HAW HAMBURG

Bachelorarbeit

Jannik Stoewing

Entwicklung von White Label Web Applikationen mit Hilfe eines modularen Architekturstils - Eine kritische Bewertung der Wiederverwendbark

Betreuung durch: Prof. Dr. Lars Hamann / Prof. Dr. Stefan Sarstedt
Eingereicht am: 11.01.2024

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Jannik Stoewing

Thema der Arbeit

Entwicklung von White Label Web Applikationen mit Hilfe eines modularen Architekturstils - Eine kritische Bewertung der Wiederverwendbarkeit

Keywords: Frontend, Architektur, White Label, Modular, Django

Kurzzusammenfassung

In dieser Arbeit wird anhand von Prototypen, welche auf einem modularen Architekturstil beruhen, geprüft, ob sich ein modularer Architekturstil eignet, um White Label Applikationen zu realisieren. Genutzt wird dafür das Python-Framework Django. Unter ausgewählten Bewertungskriterien werden die Prototypen anhand ihrer Integration in einen Webshop bewertet.

Jannik Stoewing

Title of Thesis

Development of white label web applications using a modular architectural style - A critical evaluation of reusability.

Keywords: Frontend, Architecture, White Label, Modular, Django

Abstract

In this thesis, prototypes based on a modular architecture style are used to test whether a modular architecture style is suitable for realizing White Label applications. The Python framework Django is used for this purpose. The prototypes are evaluated based on their integration into a web store using selected evaluation criteria.

Inhaltsverzeichnis

Glossar

- **Django** Django ist ein High-Level Python Web-Framework, welches das Erstellen von Webanwendungen vereinfachen soll. Durch eine strikte Trennung der Komponenten und Zuständigkeiten soll die Entwicklung vereinfacht werden und eine höhere Wiederverwendbarkeit und Skalierbarkeit gesichert werden.
- **HTTP Requests und Responses** HTTP (Hypertext Transfer Protocol) ist ein Kommunikationsprotokoll, welches in nahezu jeder Onlineanwendung genutzt wird. Requests sind Anfragen an eine Webanwendung, welche häufig mit einem sogenannten Body an Parametern gesendet werden. Als Antwort darauf erhält das System, oder der Nutzer eine HTTP Response, diese kann von einem einfachen Status Code, bis hin zu ganzen Webseiten alles enthalten.
- **Navbar** Eine Navbar ist ausformuliert die Navigation Bar auf einer Website. Diese ist Zuständig für die Navigation zwischen den Websiten zum Beispiel eines Shops und ist als Header auf fast jeder Seite identisch zu finden.

1 Einleitung

1.1 Motivation

Die Wahl der Architekturmethoden und -patterns wird immer wichtiger in der Entwicklung von jeder Art Applikationen heutzutage. Der Fokus liegt dabei häufig auf dem Backend, mit Microservices die Monolithen ablösen und Service Meshes, entwickelt sich die Architektur in diesen Bereichen stetig weiter. Auch im Frontend wird das Thema der Architektur immer wichtiger. Schon lange ist das Frontend kein einfaches HTML Konstrukt mehr, sondern viel Logik befindet sich mittlerweile im Frontend der heutigen Web-Applikationen. Je mehr Logik das Frontend aufnimmt, desto mehr ähnelt es dem Backend und benötigt auch eine Strukturierung durch verschiedene Architekturstile.
In demselben Atemzug werden Webanwendungen immer beliebter und gefragter. Während früher alle im Bus bezahlten, muss man sich heute an einigen Orten, wie zum Beispiel in Hamburg nach Berichten des NDR [Unk24], ab jetzt fast immer einer App bedienen. Verkehrsunternehmen brauchen Apps und Friseure brauchen Websites, alle Bereiche des Lebens welche früher analog funktionierten, sind dem Wandel der Digitalisierung ausgesetzt. Für viele Branchen ist es nun wichtig eine günstige, aber gute Lösung für diese Digitalisierung zu finden und in diesem Kontext kommt immer häufiger der Begriff White Label auf. Das Ziel ist es viele Produkte an unterschiedlichste Kunden zu bringen, ohne großen Aufwand bei der Konfigurierung. So können die Kunden ihr eigens gebrandetes Produkt an den Endnutzer geben und die Entwicklungsfirmen haben weniger Aufwand, da sie keine vielen individuellen Applikationen verwalten müssen, sondern ein Grundprojekt, welches für jeden Kunden nur konfiguriert wurde.

1.2 Zielsetzung der Arbeit

Das Ziel dieser Arbeit ist es, anhand von unterschiedlichen modularen Prototypen festzustellen, ob sich ein modularer Architekturstil eignet, um White Label Applikationen zu realisieren. Genutzt wird dafür in dieser Arbeit das Framework Django. Das Fazit am Ende der Arbeit soll eine Auskunft für zukünftige Projekte geben, ob modulare Web Applikationen ein guter Weg sind, um White Label Apps umzusetzen.

1.3 Gliederung

Diese Arbeit besteht aus 7 Kapiteln. Im aktuellen Kapitel, der Einleitung, wurde die Motivation dieses Themas beschrieben, außerdem wurde die Zielsetzung der Arbeit erklärt und der Aufbau.
Darauf folgt das Kapitel der Grundlagen, welches näher auf den Begriff White Label eingeht. Des Weiteren wird ein beliebtes Frontend-Architekturpattern beschrieben, welches in der Arbeit zum Einsatz kommen wird. Als letzten Punkt der Grundlagen wird das Framework beschrieben, in welchem die Prototypen dieser Arbeit geschrieben wurden und eine Architekturabweichung vom zuvor beschriebenen Pattern in diesem Framework.
Das dritte Kapitel enthält die Bewertungskriterien, anhand welchen später die Integration der Module in den Hauptprototypen bewertet werden.
Im vierten Kapitel werden nun alle Prototypen detailliert vorgestellt. In der Einleitung des Kapitels wird der grobe Plan beschrieben und in den Prototyp spezifischen Unterkapiteln danach, wird jeder Prototyp im Detail beschrieben mit allen seiner Besonderheiten.
Das fünfte Kapitel wird die Zusammenführung der Prototypen behandeln. Die Architektur des Gesamtprototyps wird jeweils beachtet und es wird eine Schritt-für-Schritt-Anleitung erstellt, welche den Integrationsprozess der Module in das Hauptprojekt beschreibt.
Diese Integration ist die Grundlage für die Bewertung in Kapitel sechs. Beide Integrationen werden anhand der Bewertungskriterien unabhängig voneinander bewertet.
Das Fazit, ob soeben dieser Prototyp durch seine modulare Architekturweise und das Framework Django eine funktionale White Label Applikation geworden ist, wird anhand der Bewertung im letzten Kapitel erstellt. Danach folgt noch ein Ausblick auf die Möglichkeiten von modularen Applikationen und Django als Framework in diesem Bereich.

2 Grundlagen

2.1 White Label

Der Begriff White Label ist in der Softwareentwicklung nicht einheitlich definiert, doch gibt es Paradigmen, welche den Begriff verinnerlichen. Grundlegend kann ein White-Label-Produkt als ein Produkt beschrieben werden, welches vor dem Verkauf an unterschiedliche Kunden auf deren Wünsche angepasst wird. Somit hat jeder Kunde das gleiche Produkt bekommen, außer das dieses an das Branding und Features der Kunden angepasst wurde. Zwei Konzepte in der Softwareentwicklung passen mit dem Begriff White Label wie man ihn im Alltag kennt zusammen, das sind einmal Highly Configurable Systems (HCS) und Software Product Lines (SPL).

HCS konzentrieren sich auf die Möglichkeit ein System fein-granular anzupassen und zu konfigurieren. Um dieses Ziel im Laufe einer Projektarbeit zu erreichen werden zuerst alle variablen und nicht-variablen Punkte identifiziert und dadurch das System vom Entwurf bis zur Implementierung geplant. Diese variablen Punkte finden sich später in sogenannten Features wieder. Die Stakeholder haben im ersten Schritt beschrieben, was das System können soll und wo es konfigurierbar sein soll, danach kann das Entwicklungsunternehmen weitere konfigurierbare Punkte bestimmen und somit Design und Implementierungen bestimmen, welche im Endprodukt leicht veränderbar sind. Über die Zeit haben sich einige Definitionen für Features durchgesetzt, sie werden von unter anderem Apel [ABKS16] als vom Nutzer sichtbare Merkmale oder Verhaltensweisen definiert, nach Batory [Bat05] kann man sie aber auch einfach als Ergänzung der Systemfunktionalität bezeichnen.

Die Einbindung von Features passiert in HCS in 3 unterschiedlichen Phasen:

- Kompilierzeit: Features müssen ein und ausgeschaltet werden, oder anders bestimmt werden, bevor das System in Betrieb genommen wird, zum Beispiel in Datenbanken oder im Code selbst.
- Ladezeit: Während des Systemstarts können Features durch Secrets, Umgebungsvariablen und Keys berechnet werden.
- Laufzeit: Änderungen sind möglich, während das System bereits läuft und werden auf das laufende System angewendet. Benutzer konfigurierbare Systeme ermöglichen Änderungen zur Laufzeit durch Tools wie Konfigurationsdateien. Die NASA

beispielsweise verwendet ein Modell zum Aktivieren/Deaktivieren von Modulen auf der Grundlage des Missionsstatus ohne menschliches Eingreifen.

Software Product Lines (SPL) sind eine Entwicklungsmethode für Softwareprodukte in welchem Techniken zur Code-Wiederverwendung genutzt werden. Hierbei bilden wiederverwendbare Artefakte die Grundlage. Ein Kernprodukt soll einen einheitlichen Kern haben und durch verschiedenste aus und Einkopplungen veränderbar sein. Auch in diesem Konzept soll das System für jeden Kunden speziell anpassbar sein.

