Entwicklung einer Daten- und Planungsplattform zur agentenbasierten Simulation von Infrastruktur für Bevölkerung

Masterprojektarbeit
von
Herrn Christian Grünewald

im Studiengang Elektro-/Informationstechnik
mit Schwerpunkt Ingenieurinformatik

an der HAWK Hochschule für angewandte Wissenschaft und Kunst
Hildesheim / Holzminden / Göttingen
Fakultät Ingenieurwissenschaften und Gesundheit in Göttingen

Prüfer: Prof. Dr. Rer. Nat. Roman Gorthausmann

April 2023

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Abschlussarbeit selbstständig, ohne fremde Hilfe und nur unter Verwendung der angegebenen Literatur angefertigt habe. Alle fremden, öffentlichen Quellen sind als solche kenntlich gemacht. Mir ist bekannt, dass ich für die Quellen Dritter in dieser Arbeit die Nutzungsrechte zur Verwendung in dieser Arbeit benötige. Weiterhin versichere ich, dass diese Abschlussarbeit noch keiner anderen Prüfungskommission vorgelegen hat.

Göttingen, den 26. April 2023 Unterschrift

Erklärung zu Nutzungsrechten und Verwertungsrechten[1]

Ich bin hiermit einverstanden, dass von meiner Abschlussarbeit (ggf. nach Ablauf der Sperre) 1 Vervielfältigungsstück erstellt werden kann, um es der Bibliothek der HAWK zur Verfügung zu stellen und Dritten öffentlich zugänglich zu machen. Ich erkläre, dass Rechte Dritter der Veröffentlichung nicht entgegenstehen.

Göttingen, den 26. April 2023 Unterschrift

Sperrvermerk der Abschlussarbeit

- ☐ NEIN[2]
- ☐ JA / Dauer der Sperre: ☐ 3 Jahre[2] ☐ 5 Jahre[2]

Göttingen, den 26. April 2023 Unterschrift

[1] Dadurch räumen Sie der HAWK ein einfaches, zeitlich unbeschränktes, unentgeltliches Nutzungsrecht nach §§ 15 Abs. 2 Nr. 2, 16, 17, 19a, 31 Abs.2 UrhG ein.

[2] Zutreffendes bitte ankreuzen

Inhaltsverzeichnis

1 Einleitung

Eine aktuelle Nahverkehr-Studie zeigt, dass rund 55 Millionen Bürger der Bundesrepublik Deutschland unzureichende Anbindungen an öffentliche Verkehrsmittel haben. Bushaltestellen und Bahnhöfe werden zu selten bedient. Das bestehende Netz an öffentlichen Verkehrsmitteln muss aus diesem Grund ausgebaut werden [1]. Der Ausbauprozess ist mit vielen Planungs- und bürokratischen Hürden verbunden. Ein anderes Beispiel aus dem Gesundheitswesen zeigt, dass das bestehende Netz an Ärzten und Krankenhäusern aufgrund des aktuell existierenden Fachkräftemangels in Deutschland effizient überarbeitet werden muss [2].
Diese und viele weitere planungstechnische Probleme können durch Simulationen und eine Übersicht aktueller Daten entscheidend beeinflusst und beschleunigt werden. Genau hiermit beschäftigt sich diese Arbeit. Es soll eine Applikation entwickelt werden, welche alle aktuellen planungsrelevanten Daten geografisch darstellt. Wirksamkeit und Analyse bei Änderung der Daten können mittels Simulationen überprüft werden. Es soll erforscht werden, wie sich solch eine Anwendung am besten entwickeln lässt und welche technischen Grundlagen notwendig sind. Diese Arbeit beschränkt sich dabei auf das Backend der Anwendung. Es wurde im Vorfeld festgelegt, dass eine Webanwendung entwickelt werden soll, da diese plattformunabhängig und schnell entwickelt werden kann. Webanwendungen unterteilen sich in zwei wesentliche Schichten, das Frontend und Backend. Das Frontend entspricht der Benutzeroberfläche, mit welchem die Benutzer der Anwendung interagieren können. Das Backend entspricht dem Teil der Anwendung, welches für die Verarbeitung der Daten verantwortlich ist und unzugänglich für den Benutzer ist, da diese sich auf einem Server befindet und in der Regel der Logik der Anwendung entspricht [3]. Im Folgenden wird zuerst die Projektarchitektur und anschließend der Entscheidungsprozess erläutert, welches Framework für das Backend gewählt worden ist. Danach werden alle notwendigen technischen Grundlagen beschrieben und eine Übersicht über den ersten Implementationsstand geschaffen. Zum Schluss werden alle momentanen Ergebnisse zusammengefasst, ein Zwischenfazit gezogen und ein Ausblick auf künftige Arbeiten gegeben.

