkt, ohne Auslagerung von Änderungen in Änderungsvorschläge.

### 7.1.5 Datenexport

Die Daten aller `HTLItem` Gegenstände, die aus dem *SAP ERP* System importiert wurden können unter spezieller Verarbeitung exportiert werden. Die exportierten Daten können in das *SAP ERP* System importiert werden und beinhalten u.a. Raumänderungen, die durch Inventuren aufgetreten sind. Bei der Verarbeitung der zu exportierenden Daten wird eine Funktion des `reversion` Pakets [35] genutzt. Diese Funktion besteht

darin, den Zustand eines Gegenstandes zum Zeitpunkt des letzten Imports aus dem *SAP ERP* System abzufragen. Es wird der Zustand zu dem angegebenen Zeitpunkt mit dem aktuellen Zustand verglichen. Anhand der erkannten Änderungen wird die Export-Datei erstellt. Weitere Informationen zum Datenexport sind dem Handbuch zum Server zu entnehmen.

## 7.2 Das "Stocktaking" Modul

Das "*Stocktaking*" Modul ermöglicht das Erstellen und Verwalten von Inventuren. Um das Datenbankmodell möglichst modular und übersichtlich zu gestalten, werden diverse Modelle miteinander hierarchisch verknüpft. Die oberste Ebene der Modellhierarchie bildet das `Stocktaking` Modell. In der Abbildung 7.1 ist die Hierarchie in Form eines Klassendiagramms abgebildet. Das Klassendiagramm wurde mithilfe der Erweiterungen `django_extensions` und `pygraphviz` erstellt. Folgendes Kommando wurde dafür aufgerufen [36]:

```
dev_ralph graph_models stocktaking \
-X ItemSplitChangeProposal,MultipleValidationsChangeProposal,
   ValueChangeProposal,TimeStampMixin \
-g -o stocktaking_klassendiagramm.png
```

Um die *Subklassen* der Klasse `ChangeProposalBase` auszublenden, wurden diese mit der Option `-X` exkludiert.

### 7.2.1 Das Stocktaking Modell

Eine Inszanz des `Stocktaking` Modells repräsentiert eine Inventur.

#### 7.2.1.1 Die wichtigsten Attribute

Zu den wichtigsten Attributen des `Stocktaking` Modells zählen u.a.:

- `name`: Durch dieses Attribut kann eine Inventur benannt werden. Dieser Name erscheint auf der mobilen Applikation oder dem Inventurbericht.
- `user`: Dieses Attribut referenziert einen Hauptverantwortlichen Benutzer einer Inventur.
- `date_started` und `time_started`: Diese Attribute sind Zeitstempel und werden automatisch auf den Zeitpunkt der Erstellung einer `Stocktaking` Instanz gesetzt. Eine Inventur beginnt zum Zeitpunkt ihrer Erstellung.

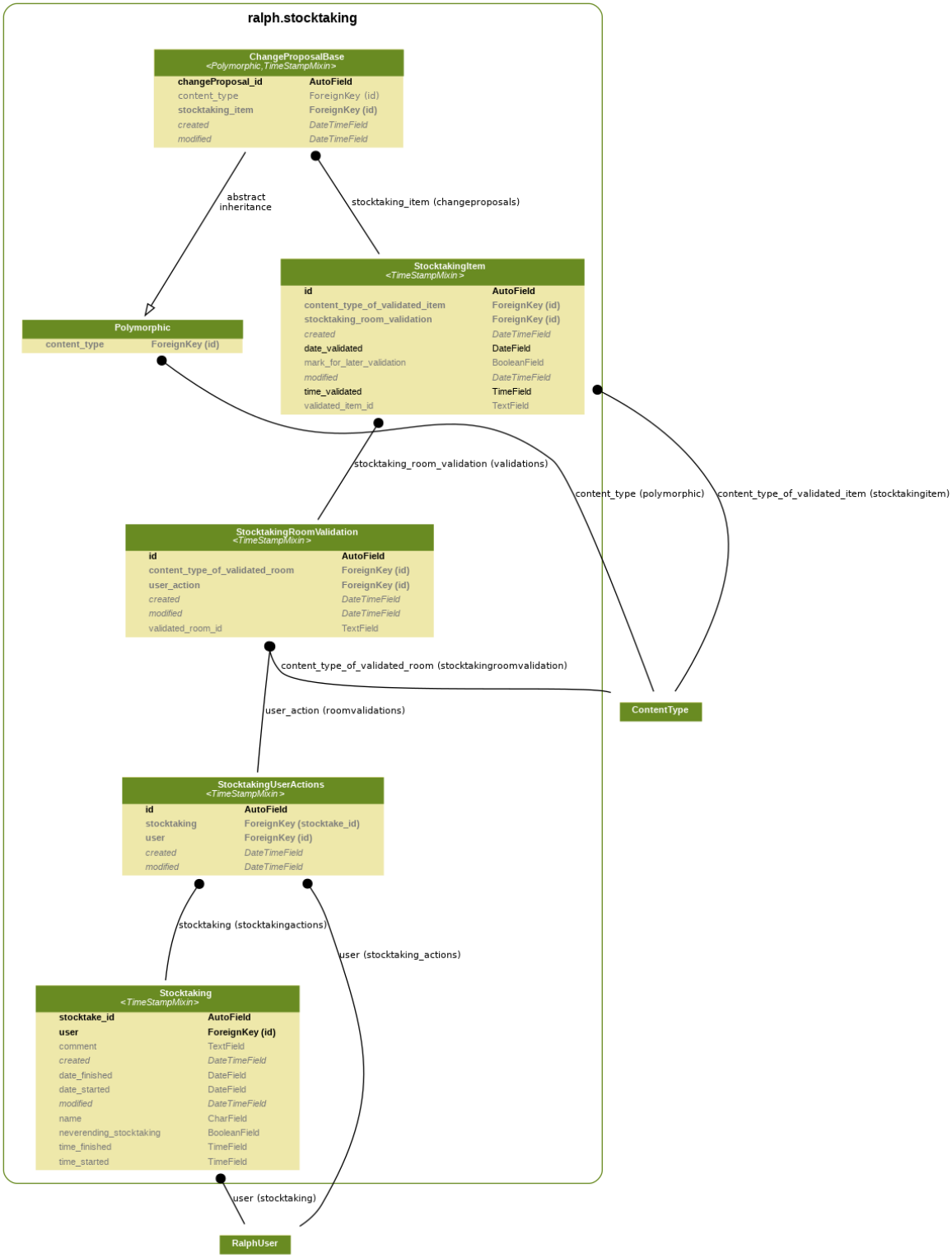



Abbildung 7.1: Das automatisch generierte Klassendiagramm der Modelle des "Stocktaking" Moduls.

- `date_finished` und `time_finished`: Diese Attribute sind Zeitstempel und werden gesetzt, wenn ein Administrator eine Inventur beenden möchte. Wenn beide Attribute mit einem Wert befüllt sind, werden keine über die mobile Applikation empfangene Validierungen verarbeitet. Der Zeitpunkt, der sich aus den beiden Attributen ergibt, darf nicht vor dem Zeitpunkt aus `date_started` und `time_started` liegen.
- `neverending_stocktaking`: Wenn dieses *Boolean* Attribut gesetzt ist, hat der Wert der `date_finished` und `time_finished` Attribute keine Bedeutung. Es werden alle Validierungen der mobilen Applikation verarbeitet. Ob in einem Raum bereits einmal validiert wurde oder die Inventur beendet ist, wird nicht mehr überprüft.

### 7.2.2 Das StocktakingUserActions Modell

Das `StocktakingUserActions` Modell ist die Verbindung zwischen einer Inventur und den Benutzern, die zu dieser Inventur beigetragen haben. Das Attribut `stocktaking` verlinkt je eine bestimmte `Stocktaking` Instant. Das Attribut `user` verlinkt je eine bestimmte `RalphUser` Instanz, die einem angemeldeten Benutzer entspricht. Validierungen, die ein Benutzer während einer Inventur tätigt, sind mit der entsprechenden `StocktakingUserActions` Instanz verknüpft. Pro Inventur kann es nur eine `StocktakingUserActions` Instanz für einen bestimmten Benutzer geben. Diese Bedingung wird durch die `unique_together` Metavariable [24] festgelegt:

```
unique_together = [["user", "stocktaking"]]
```

### 7.2.3 Das StocktakingRoomValidation Modell

Das `StocktakingRoomValidation` Modell ist die Verbindung zwischen einer `StocktakingUserActions` Instanz und einer bestimmten Raum-Instanz. Die Raum-Instanz kann dank der `GenericForeignKey` Funktionalität [18] von jedem beliebigen Modell stammen. In der Implementierung einer Client-Schnittstelle wird ein konkretes Modell spezifiziert. Zum Zweck der Inventur an der HTL Rennweg ist das `HTLRoom` Modell in der entsprechenden Client-Schnittstelle spezifiziert. Eine Instanz des `StocktakingRoomValidation` Modells repräsentiert das Validieren von Gegenständen in einem bestimmten Raum.

### 7.2.4 Das StocktakingItem Modell

Das `StocktakingItem` Modell repräsentiert die Validierung einer Gegenstands-Instanz während einer Inventur. Die Gegenstands-Instanz kann dank der `GenericForeignKey` Funktionalität [18] von jedem beliebigen Modell stammen. In der Implementierung einer Client-Schnittstelle wird ein konkretes Modell spezifiziert. Zum Zweck der Inventur an

der HTL Rennweg ist das `HTLItem` Modell in der entsprechenden Client-Schnittstelle spezifiziert. Durch die Beziehungen zu den Modellen `StocktakingRoomValidation` und dadurch zu `StocktakingUserActions` und `Stocktaking` ist erkennbar, in welchem Raum durch welchen Benutzer im Rahmen welcher Inventur ein Gegenstand validiert wurde. Durch das `ChangeProposalBase` Modell sind die Änderungsvorschläge einer Gegenstandsvalidierung verknüpft.

#### 7.2.4.1 Die wichtigsten Attribute

Zu den wichtigsten Attributen des `StocktakingItem` Modells zählen u.a.:

- `date_validated` und `time_validated`: Diese Attribute bilden den Zeitstempel der Gegenstandsvalidierung. Durch die Client-Schnittstelle erstellten `StocktakingItem` Instanzen wird der Wert der aktuellen Systemzeit zum Zeitpunkt des Datenempfangs zugewiesen. Wird ein Gegenstand mehrmals validiert[8], ist der Zeitstempel jener der jüngsten Validierung.
- `mark_for_later_validation`: Dieses *Boolean* Attribut dient zur erleichterten Verifikation der Validierungen durch einen Administrator. Ist das Attribut gesetzt, repräsentiert es eine unvertrauenswürdige Gegenstandsvalidierung. Das Attribut wird gesetzt, wenn der Benutzer durch die mobile Applikation angibt, sich bei einer Gegenstandsvalidierung nicht sicher zu sein und daher das "Später entscheiden" Feld setzt. Innerhalb des Inventur-Berichts werden `StocktakingItem` Instanzen, dessen `mark_for_later_validation` Attribut gesetzt ist, besonders markiert.

### 7.2.5 Änderungsvorschläge

Änderungsvorschläge beziehen sich auf eine bestimmte `StocktakingItem` Instanz. Ein Änderungsvorschlag ist eine Instanz einer *Subklasse* von `ChangeProposalBase`. Ein Änderungsvorschlag besagt, dass der aktuelle Zustand eines Datensatzes in der Datenbank nicht der Realität entspricht.

Um während einer Inventur nicht sofort jegliche erkannte Änderungen von Gegenstandseigenschaften anwenden zu müssen, werden sie in Änderungsvorschläge ausgelagert. So kann Fehlern, die während einer Inventur entstehen können, vorgebeugt werden. Dem Benutzer der mobilen Applikation wird dadurch Vertrauenswürdigkeit entzogen. Es ist die Aufgabe eines Administrators, Änderungsvorschläge als vertrauenswürdig einzustufen und diese tatsächlich anzuwenden.

[8] Dieser Fall tritt bei Inventuren auf, dessen `neverending_stocktaking` Attribut gesetzt ist. So kann derselbe Gegenstand Monate nach der ersten Validierung erneut während derselben Inventur validiert werden.