Zwischen SPL und HCS ist somit ein starker Zusammenhang zu erkennen und man kann sagen, dass SPL ein Teil von HCS als übergeordnetes Pattern sind. Während Highly Configurable Systems als Kernaspekt das Konfigurieren von einem System beschreibt, erreichen Software Product Lines genau dieses durch die Auslagerung von Code und dessen Wiederverwendung in anderen Produktzusammenstellungen. Diese modulare Herangehensweise ist eine Art wie Features in ein System zur Kompilierzeit eingebracht werden können.

2.2 Frontendarchitekturen

2.2.1 Model View Controller

Um modulare Anwendungen zu entwickeln und die Entwicklung von diesen einfacher zu gestalten, ist eine strenge Abtrennung der Komponenten wichtig. Das Model-View-Controller (MVC) Pattern bietet genau diese Abtrennung. In diesem Pattern wird zwischen drei Ebenen entschieden, welche alle unterschiedliche Aufgaben übernehmen. Der ideale Ablauf einer Anfrage an eine Applikation, welche auf dem MVC Pattern basiert, würde folgendermaßen ablaufen:

1. Ein Nutzer stellt einen Request an den Controller
2. Der Controller verarbeitet diesen Request und leitet ihn an die View weiter
3. Die View nutzt die Models, um eine Response an den Nutzer zu kreieren
4. Die View schickt eine Response an den Nutzer, welche den aktuellen Status der Models widerspiegelt

Die Ebenen werden hier als einzelne Komponenten bezeichnet, um den Ablauf einfacher zu erklären, in Wahrheit können die Ebenen allerdings aus mehreren Komponenten, Objekten, Services bestehen.
Wie Deacon [Dea09] es schon beschrieb, kann die Model-Ebene in diesem Pattern in zwei Teile aufgeteilt werden. Ein Teil welcher rein die Klassen und internen Funktionen enthält und sprichwörtlich von der Außenwelt abgeschottet ist. Und ein weiterer Teil welcher zwar nichts von den direkten Abläufen auf den anderen Ebenen weiß, aber eine Art Interface bieten kann, um Objekte der Modelle zu kreieren, verwalten und zu löschen.
Wie im Ablauf oben beschrieben ist die View-Ebene für die grafische Repräsentation der Anfrage gedacht. Diese kann je nach Anwendung natürlich über verschiedene Wege erreicht werden, zum Beispiel über tatsächliche Websites mit grafischen User Interfaces, aber auch unter anderem auch über das Command Line Interface. Während die Views die Repräsentation vorbereiten, sind diese es auch welche die Objekte über das Model-Interface bearbeiten und in eine Form bringen, wie der Nutzer sie sehen kann. Um aber zu wissen, was der Nutzer möchte, brauchen die Views Anweisungen von der Controller-Ebene. Diese ist rein dafür zuständig auf Nutzerinteraktionen zu reagieren und die Aufgaben zu verteilen. Die Controller-Ebene ist oft technischer als die restlichen Ebenen, da diese sich damit auseinandersetzen muss welche Requests hereinkommen können, ob jeder Nutzer auf alle Funktionen Zugriff haben darf, oder auch ob bestimmte Geräte, oder Quellen die Erlaubnis haben auf die Applikation zuzugreifen.

2.3 Django

In dieser Bachelorarbeit werden die Prototypen unter Einsatz des Frameworks Django entwickelt, Django ist ein High-Level Python Web-Framework, welches es ermöglicht Websites, ohne großen Aufwand zu erstellen. Das Framework selbst übernimmt dabei die Aufgabe von Server, Backend und Frontend in einem, was es Entwicklern ermöglicht sich auf das Strukturieren, Designen und Kreieren zu konzentrieren. Bei der Projektstruktur setzt Django auf das Model-View-Controller Pattern.
Django eignet sich mit diesem Pattern für die modulare und White-Label-Entwicklung, da wie schon von Krasner und Pope [KP88] beschrieben, sich die Aufteilung einer Anwendung in unterschiedliche funktionelle Einheiten lohnt. Die Entwickler können sich so besser im Projekt zurechtfinden, außerdem ist es Personen in Django auch noch möglich Ebenen übergreifend zu arbeiten. Damit ist gemeint, dass Django im sogenannten Frontend, also der View und den Templates, eine enge Codeverbindung besitzt. So können

Entwickler ihre Expertise in Frontendarbeit mit Backendarbeit und Services verbinden.

2.3.1 Model Template View

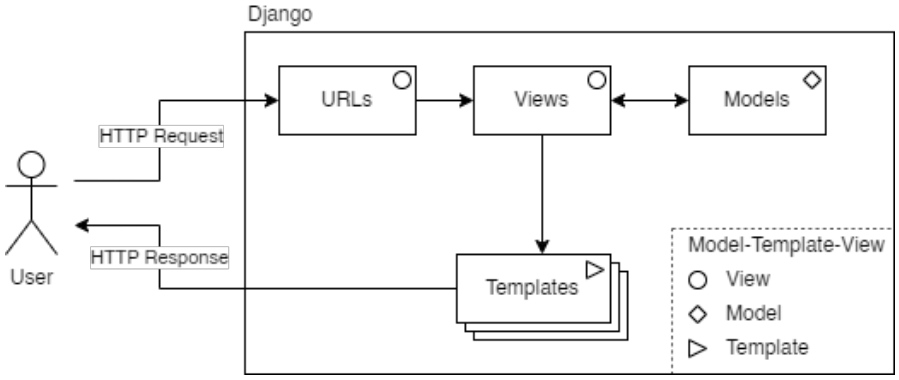

Abbildung 1: Visualisierung des MTV Patterns

Während die klassische Aufteilung von MVCs aus Models, Views und dem Controller besteht, setzt Django auf ein leicht abgewandeltes Pattern: MTV. Dies steht für Model-Template-View. Als leicht abgewandeltes MVC Pattern ähnelt es dem Prinzip eines MVC und hat weiter die strikte Aufteilung in Komponenten vor Augen.
Die Modelle der Model-Ebene sind hier für das Verwalten der Daten zuständig. Die Objekte werden hier definiert und außerdem die Businesslogik für den Umgang mit den Objekten. Die View-Ebene ist in diesem Pattern nicht nur als visuelle Repräsentation der Website definiert, genaugenommen zeigt die View-Ebene dem Nutzer nichts an.
In Abbildung 1 ist zu erkennen, dass die Namensgebenden Views nicht allein die View-Ebene ausmachen, viel mehr ist es ein Zusammenspiel verschiedener Komponenten. Einer Art URL Interface welche URLs der Website exposed und Requests ermöglicht, zusammen mit den Views; welche dafür sorgen, dass der Inhalt korrekt aufgearbeitet wird. Diese Daten werden zuletzt dann an ein Template, welches häufig HTML Dateien sind, weitergeleitet. Django bietet hier die Möglichkeit dynamisch die aufgearbeiteten Daten aus den Views zu nutzen und in die ansonsten statischen HTML Templates einzubinden.

Wie unter anderem Correia und Barbosa [CB19] es definiert haben, kann man zusammenfassend sagen, dass die MVC-View-Ebene im MTV-Pattern unterteilt ist in die Template-Ebene, welche die Daten dem Nutzer anzeigt und die View-Ebene, welche zuständig ist welche Daten angezeigt werden sollen. Die Models haben eine ähnliche Aufgabe und viel

der Businesslogik welche dem Controller zugefallen wäre, übernimmt Django im Hintergrund automatisch.

3 Bewertungskriterien

- **Komplexität** Die Komplexität bezieht sich auf den Unterschied der Projektstruktur von Vor und Nach der Integration des Moduls. Außerdem spielt das Level an Abhängigkeiten, welche im Zuge der Integration entstanden sind und wie diese die Übersichtlichkeit und damit Wartbarkeit beeinträchtigen, eine größere Rolle.
- **Integration** Mit der Integration wird der Weg bewertet, welcher absolviert werden muss, um ein Modul in das bestehende Projekt einzufügen. Sowohl die Anzahl an Schritten, als auch die Tiefe und Anzahl der Abhängigkeiten wird hier beachtet.
- **Konfigurierbarkeit** White Label Applikationen sollen konfigurierbar sein, dies bezieht sich sowohl auf die universelle Nutzbarkeit eines Moduls, als auch auf die Veränderbarkeit auf verschiedenen Granularitätsstufen. Dynamische Veränderung und Bereitstellung von Informationen spielt hierbei auch eine wichtige Rolle.
- **Wiederverwendbarkeit** Der grundlegende Begriff White Label bezeichnet nach unter anderem Silva [SSM20] immer noch Produkte, welche vor dem Verkauf an unterschiedliche Kunden auf deren Wünsche angepasst werden. Genau dafür müssen Module wiederverwendbar sein und in verschiedenen Apps einfach integriert werden können. Gleichzeitig darf ein Modul, welches vor dem Hintergrund eines White-Label-Produktes entwickelt wurde, nicht zu Abhängig von dem zu integrierenden Produkt sein und muss häufig eine abstrakte Form der Objekte annehmen.
- **Pluggabiltiy** Pluggability bezeichnet die Fähigkeit eines Moduls einfach aus dem Grundprojekt ein- und aus gestöpselt werden zu können. Damit ist gemeint, dass die Integration und Extraktion eines Features ohne Probleme und schnell vonstattengehen kann. Features sollen mit dem ëinstecken" sofort funktionieren, dieses Phänomen wird häufig als Plug-and-Play bezeichnet.
- **Skalierbarkeit** Unter diesem Punkt betrachten wir zwei Dinge. Einmal die Skalierbarkeit innerhalb eines Moduls, also die Erweiterung des Moduls um eigene Feature und dann die Skalierbarkeit bezogen auf den Typ an Modul. Bei diesem wird überprüft wie aufwendig es ist viele Module dieser Art hinzuzufügen, ohne die Leistungsfähigkeit und Wartbarkeit des Systems zu sehr zu belasten.