2 Planung und Aufbau des Projekts

Es wurde sich entschieden, die Anwendung mittels sogenanntem Microservices Ansatz umzusetzen. Dabei handelt es sich um ein Architekturmuster in der Informationstechnik, bei dem die Anwendungssoftware aus unabhängigen Prozessen erzeugt wird. Die Anwendungsteile werden modular über sprachunabhängige Schnittstellen unterteilt. Jede Teilaufgabe der Software wird in kleine Dienste entkoppelt, welche unabhängig voneinander laufen [4]. Abbildung 1 beschreibt die geplante Struktur der Anwendung.

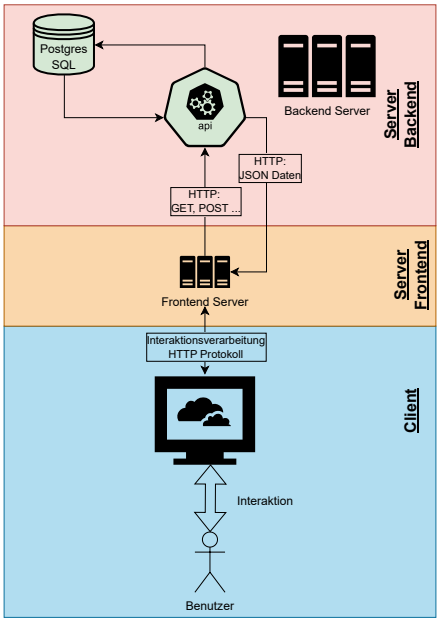

Abbildung 1: Geplante Architektur der Anwendung.

Front- und Backend sollen komplett getrennt voneinander laufen und über Schnittstellen kommunizieren. Auf dem Frontendserver werden auf Benutzerinteraktionen reagiert und verarbeitet. Bei dem Backendserver werden auf die vom Frontend geforderte Anfragen reagiert und Daten von einer Datenbank verarbeitet und über ein Interface formatiert ausgegeben. Es handelt sich um sogenannte Application Programming Interface (kurz API). Eine API kann auf eine Anfrage über ein standardisiertes Protokoll reagieren und Daten formatiert zurückgeben. Im Falle einer Webapplikation handelt es sich bei dem Protokoll um das Hyper Text Transfer Protokoll (kurz HTTP). Üblicherweise werden die Daten mittels Javascript Object Notation (kurz JSON) oder Extensible Markup Lan-

guage (kurz XML) formatiert ausgetauscht. Die gewählte Struktur hat den Vorteil, dass die Anwendung systematisch und schrittweise aufgebaut werden kann. Soll dabei die Applikation erweitert werden, so kann dies komplett unabhängig von bereits vorhandenen und funktionierenden Features passieren.

2.1 Entscheidungsprozess für die Wahl des Backendframeworks

Um das Backend einer Webanwendung zu realisieren, gibt es eine Reihe an Frameworks in diversen Programmier- und Interpretersprachen. Zu Beginn wurde untersucht, welche üblichen Programmier- und Interpretersprachen verwendet werden und was derzeit die beliebtesten Programmiersprachen sind. Laut Popularity of Programming Language Index (kurz: PYPL-iundex) ist derzeit Python die beliebteste Programmiersprache weltweit. Dabei hat Python einen deutlich höheren Anteil mit 27.7 % im Vergleich zu der zweit beliebtesten Programmiersprache Java (16.79 %) [5]. In der Entwicklung des Backends werden Sprachen verwendet wie PHP, C++, C#, Java oder auch Python. Dabei bietet jede Sprache Vor- und Nachteile. Beispielsweise wurde PHP 1994 entwickelt und wird bis heute in vielen Anwendungen eingesetzt. Der wesentliche Nachteil von PHP obliegt darin, dass es nicht mehr modern ist und im Vergleich zu modernen Sprachen eine aufwendigere Syntax hat als z.B. Python [6].
Aus diesen Gründen wurde Python als Programmiersprache im Backend gewählt. Im nächsten Schritt wurde untersucht, wie sich ein Backend in Python realisieren lässt. Aufgrund der hohen Anzahl an Python Modulen existieren die unterschiedlichsten Möglichkeiten das Backend zu realisieren. Die Module Django und Flask werden genauer betrachtet.

Tabelle 1 gibt dabei einen Überblick über die jeweiligen Module. Man kann deutlich erkennen, dass beide Module ihre Vor- und Nachteile mit sich bringen. Zwar bietet Flask die Möglichkeit das System flexibel und mit einer guten Performance zu implementieren, jedoch muss von Grund aus viele selber implementiert werden. Django bietet aufgrund seines Umfangs, Sicherheit und der Möglichkeit eine Datenbank direkt mit einzubinden, die wesentlichen Aspekte, die für die Anwendung notwendig sind. Es wäre zwar möglich

Tabelle 1: Gegenüberstellung zwischen Django und Flask [7].