Als Basis für Änderungsvorschläge dient die Klasse `ChangeProposalBase`. Sie beschreibt nicht, welche Änderung während einer Inventur festgestellt wurde. Sie erbt von der Klasse `Polymorphic`. Die Klasse `Polymorphic` ermöglicht die Vererbung von Modellklassen auf der Datenbankebene. So können alle *Subklassen* der Klassen `ChangeProposalBase` einheitlich betrachtet werden. Über die Verbindung von `StocktakingItem` zu `ChangeProposalBase` in der Abbildung 7.1 sind nicht nur alle verknüpften `ChangeProposalBase` Instanzen, sondern auch alle Instanzen der *Subklassen* von `ChangeProposalBase`, verbunden. Eine *Subklassen* von `ChangeProposalBase` beschreibt eine Änderung, die von einem Benutzer während einer Inventur festgestellt würde. Es sind 4 *Subklassen* von `ChangeProposalBase` und damit 4 Arten von Änderungsvorschlägen implementiert:

1. `ValueChangeProposal`
2. `MultipleValidationsChangeProposal`
3. `ItemSplitChangeProposal`
4. `SAPExportChangeProposal`

Weitere Informationen über die Arten von Änderungsvorschlägen und deren Handhabung sind dem Handbuch zum Server zu entnehmen.

### 7.2.6 Die Client-Schnittstelle

Jede Art der Inventur benötigt eine eigene Client-Schnittstelle. Eine Art der Inventur unterscheidet sich von anderen durch das Gegenstands- und Raummodell, gegen welches inventarisiert wird. Ein Beispiel ist die Inventur für die Gegenstände der HTL Rennweg, durch die gegen die `HTLItem` und `HTLRoom` Modelle inventarisiert wird.

Um die Kommunikation zwischen Server und mobiler Client-Applikation für eine Art der Inventur zu ermöglichen, werden 2 `APIView` Klassen erstellt (siehe Kapitel 6 Abschnitt “Views”):

- Eine Klasse, die von der Klasse `BaseStocktakeItemView` erbt.
- Eine Klasse, die von der Klasse `BaseStocktakeRoomView` erbt.

In den *Subklassen* wird u.a. definiert, um welche Gegenstands- und Raummodelle es sich bei der Art der Inventur handelt. Die beiden vordefinierten Klassen `BaseStocktakeItemView` und `BaseStocktakeRoomView` wurden dazu konzipiert, möglichst wenig Anpassung für die Implementierung einer Art der Inventur zu benötigen. Um das Anpassungspotenzial allerdings nicht einzuschränken, sind die Funktionalitäten der Klassen möglichst modular. Eine Funktionalität wird durch eine Methode implementiert, die von den *Subklassen* angepasst werden kann. In dem unten angeführten Beispiel repräsentiert jede Methode eine Anpassung gegenüber dem von

`BaseStocktakeItemView` und `BaseStocktakeRoomView` definierten Standardverhalten.

Zu Demonstrationszwecken wurde eine voll funktionstüchtige Client-Schnittstelle für die Inventur des Gegenstandsmodells `BackOfficeAsset` und Raummodells `Warehouse` in weniger als 30 Code-Zeilen erstellt.

```
class BackOfficeAssetStocktakingView(StocktakingGETWithCustomFieldsMixin,
                                     ClientAttachmentsGETMixin,
                                     BaseStocktakeItemView):
    item_model = BackOfficeAsset
    pk_url_kwarg = None
    slug_url_kwarg = "barcode_final"
    slug_field = "barcode_final"

    display_fields = ("hostname", "status", "location", "price", "provider")
    extra_fields = (
        "custom_fields", "office_infrastructure", "depreciation_rate",
        "budget_info", "property_of", "start_usage"
    )
    explicit_readonly_fields = ("hostname", "price")

    def get_queryset(self, request=None, previous_room_validation=None):
        # annotate custom barcode as either "barcode" or "sn"
        return super(BackOfficeAssetStocktakingView, self).get_queryset(
            request=request, previous_room_validation=previous_room_validation
        ).annotate(
            barcode_final=Case(
                When(Q(barcode__isnull=True) | Q(barcode__exact=""),
                     then=(F("sn"))),
                default=F("barcode"), output_field=CharField()))

    def get_room_for_item(self, item_object):
        return str(item_object.warehouse)

    def item_display_name_getter(self, item_object):
        return str(item_object)

    def item_barcode_getter(self, item_object):
        return str(
            item_object.barcode_final
        ) or "" if hasattr(
            item_object, "id"
        ) else ""
```

```
class WarehouseStocktakingView(StocktakingPOSTWithCustomFieldsMixin,
                               ClientAttachmentsPOSTMixin,
                               BaseStocktakeRoomView):
    room_model = Warehouse
    item_detail_view = BackOfficeAssetStocktakingView()

    item_room_field_name = "warehouse"
```

Die Modelle `BackOfficeAsset` und `Warehouse` sind in dem unveränderten "Ralph" System vorhanden.

Die Klassen `StocktakingGETWithCustomFieldsMixin`, `StocktakingPOSTWithCustomFieldsMixin`, `ClientAttachmentsGETMixin` und `ClientAttachmentsPOSTMixin` ermöglichen das Miteinbeziehen von *Custom-Fields* und Änhängen (siehe Abschnitt "Speichern von Bildern und Anhängen"). Weitere Informationen zu den `Mixin`-Klassen kann der im Source-Code enthaltenen Dokumentation entnommen werden.

#### 7.2.6.1 Kommunikationsformat

Die Kommunikation zwischen Server und mobilem Client erfolgt über HTTP Abfragen. Das Datenformat wurde auf das JSON-Format [34] festgelegt. Grund dafür ist die weit verbreitete Unterstützung und relativ geringe Komplexität des Formats.

#### 7.2.6.2 Modell-Voraussetzungen

Um eine Client-Schnittstelle für die Inventarisierung bestimmter Gegenstands- und der dazugehörigen Raummodelle zu implementieren, müssen diese bestimmte Voraussetzungen erfüllen. Die Voraussetzungen werden in diesem Abschnitt beschrieben.

**Raum-Gegenstand-Verknüpfung** Das Gegenstandsmodell muss ein Attribut der Klasse `ForeignKey` besitzen, das auf das Raummodell verweist. Der Name dieses Attributs wird in der Klasse, die von `BaseStocktakeRoomView` erbt, in dem Attribut `item_room_field_name` angegeben. Zusätzlich kann in der von `BaseStocktakeItemView` erbenden Klasse durch die `get_room_for_item()` Methode der Raum einer Gegenstandsinstanz als *String* zurückgegeben werden. Ein Implementierungsbeispiel ist in der o.a. Client-Schnittstelle zu finden.

**String-Repräsentation** Gegenstands- und Rauminstanzen müssen eine vom Menschen lesbare Repräsentation ermöglichen. Standardmäßig wird dazu die `__str__()` Methode der jeweiligen Instanz aufgerufen. Das Verhalten kann durch das Überschreiben der Methoden `item_display_name_getter()` (aus der Klasse `BaseStocktakeItemView`) und `room_display_name_getter()` (aus der Klasse `BaseStocktakeRoomView`) angepasst werden.

**Barcode eines Gegenstandes** Es muss für eine Instanz des Gegenstandsmodells ein Barcode generiert werden können. Diese Funktion wird in der Methode `item_barcode_getter()` der von `BaseStocktakeItemView` erbenden Klasse definiert. Um über den von der mobilen Applikation gescannten Barcode auf einen Gegenstand schließen zu können, müssen die Attribute `slug_url_kwarg` und `slug_field` auf den Namen eines Attributes gesetzt werden, das den Barcode eines Gegenstandes repräsentiert. Dieses Attribut muss für jeden Datensatz, der durch die `get_queryset()` Methode entsteht, vorhanden sein. In dem o.a. Beispiel wird dieses Attribut in der `get_queryset()` Methode durch `annotate()` [26] hinzugefügt.

#### 7.2.6.3 Einbindung und Erreichbarkeit der APIView-Klassen

Um die für eine Art der Inventur implementierten Klassen über das *DRF* ansprechbar zu machen, werden diese in die Variable `urlpatterns` [30] eingebunden. Die Einbindung erfolgt beispielsweise in der Datei `api.py` eines beliebigen Pakets. Bei der Einbindung müssen den jeweiligen Schnittstellen zur späteren Verwendung interne Namen vergeben werden. Folgendes Implementierungsbeispiel bindet die oben definierten Klassen `BackOfficeAssetStocktakingView` und `WarehouseStocktakingView` ein:

```
urlpatterns = [
    url(
        r"^warehousesforstocktaking/(?:(?P<pk>[\w/]+)/)?$",
        WarehouseStocktakingView.as_view(),
        name="warehousesforstocktaking"),
    url(
        r"^backofficeassetsforstocktaking/(?:(?P<barcode_final>[\w/]+)/)?$",
        BackOfficeAssetStocktakingView.as_view(),
        name="backofficeassetsforstocktaking"),
]
```

Die Namen der Schnittstellen werden auf `"warehousesforstocktaking"` und `"backofficeassetsforstocktaking"` gesetzt. In der Einbindung der Klasse `BackOfficeAssetStocktakingView` wird definiert, dass der Name des Barcode-Attributes eines Gegenstandes `barcode_final` lautet. Das `barcode_final` Attribut

wird in der Definition von `BackOfficeAssetStocktakingView` hinzugefügt (siehe auch: Abschnitt “Barcode eines Gegenstandes”).

Die Klassen `BackOfficeAssetStocktakingView` und `WarehouseStocktakingView` sind dadurch über die *URLs* `/api/backofficeassetsforstocktaking/` und `/api/warehousesforstocktaking/` erreichbar.

Für die Implementierung der Inventur der HTL Rennweg wurden die *URLs* `/api/htlinventoryitems/` und `/api/htlinventoryrooms/` definiert.

#### 7.2.6.4 Statische Ausgangspunkte

Der mobilen Client-Applikation müssen notwendige Informationen der dynamisch erweiterbaren Inventuren und Inventurarten über statisch festgelegte *URLs* mitgeteilt werden:

**Verfügbare Inventuren** Der mobilen Client-Applikation muss mitgeteilt werden, welche Inventuren zurzeit durchgeführt werden können. Für das Modell `Stocktaking`ist eine *API*-Schnittstelle implementiert (siehe Kapitel 6 Abschnitt “API und DRF”). Über diese Schnittstelle können durch folgende Abfrage-URL mittels eines HTTP GET-Requests [33] alle verfügbaren Inventuren und dessen einzigartige *IDs* abgefragt werden:

```
/api/stocktaking/?date_finished__isnull=True&time_finish_isnull=True&format=json
```

Durch `&format=json` wird festgelegt, dass der Server eine Antwort im JSON-Format [34] liefert. Die Antwort des Servers auf die o.a. Abfrage sieht beispielsweise wie folgt aus (Die Daten wurden zwecks Lesbarkeit auf die wesentlichsten Attribute gekürzt.):

```
[
    {
        "stocktake_id": 1,
        "name": "Inventur 1",
        "comment": "Diese Inventur endet nie!",
        "neverending_stocktaking": true,
    },
    {
        "stocktake_id": 2,
        "name": "Inventur 2",
        "comment": "",
        "neverending_stocktaking": false,
```

```
    }
]
```

Laut der Antwort des Servers sind aktuell 2 Inventuren - “Inventur 1” und “Inventur 2” -verfügbar. Die mobile Client-Applikation benötigt den Wert des Attributs `"stocktake_id"`, um bei späteren Abfragen festzulegen, welche Inventur von einem Benutzer ausgewählt wurde.

**Verfügbare Arten der Inventur** Der mobilen Client-Applikation muss mitgeteilt werden, welche Arten der Inventur verfügbar sind und über welche *URLs* die jeweiligen `APIView` Klassen ansprechbar sind. Unter der *URL* `/api/inventoryserializers/` werden je eine Beschreibung der Inventurart und die benötigten *URLs* ausgegeben. Bei korrekter Implementierung der Inventurarten für die Gegenstandsmodelle `HTLItem` und `BackOfficeAsset` liefert der Server folgende Antwort im JSON-Format [34]:

```
{
    "HTL": {
        "roomUrl": "/api/htlinventoryrooms/",
        "itemUrl": "/api/htlinventoryitems/",
        "description": "Inventur der HTL-Items"
    },
    "BackOfficeAsset-Inventur": {
        "roomUrl": "/api/warehousesforstocktaking/",
        "itemUrl": "/api/backofficeassetsforstocktaking/",
        "description": "Demonstrations-Inventur fuer BackOfficeAssets"
    }
}
```

Um eine Art der Inventur in diese Ausgabe miteinzubeziehen, muss in der Datei `api.py` des “stocktaking” Pakets die Variable `STOCKTAKING_SERIALIZER_VIEWS` entsprechend erweitert werden. Dazu wird der o.a. Name der Schnittstelle verwendet, wie er in der Variable `urlpatterns` gesetzt wurde. Die zu dem angeführten Beispiel gehörige `STOCKTAKING_SERIALIZER_VIEWS` Variable ist wie folgt definiert:

```
STOCKTAKING_SERIALIZER_VIEWS = {
    "HTL": (
        "htlinventoryrooms",
        "htlinventoryitems",
        "Inventur der HTL-Items"
    ),
    "BackOfficeAsset-Inventur": (
        "warehousesforstocktaking",
        "backofficeassetsforstocktaking",
        "Demonstrations-Inventur fuer BackOfficeAssets"
```

```
    )
}
```

`"BackOfficeAsset-Inventur"` ist der Name der Inventurart für das Gegenstandsmodell `BackOfficeAsset`. `"warehousesforstocktaking"` ist der Name der Schnittstelle für die Klasse `WarehouseStocktakingView`. `"backofficeassetsforstocktaking"` ist der Name der Schnittstelle für die Klasse `BackOfficeAssetStocktakingView`. `"Demonstrations-Inventur fuer BackOfficeAssets"` ist eine Beschreibung der Inventurart.