4 Prototypen

Anhand der folgenden Prototypen möchte ich die modulare Architekturweise einer durch Django erstellten Web-Applikation auf verschiedene Bewertungskriterien und schlussendlich auf ihre White Label Fähigkeiten bewerten. Wichti ist zu erwähnen, dass im Folgenden häufig das Wort 'App' genutzt wird. In Django sind Apps eine Art von Modulen. Mit einer App ist also immer das Modul gemeint.
Diese Prototypen werden unterschiedliche Aufgaben haben, welche jeweils in dem dazugehörigen Abschnitt näher erläutert werden. Grob werden drei Prototypen angefertigt, ein Webshop-Modul, ein Warenkorb-Modul und ein News-Modul. Das Webshop-Modul wird dabei die Grundlage unseres Projektes und die Integration der beiden anderen Module wird die Bewertungsgrundlage werden.

4.1 Shop

Der Shop ist eine einfache Django App, welche einen Ticketshop simulieren soll. Der Ticketshop wird die Basis des späteren Projektes sein, so ist es auch diese App, welche das User Handling handhaben wird und die weiteren Prototypen integrieren soll. Wie im folgenden genauer zu betrachten sein wird, hat der Shop initial keine Warenkorbfunktion, sondern bietet nur die Möglichkeit bestimmte Tickets einzeln zu kaufen. Der Kaufprozess ist dabei nicht wichtig um unseren Shop zu testen, weshalb dieser nur Oberflächlich implementiert ist und ein Flag im Hintergrund im Falle eines Kaufs auf True setzen wird.

Die Struktur der App ist wie folgt:

```
baPrototype
├── baPrototype [Projekt]
├── baShop [App]
│   ├── admin.py
│   ├── models.py
│   ├── urls.py
│   └── views.py
├── static_file [Obr19]
├── templates
│   ├── base.html
│   ├── checkout.html
│   ├── footer.html
│   ├── home.html
│   ├── navbar.html
│   ├── product.html
│   └── scripts.html
├── db.sqlite3
└── manage.py
```

4.1.1 URLs

Einem Interface ähnlich, ist die Aufgabe einer jeden URL Datei in Django, dem Nutzer und anderen Projekten und Apps Zugriff auf verschiedene Views zu geben. Wie schon in Abschnitt 2.3.1 beschrieben, wird dafür ein HTTP Request an eine der verfügbaren URLs gesendet. Diese URL leitet diese Anfrage dann an die Views weiter, welche die Anfragen verarbeiten und eine geeignete Antwort zurückgeben.

Listing 1: Shop URLs

```
<!-- Project: baPrototype -->
<!-- App: baShop -->
<!-- File: urls.py -->
from django.urls import path, include
from baShop.views import (
    HomeView,
    TicketDetailView,
    CheckoutView,
    checkout_builder
)

app_name = 'baShop'

urlpatterns = [
    path('', HomeView.as_view(), name='home'),
    path('product/<slug>/', TicketDetailView.as_view(), name='product'),
    path('checkout_builder/<product_request>/', checkout_builder, name='
        checkout_builder'),
    path('checkout/', CheckoutView.as_view(), name='checkout')
]
```

Wie im Codeabschnitt zu sehen, wird in der App *baShop* alle URLs des Projektes, welches den grundlegenden Shop abbildet, integriert. Jede URL in den urlpatterns ist mit einer View verbunden, wobei diese klassenbasierter Natur, oder auch funktionsbasiert sein kann. Der Namespace erlaubt es uns in den Views einen "reverse()" Aufruf durchzuführen und somit die Views aus diesem Projekt untereinander zu verlinken. In diesem Fall wird das Beispielweise der *checkout_ builder* sein, welcher nach Verarbeitung die App eigene CheckoutView aufruft.

4.1.2 Models

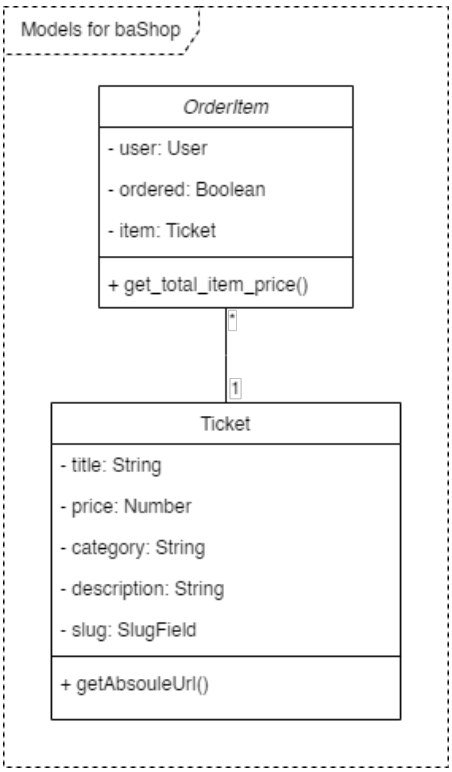

Abbildung 2: UML Diagramm der Models des Shop Projektes

Das Kernprojekt soll einen einfachen Ticketshop darstellen, mit der Funktion ein Ticket zurzeit kaufen zu können. Initial wird es in der Datenbank verschiedene Tickettypen geben, welche keinem Nutzer besonders zugeordnet sind und in Abbildung 2 durch das Objekt Ticket dargestellt werden. Das Ticket hat die erwarteten Attribute wie Titel, Preis eine Kategorie und eine Beschreibung. Die Bezeichnung Slug ist schon im URL Kapitel zu dieser App aufgetaucht und bezeichnet in Django einen lesbaren Query Parameter, welcher in der URL angezeigt wird. In diesem Objekt kann man das SlugField als eine Art Produkt-ID verstehen, über welche über die *'products/<slug>/'* die Details des jeweiligen Produktes abgerufen werden können.
Um ein Ticket kaufen zu können, muss ein Nutzer sich anmelden und hat dann die Auswahl zwischen den verschiedenen Produkten auf der Homepage. Das *OrderItem* Objekt wird genutzt, sollte ein Nutzer sich dazu entscheiden ein Ticket kaufen zu wollen. Als Attribute hat es den User, welcher den Request gestellt hat, ein Flag um den Bestellstatus zu symbolisieren und ein Ticket Objekt des Tickets, welches gekauft werden soll.

4.1.3 Views

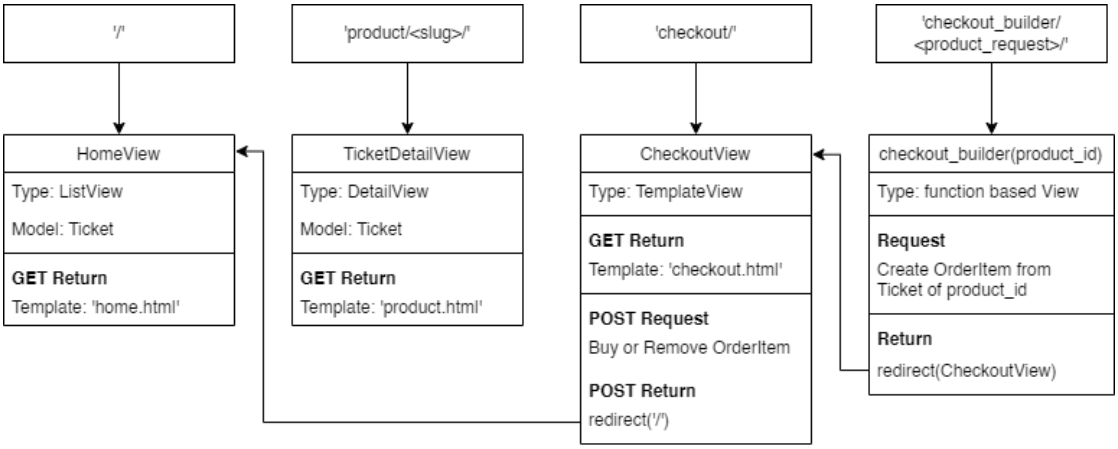

Abbildung 3: Diagramm der Views des baShop Projektes

Die URLs haben bereits einen groben Überblick über alle Verfügbaren Views in dieser App gegeben. Etwas genauer aufgeschlüsselt werden sie in Abbildung 3. Die Views sind das Kernelement eines jeden Django Projektes und haben als Aufgabe, den Request des Nutzers zu verarbeiten und endgültig eine Seite zu präsentieren, welche mit allen nötigen Informationen befüllt ist. Der Shop verfügt über drei klassenbasierte Views, einmal die HomeView, die TicketDetailView und die CheckoutView.

Die HomeView ist die Startseite der Applikation und enthält eine Auflistung aller existierenden Tickettypen. Um dies zu realisieren, bedient sich die Home View der Django-generischen ListView, welche darauf ausgelegt ist einen Zugriff auf alle existierenden Objekte eines Models im Form einer Liste zuzugreifen. Diese Liste an in diesem Fall Tickets wird mit dem Return an das 'home.html' Template gegeben, damit dieses dynamisch alle Objekte anzeigen kann.
Mit dem klicken auf eines der Tickets öffnet sich dann die TicketDetailView für eben jenes Ticket. Auch diese wird durch eine der generischen Klassen abgebildet und nutzt diesmal die DetailView. Wie der Name bereits andeutet, ist die DetailView in Django darauf ausgelegt ein bestimmtes Objekt eines Models anzuzeigen.
Der Checkout Button auf den Produktseiten führt daraufhin zu einem Request an die funktionsbasierte View *checkout_builder(product_id)* mit dem passenden Slug als product_id. Die Aufgabe dieser View ist in diesem Fall nicht direkt ein Template Response herbeizuführen, sondern sich um das Verarbeiten eines neuen Objektes zu kümmern. An-

hand des Slugs wird der Tickettyp festgestellt und sich das Ticket Objekt herausgesucht, dieses wird dann in einer neuen Instanz eines OrderItems hinzugefügt und dem OrderItem wird der momentane Nutzer angefügt. Nachdem dies erledigt ist, wird der Nutzer weitergeleitet zur CheckoutView, welche als HTTP Response mit dem 'checkout.html' Template antwortet. Befüllt wird dieses Template zuvor mit dem aktuellen OrderItem. Diese View hat einen weiteren spezifizierten HTTP Request für die POST Methode, welcher dafür verantwortlich ist das OrderItem zu kaufen und somit den Status ordered auf True zu setzen, oder das Item zu löschen. In beiden Fällen wird der Nutzer danach wieder auf die Homepage weitergeleitet, um seinen Einkauf fortzusetzen.