	Flask	Django
Sicherheit	Manuelle Pflege der Sicherheit	Viele Sicherheitsmechanismen sind standardmäßig aktiv (wie Verschlüsselung von Benutzerpasswörtern)
Umfang	Mikro Framework mit vielen manuellen Anpassungsmöglichkeiten	Makro Framework mit vielen Features
Flexibilität	Viele Anpassungsmöglichkeiten	Geringere Anpassungsmöglichkeit
Performance	Aufgrund der Größe besser	Trotz des Umfangs gute Laufzeiten
Verbreitung	Sehr beliebt	Sehr beliebt
Datenbankanbindung und Modellierung	Externes Modul notwendig	Sehr einfach dank Django ORM
Support	Große Community	Große Community

mit Flask das Backend zu realisieren, jedoch wäre dies mit deutlich mehr Aufwand verbunden. Weil die Anwendung im späteren Verlauf mit empfindlichen Daten arbeiten soll, sowie eine Datenbank Anbindung benötigt, wird das Backend der Anwendung mithilfe in Python und dem Django Webframework realisiert.

3 Technische Grundlagen

Dieses Kapitel soll einen Überblick schaffen über die Grundlagen, welche notwendig sind, um die entstandenen Quellcodes zu verstehen. Die Anwendung wird Kartendaten und Einwohnerdaten verarbeiten müssen. Aus diesem Grund ist es wichtig zu wissen, wie geografische Daten in einem Computer dargestellt werden. Dies wird im ersten Teil erläutert. Im nächsten Teil geht es um die Funktionsweise des in der Arbeit verwendeten Webframeworks und das gewählte Datenaustauschformat.

3.1 Geografische Daten

Um Karten darzustellen, benötigt man geografische Daten. Diese sind digitale Informationen, denen auf der Erdoberfläche eine räumliche Lage zugewiesen wird. Es werden Standortinformationen (geografische Koordinaten der Erde) und Attributinformationen mit zeitlichen Informationen kombiniert. In der Regel werden große Mengen räumlicher Daten verarbeitet. Diese können zudem Informationen wie Bevölkerungszahlen, Wetterdaten oder Handydaten enthalten [8].

3.1.1 Mathematische Grundlagen zu geografischen Koordinaten

Um Geodaten darzustellen, benötigt man ein Koordinatensystem, welches klassischerweise dreidimensional ist. Die Erde ist keine Kugel, sondern ähnelt einem Ellipsoid. Mathematisch betrachtet lassen sich mit Ellipsoiden besser rechnen, weshalb die Erde als Ellipsoid angenähert wird. Aus diesem Grund ist das hier zugrunde liegende Koordinatensystem ein Kugelkoordinatensystem. Sogenannte geografische Koordinaten beschreiben Kugelkoordinaten, mit denen die Lage eines Punktes auf der Erdoberfläche darstellen lassen, indem ein sogenanntes Gradnetz auf der Erdoberfläche gespannt wird (vgl. Abbildung 2). Das Gradnetz beschreibt ein konstruiertes Koordinatensystem auf der Erdoberfläche. Dabei wird die Erde in 180 Breitengraden und 360 Längengraden aufgeteilt. Längen- und Breitengraden sind orthogonal zueinander. Das Koordinatensystem dient zur geografischen Ortsbestimmung. Breitengraden werden vom Äquator aus gezählt, Nord- bzw. Südpol liegen bei 90°. Bei den Längengraden werden willkürlich vom

Nullmeridian nach Osten und Westen gezählt bis je 180°. Bei den Koordinaten ist jedoch zu beachten, dass aufgrund der geometrischen Form der Erde eine Verschiebung von bis zu 20 km wirken können. Weshalb auch aufgrund zunehmenden Erkenntnis der Forschung unterschiedliche Bezugssysteme existieren. Standardmäßig wird jedoch meist das World Geodetic System 1982 (kurz WSG84 oder EPSG (Geodetic Parameter Dataset) 4326) genutzt [9, 10].

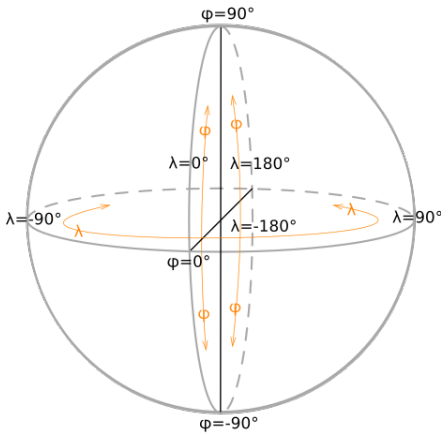

Abbildung 2: Geografisches Koordinatensystem der Erde im Gradnetz [10].