#### 7.2.6.5 JSON Schema

Die Daten, die durch die implementierten `APIView` Klassen gesendet oder empfangen werden, müssen einer bestimmten Struktur folgen. In diesem Abschnitt werden die durch die Klassen akzeptierten HTTP-Methoden [33] aufgezählt und deren JSON Schema anhand mehrerer Beispiele dargestellt. Die Beispiele können zwecks Lesbarkeit gekürzt sein. Gekürzte Bereiche werden mit `[...]` markiert.

**`BaseStocktakeItemView` GET-Methode** Die Klasse `BaseStocktakeItemView` akzeptiert 2 unterschiedliche Abfragen der GET-Methode.

Eine Abfrage über die *URL* ohne weitere Zusätze liefert eine Liste aller Gegenstände und deren Eigenschaften:

`GET /api/htlinventoryitems/` liefert:

```
{
    "items": [{
            "itemID": 2,
            "displayName": "Tischlampe",
            "displayDescription": "",
            "barcode": "46010000",
            "room": "111 (Klassenraum)",
            "fields": {
                "anlagenbeschreibung": "Tischlampe",
                [...]
            },
            "attachments": []
        },
        [...]
    ]
}
```

Eine Abfrage über die *URL* mit Zusatz des Barcodes eines Gegenstandes liefert eine Liste aller Gegenstände mit dem entsprechenden Barcode und dessen Eigenschaften:

`GET /api/htlinventoryitems/46010000/` liefert:

```
{
    "items": [{
            "itemID": 2,
            "displayName": "Tischlampe",
            "displayDescription": "",
            "barcode": "46010000",
            "room": "111 (Klassenraum)",
            "fields": {
                "anlagenbeschreibung": "Tischlampe",
                [...]
            },
            "attachments": []
        }
    ]
}
```

**`BaseStocktakeItemView` OPTIONS-Methode** Der `OPTIONS` Request wird von der mobilen Client-Applikation benötigt, um ein Formular für Gegenstandsdaten zu erstellen. Dafür werden die Eigenschaften eines Gegenstandes nach Relevanz in `"displayFields"` (hohe Priorität; wichtige Eigenschaften) und `"extraFields"`(niedrige Priorität; unwichtige Eigenschaften) geteilt. Welche Eigenschaften welcher Kategorie zugeordnet werden ist in der Implementierung der `BaseStocktakeItemView` *Subklasse* durch die Variablen `display_fields` und `extra_fields` definiert. Jede Eigenschaft wird anhand folgender Kenngrößen beschrieben:

- `"verboseFieldName"`: Eine vom Menschen lesbare Beschreibung der Eigenschaft.
- `"readOnly"`: Dieses *Boolean* Feld ist `true`, wenn die Eigenschaft durch Eintragen in die `explicit_readonly_fields` Variable der `BaseStocktakeItemView` *Subklasse* als schreibgeschützt markiert wurde.
- `"type"`: Dieses Feld gibt die Art der Eigenschaft an. Beispiele sind `"string"`, `"boolean"`, `"integer"`, die jeweils eine Zeichenkette, *Boolean* oder ganzzahlige Nummer identifizieren. Besondere Eigenschaftsarten sind `"choice"` und`"dict"`.
- `"choices"`: Dieses Feld ist nur angeführt, wenn das `"type"` Feld den Wert `"choice"` hat. Es beinhaltet eine Liste aller möglichen Werte inkl. vom Menschen lesbare Repräsentation der Werte für die Eigenschaft.
- `"fields"`: Dieses Feld ist nur angeführt, wenn das `"type"` Feld den Wert `"dict"` hat. Es beinhaltet eine Kollektion an Eigenschaften, je mit eigenen `"verboseFieldName"`,

`"readOnly"` und `"type"` Feldern. Dadurch können Felder rekursiv verschachtelt werden. Ein Beispiel ist unten angeführt.

`OPTIONS /api/htlinventoryitems/` liefert:

```
{
    "displayFields": {
        "anlagenbeschreibung_prio": {
            "verboseFieldName": "interne Gegenstandsbeschreibung",
            "readOnly": false,
            "type": "string"
        },
        "item_type": {
            "verboseFieldName": "Kategorie",
            "readOnly": false,
            "type": "choice",
            "choices": [
                {
                    "value": null,
                    "displayName": "--- kein Wert ---"
                },
                {
                    "value": 1,
                    "displayName": "IT-Infrastruktur"
                },
                [...]
            ]
        },
        "comment": {
            "verboseFieldName": "Anmerkung",
            "readOnly": false,
            "type": "string"
        },
        "label": {
            "verboseFieldName": "Gegenstand benötigt ein neues Etikett",
            "readOnly": false,
            "type": "boolean"
        },
        [...]
    },
    "extraFields": {
        "anlagenbeschreibung": {
            "verboseFieldName": "SAP Anlagenbeschreibung",
            "readOnly": true,
            "type": "string"
```

```
        },
        "custom_fields": {
            "verboseFieldName": "Custom Fields",
            "readOnly": false,
            "type": "dict",
            "fields": {
                "Beispiel-Custom-Field + Auswahl": {
                    "verboseFieldName": "Beispiel-Custom-Field + Auswahl",
                    "type": "choice",
                    "readOnly": false,
                    "choices": [
                        {
                            "value": "A",
                            "displayName": "A"
                        },
                        {
                            "value": "B",
                            "displayName": "B"
                        },
                        {
                            "value": "C",
                            "displayName": "C"
                        }
                    ]
                },
                "Beispiel-Custom-Field": {
                    "verboseFieldName": "Beispiel-Custom-Field",
                    "type": "string",
                    "readOnly": false
                }
            }
        },
        "sponsor": {
            "verboseFieldName": "Sponsor",
            "readOnly": false,
            "type": "string"
        },
        "is_in_sap": {
            "verboseFieldName": "Gegenstand ist in der SAP Datenbank",
            "readOnly": true,
            "type": "boolean"
        }
    }
}
```

**`BaseStocktakeRoomView` GET-Methode** Die Klasse `BaseStocktakeRoomView` akzeptiert 2 unterschiedliche Abfragen der GET-Methode. Beide Abfragen benötigen die *ID* der ausgewählten Inventurinstanz als *URL* -Parameter `stocktaking_id`.

Eine Abfrage über die *URL* ohne weitere Zusätze liefert eine Liste aller Räume:

`GET /api/htlinventoryrooms/?stocktaking_id=1` liefert:


```
{
    "rooms": [{
            "roomID": 1,
            "displayName": "111",
            "displayDescription": "Klassenraum",
            "barcode": "11111111",
            "itemID": null
        }, {
            "roomID": 2,
            "displayName": "222",
            "displayDescription": "Labor",
            "barcode": "22222222",
            "itemID": null
        },
    ]
}
```


Eine Abfrage über die *URL* mit Zusatz der *ID* einer Rauminstanz[9] liefert die Details der angegebenen Rauminstanz inkl. aller Gegenstände, die sich in dem Raum befinden sollten, deren Eigenschaften und allen "Subräumen" der Rauminstanz:

`GET /api/htlinventoryrooms/1/?stocktaking_id=1` liefert:


```
{
    "roomID": 1,
    "displayName": "111",
    "displayDescription": "Klassenraum",
    "barcode": "11111111",
    "itemID": null,
    "subrooms": [],
    "items": [{
            "times_found_last": 1,
            "itemID": 2,
            "displayName": "Tischlampe",
            "displayDescription": "",
```


[9] Diese ID wird etwa der Antwort auf die Abfrage ohne URL-Zusatz entnommen. Die entsprechende Eigenschaft ist `"roomID"`

```
            "barcode": "46010000",
            "room": "111 (Klassenraum)",
            "fields": {
                "anlagenbeschreibung": "Tischlampe",
                [...]
            },
            "attachments": []
        }
    ]
}
```

**`BaseStocktakeRoomView` POST-Methode** Mit dieser Methode sendet die mobile Client-Applikation den aufgezeichneten Ist-Zustand der in einem Raum befindlichen Gegenstände. Eine Abfrage kann nur über die *URL* mit Zusatz der *ID* einer Rauminstanz[10] getätigt werden. Die *ID* der ausgewählten Inventurinstanz muss als Teil der übermittelten Daten im Feld `"stocktaking"` angegeben werden. Unter dem Feld `"validations"` werden alle validierten Gegenstände mit mindestens deren *ID* als Feld `"itemID"` angegeben. Zusätzlich können alle Gegenstandseigenschaften spezifiziert werden. Die Eigenschaften jedes Gegenstandes werden von der Klasse `BaseStocktakeRoomView` oder der davon erbenden Klasse verarbeitet und mit dem aktuell in der Datenbank eingetragenen Wert verglichen. Bei einer Differenz wird automatisch ein Änderungsvorschlag für diesen Gegenstand erstellt.

`POST /api/htlinventoryrooms/1/` mit folgenden Daten

```
{
    "stocktaking": 1,
    "validations": [
        {
            "itemID": 1
        },
        {
            "itemID": 2
        }
    ]
}
```

liefert die Antwort:

```
{
    "success:": true
}
```

[10] Diese ID wird etwa der Antwort auf die Abfrage ohne URL-Zusatz entnommen. Die entsprechende Eigenschaft ist `"roomID"`

Tritt ein Fehler während der Verarbeitung der Daten auf, wird der Zustand der Datenbank vor dem Empfangen der Daten wiederhergestellt. In diesem Fall teilt der Server dem Client in seiner Antwort mit, welcher Fehler aufgetreten ist.

`POST /api/htlinventoryrooms/1/` mit folgenden Daten

```
{
    "stocktaking": 100,
    "validations": [
        {
            "itemID": 1
        },
        {
            "itemID": 2
        }
    ]
}
```

liefert die Antwort:

```
{
    "errors": [
        "No stocktaking with ID 100"
    ]
}
```

Um eine Gegenstandsvalidierung zur erneuten Validierung zu einem späteren Zeitpunkt durch einen Administrator zu markieren, kann das Feld `"mark_for_later_validation"` auf `true` gesetzt werden:

```
{
    "stocktaking": 1,
    "validations": [
        {
            "itemID": 1,
            "mark_for_later_validation": true
        }
    ]
}
```

Dadurch wird die *Boolean*-Eigenschaft der erstellten `StocktakingItem` Instanz gesetzt (siehe Abschnitt “Das StocktakingItem Modell”).

Um einen Gegenstand in einem “Subraum” (siehe Abschnitt “Subräume”) zu validieren,

kann eine der folgenden 3 Maßnahmen gesetzt werden:

1. Eine separate POST-Abfrage auf die *URL* mit Zusatz der *ID* des Subraums mit den Daten der entsprechenden Gegenstände. Für einen `HTLRoom`-Subraum mit *ID* 100 ist die anzuwendende *URL* `/api/htlinventoryrooms/100/`.
2. Das Setzen des `"room"` Feldes auf die *ID* des Subraums innerhalb einer Gegenstandsvalidierung. Ein Beispiel bietet die folgende Abfrage. Die *ID* des Subraums ist 100. Die *ID* des übergeordneten Raumes ist 1.

`POST /api/htlinventoryrooms/1/` mit folgenden Daten:

```
{
    "stocktaking": 1,
    "validations": [
        {
            "itemID": 1,
            "room": 100
        }
    ]
}
```

3. Das Auslagern der Gegenstandsvalidierungen in das Feld `"subroomValidations"` wie in folgendem Beispiel. Das Ergebnis gleicht dem Beispiel aus Alternative 2:

`POST /api/htlinventoryrooms/1/` mit folgenden Daten:

```
{
    "stocktaking": 1,
    "validations": [
    ],
    "subroomValidations": [{
            "roomID": 100,
            "validations": [{
                    "itemID": 1
                }
            ],
            "subroomValidations": [
            ]
        }
    ]
}
```

`"subroomValidations"` können rekursiv definiert werden. In dem Beispiel aus Alternative 3 muss darauf geachtet werden, dass der Raum mit *ID* 100 direkter “Subraum”

des Raums mit *ID* 1 ist.