4.1.4 Templates

Das baShop Modul beinhaltet die meisten Templates, da es als Modul des Hauptprojektes später, die meisten Aufgaben zu erfüllen hat und die Templates für zum Beispiel Footer, Header und das Grundgerüst, die 'base.html' bereitstellen muss.
Zugegriffen kann von außen nur auf zwei der Templates, einmal die 'home.html' und die 'product.html'. Django bietet in jedem Template, welches von einer View mit Daten befüllt wird, eine dynamsiche Art an diese Daten einzubinden.

Listing 2: Shop Homepage Template

```
1  <!-- Project: baPrototype -->
2  <!-- App: baShop -->
3  <!-- File: home.html (Vereinfacht) -->
4  <div class="row">
5      {% for item in object_list %}
6      <div class="col-lg-3">
7      <div class="card">
8        <div class="card-body text-center">
9          <strong class="grey-text">
10           <h5>{{ item.get_category_display }}</h5>
11         </strong>
12         <h5>
13           <strong>
14             <a href="{{ item.get_absolute_url }}" class="dark-grey-text">
                   {{ item.title }}
15             </a>
16           </strong>
17         </h5>
18         <h4 class="font-weight-bold">
19           <strong>
```

```
20              {{ item.price }}
21                < /strong>
22            </h4>
23          </div>
24        </div>
25  </div>
26  {% endfor %}
```

Der Code Abschnitt ist vereinfacht um Platz zu sparen, die Originale Struktur vieler der Templates ist unter anderem inspieriert von [Fre20]. In dem Code Abschnitt ist zu erkennen, dass das Template eine native object_list hat. Diese wird automatisch von Django durch die ListView, mit den existierenden Model Objekten des ausgewählten Models (hier Ticket), gefüllt. Die dynamische Integration erlaubt es die HomePage mit beliebig vielen Ticket Typen zu füllen und ein vorher festgelegtes Design auf diese anzuwenden, was zu einer einheitlichen und übersichtlichen Seite führt.

4.2 Warenkorb

Der Warenkorb ist eine weitere Django App mit dem Namen baShoppingCart, welche einen Warenkorb simulieren soll. Der Warenkorb wird ein Feature simulieren, welches in den Shop Prototypen integriert werden soll. Aufgrund der engen Verbindung, welche ein Warenkorb mit einem Webshop hat, werden mehrere tiefere Abhängigkeiten zwischen den Apps entstehen. Kurzgefasst wird der Warenkorb über die Optionen verfügen, Items hinzuzufügen und zu löschen, wie auch eine HTML-Repräsentation der Items zu liefern und diese Items als Liste zurückzugeben.

Die Struktur dieser App ist wie folgt:

```
baPrototype2
├── baPrototype2 [Projekt]
├── baShoppingCart [App]
│   ├── templates
│   │   └── order_summary.html
│   ├── admin.py
│   ├── models.py
│   ├── context_processors.py
│   ├── urls.py
│   ├── utility.py
│   └── views.py
├── static_files [Obr19]
├── db.sqlite3
└── manage.py
```

4.2.1 URLs

Listing 3: Shopping Cart URLs

```
<!-- Project: baPrototype2 -->
<!-- App: baShoppingCart -->
<!-- File: urls.py -->
from django.urls import path
from baShoppingCart.views import (
    add_to_cart,
    remove_from_cart,
    remove_single_item_from_cart,
    OrderSummaryView
)

app_name = 'baShoppingCart'

urlpatterns = [
    path('order-summary/', OrderSummaryView.as_view(), name='
        order-summary'),
    path('add-to-cart/<item_id>/<item_name>/<item_price>/', add_to_cart,
        name='add-to-cart'),
    path('remove-single-item-from-cart/<item_id>/',
        remove_single_item_from_cart, name='remove-single-item-from-cart'
        ),
    path('remove-from-cart/<item_id>/', remove_from_cart, name='
        remove-from-cart')
]
```

Die URLs der Warenkorb App sind mit dem Ziel erstellt worden, dem Entwickler gleich klarzumachen, welche Daten benötigt werden. Drei der URLs verweisen auf funktionsbasierte Views, welche für die Handhabung der Artikel im Warenkorb zuständig sind. Die einzige klassenbasierte View hat dann nur noch die Aufgabe den Warenkorb zu präsentieren, erwartet aus diesem Grund auch als einzige URL kleine Query Parameter beim Aufruf.

4.2.2 Models

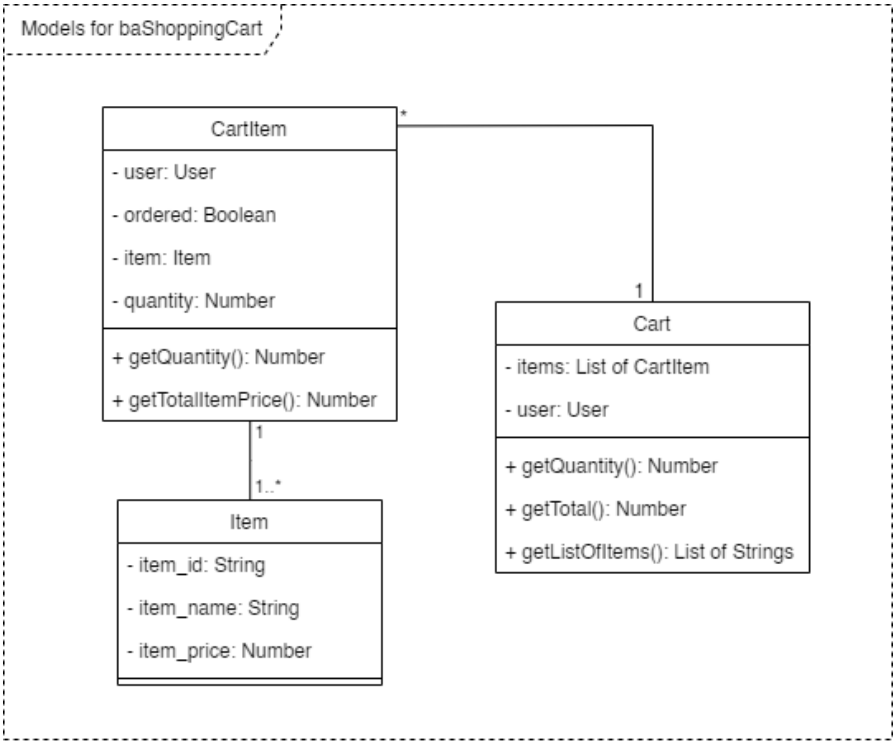

Abbildung 4: UML Diagramm der Models des Shopping Cart Projektes

Der Warenkorb wird entwickelt, um in andere Projekte integriert zu werden und kann dadurch keine Objekte als Itembasis verwenden. Um dieses Problem zu umgehen wird der Warenkorb selbst nur bestimmte Werte empfangen. Es wird erwartet, dass beim Hinzufügen eines Artikels in den Warenkorb jeweils ein Name, der Preis des einzelnen Produkts und eine Produkt-ID übergeben werden. Diese können in Abbildung 4 unter den Attributen item_id, item_name und item_price im Item Objekt gefunden werden. Das CartItem verbindet ein in den Warenkorb eingefügtes Item mit dem User, welcher den Request für das Hinzufügen stellt. Die Beziehung zwischen dem CartItem Objekt und dem einzelnen Item Objekt ist eine 1:1..n-Beziehung, ein CartItem hat immer exakt ein zugehöriges Item, die Items können allerdings in verschiedenen CartItems genutzt werden. Sollte ein weiteres Item mit derselben Produkt-ID in den Warenkorb hinzugefügt werden, dann wird kein weiteres Item in das Objekt eingefügt, sondern es wird die Quantity, also die Anzahl der Objekte dieses Typs hochgesetzt.
Das Cart-Objekt selbst überwacht dabei immer die Anzahl der CartItems, welche momen-

tan im Warenkorb vorhanden sind. Um dynamisch die momentane Anzahl an CartItems abrufbar zu machen, stellt das Objekt die Methode getQuantity() zur Verfügung. Diese Methode kann dann im Projekt selbst, oder aber über Kontextprozessoren anderen Projekten zugänglich gemacht werden. Die Methode getListOfItems() gibt eine Liste an item_ids zurück, diese Methode wird also abrufbar gemacht werden für das integrierende Projekt, um einen Checkout Prozess realisieren zu können.