Karten werden jedoch nicht dreidimensional dargestellt. Um die Position auf einer Erdoberfläche als Karte darzustellen, benötigt man eine Projektion dieses Koordinatensystems, weil Karten als Ebene dargestellt werden. Es gibt dabei unterschiedliche Möglichkeiten, die Erde als Ebene zu projektieren (vgl. Abbildung 3). Der Prozess zur Erstellung von Kartenprojektionen lässt sich veranschaulichen, indem eine Lichtquelle in einem durchsichtigen Globus positioniert wird. Die undurchsichtigen Merkmalsumrisse werden dann projektiert. Verschiedene Projektionsformen können erzeugt werden, indem man den Globus zylindrisch, als Kegel oder sogar als ebene Fläche umgibt. Jede dieser Methoden erzeugt eine sogenannte Kartenprojektionsfamilie. Daher gibt es eine Familie von planaren Projektionen, eine Familie von zylindrischen Projektionen und eine andere, die konische Projektionen genannt wird. Aufgrund der Kartenprojektionsprozesse weist jede Karte Verzerrungen der Winkelkonformität, Entfernung und Fläche auf. Eine Kartenprojektion kann mehrere dieser Eigenschaften kombinieren oder kann ein Kompromiss sein, der alle Eigenschaften von Fläche, Entfernung und Winkelkonformität innerhalb einer akzeptablen Grenze verzerrt.

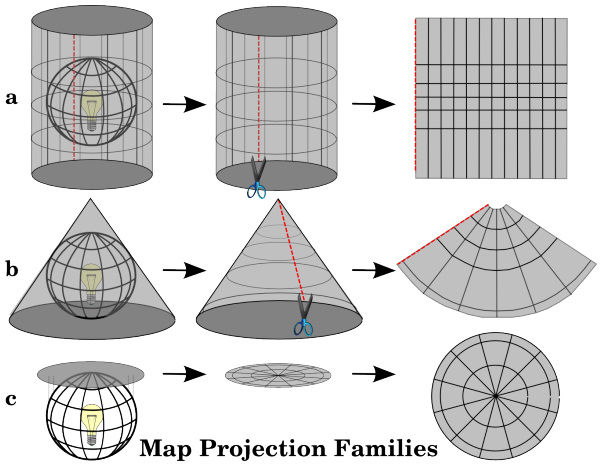

Abbildung 3: Möglichkeiten zur Projektion der Erde als Ebene [11].

Um nun eine Karte darzustellen, gibt es unterschiedliche Koordinatenbezugssysteme (kurz KBS). Allgemein können KBS in projizierte Koordinatenbezugssysteme (auch kartesische oder rechtwinklige Koordinatenbezugssysteme genannt) und geografische Koordinatenbezugssysteme unterteilt werden. Ein zweidimensionales Koordinatenbezugssystem wird üblicherweise durch zwei Achsen definiert. Rechtwinklig zueinander bilden sie eine XY-Ebene. Die horizontale Achse wird normalerweise mit X bezeichnet und die vertikale Achse mit Y. Jeder Punkt, der in sphärischen Koordinaten ausgedrückt wird, kann als X-Y-Koordinate ausgedrückt werden. Jedes Bezugssystem ist in einer anderen Einheit dargestellt, EPSG 4326 hat als Einheit Grad. In Deutschland wird EPSG 25832-25833 in der Einheit Meter verwendet [11]. Sogenannte EPSG-Codes sind ein System weltweit eindeutiger, 4- bis 5-stelliger Schlüsselnummern (SRIDs) für Koordinatenreferenzsysteme und andere geodätische Datensätze [12].

3.1.2 Darstellung geografischer Informationen in Computerdaten

Um geografische Objekte zu speichern und verarbeiten zu können, wird ein spezielles Datenformat benötigt, sogenannte Shapefile-Format. Es handelt sich hierbei um ein allgemeines Format, um Vektor-GIS (Geo Informations System)-Daten zu speichern. Neben den Geometrieinformationen wie Position und Form können weitere Attributdaten gespeichert werden. Shapefiles wurden von Esri (Environmental Systems Research Institute) entwickelt. Obwohl der Name eine einzelne Datei angibt, ist ein Shapefile eine

Sammlung von mindestens drei Basisdateien: .shp, .shx und .dbf. Jede der drei Dateien muss im gleichen Verzeichnis sein, um diese darstellen zu können. Es können Punkte, Flächen, Polygone und Multipolygone als Geometrie dargestellt werden [13]. Das Datenformat eignet sich jedoch nicht, um zwischen Front- und Backend Informationen auszutauschen. Aus diesem Grund werden die Daten mittels Javascript Objekt Notation ausgetauscht.