Weitere Details zur Funktionsweise und Anpassung der Client-Schnittstelle ist der im Source-Code enthaltenen Dokumentation zu entnehmen.

### 7.2.7 Pull-Request

Es wurde versucht, das `Stocktaking` Modul durch einen *Pull-Request* in das offizielle Ralph-System einzubinden. Durch der in den Abschnitten "Das StocktakingRoomValidation Modell" und "Das StocktakingItem Modell" behandelten `GenericForeignKey` Funktionalität [18] ist es möglich, das Stocktaking-Modul für jegliche Modellklassen zu verwenden. Es gibt keine Beschränkung auf die durch das Diplomarbeitsteam implementierten Klassen.

Die erstellte Lösung wurde im Entwicklungsforum der Ralph Plattform vorgestellt [39] [85] und ein Pull-Request auf der GitHub-Plattform wurde durchgeführt.

# 8 Einführung in die Infrastruktur

## 8.1 Technische Umsetzung: Infrastruktur

Um allen Kunden einen problemlosen Produktivbetrieb zu gewährleisten, muss ein physischer Server aufgesetzt werden. Auf diesem können dann alle Komponenten unseres Git-Repositories geklont und betriebsbereit installiert werden. Dafür gab es folgende Punkte zu erfüllen:

- das Organisieren eines Servers
- das Aufsetzen eines Betriebssystemes
- die Konfiguration der notwendigen Applikationen
- die Konfiguration der Netzwerkschnittstellen
- das Testen der Konnektivität im Netzwerk
- die Einrichtung des Produktivbetriebes der Applikation
- das Verfassen einer Serverdokumentation
- die Absicherung der Maschine
- die Überwachung des Netzwerks

In den folgenden Kapiteln wird deutlich gemacht, wie die oben angeführten Punkte, im Rahmen der Diplomarbeit “Capentory” abgearbeitet wurden.

### 8.1.1 Anschaffung des Servers

Den 5. Klassen wird, dank gesponserter Infrastruktur, im Rahmen ihrer Diplomarbeit ein Diplomarbeitscluster zur Verfügung gestellt. Damit können sich alle Diplomarbeitsteams problemlos Zugang zu ihrer eigenen virtualisierten Maschine verschaffen. Die Virtualisierung dieses großen Servercluster funktioniert mittels einer ProxMox-Umgebung.

#### 8.1.1.1 ProxMox

Proxmox Virtual Environment (kurz PVE) ist eine auf Debian und KVM basierende Virtualisierungs-Plattform zum Betrieb von Gast-Betriebssystemen. Vorteile:

- läuft auf fast jeder x86-Hardware
- frei verfügbar
- ab 3 Servern Hochverfügbarkeit

Jedoch liegen alle Maschinen der Diplomarbeitsteams in einem eigens gebaut und gesicherten Virtual Private Network (VPN), sodass nur mittels eines eingerichteten Tools auf den virtualisierten Server zugegriffen werden kann.

#### 8.1.1.2 FortiClient

FortiClient ermöglicht es, eines VPN-Konnektivität anhand von IPsec oder SSL zu erstellen. Die Datenübertragung wird verschlüsselt und damit der enstandene Datenstrom vollständig gesichert über einen sogenannten “Tunnel” übertragen.

Da die Diplomarbeit “Capentory” jedoch Erreichbarkeit im Schulnetz verlangt, muss die Maschine in einem Ausmaß abgesichert werden, damit sie ohne Bedenken in das Schulnetz gehängt werden kann. Dafür müssen folgende Punkte gewährleistet sein:

- Konfiguration beider Firewalls (siehe Punkt 8.1.7)
- Wohlüberlegte Passwörter und Zugriffsrechte

### 8.1.2 Wahl des Betriebssystems

Neben den physischen Hardwarekomponenten wird für einen funktionierenden und leicht bedienbaren Server logischerweise auch ein Betriebsystem benötigt. Die erste Entscheidung, welche Art von Betriebssystem für die Diplomarbeit “Capentory” in Frage kam wurde rasch beantwortet: Linux. Weltweit basieren die meisten Server und andere Geräte auf Linux. Jedoch gibt es selbst innerhalb des OpenSource-Hersteller zwei gängige Distributionen, die das Diplomarbeitsteam während deren Schulzeit an der Htl Rennweg kennenlernen und Übungen darauf durchführen durfte:

- Linux CentOS
- Linux Ubuntu

#### 8.1.2.1 Linux CentOS

CentOS ist eine frei verfügbare Linux Distribution, die auf Red Hat Enterprise Linux aufbaut. Hinter Ubuntu und Debian ist CentOS die am dritthäufigsten verwendete Software und wird von einer offenen Gruppe von freiwilligen Entwicklern betreut, gepflegt und weiterentwickelt.

#### 8.1.2.2 Linux Ubuntu

Ubuntu ist die am meist verwendete Linux-Betriebssystemsoftware für Webserver. Auf Debian basierend ist das Ziel der Entwickler, ein einfach zu installierendes und leicht zu bedienendes Betriebssystem mit aufeinander abgestimmter Software zu schaffen. Hauptsponsor des Ubuntu-Projektes ist der Software-Hersteller Canonical, der vom südafrikanischen Unternehmer Mark Shuttlerworth gegründet wurde.

#### 8.1.2.3 Vergleich und Wahl

**CentOS**

- Kompliziertere Bedienung
- Keine regelmäßigen Softwareupdates
- Weniger Dokumentation vorhanden

**Ubuntu**

- Leichte Bedienung
- Wird ständig weiterentwickelt und aktualisiert
- Zahlreich brauchbare Dokumentation im Internet vorhanden
- Wird speziell von Ralph empfohlen

Aus den angeführten Punkten entschied sich “Capentory” klarerweise das Betriebsystem Ubuntu zu verwenden, vorallem auch weil der Hersteller deren Serversoftware die Verwendung von diesem Betriebsystem empfiehlt. Anschließend wird die Installation des Betriebsystemes genauer erläutert und erklärt.

### 8.1.3 Installation des Betriebsystemes

Wie bereits unter Punkt Anschaffung des Servers (siehe 8.1.1) erwähnt, wird uns von der Schule ein eigener Servercluster mit virtuellen Maschinen zur Verfügung gestellt.

Durch die ProxMox-Umgebung und diversen Tools, ging die Installation der Ubuntu-Distribution rasch von der Hand. In der Virtualisierungsumgebung der Schule musste nur ein vorhandenes Linux-Ubuntu 18.04 ISO-File gemountet und anschließend eine gewöhnliche Betriebsysteminstallation für Ubuntu durchgeführt werden. Jedoch kam es beim ersten Versuch zu Problemen mit der Konfiguration der Netzwerkschnittstellen, die im nächsten Punkt genauer erläutert werden.

### 8.1.4 Konfiguration der Netzwerkschnittstellen

#### 8.1.4.1 Problematische Ereignise bei der Konfiguration der Schnittstellen

Um eine funktiorierende Internetverbindung zu erstellen, durfte die Konfiguration der Schnittstellen nicht erst "später durchgeführt" werden, da mit dem Aufschub der Schnittstellen-Konfiguration das Paket "NetworkManager" nicht installiert wurde. Der "NetworkManager" ist verantwortlich für den Zugang zum Internet und der Netzwerksteuerung auf dem Linuxsystem. Und da im Nachhinein dieses Paket nicht installiert war (und aufgrund fehlender Internetverbindung nicht installiert werden konnte), half auch die fehlerfreie Interface-Konfiguration nicht, um eine Konnektivität herzustellen. Dadurch musste ein zweiter Installationsdurchgang durchgeführt werden, worauf dann alles fehlerfrei und problemlos lief.

#### 8.1.4.2 Konfiguration

Im Rahmen des Laborunterrichts an der Htl Rennweg, bekamen die Schüler für diverse Unklarheiten ein sogenanntes Cheat-Sheet für Linux-Befehle zur Verfügung gestellt. In diesem Cheat-Sheet finden sich unteranderem Anleitungen für die Konfiguration einer Netzwerkschnittstelle auf einer CentOS/RedHat sowie Ubuntu/Debian-Distribution. Den Schülern der fünften Netzwerktechnikklasse sollte diese Kurzkonfiguration jedoch schon leicht von der Hand gehen, da sie diese Woche für Woche benötigen.

Eine Netzwerkkonfiguration mit statischen IPv4-Adressen für eine Ubuntu-Distribution könnte wie folgt aussehen:

In `/etc/network/interfaces`:

```
auto ens32
iface ens32 inet static
address 192.168.0.1
netmask 255.255.255.0
```

Eine Netzwerkkonfiguration mit Verwendung eines IPv4-DHCP-Servers für eine Ubuntu-Distribution könnte wie folgt aussehen:

In `/etc/network/interfaces`:

```
auto ens32
iface ens32 inet dhcp
```

#### 8.1.4.3 Topologie des Netzwerkes

Unter Abbildung 5.1 wird der Netzwerkplan veranschaulicht. Auf der linken Seite wird der Servercluster der Diplomarbeitsteams dargestellt, worauf die virtuelle Maschine von "Capentory" gehostet wird. In der Mitte ist die FortiGate-Firewall zu sehen, die nicht nur als äußerster Schutz vor Angriffen dient, sondern auch die konfigurierte VPN-Verbindung beinhaltet und nur berechtigten Teammitgliedern den Zugriff gewährleistet. Des Weiteren ist die moderne Firewall auch für die Konnektivität der virtuellen Maschine im Schulnetz zuständig, aber dazu später (unter Punkt "Absicherung der virtuellen Maschine") mehr.

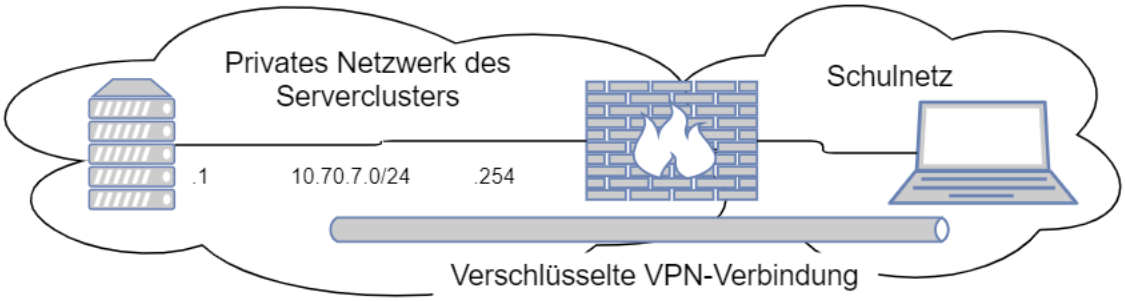



Abbildung 8.1: Netzwerkplan

#### 8.1.4.4 Testen der Konnektivität im Netzwerk

### 8.1.5 Installation der notwendigen Applikationen

Damit die Ubuntu-Maschine für den Produktivbetrieb startbereit ist, müssen im Vorhinein noch einige wichtige Konfigurationen durchgeführt werden. Die wichtigste (Netzwerkkonfiguration) wurde soeben ausführlich erläutert doch ohne der Installation von diversen Applikationen, wäre das System nicht brauchbar.

#### 8.1.5.1 Advanced Packaging Tool

Jeder Ubuntu Benutzer kennt es. Mit diesem Tool werden auf dem System die notwendigen Applikationen heruntergeladen, extrahiert und anschließend installiert. Insgesamt stehen einem 18 apt-get commands zur Verfügung. Genauere Erklärungen sowie die Syntax zu den wichtigsten commands folgen.

**apt-get update** Update liest alle in `/etc/apt/sources.list` sowie in `/etc/apt/sources.list.d/` eingetragenen Paketquellen neu ein. Dieser Schritt wird vor allem vor einem upgrade-command oder nach dem Hinzufügen einer neuen Quelle empfohlen, um sich die neusten Informationen für Pakete ansehen zu können.

Syntax: `[sudo] apt-get [Option(en)] update`

**apt-get upgrade** Upgrade bringt alle bereits installierten Pakete auf den neuesten Stand.

Syntax: `[sudo] apt-get [Option(en)] upgrade`

**apt-get install** Install lädt das Paket bzw. die Pakete inklusive der noch nicht installierten Abhängigkeiten (und eventuell der vorgeschlagenen weiteren Pakete) herunter und installiert diese. Außerdem besteht die Möglichkeit, beliebig viele Pakete auf einmal anzugeben, indem sie mittels eines Leerzeichens getrennt werden.