4.2.3 Views

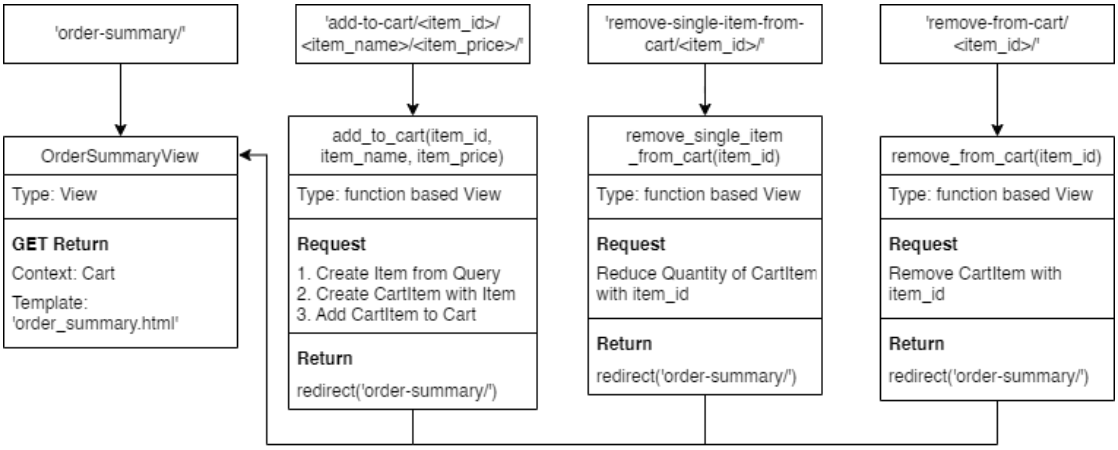

Abbildung 5: Diagramm der Views des baShoppingCart Projektes

Im Projekt des Warenkorbs haben wir es mit 3 funktionsbasierten und einer klassenbasierten View zu tun. Die klassenbasierte View ist das Kernelement dieser App da sie den Warenkorb selbst definiert. Die OrderSummaryView basiert auf einer einfachen View und kann so am meisten umgestaltet werden. Als einzige Methode haben wir hier die HTTP Get Methode, welche keine Argumente empfängt. Die Aufgabe dieser View ist es, das Cart Objekt des Users, welcher die Anfrage stellt, zu holen und das Template für den Warenkorb örder_summary.html" mit dem Cart als Kontextobjekt zu füllen und als HTTP Response zurückzugeben. Aufgerufen wird diese Methode von dem Projekt, welches den Warenkorb nutzen wird, in den meisten Fällen beim Hinzufügen eines Produktes, oder beim Hinnavigieren über eine Navbar.

Die 3 funktionsbasierten Views erfüllen jeweils eine eigene Aufgabe. Das Hinzufügen von Produkten in den Warenkorb übernimmt die *add_to_cart* View, das Entfernen von einem einzelnen Objekt eines Produktes die *remove_single_item_from_cart* View und

das vollständige Entfernen eines Produktes aus dem Warenkorb die *remove_from_cart* View. Aufgerufen werden diese Views jeweils mit Parametern, welche für die Bearbeitung der Anfrage nötig sind. Die add_to_cart() View benötigt eine Item-ID, den Namen und den Preis des Items um wie in Abschnitt 4 zu sehen ein Item Objekt zu erstellen. Daraufhin wird dieses Item Objekt genutzt, um ein CartItem zu erstellen und jenes CartItem wird mit dem Nutzer verknüpft, welcher die Anfrage gestellt hat. Danach wird das CartItem zum Warenkorb des Nutzers hinzugefügt und der Warenkorb wird über einen Redirect geöffnet.

Für das Entfernen von einzelnen Objekten, oder ganzen Itemtypen aus dem Warenkorb, wird jeweils nur die Item-ID benötigt. Im ersten Fall wird die Quantity des CartItems um 1 reduziert und das CartItem nur im Fall 'Quantity=0' vollständig aus dem Warenkorb gelöscht. Die remove_from_cart() View allerdings löscht das CartItem aus dem Warenkorb, unabhängig davon wie hoch die Anzahl im CartItem ist. Genutzt werden können diese Views von integrierenden Projekten, aber auch die 'order_summary.html' Seite hat über Icons für das Hinzufügen, Reduzieren und Entfernen einen direkten Zugriff auf diese Views.

4.2.4 Kontextprozessoren

Listing 4: baShoppingCart Kontextprozessoren

```
1  <!-- Project: baPrototype2 -->
2  <!-- App: baShoppingCart -->
3  <!-- File: context_processors.py -->
4  from shopping_cart.models import Cart
5
6
7  def cart_quantity(request):
8      quantity = 0
9      if request.user.is_authenticated:
10         cart = Cart.objects.get(user=request.user)
11         quantity = cart.get_quantity()
12     return {'cart_quantity': quantity}
```

Wie zuvor in Abbildung **??** zu sehen, befinden sich im baShoppingCart Projekt Kontextprozessoren. Diese sind im Grunde einfache Python Methoden, welche einen Request als Parameter nehmen und ein Dictionary an Elementen zurückgeben, welche dann in einer View, oder einem Template genutzt werden können.

Das Besondere an den Kontextprozessoren ist, dass sie die Daten jederzeit global zur Verfügung stellen und diese somit in jeder installierten App abgerufen werden können. Genutzt wird diese globale Variable hier als eine Möglichkeit auf die aktuelle Anzahl an Tickets, welche bereits im Warenkorb vorhanden sind, zuzugreifen. Wichtig ist zu beachten, dass diese globale Variable nur lesbar ist, also nicht von außen verändert werden kann.
Die Funktionsweise ist im oberen Codeabschnitt zu sehen, 'cart_quantity' bietet als Kontextparameter einen direkten Zugriff auf die *get_ quantity()* Methode des Carts des aktuellen Users.

4.2.5 Utility Klasse

Listing 5: baShoppingCart Utility Klasse

```
1  <!-- Project: baPrototype2 -->
2  <!-- App: baShoppingCart -->
3  <!-- File: utility.py -->
4  from django.contrib import messages
5
6  from .models import CartItem
7  from .views import remove_single_item_from_cart  # Import your
       remove_from_cart view
8
9  def delete_item_from_cart(request, item_id):
10     cart_item = CartItem.objects.filter(
11         item__item_id=item_id,
12         user=request.user,
13         ordered=False
14     ).first()
15
16     if cart_item:
17         remove_single_item_from_cart(request, item_id)  # Reuse the logic
               from your existing view
18         messages.info(request, "Item removed from the cart.")
19     else:
20         messages.info(request, "Item not found in the cart.")
```

Während der Entwicklung ist im letzten Schritt ein Problem aufgetreten. Die Frage war wie die Items aus dem Warenkorb gelöscht werden können nach dem Kauf. Die Views für das eigentliche Reduzieren und Entfernen sind dabei nicht nutzbar, da jede von diesen

Views redirected auf das ShoppingCart, doch muss dem integrierenden Projekt trotzdem eine Methode bereitgestellt werden, welches ein Objekt aus dem Warenkorb löschen kann. Eine Lösung dafür war eine Utility Klasse *utility.py*, welche wie im Codeabschnitt zu sehen die Methode 'delete_item_from_cart(request, item_id) enthält. Die Methode erlaubt es uns die gewünschte View aufzurufen, aber gleichzeitig muss sie die Response, also den redirect() auf den Warenkorb nicht ausführen.
Diese Methode muss direkt importiert werden, da sie nicht wie die Kontextprozessoren nur lesbares enthält, sondern etwas in den Objekten verändert.

4.2.6 Templates

Dieses Modul beinhaltet nur ein Template, dieses ist für das Darstellen einer Liste der CartItems zuständig ist. Nicht unähnlich der ListView aus Abschnitt 4.1.4 wird hier zwar kein Model direkt übergeben, die OrderSummaryView übergibt stattdessen allerdings das Cart Objekt des Nutzers als Kontext. Dieses kann dann wieder über eine dynamische for-Schleife '{% for cart_item in object.items.all %}' im HTML genutzt werden.

4.3 Neuigkeiten

Der Bereich der News wird auch als eine weitere Django App erstellt, mit dem Namen baNews, welche ein News-System simulieren soll. Dieser News Bereich wird ein weiteres Feature des Shops und benötigt auch nur eine Integration in selbigen, nicht in die Warenkorb-App. Diese App soll die Integration eines abgeschlossenen, eigenständigen Modul simulieren. Es wird keine enge Verbindung zum Shop aufgebaut, da keine Befehle von jenem ausgehen müssen. Einzig eine Verlinkung in der Navbar des Projektes wird erwartet. Die App selbst wird eine Repräsentation der News Objekte bieten, sowohl als Liste, als auch einzelne Elemente im Detail.

Die Struktur dieser App ist wie folgt:

```
baPrototype3
├── baNews [App]
├── baPrototype3 [Projekt]
│   ├── templates
│   │   ├── news-section.html
│   │   └── news.html
│   ├── admin.py
│   ├── models.py
│   ├── urls.py
│   └── views.py
├── static_files
│   └── css
│       └── bootstrap.css
├── db.sqlite3
└── manage.py
```

4.3.1 URLs

Listing 6: News URLs

```
<!-- Project: baPrototype3 -->
<!-- App: baNews -->
<!-- File: urls.py -->
from django.urls import path, include
from baNews.views import (
    NewsSectionView,
    NewsDetailView
)

app_name = 'baNews'

urlpatterns = [
    path('news-section/', NewsSectionView.as_view(), name='news-section')
        ,
    path('news/<slug>/', NewsDetailView.as_view(), name='news')
]
```

Auffällig in den urlpatterns dieses Projektes ist das Fehlen von URLs für das Hinzufügen, Verändern und Entfernen von News-Objekten. Dieses Design ist beabsichtigt, da nur Administratoren Zugriff auf die Erstellung von Neuigkeiten haben sollen, dieses kann über das Admin-Portal geschehen. Mit der *'news-section/'* Seite, hätten wir dann die HomePage für diese App, welche eine Übersichtsseite für alle News ist und durch die NewsDetailView komplettiert wird, welche einzelne News, identifiziert durch den slug, anzeigen soll.