3.1.3 Javascript Objekt Notation als Datenaustauschformat

Die Javascript Objekt Notation ist ein bekanntes und kompaktes Datenformat zum Informationsaustausch zwischen Anwendungen, ähnlich wie Extensible Markup Language (kurz XML). Dabei ist JSON unabhängig von der Programmiersprache, da in allen gängigen Sprachen Parser existent sind. Es wurde erstmals 1997 von Dougles Crockford entwickelt. Vor allem in Webanwendungen wird dieses Datenformat sehr häufig eingesetzt. Aus diesem Grund wird dieser auch für diese Anwendung benutzt. JSON erlaubt eine beliebige Verschachtlung der Daten. Es wird standardmäßig in Unicode codiert (UTF-8). In JSON gibt es dabei unterschiedliche Typen von Elementen wie Zahlen. Aufgrund der Variabilität von Daten gibt es JSON Erweiterungen. Diese verfügen neben den Standard JSON Elementtypen auch noch über spezifische Zusätze. Weil sich das Projekt mit Kartendaten befassen wird, wird die JSON Erweiterung GeoJSON benutzt. GeoJSON dient zur Repräsentation von geografischen Daten [14]. GeoJSON unterscheidet sich zu JSON nur minimal, es werden die geografischen Typen von Geometrien, welche notwendig sind, um komplette Karten darzustellen. Die Position wird in der Regel mit einer Koordinate aus Längen- und Breitengrad angegeben. Wie auch bei den Shapefiles werden grundlegende geometrische Formen genutzt. Eine GeoJSON formierte Datei sieht dabei wie im Quellcode 1 dargestellt aus. Jede GeoJSON beginnt immer mit der Beschreibung des Typs, hierbei können einzelne Elemente oder Sammlungen wie FeatureCollection erzeugt werden. Innerhalb einer FeatureCollection können mehrere geometrische Informationen und Attributinformationen wie Name oder Adresse enthalten sein [15].

```
1 {
2   "type": "FeatureCollection",
3   "features": [
4   {
```

```
    "type": "Feature",
    "properties": {
      "name": "Muster AG",
      "address": "Musterstrasse 1"
    },
    "geometry": {
      "type": "Point",
      "coordinates": [12.0069, 51.1623]
    }
  }
  ]
}
```

Quellcode 1: Beispiel einer Geo JavaScript Objekt Notation Formatierten Datei.

GeoJSON formatierte Daten können dabei beliebig komplex und groß aufgebaut sein. Es können sämtliche notwendige Informationen formatiert gespeichert oder übertragen werden. GeoJSON hat den wesentlichen Vorteil, dass diese in jeder Programmiersprache genutzt werden, welche JSON beherrscht, deshalb wurde dieses Format auch gewählt.

3.2 Webentwicklung mit Django

Das Django Webanwendungs-Framework ist ein in Python entwickeltes, quelloffenes Webframework, dass die Erstellung von Webseiten deutlich vereinfacht. Bei der Entwicklung von Webanwendungen werden immer wieder ähnliche Elemente benötigt, z. B. wie das Anbinden einer Datenbank, das Verschlüsseln von Formulardaten oder immer wiederkehrende Codeteile. Diese Elemente bieten Webframeworks als kleine Bausteine anwendungsbereit an. Django wurde im Jahr 2005 erstmals veröffentlicht und wird bis heute stetig weiterentwickelt [16, 17].

3.2.1 Projektstrukturen von Django

In Django gibt es zwei Arten von Strukturen, welche unterschieden werden: Projekte und Apps. Django Projekte sind vollständig neue Server und Anwendungen, welche unabhängig voneinander laufen und entwickelt werden können, da jeder seinen eigenen Server mit den eigenen Einstellungen und Datenbank betrieben werden können. In diesem Teil wird das Verhalten von Server und Datenbank oder sonstige Einstellungen definiert. Django Apps sind kleine Bausteine von Django Projekten. Diese sind abhängig von Django Projekten und können nur innerhalb von diesen generiert werden. In diesem Teil der gesamten Django Anwendung werden Funktionalität und Design realisiert. Zuerst werden Django Projekte, danach Django Apps, anhand des angelegten Projekts erläutert. Als Erstes muss ein Projektordner anlegt werden, eine virtuelle Umgebung von Python erzeugt und aktiviert werden, wie in Quellcode 2 gezeigt.

```
1  #!/bin/bash
2  $ mkdir backend_django
3  $ cd ./backend_django
4  $ python -m venv env
5  $ env/Scripts/Activate
6
```

Quellcode 2: Anlegen einer virtuellen Python Umgebung für ein Django Projekt.

Als Nächstes muss Django installiert werden. Danach kann ein neues Django Projekt erzeugt werden, wie in Quellcode 3 gezeigt.

```
1  #!/bin/bash
2  $ pip install django
3  $ django-admin startproject morizon_backend .
```

Quellcode 3: Installation von Django und Anlegen eines neuen Projekts.

Wie man sieht, generiert Django eine ganze Reihe an Ordnern und Python-Dateien. Das erste Argument beschreibt dabei den Namen des Projekts, das zweite Argument legt den Pfad des Projekts fest. Wenn ein „.“ eingegeben wird, wird der aktuelle Ordner als Pfad angelegt. Nun ergibt sich die wie in Abbildung 4 beschriebene Ordnerstruktur [18, 19].

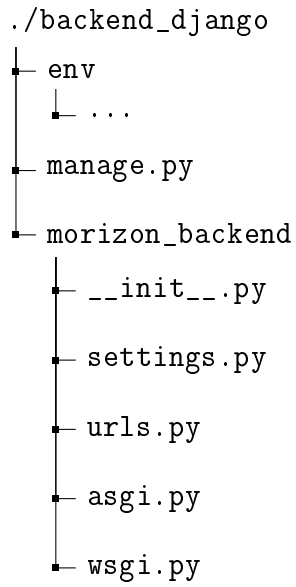

Abbildung 4: Struktur eines Django Projekts.