Syntax: `[sudo] apt-get [Option(en)] install PAKET1 [PAKET2]`

Falls eine bestimmte Version installiert werden soll:

```
[sudo] apt-get [Option(en)] install PAKET1=VERSION [PAKET2=VERSION]
```

**apt-get remove** Wie bereits erkannt, gibt es die Möglichkeit, sich ein beliebiges Paket zu installieren. Doch was soll geschehen falls dieses Paket nicht mehr benötigt wird? Daher gibt es prakitscherweise den remove-command, der, wie schon im Namen deutlich wird, ein oder mehrere Paket/e vollständig vom System entfernt.

Syntax: `sudo apt-get [Option(en)] remove PAKET1 [PAKET2]`

#### 8.1.5.2 Installierte Pakete mittels apt-get

**NGINX** NGINX ist der am Häufigsten verwendete OpenSource-Webserver unter Linux für diverse Webanwendungen. Große Unternehmen wie Cisco, Microsoft, Facebook oder auch IBM schwören auf die Verwendung dieses genialen Paketes. Unteranderem wird NGINX auch als Reverse-Proxy, HTTP-Cache und Load-Balancer verwendet. Wie genau NGINX für den Produktivbetrieb funktioniert wird im Laufe des Punktes 8.1.6.3 erläutert.

Installation: `[sudo] apt-get install nginx`

**Docker** Die OpenSource-Software Docker ist eine Containervirtualisierungstechnologie, die die Erstellung und den Betrieb von Linux Containern ermöglicht. Wie genau dies funktioniert, wird später unter Punkt 8.1.6.3 genauer beschrieben und erklärt.

Installation: `[sudo] apt-get install docker`

**docker-compose** Jeder Linux-Benutzer hat mindestens einmal in seinem Leben etwas über das `docker-compose.yml` File gehört. Doch was ist docker-compose eigentlich? Nun, die Verwaltung und Verlinkung von mehreren Containern kann auf Dauer sehr nervenaufreibend sein. Die Lösung dieses Problems nennt sich docker-compose. Wie docker-compose jedoch genau funktioniert, wird ebenfalls wie das Grundkonzept von Docker unter Punkt 8.1.6.3 präziser erläutert.

Installation: `[sudo] apt-get install docker-compose`

**MySQL** MySQL ist ein OpenSource-Datenverwaltungssystem und die Grundlage für die meisten dynamischen Websiten. Darauf werden die Inventurdatensätze der Htl Rennweg gespeichert. Nähere Informationen finden sich Punkt 8.1.6.3 wieder.

Installation: `[sudo] apt-get install mysql`

**Redis** Redis ist eine In-Memory-Datenbank mit einer Schlüssel-Wert-Datenstruktur (Key Value Store). Wie auch MySQL handelt es sich um eine OpenSource-Datenbank.

Installation: `[sudo] apt-get install redis`

**Nagios** Nagios ist ein Monitoring-System, mit dem sich verschiedene Geräte und auf solche laufende Dienste (oder auch Eigenschaften) überwachen lassen. Ziel ist es dabei

schnell Ausfälle festzustellen und diese dem zuständigen Administrator mit zu teilen, so dass dieser dann schnell darauf reagieren kann.

Installation: `[sudo] apt-get install nagios`

**virtualenv** Bei virtualenv handelt es sich um ein Tool, mit dem eine isolierte Python-Umgebung erstellt werden kann. Eine solch isolierte Umgebung besitzt eine eigene Installation von diversen Services und teilt ihre libraries nicht mit anderen virtuellen Umgebungen (im optionalen Fall greifen sie auch nicht auf die global installierten libraries zu). Dies bringt vor allem den großen Vorteil, dass im Testfall virtuelle Umgebungen aufgesetzt werden können, um nicht die globalen Konfigurationen zu gefährden.

Installation: `[sudo] apt-get install virtualenv`

**Python** Python ist einer der Hauptbestandteile auf dem Serversystem der Diplomarbeit "Capentory". Das Backend (=Serveranwendung) basiert wie bereits erwähnt auf dem Python-Framework "Django". Um dieses Framework auf dem System installieren zu können wird jedoch noch ein weiteres "Packaging-Tool", speziell für Python-Module, benötigt.

Installation: `[sudo] apt-get install python3.x`

#### 8.1.5.3 pip

Und dieses Tool nennt sich Pip. Pip ist ein rekursives Akronym für **P**ip **I**nstalls **P**ython und ist, wie bereits erwähnt, das Standardverwaltungswerkzeug für Python-Module. Die Funktion sowie Syntax kann relativ gut mit der von apt verglichen werden.

Installation von pip3 (vorrausgesetzt python3.x ist auf dem System bereits installiert):

```
[sudo] apt-get install python3-pip
```

#### 8.1.5.4 Installierte Pakete mittels pip3

**uWSGI** Das eigentliche Paket, mit dem der Produktivbetrieb schlussendlich gewährleistet wurde, nennt sich uWSGI. Speziell wurde es für die Produktivbereitstellung von Serveranwendungen (wie eben der Django-Server von Team "Capentory") entwickelt und harmoniert eindrucksvoll mit der Webserver-Software NGINX. Die grundlegende

Funktionsweise von uWSGI, sowie eine Erklärung, warum schlussendlich dieses Paket und nicht Docker verwendet wurde, wird unter Punkt 8.1.6 veranschaulicht.

Installation: `[sudo] pip3 install uwsgi`

**Django** Django ist ein in Python geschriebenes Webframework, auf dem unsere Serveranwendung basiert. Genauere Informationen wurden jedoch schon unter Punkt 6.1 übermittelt.

Installation: `[sudo] pip3 install Django`

**runsslserver** Runsslserver ist ein Python-Paket mit dem eine Entwicklungsumgebung über https erreichbar gemacht werden kann.

Installation: `[sudo] pip3 install runsslserver`

### 8.1.6 Produktivbetrieb der Applikation

Das Aufsetzen beziehungsweise die Installation der Produktivumgebung ist der wichtigste, aber auch aufwendigste, Bestandteil jeder Serverinfrastruktur. Eine Produktivumgebung soll von einer Testumgebung möglichst weit getrennt sein, damit die zu testende Software keinen Schaden für den produktiven Betrieb anrichten kann. Weiters soll durch den Produktivbetrieb der Server um einiges perfomanter sein, da dieser, im Falle der Diplomarbeit “Capentory”, einen optimierten Webserver (NGINX) verwendet.

#### 8.1.6.1 Entwicklungsumgebung

Die Entwickler des Grundservers “Ralph” haben eine eigene Entwicklungsumgebung für den Django-Server erstellt. Normalerweise findet sich in einem klassischen Python-Projekt oder einem Projekt, dass auf einem Python-Framework (wie zum Beispiel Django) basiert, ein sogenanntes “`manage.py`”-File. Hierbei handelt es sich um ein Script, dass das Management eines beliebigen Python-Projektes unterstützt. Mit dem Script ist man unter anderem in der Lage, den Webserver auf einem unspezifischen Rechner zu starten, ohne etwas Weiteres installieren zu müssen.

Jedoch wurde diese Datei im Projekt von “Ralph” und daher auch von “Capentory” in eine eigene spezifische Entwicklungsumgebung umimplementiert. Der Server startet sich nach der eigenen Installation (welche ab Seite 4 im Dokument "Serverdokumentation Schritt für Schritt erklärt wird) nicht mehr mittels:

```
python manage.py runserver
```

sondern mit:

```
dev_ralph runserver 0.0.0.0:8000
```

Obwohl die Befehle dieses Serverstarts nicht wirklich ident aussehen, führen sie im Hintergrund aber die gleichen Unterbefehle aus, um eine sichere Verwendung der Entwicklungsumgebung zu gewährleisten.

Da das Diplomarbeitsteam "Capentory" diesen Entwicklungsumgebung vorerst auf dem Produktivserver für Testzwecke verwendete, wurde ein eher unbekanntes Paket namens "runsslserver" für die Entwicklungsumgebung installiert und eingebaut. Zuerst musste, üblich um eine https-Verbindung einzurichten, beispielsweise mit dem Befehl

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048
-keyout /ssl/nginx.key -out /ssl/cert.crt
```

ein Zertifikat mit dem zugehörigen Schlüssel generiert werden.

Kurzerklärung der Befehlsoptionen

- `-x509` Gibt an, statt eines CSR gleich ein selbstsigniertes Zertifikat auszustellen.
- `-nodes` Zertifikat wird nicht über ein Kennwort geschützt. Damit kann der Server ohne weitere Aktion (Eingabe des Kennworts) gestartet werden.
- `-days` Beschreibt die Gültigkeit des Zertifikates für 365 Tage.
- `-newkey rsa:2048` Generiert das Zertifikat und einen 2048-bit langen RSA Schlüssel.
- `-keyout` Gibt die Ausgabepfad und -datei für den Schlüssel an.
- `-out` Gibt die Ausgabepfad und -datei des Zertifikates an.

Der Befehl "runsslserver" muss jetzt nur noch in den "Installed-Apps" des Projektes (im Falle von "Ralph" und "Capentory" befinden sich diese in der Datei "`base.py`") hinzugefügt werden.

Dann kann der Server problemlos mit

```
dev_ralph runsslserver 0.0.0.0:443
```

gestartet werden.

### 8.1.6.2 Probleme der Produktivumgebung

Der Webserver selbst kann für unsere Zwecke nicht mit Docker für eine Produktivumgebung bereitgestellt werden, da die bereits vorhandenen Dockerfiles von Ralph nur für deren Lösung entwickelt wurden (d.h. Ralph installiert deren Webserver sehr komplex wie in etwa mit einem „apt-get install ralph“ Befehl in diversen Skripten). Jedoch gibt es eine eigene Docker-Variante für die Entwicklungsumgebung die rasch mit uWSGI in eine Produktivumgebung umgewandelt werden kann.

### 8.1.6.3 Funktionsweise der Produktivumgebung

**Funktion von uWSGI** Es wurde bereits desöfteren erklärt warum die vorhandene “Ralph-Dockerlösung” für die Zwecke von “Capentory” nicht brauchbar sind (siehe 8.1.6.2). Jedenfalls wurde nun die Einrichtung der Produktivumgebung mittels uWSGI erfolgreich durchgeführt. In der folgenden Grafik wird die Funktion von uWSGI genauer dargestellt.

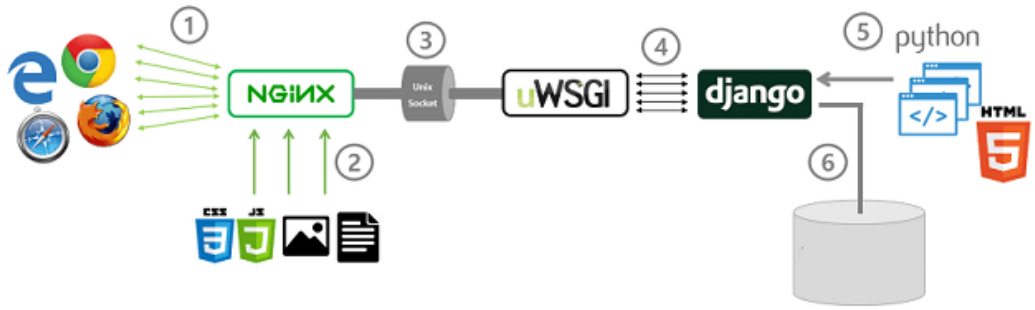



Abbildung 8.2: Funktionsweise von uWSGI

Da die meisten Serverbenutzer alleine mit der Grafik nicht wirklich viel anfangen können, folgt nun eine Erklärung dieser:

1. Ein beliebiger Benutzer eines Webbrowsers (zum Beispiel Google Chrome oder Mozilla Firefox) sendet einen sogenannten “Webrequest” auf den https-Port “443” (Abschnitt 8.1.6.2).
2. Ein beliebig gewählter, optimierter Webserver (im Fall von “Capentory” NGINX) stellt Dateien wie Javascript, CSS oder auch Bilder bereit und macht diese für den Benutzer abrufbar.

3. Hier wird die Kommunikation zwischen dem hochperfomanten Webserver NGINX und dem Webinterface uWSGI mit Verwendung eines klassischen Websockets veranschaulicht. (Anm.: Bei einem Websocket handelt es sich um ein auf TCP basierendes Protokoll, dass eine bidirektionale Verbindung zwischen einer Webanwendung und einem Webserver herstellt).
4. Das eigentliche Interface von uWSGi ist hier zu sehen. Diese Schnittstelle sorgt für die Kommunikation des oben genannten Websockets mit dem verwendeten Python-Framework.
5. Django reagiert nun auf die Anfrage des Benutzers und lässt diesem (falls dessen Zugriffsrechte darauf es erlauben) die gewünschten Daten.
6. Die vom Benutzer gewünschten Daten werden (bei "Capentory") in einer MySQL-Datenbank gespeichert.

Grundsätzlich kann die Grafik auch mittels