4.3.2 Models

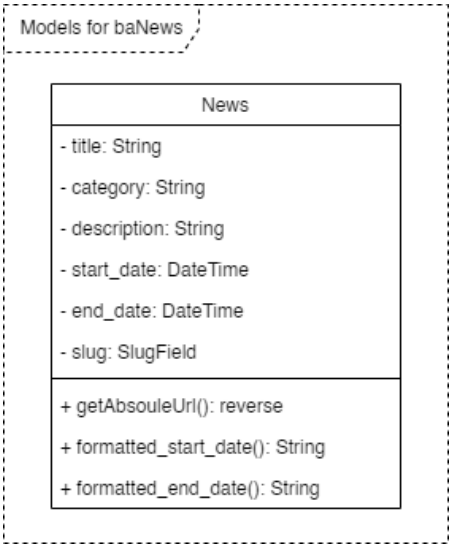

Abbildung 6: UML Diagramm der Models des News Projektes

Ohne die Notwendigkeit für userszpezifische Objekte kommt das baNews Projekt mit einem einfachen News-Objekt aus. Um eine dynamische Übersicht zu ermöglichen, in welcher nur aktuelle Nachrichten angezeigt werden, besitzt das News-Objekt ein Startdatum und ein Enddatum. Um außerdem, wie im Shop in Abschnitt 4.1.3, eine Detailansicht für einzelne Neuigkeiten zu ermöglichen, wird auch dieses Objekt ein SlugField erhalten. Die Methode *getAbsoluteUrl()* wird dann direkt auf die Detailsicht der News verlinken.

4.3.3 Views

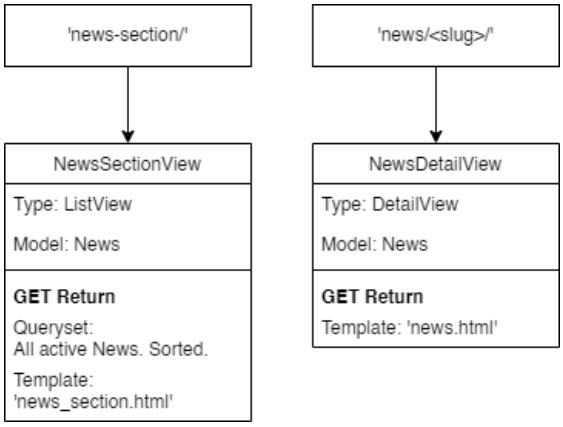

Abbildung 7: Diagramm der Views des baNews Projektes

Wie bereits im Abschnitt 4.3.1 über die URLs dieses Projektes erwähnt, haben wir nur zwei Views welche rein für das Anzeigen von Daten da sind. Vergleichbar mit der HomeView und TicketDetailView aus Abschnitt 4.1.3 ist die Aufgabe der NewsSectionView in dieser App, alle Neuigkeiten anzuzeigen. Besonders ist dabei, dass die native get_queryset() Methode einer ListView hier überschrieben wird. Sie wird überschrieben, um uns nicht immer alle existierenden News Objekte zurückzugeben, sondern immer nur aktuelle. Dafür bedient sich die Methode der Startzeit und Endzeit in den Objekten und gleicht die aktuelle Zeit des Nutzers mit diesen ab. Sollte die aktuelle Zeit zwischen Start und Ende der News liegen, wird das Objekt in die Liste mit aufgenommen, anderenfalls wird es ignoriert. Das Template 'news_section.html' empfängt durch die ListView und überschriebene Methode dann eine Liste an aktuellen News.
Die NewsDetailView wiederum funktioniert genauso wie die TicketDetailView des Shop-Prototyps und zeigt im 'news.html' Template ein bestimmtes News-Objekt an, welches mit dem Slug übereinstimmt.

4.3.4 Templates

Das News-Modul beinhaltet zwei Templates, diese sind den Templates der Homepage und der Ticket Detail Seite des baShop Moduls sehr ähnlich. Die ListView und die DetailView, welche die Templates 'news_section.html' und 'news.html' befüllen, sind bis auf die genutzten News-Objekte vom Verhalten her identisch.

5 Zusammenführen der Prototypen

5.1 Shop und Warenkorb

5.1.1 Architektur

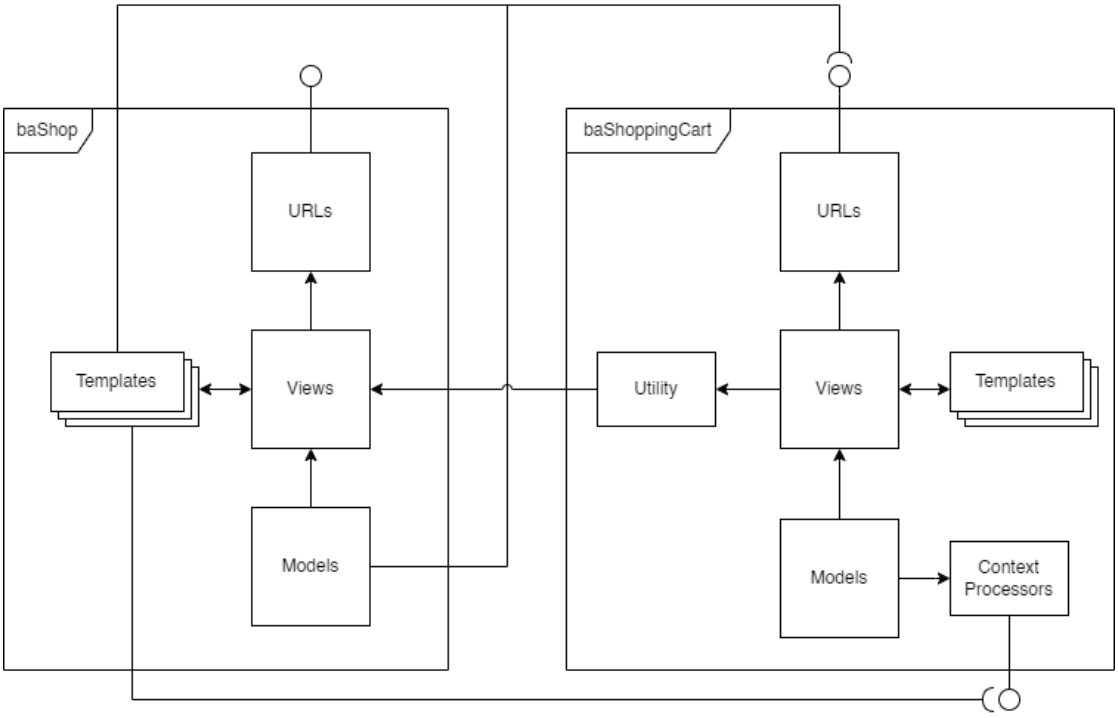

Abbildung 8: Komponenten-Diagramm der zusammengefügten Prototypen

Die Komponenten sollen als Apps zusammengefügt werden. Als Grundlage wird davon das geklonte baPrototype Projekt genutzt, welches nur die App baShop enthält und somit nur den grundlegenden Webshop, ohne Warenkorbfunktion.
In Abbildung 8 ist der Aufbau der zusammengefügten Django Applikation zu sehen. Das Kernprojekt baShop bietet weiterhin über die URLs die Schnittstellen für die Nutzer, um zur Homepage und den Produktseiten zu gelangen an. Über die URLs aus der *url.py* des Warenkorb-Projektes [Abschnitt 4.2.1] soll der Warenkorb an den Webshop angebunden werden. Die URLs werden in die Models und einige Templates integriert und über diese hat die Anwendung daraufhin Zugriff auf die Funktionen des Warenkorbs. Wie die Abschnitte 4.2.4 und 4.2.5 zeigen, müssen auch die Kontextprozessoren und Utility Klassen

eingebunden werden. Erstere werden als globale Variable mit Leseberechtigung bereitgestellt, während die Utility Klasse direkt im Projekt integriert werden muss.

An der Projektstruktur ändert sich in diesem Fall nichts, außer, dass die neue App baShoppingCart sich mit im Projekt befindet. An den inneren Dateien ändert sich nichts.

```
BA-Prototyp-Combi
├── baPrototype [Projekt]
├── baShop [Webshop 4.1]
├── baShoppingCart [Warenkorb 4.2]
├── static_files
├── templates
├── db.sqlite3
└── manage.py
```

5.1.2 Integration

1. Hinzufügen der App durch einfaches Copy&Paste

   Integration in Projekt settings.py und urls.py

2. Füge die App in den INSTALLED_APPS hinzu
3. Füge den Kontextprozessor in die 'context_processors' ein
4. Füge die urlpatterns der neuen App in den Projekt-URLs hinzu

   Integration in App baShop

5. Füge die urlpatterns des baShoppingCarts in den App-URLs hinzu
6. Füge einen Link über die Navbar zum Warenkorb hinzu
   a) Füge eine aktuelle Itemanzahl des Warenkorbs hinzu, nutze dafür den Kontextprozessor
7. Füge die Warenkorb-URLs für das Hinzufügen und Entfernen eines Items in das Ticket Model ein
8. Füge einen Button in der 'product.html' hinzu, welcher die neue Ticket Methode

nutzt, um die URL für das Hinzufügen in den Warenkorb aufzurufen

9. Verändere die Checkout Logik, um eine Liste als Query Parameter zu empfangen
10. Füge der Checkout Logik einen Aufruf der Utility Klasse hinzu um das gekaufte Item aus dem Warenkorb zu entfernen

Veränderungen in App baShoppingCart

11. Lass die 'order_summary.html' die 'base.html' erweitern
12. Füge dem Checkout Button einen Aufruf der 'checkout_builder' View des baShop Projektes hinzu, mit dem Return der 'getListOfItems()' Methode als Query Parameter

Die Integration wird mit dem Befehl 'python ./manage.py migrate' in der CLI beendet, so werden alle Models der neuen App in die Projektdatenbank migriert.