Innerhalb des Ordners backend_django liegen jetzt sämtliche Einstellungs- und Verwaltungsdaten. Auf Details werden im nächsten Unterkapitel eingegangen. Im Folgenden wird kurz erläutert, was welche Datei enthält, und was wo beachtet werden sollte.

- Das äußere „backend_django/“ Wurzel-Verzeichnis ist ein Container für das Projekt. Sein Name spielt für Django keine Rolle. Diese kann beliebig umbenannt werden.
- „manage.py": Ein Befehlszeilenprogramm, mit dem auf verschiedene Weise mit dem Django-Projekt interagiert werden kann. Es gibt eine Reihe an Befehlen, welche durchgeführt werden können. Wie z. B. das Starten des Servers oder das Ausführen von Datenbank-Operationen (ausführen von SQL).
- Das innere Verzeichnis „backend_django/“ ist das eigentliche Python-Paket für das Projekt. Sein Name ist der Python-Paketname, der verwendet werden muss, um alles darin zu importieren (z. B. mysite.urls).
  - „morizon_backend/__init__.py“: Eine leere Datei, die Python mitteilt, dass dieses Verzeichnis als Python-Paket betrachtet werden soll.
  - „morizon_backend/settings.py“: Konfiguration für das Django-Projekt. Beispielsweise kann hier die Verbindung zur Datenbank eingestellt werden.

- „morizon_backend/urls.py“: Die Uniform Resource Locator (kurz URL) - Deklarationen für das Django-Projekt. Was eine Art „Inhaltsverzeichnis“ der Django-betriebenen Website darstellt.
- „morizon_backend/asgi.py“ und „morizon_backend/wsgi.py“: synchroner und asynchroner Einstiegspunkt für den Webserver [18]

Ein Django Webserver kann auf zwei Arten Daten verarbeiten. WSGI (steht für Web Server Gateway Interface) in dieser Betriebsart arbeitet der Server alle Anfragen synchron ab, d.h. nacheinander und synchron. ASGI steht für Asynchronous Server Gateway Interface. Hier werden die Anfragen parallel abgearbeitet und je nach Aufwand kommen die Antworten vom Server zeitversetzt, also asynchron (`def Funktion()...`). Soll eine Funktion asynchron laufen, muss das Python mit dem Schlüsselwort `async def Funktion()` kenntlich gemacht werden. Standardmäßig werden alle Anfragen in Django synchron abgearbeitet [20]. Um nun eine Funktionalität zu implementieren, muss eine Django App generiert werden, dies kann wie in Quellcode 4 realisiert werden.

```
#!/bin/bash
$ python manage.py startapp perimeter_search
```

Quellcode 4: Anlegen einer neuen Django App.

Die wie in Abbildung 5 entstandene Ordnerstruktur wird erzeugt. Dabei beschreibt „perimeter_search“ den Namen der Django App. Nun kann ein neuer Ordner namens „perimeter_search“ in dem Wurzelpfad gefunden werden. Innerhalb dieses erzeugt Django dabei nun wieder eine Reihe von Ordner und Dateien. Die wichtigsten sind dabei „views.py“, „models.py“ und „admin.py“.

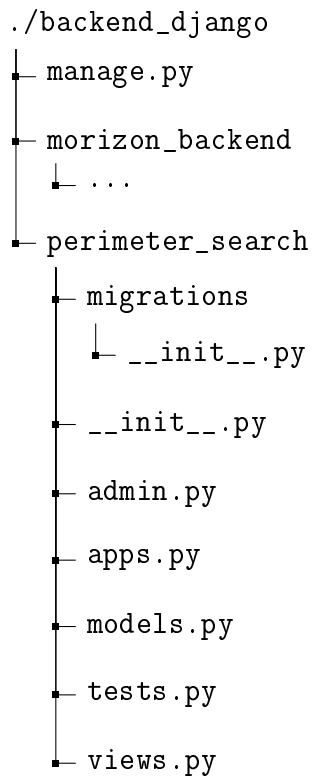

Abbildung 5: Struktur einer Django App.

In „models.py“ werden Datenbankmodelle mithilfe von Django erzeugt. Diese können auf der automatischen und bereits einsatzbereiten Django Administrations Oberfläche angezeigt und verwaltet werden, indem sie „admin.py“ registriert werden. In „views.py“ werden Django Views, also die Funktionalität der Backends programmiert [18]. Durch Djangos Unterteilung in Projekte und Apps wird eine Django Anwendung übersichtlich und kann modular erweitert werden. Um Django kenntlich zumachen, dass diese App auch verwendet werden soll, wird im Django Projekt unter „settings.py“ eingestellt, indem die Liste `INSTALLED_APPS` um einen Eintrag als Namen erweitert wird als Zeichenkette (vergleiche Quellcode 7).