```
the web client <-> the web server <-> the socket <-> uwsgi <-> Django
```

als normale ASCII-Zeichenkette dargestellt werden.

**Funktion von Docker** Docker wird im Projekt "Capentory" ausschließlich für die Bereitstellung der Datenbank verwendet. Der Befehl

```
docker-compose -f docker/docker-compose-dev.yml up -d
```

im Projektstammverzeichnis erstellt und startet die in der folgenenden Grafik zu sehenden Container, die das Datenbanksystem darstellen:

#### 8.1.6.4 Erreichbarkeit mittels HTTPS

Wie es sich für eine Netzwerktechnikklasse gehört, wurde auch an die Erreichbarkeit über HTTPS gedacht. NGINX wurde mit einem sogenannten Self Signed Certificate ausgestattet, um eine sichere Verbindung des Benutzers mit dem Webserver zu gewährleisten. Die Konfiguration von NGINX für den Gebrauch von uWSGI mit HTTPS ist auf Seite 9 der Serverdokumentation zu finden. Im Webbrowser "Microsoft Edge" würde der Aufruf des Servers nun wie folgt aussehen:

Der Zertifikatsfehler, der am Bild deutlich zu sehen ist, bedeudet jedoch nichts anderes als dass das Zertifakt des Produktivservers von "Capentory" nicht von einer offiziellen Zertifizierungsstelle signiert wurde. Da der Server aber sowieso nur im Schulnetz der HTL 3 Rennweg beziehungsweise über einen VPN-Tunnel erreichbar ist, war es nicht nötig sich an solch eine Zertifizierungsstelle zu wenden.

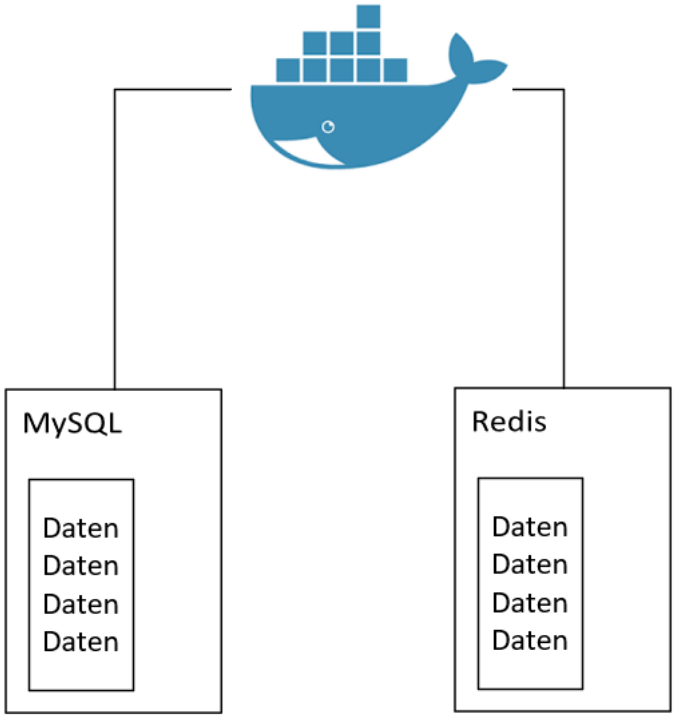



Abbildung 8.3: Datenbanksystem mit Docker

#### 8.1.6.5 Verwendung einer .ini-Datei

Bei einem File mit einer .ini-Endung handelt es sich um eine Initialisierungsdatei. Diese beinhaltet beispielsweise Konfigurationsmöglichen die zum Start eines bestimmten Dienstes benötigt werden. Glücklicherweise funktioniert so ein File auch wunderbar mit uWSGI. Durch die Implementierung einer ralph.ini-Datei wurden die Befehlsoptionen des `uwsgi`-Befehls ausgelagert und dieser somit für den Serverstart verkürzt.

Ein kleiner Vergleich:

```
uwsgi --socket mysite.sock --module mysite.wsgi --chmod-socket=664 --proceses 10
```

```
uwsgi --ini ralph.ini
```

Beide Befehle liefern schlussendlich das gleiche Ergebnis, jedoch ist der unter Befehl (Verwendung einer .ini-Datei) um einiges kürzer und erspart somit nervernaufreibende Schreibarbeit.

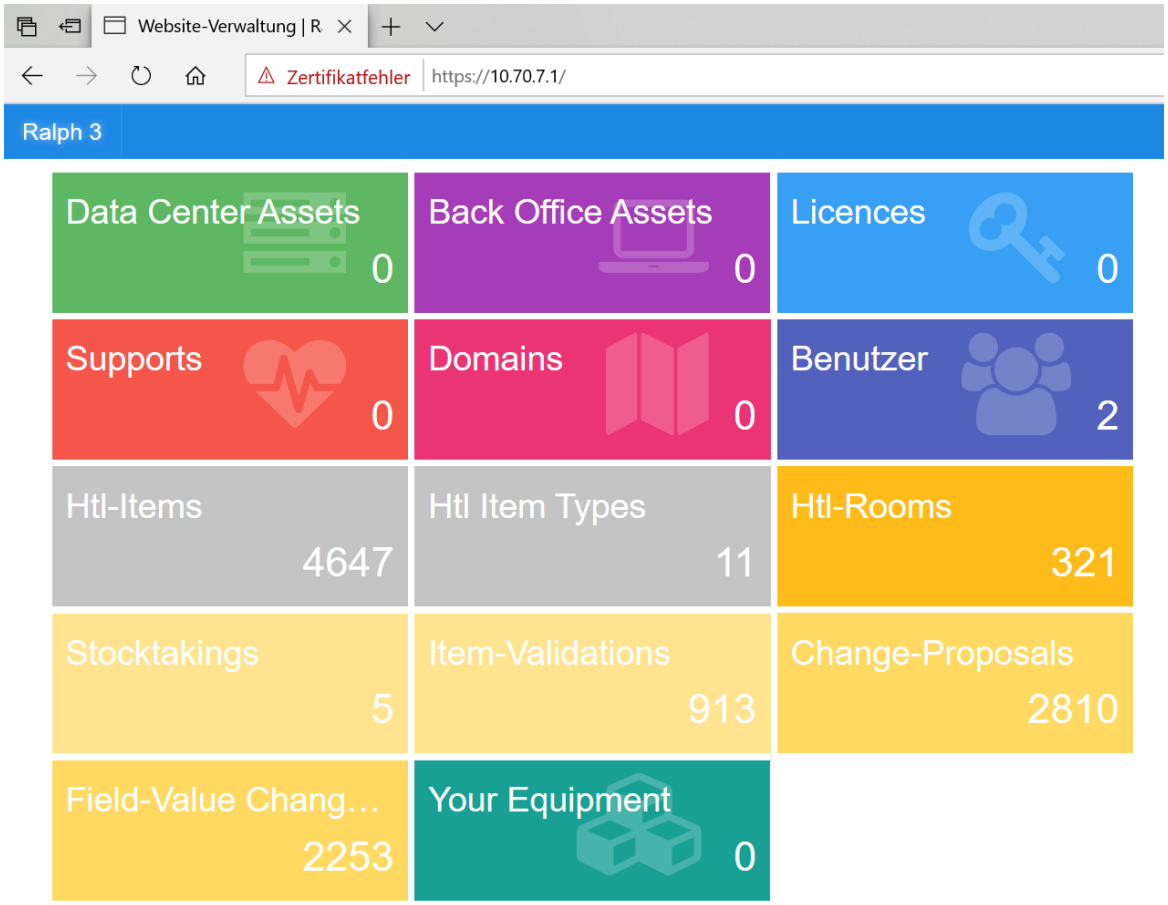



Abbildung 8.4: Aufruf des Servers über HTTPS

#### 8.1.6.6 Neustartverhalten mittels Service

Für den Gebrauch von uWSGi gibt es einige Lösungen, um den automatischen Neustart, beispielsweise bei Eintritt eines Stromausfalls, zu gewährleisten. “Capentory” hat sich, mit der Implementierung eines eigenen “Ralph-Service”, für den klassischen Linuxweg entschieden.

Nach dem korrekten Erstellen der `ralph.service`-Datei kann dieser als ganz normaler Linux-Service behandelt werden. Mit