Insgesamt werden 12 Schritte benötigt, um den Code der neuen App zu integrieren. Es werden in den Projektdateien 3 Abhängigkeiten geschaffen (Schritt 2-4). In der baShop App wurden direkte Abhängigkeiten in einer HTML-Datei (Schritt 6), einem Model (Schritt 7), sowie einer View (Schritt 10) geschaffen. Die baShoppingCart App hat nur in einer HTML-Datei Abhängigkeiten geschaffen.

5.2 Shop mit Warenkorb und News

5.2.1 Architektur

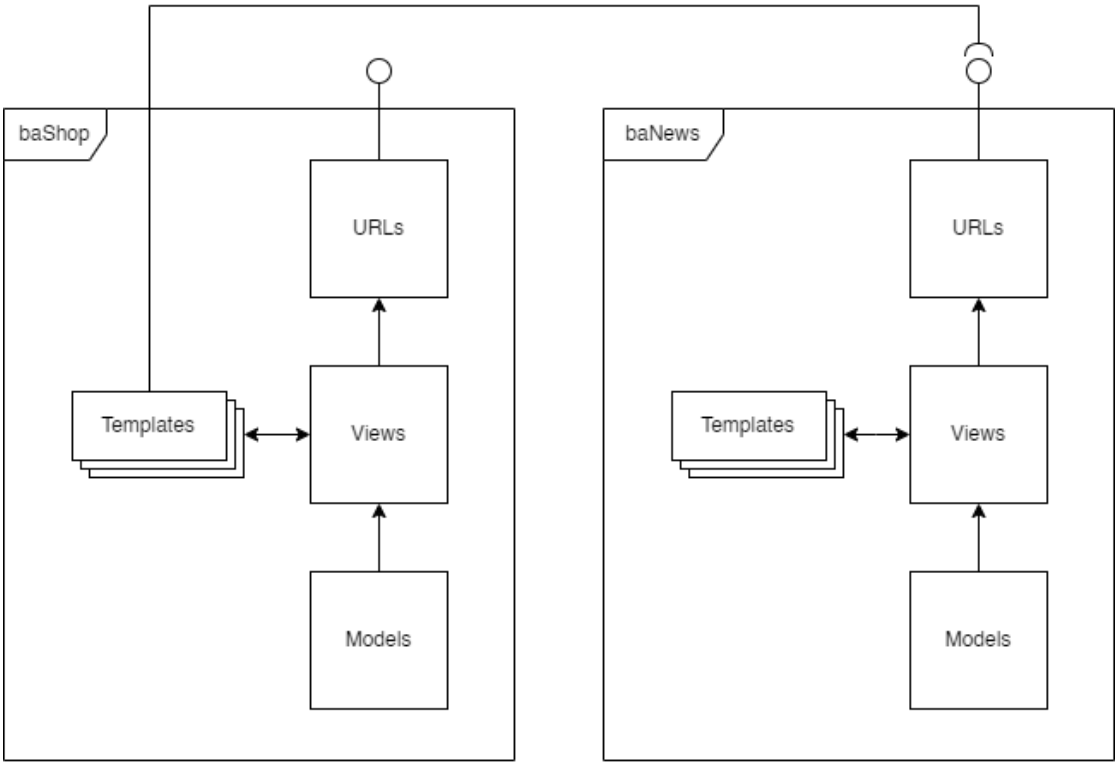

Abbildung 9: Komponenten-Diagramm der zusammengefügten Prototypen

In Abbildung 5.2 werden nur die beiden Apps baShop und baNews angezeigt, das liegt daran, dass keine Verbindung mit dem Warenkorb entsteht und somit für die Übersicht jener weggelassen würde. Nach dieser Integration würde das korrekte Komponenten-Diagramm alle 3 Apps enthalten.
Der Aufbau der zusammengefügten Applikationen ist diesesmal deutlich einfacher, da die baNews App ein abgeschlossenes System darstellen soll. Es werden keine Cross References benötigt, da das Hinzufügen und somit Verwalten aller Neuigkeiten von der Django Admin Seite verwaltet wird und alles andere intern in der App funktioniert. Es wird nur eine der URLs aufgerufen, welche die baNews App bereitstellt. Das Kernprojekt baShop bietet weiterhin über die URLs die Schnittstellen für die Nutzer, um zur Homepage, den Produktseiten und auch zu dem Warenkorb zu an. Über die URLs der baNews App [Abschnitt 4.3.1] soll die neue App wieder eingebunden werden. Die URL zur News-Seite

wird hier nur in eines der Templates eingebunden.

An der Projektstruktur ändert sich in diesem Fall wieder nichts, außer, dass die neue App baNews sich mit im Projekt befindet. An den inneren Dateien ändert sich auch hier nichts.

```
BA-Prototyp-Combi
├── baNews [News 4.3]
├── baPrototype [Projekt]
├── baShop [Webshop 4.1]
├── baShoppingCart [Warenkorb 4.2]
├── static_files
├── templates
├── db.sqlite3
└── manage.py
```

5.2.2 Integration

1. Hinzufügen der App durch einfaches Copy&Paste

   Integration in Projekt settings.py und urls.py

2. Füge die App in den INSTALLED_APPS hinzu
3. Füge die urlpatterns der neuen App in den Projekt-URLs hinzu

   Integration in App baShop

4. Füge die urlpatterns der baNews App in den App-URLs hinzu
5. Füge einen Link über die Navbar zu der News Übersicht hinzu

Die Integration wird wieder mit dem Befehl 'python ./manage.py migrate' in der CLI beendet, so werden alle Models der neuen App in die Projektdatenbank migriert.

Insgesamt werden dieses Mal nur 5 Schritte benötigt, um den Code der neuen App zu integrieren. Es werden in den Projektdateien 2 Abhängigkeiten geschaffen (Schritt 2-3). In der baShop App wurde eine direkte Abhängigkeit in einer HTML-Datei (Schritt 5) geschaffen. Die integrierte App benötigte keine weitere Veränderung.

6 Bewertung der Integrationen

Im Folgenden Abschnitt wird die Integration der Module bewertet. Zusammengefasst bestand der Django Prototyp zuerst nur aus einem Webshop mit Kaufoptionen für verschiedene Tickets. Im ersten Schritt wurde ein Warenkorb Modul hinzugefügt, welche es dem Projekt erlauben sollte, Tickets in einen Warenkorb hinzuzufügen, diesen zu verwalten und die Tickets dann über das Shop-Modul zu kaufen. Als weiteres Modul wurde dann ein News-Bereich hinzugefügt, dieser simulierte die Integration eines eigenständigen Moduls, bei welchem die Funktionsweise nichts mit dem ursprünglichen Shop zu tun hatte und so keine Tiefe Integration nötig war.

6.1 Integration Warenkorb in den Shop

- **Komplexität** Wie schon an der Abbildung 8 zu erahnen, ist die Komplexität des ursprünglichen Shops deutlich gestiegen. Dies ist durchaus zu erwarten bei einer Modul-Integration, welche auf der ursprünglichen Funktionalität aufbaut und diese auch noch erweitert. Allerdings werden Abhängigkeiten auf allen 3 der MTV-Ebenen unweigerlich zu unübersichtlicheren Verknüpfungen führen.
- **Integration** Die Integration einer App in ein anderes Projekt ist grundsätzlich durch Django sehr angenehm. Als Framework welches auf modularisierte Anwendungen spezialisiert ist, ist die erste Code-Integration durch einfaches Copy&Paste leicht zu realisieren. Mit 12 Schritten bis zur vollständigen Integration ist der Prozess aber durchaus verwinkelt. Mit einer entsprechenden Anleitung sollte eine Integration allerdings kein Problem darstellen. Komplizierter könnte es bei den Erklärungen für Kontextprozessoren und Utility Klassen werden, da diese auf unterschiedliche Art und Weisen eingebunden werden, herrscht hier höheres Fehlerpotential.
- **Konfigurierbarkeit** Mit Hinblick auf den White Label Aspekt ist dieses Modul recht solide aufgebaut. Positiv hervorzuheben sind die simpel gehaltenen Designs und die Dynamik bei den Item-Objekten. Die Item-Objekte stellen bei einer Erweiterung allerdings möglicherweise ein Problem dar, zwar ist dieses Model sehr generisch, es muss aber bei der Integration durch ein anderes Projekt unter Umständen auf gewünschte Details verzichtet werden. Eigenschaften wie die Kontextprozessoren sind wiederum ein großes Plus, da sie dem gesamten Projekt einfachen Zugriff

auf bestimmte Attribute geben.

- **Wiederverwendbarkeit** Die Veränderungen, welche im Warenkorb Modul nötig sind, um dieses mit einem anderen Projekt kompatibel und funktionsfähig zu machen, sind natürlich nicht ideal. Trotzdem nutzt dieses Modul abstrakte Objekte, um eine Integration nicht von bestimmten Objekten abhängig zu machen, was nach Jee [Jee] eine wichtige Eigenschaft in diesem Feld ist. Um eine schnelle Variante eines Warenkorbs zu integrieren, eignet sich das Modul, da das Thema, die Objekte und die Funktionalität des integrierenden Shops dem Warenkorb gleichgültig sind.
- **Pluggabiltiy** Wie zu erwarten lässt die Komplexe Integration des Moduls diese Eigenschaft deutlich schlechter aussehen. Zwar würde der Webshop weiter funktionieren, entferne man einfach das Warenkorb-Modul, doch werden doch einige nacharbeiten nötig. Die Integration in der Navbar, die Buttons in der TicketDetailView und die gesamten Code-Referenzen im Hintergrund, wie zum Beispiel der veränderte Checkout Prozess. Das Plug-In und Plug-Out sind somit immer mit einer gewissen Arbeit verbunden, welche aber auch nicht zu viel Zeit in Anspruch nehmen sollte.
- **Skalierbarkeit** Betrachten wir hier erst einmal die Erweiterung des Warenkorb-Moduls. Mit weiteren Features, welche eine Integration in andere Projekte erwarten, würden nur mehr Abhängigkeiten geschaffen werden und so der Übersicht im Projekt Schaden. Außerdem kann unter Umständen nicht mehr isoliert an einem der Module gearbeitet werden. Sollte es sich um Modul interne Features handeln, welche sich rein um den Warenkorb kümmern und nur aus diesem gesteuert werden können, sollte dies keine Probleme bereiten.