3.2.2 Funktionsweise von Django

Django ist im Grunde ein Model View Controller (kurz MVC) Framework. Das MVC ist ein Entwurfsmuster aus der Softwareentwicklung. Es ist darauf gezielt die Webanwendung in drei miteinander verbundene Teile zu unterteilen.

- Model: Dies ist die Schicht des Entwurfsmuster, was die Schnittstelle mit der Datenbank enthält, welche die Anwendungsdaten enthalten.
- Ansicht: In dieser Schicht wird geregelt, welche Informationen dem Benutzer gezeigt, sowie neu gespeichert werden.
- Controller: Der Controller enthält die gesamte Logik der Anwendung und dient als Schnittstelle zwischen Model und View.

Django verwendet in der Implementierung des MVC eine etwas andere Terminologie. Statt Model View Controller verwendet Django das sogenannte Model View Template, wie in Abbildung 6 gezeigt.

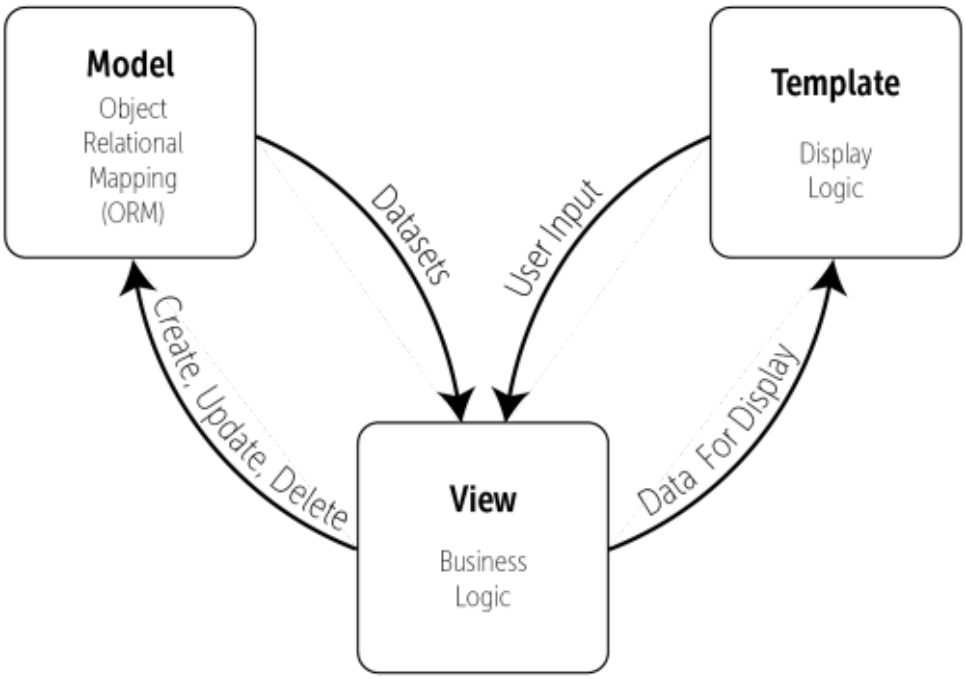

Abbildung 6: Softwareentwurfsmuster von Django [21].

- Modell (engl. Model): Funktional gleich wie in dem MVC-Muster. Django verwendet hier eine sogenannte objektrelationale Abbildung (engl. Object-relational mapping, kurz ORM) um die Daten zwischen Datenbank und Oberfläche auszutauschen (mehr zum ORM später).
- Ansicht (engl. View): Verwaltet den größten Teil der Anwendungsdatenverarbeitung. Anwendungslogik sowie Nachrichtenübermittlung sind hier implementiert. Dies passiert in der Regel auf einem Webserver im Hintergrund.
- Vorlage (engl. Template): Diese Schicht stellt die Anzeigelogik der Webanwendung dar und dient als Schnittstelle zwischen Benutzer und der Django-Anwendung.

Sämtliche Schichten Djangos MVT werden nun genauer erläutert [21, 22].

3.2.3 Datenbankmodellierung in Django mithilfe Objektrelationaler Abbildung

Benutzerdaten werden in der Regel in einer Datenbank gespeichert und verwaltet. In Django werden, wie bereits erwähnt, Daten als ORMs bereitgestellt. ORM ist eine Programmiertechnik, welche das Arbeiten mit relationalen Datenbanken erheblich erleichtern soll. Datenbanken werden üblicherweise in Structured Query Language (kurz SQL) programmiert. SQL kann je nach Anforderung an die Datenbank komplex werden. ORM ist in der Lage eine einfache Zuordnung zwischen einem Objekt (das „O“ in ORM) und der zugrunde liegenden Datenbank herzustellen. Dabei muss der Programmierer weder die Datenbankstruktur noch SQL zum Bearbeiten und Abrufen von Daten kennen. In Django ist das Modell das Objekt, dass der Datenbank zugeordnet wird. Sobald in Django ein Modell erstellt wird, so wird eine entsprechende Tabelle in der Datenbank generiert, ohne eine einzige Zeile SQL schreiben zu müssen. Wie in Abbildung 7 beispielhaft dargestellt, werden in Django Tabellen in einer Datenbank angelegt.