```
systemctl enable ralph
systemctl start ralph
```

wird der frisch angelegte Service nun immer wenn die virtuelle Maschine gestartet wurde gestartet.

### 8.1.7 Absicherung der virtuellen Maschine

Ein weiterer wichtiger und sensibler Teil der Serverinfrastruktur ist die Absicherung der virtuellen Maschine gegen Angriffe. Da es sich im Rahmen der Diplomarbeit "Capentory" um geheime Daten der Schulinventur handelt, war es ein Anliegen der Verantwortlichen, dass mit diesen Daten verantwortungsvoll und vorsichtig umgegangen wird.

#### 8.1.7.1 Firewall

Unter Punkt 8.1.4.3 wird der Plan des Netzwerkes veranschaulicht. Die in der Mitte liegende FortiGate-Firewall stellt, wie bereits in oben genannten Punkt erwähnt, die Verfügbarkeit des Servers im Schulnetz bereit. Dadurch dass der Server nur über eine VPN-Verbindung konfiguriert werden kann, denkt man sich bestimmt, dass die virtuelle Maschine schon genug gesichert sei. Jedoch ist es sinnvoll den Server doppelt abzusichern und somit wurden auf der Linux-Maschine ebenfalls noch Firewallregeln konfiguriert.

**Erlauben einer SSH-Verbindung** Die virtuelle Maschine wurde aufgrund der nicht-vorteilshaften Konsole von ProxMox immer über SSH mit einer beliebigen externen Konsole (beispielsweise Putty) konfiguriert. Dies ist wegen der FortiGate-Konfiguration nur mit VPN-Verbindung möglich. Direkt auf dem Server wurde daher eine Firewallregel für die Erlaubnis von SSH implementiert.

Regel: `sudo ufw allow ssh`

oder: `sudo ufw allow 22`

**Erlauben des Datenaustausches mittels HTTP** Obwohl der Webserver den Datenaustausch mit HTTPS bevorzugt, wurde auf der zweiten Ebene auch eine Regel für die HTTP-Verbindung konfiguriert.

Regel: `sudo ufw allow http`

oder: `sudo ufw allow 80`

**Erlauben des Datenaustausches mittels HTTPS** Für den Datenaustausch über HTTPS musste ebenso eine Regel aktiviert werden.

Regel: `sudo ufw allow https`

oder: `sudo ufw allow 443`

**Verbieten aller restlichen Verbindungen** Zuguterletzt müssen alle restlichen (die nicht von Administratoren gebrauchten) Verbindungen deaktiviert werden um Angriffslücken zu schließen.

Regel: `ufw default deny`

#### 8.1.7.2 Mögliche Angriffsszenarien

Da der Server im Schulnetz erreichbar ist, gelten unteranderem auch die Schüler der HTL 3 Rennweg als potenzielle Angreifer.

**Malware** Bei Malware handelt es sich um Schadsoftware zudem unteranderem **Viren**, **Würmer** und **Trojaner** zählen.

**Angriffe auf Passwörter** Neben dem Raten und Ausspionieren von Passwörtern ist die Brute Force Attacke weit verbreitet. Bei dieser Attacke versuchen Hacker mithilfe einer Software, die in einer schnellen Abfolge verschiedene Zeichenkombinationen ausprobiert, das Passwort zu knacken. Je einfacher das Passwort gewählt ist, umso schneller kann dieses geknackt werden.

Vorbeugung von Team "Capentory": Die festgelegten Passwörter sind komplex aufgebaut und sicher in den Köpfen der Mitarbeiter gespeichert.

**Man-in-the-middle Attacken** Bei der „Man in the Middle"-Attacke nistet sich ein Angreifer zwischen den miteinander kommunizierenden Rechnern. Diese Position ermöglicht ihm, den ausgetauschten Datenverkehr zu kontrollieren und zu manipulieren. Er kann z.B. die ausgetauschten Informationen abfangen, lesen, die Weiterleitung kappen usw. Von all dem erfährt der Empfänger aber nichts.

Vorbeugung von Team "Capentory": Der Datenaustausch verläuft über HTTPS und somit verschlüsselt.

**Sniffing** Unter Sniffing (Schnüffeln) wird das unberechtigte Abhören des Datenverkehrs verstanden. Dabei werden oft Passwörter, die nicht oder nur sehr schwach verschlüsselt sind, abgefangen. Andere Angreife bedienen sich dieser Methode um

rausfinden zu können, welche Teilnehmer über welche Protokolle miteinander kommunizieren. Mit den so erlangten Informationen können die Angreifer dann den eigentlichen Angriff starten.

Vorbeugung von Team “Capentory”: Der Datenaustausch verläuft über HTTPS und somit verschlüsselt.

### 8.1.8 Überwachung des Netzwerks

Sollte der Fall eintreten, dass der Server nicht mehr erreichbar ist, wurde ein eigener Monitoring-Server im Netzwerk eingehängt und installiert. Dadurch wird der Produktivserver rund um die Uhr überwacht. Weiters wird das Diplomarbeitsteam “Capentory” bei etwaigen Komplikationen per E-Mail benachrigtigt, damit das aufgetretene Problem beziehungsweise die aufgetretenen Probleme möglichst schnell behoben werden können.

#### 8.1.8.1 Topologieänderung

Im Netzwerk wurde ein zweiter Server installiert, der mit dem OpenSource-Programm “Nagios” als Monitoring-Maschine für den Produktivserver dient.

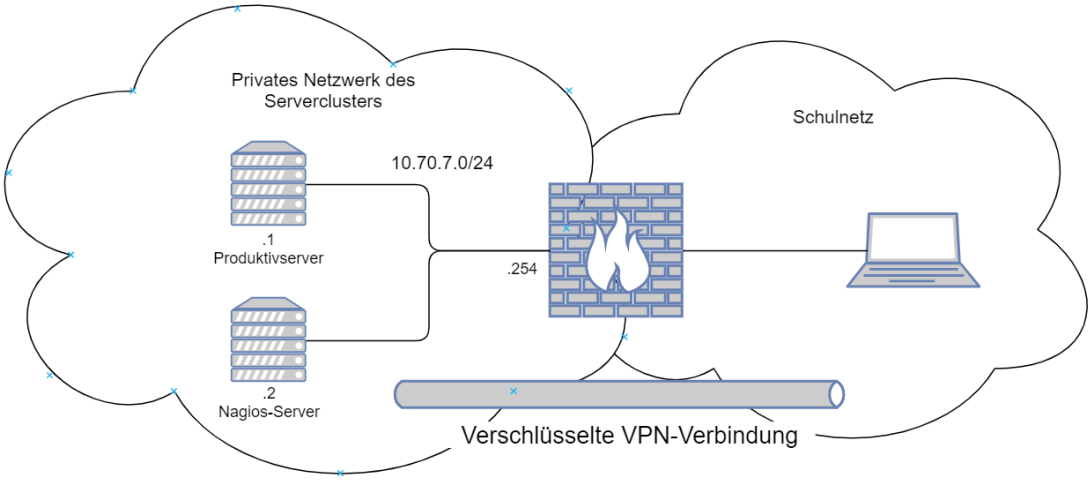



Abbildung 8.5: Ergänzter Netzwerkplan

#### 8.1.8.2 Nagios

Worum es sich bei diesem tollen Tool handelt wurde bereits unter Punkt 8.1.5.2 erklärt. Nun folgt ein vertiefender Einblick in diese Monitoring-Software.

**Hosts** Hosts sind bei Nagios definierte virtuelle Maschinen, die überwacht werden sollen. "Capentory" überwacht nicht nur den Produktivserver, sondern auch den aufgesetzten Nagios-Server selbst. Falls dieser Probleme aufweist, wird das Team ebenfalls per E-Mail benachrichtigt aber dazu unter Punkt 8.1.8.2 mehr.

Im Webbrowser sehen die zu überwachenden Hosts wie folgt aus:

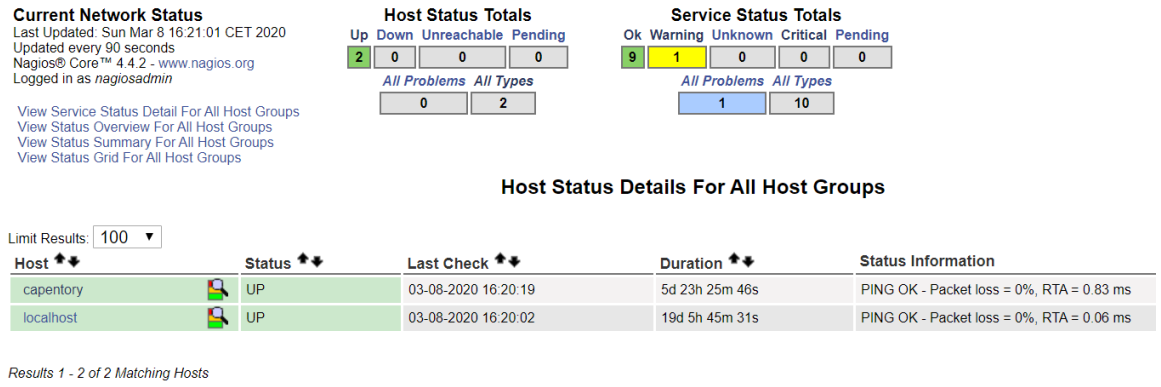



Abbildung 8.6: Die definierten Hosts der Diplomarbeit

**Capentory** ist der Name des Hosts des Produktivservers.

**Localhost** heißt der Host des Nagios-Servers.

Momentan scheint alles ohne Probleme zu funktionieren. Jedoch kann sich soetwas schlagartig ändern:

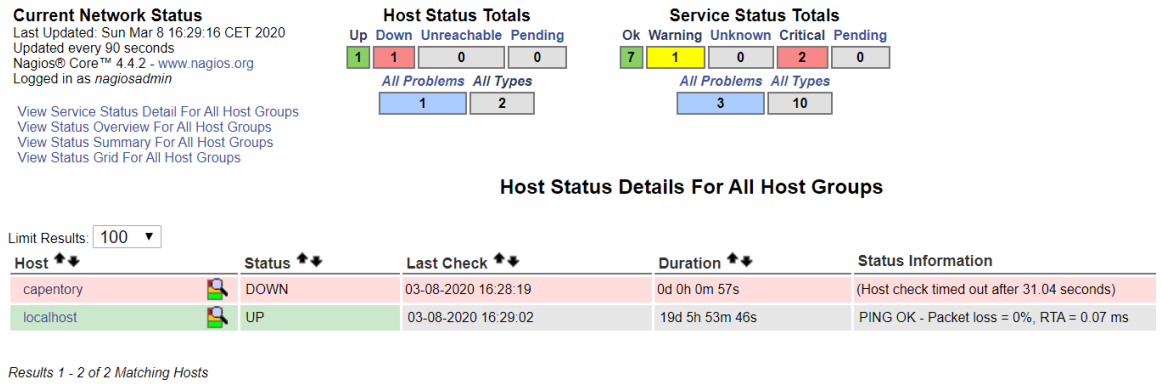



Abbildung 8.7: Der heruntergefahrene Produktivserver

Hier wird deutlich gemacht, dass der Produktivserver abgestürzt ist.

**Services** Nagios-Services sind, wie der Name schon verrät, die einzelnen zu überwachenden Services eines Hosts. Zurzeit wird am Nagios-Server aus Testzwecken eine Menge an Services überwacht. Auf dem Produktivserver hingegen werden nur Probleme,

die mit der Verbindung über Port 22 (SSH) oder Port 443 (HTTPS) zu tun haben, erfasst.

So sehen die überwachten Services am Admin-Dashboard des Nagios-Servers aus:

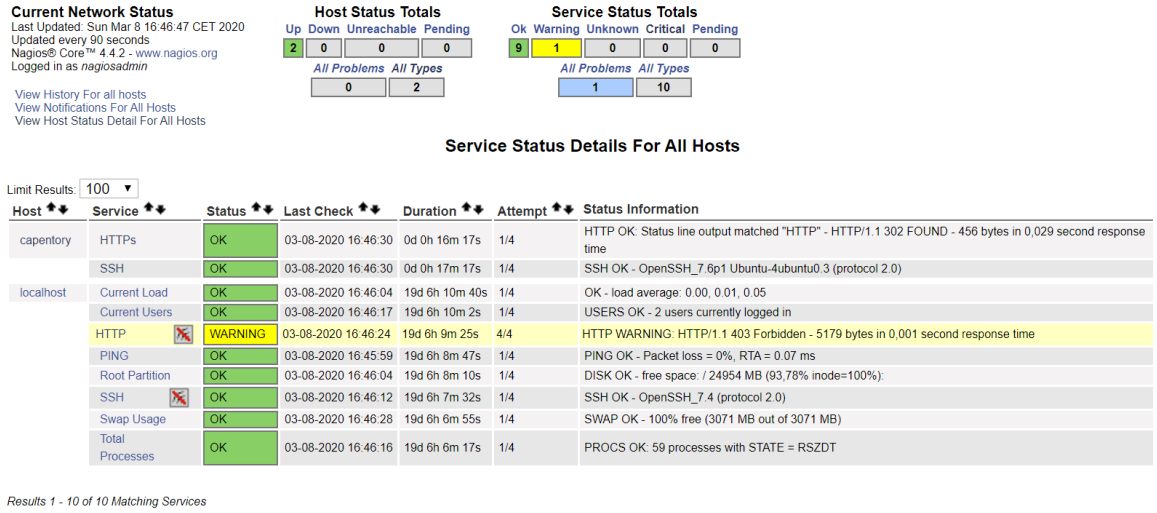



Abbildung 8.8: Die überwachten Services

Im Moment weist kein Service der beiden Hosts ein kritisches Problem auf. Auf dem Nagios-Server ist der HTTP-Service zwar gelb markiert, hierbei handelt es sich jedoch nur um eine harmlose Warnung.

Wenn auf dem Produktivserver hingegen der Webserver NGINX ausfällt sieht das Dashboard jedoch wiederum anders aus:

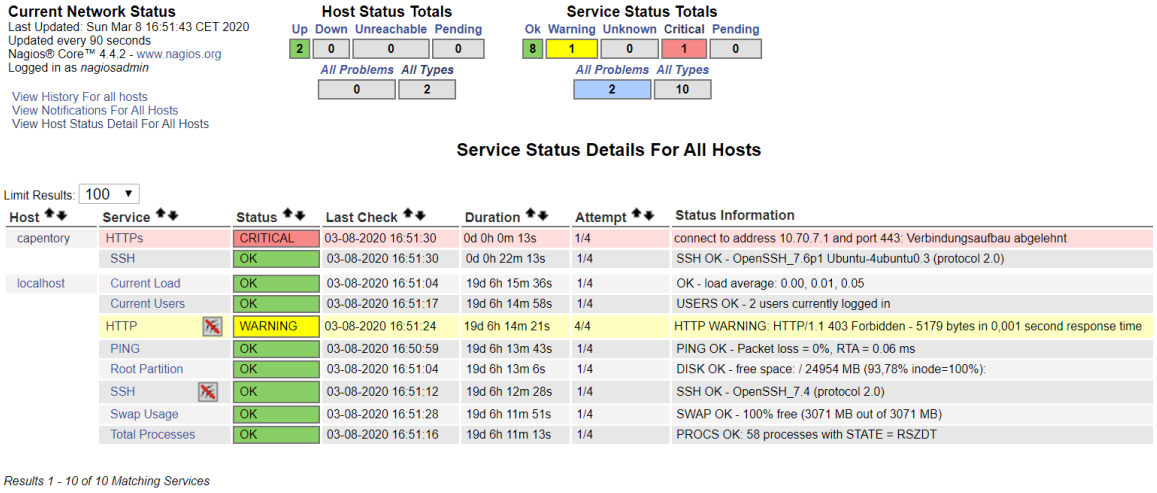



Abbildung 8.9: Die überwachten Services

**Notifications** Zuguterletzt gibt es noch das Feature der Notifications. Falls ein Host oder ein Service der Hosts ausfällt, benachrichtigt der Nagios-Server das Diplomar-

beitsteam per E-Mail über die aufgetretenen Probleme.

### 8.1.9 Verfassen einer Serverdokumentation

Im Laufe dieses Kapitels wurde der Inhalt von den für die Produktivumgebung zu erstellenden Dateien aufgelistet und Zeile für Zeile erklärt. Da nur mit diesen Dateien alleine jedoch kein Server funktionieren kann war ein weiteres großes Ziel die Erstellung einer Guideline für interessierte Benutzer, damit diese ebenfalls in der Lage sind, den Server der Diplomarbeit “Capentory” für eine Entwicklungsumgebung sowie eine Produktumgebung aufzusetzen.

# 9 Planung

# A Anhang 1

was auch immer: technische Dokumentationen etc.

Zusätzlich sollte es geben:

- Abkürzungsverzeichnis
- Quellenverzeichnis (hier: Bibtex im Stil plaindin)

# Index

# Literaturverzeichnis

[1] *Android Architecture Components.* https://developer.android.com/topic/libraries/architecture, Abruf: 2020-02-09

[2] *Android Volley Library.* https://developer.android.com/training/volley/request, Abruf: 2020-02-09

[3] *Anteile der weltweiten Androidversionen.* https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide, Abruf: 2020-01-08

[4] *Anteile der weltweiten mobilen Betriebssysteme.* https://gs.statcounter.com/os-market-share/mobile/worldwide, Abruf: 2020-01-08

[5] *App-Development Trends.* https://www.spinutech.com/dev/development/why-native-apps-are-dying/, Abruf: 2020-01-08

[6] *Artikel zu God Activity Architecture (Sarkastisch gehaltener Artikel!).* https://medium.com/@taylorcase19/god-activity-architecture-one-architecture-to-rule-them-all-62fcd4c0c1d5, Abruf: 2020-01-08

[7] *Begriffserkärung Native App.* https://de.ryte.com/wiki/Native_App, Abruf: 2020-01-08

[8] *Überblick von Django auf der offiziellen Website.* https://www.djangoproject.com/start/overview/, Abruf: 2020-01-01

[9] *Broadcasts - Offizielle Dokumentation.* https://developer.android.com/guide/components/broadcasts, Abruf: 2020-03-03

[10] *Callbacks.* https://stackoverflow.com/questions/824234/what-is-a-callback-function/7549753#7549753, Abruf: 2020-02-10

[11] *Configuration Changes - Offizielle Dokumentation.* https://developer.android.com/guide/topics/resources/runtime-changes, Abruf: 2020-02-10

[12] *Context - Offizielle Beschreibung von Android.* https://developer.android.com/reference/android/content/Context, Abruf: 2020-02-10

[13] *Datacenter-Insider Webseite mit Informationen über CMDB.* https://www.datacenter-insider.de/was-ist-eine-configuration-management-database-cmdb-a-743418/, Abruf: 2020-01-02