6.2 Integration News Bereich in den Shop

- **Komplexität** Die Komplexitätsänderung nach der Integration des News-Moduls ist nahezu unbedeutend, eine einzige neue Verbindung im Komponentendiagramm ist das Nutzen der URLs in den Templates, wie in Abbildung 9 zu erkennen ist. Dadurch, dass diese Feature Integration sich nicht auf die restliche Funktionalität des Shops auswirkt, ändert sich außer einigen Zeilen HTML Code nichts im Shop Modul, somit wird nur leicht in die Template-Ebene eingegriffen.
- **Integration** Der Beginn ist wie auch schon beim Warenkorb Modul sehr angenehm.

Mit diesmal nur 5 Schritten bis zur vollständigen Integration ist der Prozess um einiges kürzer und birgt kaum, bis gar keine Risiken. Geachtet werden muss darauf, dass die CSS Dateien übereinstimmen und das ursprüngliche 'base.html' erweitert wird. Abgesehen davon ist die Integration nicht weiter aufwendig.

- **Konfigurierbarkeit** Die Konfigurierbarkeit ist nicht sonderlich leicht zu bewerten. Dadurch, dass ein separiertes Modul sich nicht weiter um die Daten des Hauptmoduls kümmert, ist ein Vergleich kaum möglich. Hervorzuheben ist, dass die Konfigurierung der Neuigkeiten Objekte der Admin Schicht von Django überlassen wurde, was einen sehr unkomplizierten Vorgang bietet, da mit dem Einfügen des Moduls und der Migration in die Datenbank alles Nötige dafür vorbereitet ist. Designtechnisch ist das Modul stark davon abhängig wie es implementiert wurde, wenn ein Modul mit Hinblick auf die Integration in ein bestimmtes Projekt entwickelt wird, können die Style-Abhängigkeiten von vornherein mit eingeplant werden und so sollte kein Problem entstehen. Wird allerdings ein rein Fremdes Modul integriert, kann das Design und die benötigten Dateien stark abweichen.

- **Wiederverwendbarkeit** Die Konfiguration durch Djangos native Admin Schicht und der Fakt, dass dieses Modul ein abgeschlossenes System ist, ohne tiefe Integration in das Projekt, überzeugt mit einer einwandfreien Wiederverwendbarkeit.

- **Pluggabiltiy** Das Plug-In und Plug-Out brauchen in diesem Fall immer nur das Hinzufügen/Entfernen des Moduls und ein Hinzufügen/Entfernen der Verlinkung in der Navbar des integrierenden Projektes.

- **Skalierbarkeit** Betrachten wir wieder zuerst die Erweiterung des News-Moduls. Nach Østby [Øst08] ist die Veränderbarkeit des Modul, ohen Auswirkung auf das hauptprojet eines der Wichtigsten Eigenschaften. Bei diesem Prototypen sind weitere Features im Modul vollkommen ungefährlich für die Integration in ein anderes Projekt und auch Änderungen, beinträchtigen das integrierende Projekt nicht. Mit der Überlegung Kontextparameter zu kreieren lassen sich sogar noch weitere Wege erstellen, um Informationen über eine Art Schnittstelle nach außen zu geben. Bezüglich des Hinzufügens von mehr Modulen dieser Art sehe ich auch keine Probleme, da diese Module einander nicht beeinflussen und auch die Funktionalität des Shops nicht weiter verändern.

7 Fazit

In dieser Arbeit wurde sich mit Prototypen befasst, welche allesamt mit dem Framework Django erstellt wurden. Django bietet einfache Möglichkeiten Module in Projekte zu integrieren, da jede Applikation in Django gleichzeitig ihr eigenes Modul ist. Es wurde ein Prototyp eines funktionalen Features erstellt, welches die Funktion eines Webshops um die eines Warenkorbs erweitern sollte, ohne dabei Abhängig von bestimmten Implementierungen des Webshops zu sein. Der andere Prototyp, welcher für die Integration in ein vorhandenes Projekt da ist, sollte ein geschlossenes System beinhalten. Anhand von diesem soll die Integration eines Features getestet werden, welches unabhängig vom eigentlichen Projekt läuft.

In der Bewertung der Integration der Module sind einige Unterschiede aufgefallen, welche hauptsächlich auf die Integrationstiefe zurückzuführen sind. Ein Modul, welches die Funktionalität eines bestehenden Projektes erweitern soll, bedeutet verständlicherweise einen komplexeren Integrationsprozess und eine höhere Abhängigkeitsstufe, als ein Modul welches keinerlei funktionale Verbindung zum Ursprungsprojekt hat. Trotzdem ist der Aufwand für das Integrieren eines solchen Moduls überschaubar und bei korrekter Maintenance sollten keine weiteren Probleme auftreten.

Django eignet sich durch seinen modularen Architekturstil und seine Nutzung des MTV Patterns 2.3.1 hervorragend für die Entwicklung von Web Applikationen. Der White Label Aspekt wird hier in Bezug auf die Vielfalt an Features sehr deutlich erfüllt. Neue Komponenten können in den einfachen Fällen mit wenigen Zeilen Code und einfachem Copy&Paste in ein Projekt integriert werden und selbst in den komplexeren Fällen, in welchen auch das Ursprungsprojekt Änderungen benötigt, reichen vergleichsweise wenige Schritte um ein Modul zum Laufen zu bringen.
Die Struktur eines Django Projektes und der Aufbau mit Rücksicht auf viele Module mit einem Kern entspricht exakt der Definition von Software Product Lines, wie in Abschnitt 2.1 und somit einer der grundlegenden Definitionen von White Label Applikationen. Bezüglich des Detailgrades, auf welchen sich Highly Configurable Systems noch mehr konzentrieren, ist in den in dieser Arbeit genutzten Prototypen nicht viel zu machen, also kann man die Prototypen, welche hier genutzt werden als Erfolg im Bereich der modularen White Label Entwicklung bezeichnen, allerdings nicht in der Hoch Detaillierten White Label Entwicklung.

7.1 Ausblick

Einige der Funktionen welche einem durch das Implementieren in Django aufgefallen sind, zeigen noch weiteres Potenzial in der modularen White Label Entwicklung. Gerade Django hat über die Kontextprozessoren eine Option, bestimmte Werte dynamisch und jederzeit Abrufbar für alle Module im Projekt bereitzustellen. Es ist vorstellbar, über diesen Kontext sowohl Feature Toggels zu kreieren, als auch feingranularer Attribute bereitzustellen. So kann die Definition von HCS unter Umständen auch mehr erfüllt werden.
Ohne Frage ist Django bereits, hat aber auch weiterhin das Potenzial führendes Web-Framework für das Erstellen von modularen Applikationen zu sein. Django unabhängig hat dieses Projekt gezeigt, dass mit den richtigen Patterns und der richtigen Struktur, eine White Label Applikation auch deutlich einfacher umzusetzen ist, als man zuvor hätte vermuten können.

Literatur

[ABKS16] Apel, Sven; Batory, Don; Kästner, Christian ; Saake, Gunter: *Feature-oriented software product lines*. Springer, 2016

[Bat05] Batory, Don: Feature models, grammars, and propositional formulas. In: *International Conference on Software Product Lines* Springer, 2005 S. 7–20

[CB19] Correia, Renieri; Barbosa, Eiji: Detecting Design Violations in Django-based Web Applications. DOI 10.1145/3357141.3357600, 2019 S. 33–42. – ISBN 978–1–4503–7637–2

[Dea09] Deacon, John: Model-view-controller (mvc) architecture. In: *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf* 28 (2009)

[Fre20] Freire, Matthew: *Django E-commerce*. https://github.com/justdjango/django-ecommerce, 2020

[Jee] Jee, Caitlin: *Modularization in Software Engineering*. https://medium.com/@caitlinjeespn/modularization-in-software-engineering-1af52807ceed

[KP88] Krasner, Glenn; Pope, Stephen: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk80 System. In: *Journal of Object-oriented Programming - JOOP* 1 (1988), 01

[Obr19] Obrębski, Piotr: *Ecommerce Template - Bootstrap 4 Material Design*. https://github.com/mdbootstrap/Ecommerce-Template-Bootstrap, 2019

[Øst08] Østby, Torgeir: Modularization and Demodularization: Levels of a Java Web Application for Open Health. https://api.semanticscholar.org/CorpusID:63212774

[SSM20] Silva, Franklin; Souza, Renata M. ; Machado, Ivan: Taming and Unveiling Software Reuse opportunities through White Label Software in Startups. DOI 10.1109/SEAA51224.2020.00057, 2020

[Unk24] Unknown: *HVV schafft Bargeldzahlung für Tickets im Bus ab*. https://www.ndr.de/nachrichten/hamburg/

HVV-schafft-Bargeldzahlung-fuer-Tickets-im-Bus-ab,
hvv750.html. Version: 2024

Listings

Abbildungsverzeichnis

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Entwicklung von White Label Web Applikationen mit Hilfe eines modularen Architekturstils - Eine kritische Bewertung der Wiederverwendbark

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort Datum Unterschrift im Original