[14] *DataWedge - Offizielle Dokumentation.* https://www.zebra.com/content/dam/zebra_new_ia/en-us/solutions-verticals/product/Software/Mobility%20Software/datawedge/spec-sheet/dwandroid-specification-sheet-en-us.pdf, Abruf: 2020-03-03

[15] *Definition von "Syntax" im Duden.* https://www.duden.de/rechtschreibung/Syntax, Abruf: 2020-01-02

[16] *Demo-Webseite des Ralph-Systems von Allegro (Login-Daten: Benutzername: ralph/ Passwort: ralph).* https://ralph-demo.allegro.tech/, Abruf: 2020-01-02

[17] *Django Admin-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/ref/contrib/admin/, Abruf: 2020-01-01

[18] *Django ContentTypes-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/ref/contrib/contenttypes/, Abruf: 2020-02-07

[19] *Django Dateinamen-Nomenklatur.* https://streamhacker.com/2011/01/03/django-application-conventions/, Abruf: 2020-01-01

[20] *Django Datenbank-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/ref/databases/, Abruf: 2020-01-01

[21] *Django Datenbank-Modell-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/topics/db/models/, Abruf: 2020-01-01

[22] *Django Dokumentation von klassenbasierten Views (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/topics/class-based-views/, Abruf: 2020-01-02

[23] *Django Model-Instance-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/ref/models/instances/, Abruf: 2020-02-05

[24] *Django Model-Options-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/ref/models/options/, Abruf: 2020-01-01

[25] *Django Query-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/topics/db/queries/, Abruf: 2020-01-01

[26] *Django Queryset-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/ref/models/querysets/, Abruf: 2020-01-01

[27] *Django Security-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/topics/security/, Abruf: 2020-02-16

[28] *Django Signal-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/topics/signals/, Abruf: 2020-02-05

[29] *Django Template-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/topics/templates/, Abruf: 2020-01-01

[30] *Django URL-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/topics/http/urls/, Abruf: 2020-02-09

[31] *Django View-Dokumentation (Django Version 1.8).* https://docs.djangoproject.com/en/1.8/topics/http/views/, Abruf: 2020-01-02

[32] *Do not repeat yourself.* https://deviq.com/don-t-repeat-yourself/, Abruf: 2020-02-10

[33] *Dokumentation der HTTP Methoden.* https://restfulapi.net/http-methods/, Abruf: 2020-02-09

[34] *Dokumentation des JSON-Formats.* json.org/json-de.html, Abruf: 2020-02-09

[35] *Dokumentation von Django-Reversion (Version 1.8).* https://django-reversion.readthedocs.io/en/release-1.8/, Abruf: 2020-03-05

[36] *Dokumentation zu Django-Extensions: Graph-Models.* https://django-extensions.readthedocs.io/en/latest/graph_models.html, Abruf: 2020-01-02

[37] *Eingriff in den Lifecycle.* https://stackoverflow.com/questions/19219458/fragment-on-screen-rotation, Abruf: 2020-02-10

[38] *Enums sollten vermieden werden.* https://android.jlelse.eu/android-performance-avoid-using-enum-on-android-326be0794dc3, Abruf: 2020-02-10

[39] *Erster Beitrag im Entwicklungsforum von Ralph.* https://ralph.discourse.group/t/ralph-inventory-extension/131, Abruf: 2020-03-05

[40] *Fakten rund um Flutter*. https://en.wikipedia.org/wiki/Flutter_(software), Abruf: 2020-01-08

[41] *Flutter*. https://clearbridgemobile.com/mobile-app-development-native-vs-web-vs-hybrid/, Abruf: 2020-01-08

[42] *Flutter*. https://flutter.dev/, Abruf: 2020-01-08

[43] *Fragments - Offizielle Dokumentation*. https://developer.android.com/guide/components/fragments, Abruf: 2020-02-10

[44] *Generics - Offizielle Dokementation*. https://docs.oracle.com/javase/tutorial/java/generics/types.html, Abruf: 2020-03-14

[45] *Google - State Wrapper*. https://github.com/android/architecture-components-samples/blob/master/GithubBrowserSample/app/src/main/java/com/android/example/github/vo/Resource.kt, Abruf: 2020-02-10

[46] *Google unterstützt offiziel Single-Activity-Apps*. https://android-developers.googleblog.com/2018/05/use-android-jetpack-to-accelerate-your.html?m=1, Abruf: 2020-01-08

[47] *Internet-Posting über die Funtkion von REST-APIs*. https://www.cloudcomputing-insider.de/was-ist-eine-rest-api-a-611116/, Abruf: 2020-01-01

[48] *ISO-639*. https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes, Abruf: 2020-02-09

[49] *Java 8 in Android Studio*. https://developer.android.com/studio/write/java8-support, Abruf: 2020-02-10

[50] *Kotlin offiziel von Google präferiert*. https://www.heise.de/developer/meldung/I-O-2019-Googles-Bekenntnis-zu-Kotlin-4417060.html, Abruf: 2020-01-09

[51] *Lambdas in Java*. https://www.geeksforgeeks.org/lambda-expressions-java-8/, Abruf: 2020-02-10

[52] *LiveData - Offizielle Dokumentation*. https://developer.android.com/topic/libraries/architecture/livedata, Abruf: 2020-02-10

[53] *Mobile Vision API- Offizielle Dokementation*. https://developers.google.com/vision/android/barcodes-overview, Abruf: 2020-03-04

[54] *Mobile Vision API Text Recognition - Offizielle Dokementation.* https://developers.google.com/vision/android/text-overview, Abruf: 2020-03-04

[55] *MVVM Artikel.* https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata-part-1-604f50cda1, Abruf: 2020-02-09

[56] *Offiziele Google-Dokementation zu App-Architekturen.* https://developer.android.com/jetpack/docs/guide, Abruf: 2020-02-09

[57] *Offizielle Design-Grundlagen der Django-Entwicker.* https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style/, Abruf: 2020-01-02

[58] *Die offizielle Django-Website.* https://www.djangoproject.com/, Abruf: 2020-01-01

[59] *Offizielle Dokumentationsseite der Ralph Admin-Klasse von Allegro.* https://ralph-ng.readthedocs.io/en/stable/development/admin/, Abruf: 2020-01-02

[60] *Offizielle Dokumentationsseite der Ralph-API von Allegro.* https://ralph-ng.readthedocs.io/en/stable/development/api/, Abruf: 2020-01-02

[61] *Offizielle Dokumentationsseite des Paginierungs-Feature im Django Framework.* https://docs.djangoproject.com/en/3.0/topics/pagination/, Abruf: 2020-01-02

[62] *Offizielle Dokumentationsseite für Python Dekoratoren.* https://wiki.python.org/moin/PythonDecorators, Abruf: 2020-01-02

[63] *Die offizielle Flask-Website.* https://palletsprojects.com/p/flask/, Abruf: 2020-01-01

[64] *Offizielle Infopage des Django-REST Frameworks.* https://www.django-rest-framework.org/, Abruf: 2020-01-01

[65] *Die offizielle Pyramid-Website.* https://trypyramid.com/, Abruf: 2020-01-01

[66] *Die offizielle Web2Py-Website.* http://www.web2py.com/, Abruf: 2020-01-01

[67] *Die offizielle Website von "Ralph" des Unternehmens "Allegro".* https://ralph.allegro.tech/, Abruf: 2020-01-01

[68] *OWASP-Weiseite mit Informationen über CSRF.* https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF), Abruf: 2020-01-01

[69] *RecyclerView - Offizielle Dokementation.* https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView, Abruf: 2020-03-14

[70] *Retrofit vs. Volley.* https://medium.com/@sudhakarprajapati7/retrofit-vs-volley-c6cf74b3c8e4, Abruf: 2020-02-10

[71] *Singletons.* https://de.wikibooks.org/wiki/Muster:_Java:_Singleton, Abruf: 2020-03-15

[72] *Stackoverflow - Boilerplate Code.* https://stackoverflow.com/questions/3992199/what-is-boilerplate-code, Abruf: 2020-02-10

[73] *StackOverflow - State Wrapper.* https://stackoverflow.com/questions/44208618/how-to-handle-error-states-with-livedata, Abruf: 2020-02-10

[74] *TC56 Spezifikationen.* https://www.zebra.com/us/en/products/spec-sheets/mobile-computers/handheld/tc51-tc56.html, Abruf: 2020-03-03

[75] *ViewModel Anti-Patterns.* https://medium.com/androiddevelopers/viewmodels-and-livedata-patterns-antipatterns-21efaef74a54, Abruf: 2020-02-10

[76] *ViewModel Process Death.* https://developer.android.com/topic/libraries/architecture/viewmodel-savedstate, Abruf: 2020-02-10

[77] *ViewModels in Android.* https://developer.android.com/topic/libraries/architecture/viewmodel, Abruf: 2020-02-10

[78] *ViewPager2 - Offizielle Dokementation.* https://developer.android.com/guide/navigation/navigation-swipe-view-2, Abruf: 2020-03-04

[79] *WEBP.* https://developers.google.com/speed/webp, Abruf: 2020-03-04

[80] *Weiterer Vergleich Native vs. Hybrid vs. Web-App.* https://mlsdev.com/blog/native-app-development-vs-web-and-hybrid-app-development, Abruf: 2020-02-08

[81] *Wikipedia-Artikel zu MVVM.* https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel, Abruf: 2020-02-09

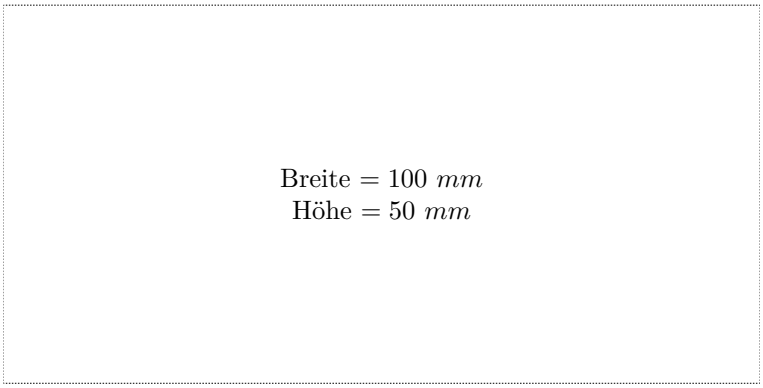
[82] *Xamarin.* https://apiko.com/blog/flutter-vs-xamarin-the-complete-2019-developers-guide-infographics-included/, Abruf: 2020-01-08

[83] *Xamarin - Offizielle Beschreibung.* https://dotnet.microsoft.com/learn/xamarin/what-is-xamarin, Abruf: 2020-02-09

[84] *Xamarin in Visual Studio.* https://visualstudio.microsoft.com/de/xamarin/, Abruf: 2020-02-09

[85] *Zweiter Beitrag im Entwicklungsforum von Ralph.* https://ralph.discourse.group/t/integration-of-ralph-inventory-extension/154, Abruf: 2020-03-05

— Druckgröße kontrollieren! —




— Diese Seite nach dem Druck entfernen! —

Diese Seite nach dem Druck entfernen!