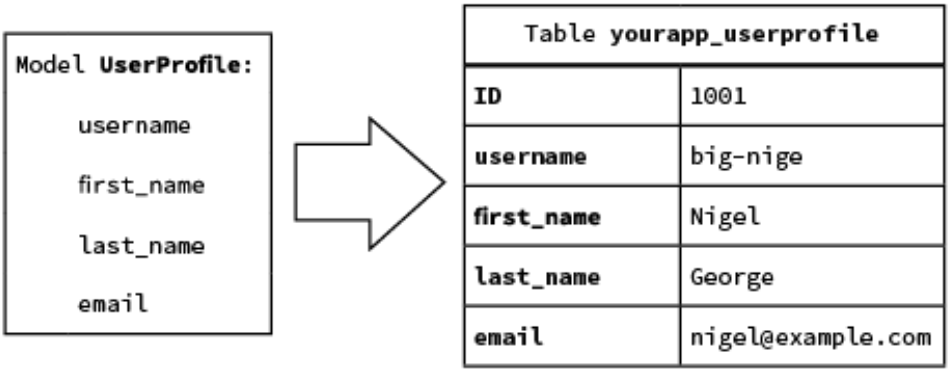

Abbildung 7: Beispielhafte in Objektrelationalen Abbildung erzeugte SQL-Relation [21].

Der Name der Tabelle entspricht dabei dem Namen der Anwendung, welche getrennt durch einen „_“ mit dem Objektnamen (z.B. entspricht das Objekt „UserProfile“ dann der SQL-Tabelle „yourapp_userprofile“, wobei „yourapp“ dann dem Namen der Anwendung entspricht). Programmiert werden ORM-Objekte in der Datei namens „models.py“. Jedes ORM entspricht einer Python-Klasse. Wie üblich in SQL lassen sich in Djangos ORM auch Tabellen miteinander verknüpfen, dies funktioniert über die Djangos Funktion „ForeignKey“. Ab Django 3.0 und höher werden alle gängigen SQL-Datenbanken unterstützt (z.B. PostgreSQL, MySQL, MariaDB, SQLite ...) [21, 22]. Webframeworks müssen in der Lage sein, wie PHP, Hyper Text Markup Language (kurz HTML) dynamisch generieren zu können. Üblich werden sie mithilfe von Templates realisiert. Ein Template enthält statische Teile, eine HTML-Ausgabe sowie eine spezielle Syntax, welche die dynamischen Inhalte des Dokuments einfügen. Standardmäßig verwendet Django seine eigene Template-Sprache zum Generieren von dynamischen Inhalten. Eine Django-Vorlage ist ein Textdokument oder eine Python-Zeichenfolge, die mit der Django-Vorlagensprache ausgezeichnet wurde. Einige Konstrukte werden von der Template-Engine erkannt und interpretiert. Ein Template wird mit einem Kontext gerendert, was als Datentyp Dictionary (deutsch Wörterbuch, Standard Python Datentyp) der Renderingmethode als Argument mitgegeben wird. Beim Prozess des Renderns wird im HTML-Dokument nach Django-Templates Zeilen gesucht. Es werden nach Variablen und Tags gesucht oder anderen Template Elementen und dynamisch durch beispielsweise HTML ersetzt. Nach diesem wird das Ergebnis über eine Django Standard Application Programming Interface (engl. für Schnittstelle Programmiersprache und Anwendung, kurz API) zu dem Client übertragen. Wie in der Python Funktion Index in Quellcode

5 dargestellt, können HTML-Dokumente übertragen werden. Dabei enthält die Variable „request“ sämtliche Informationen über die HTTP Anfrage des Servers, wie zum Beispiel GET/POST Variablen. Das zweite Argument beschreibt den Pfad zur HTML Datei, dabei muss beachtet werden, dass Django standardmäßig im Applikationsordner (nicht Projektordner) „templates“ nach dem HTML Dokument sucht. Es besteht jedoch auch die Möglichkeit, dass HTML-Dokumente in einem anderen Pfad abgelegt werden können. Dies kann explizit in den Einstellungen des Django Projekts („settings.py“) konfiguriert werden [21–23]. Eine normale Django Anwenndung besitzt auch ein Frontend zur Präsentation der Daten, diese werden mit Templates dynamisch generiert, da diese Arbeit sich jedoch mit dem Backend der Anwendung beschäftigt, werden Details dieses Bereichs in den Grundlagen ausgelassen.

3.2.4 Django Views

Dieser Bereich beschreibt die Informationsvermittler, d. h. die Logik einer Django-Anwendung. Hier werden die APIs, also die Schnittstelle zu dem Frontend programmiert. Eine View bezieht Daten aus Ihrer Datenbank (oder einer externen Datenquelle oder einem externen Dienst) und liefert sie an ein Template. Für eine Webanwendung liefert die View Webseiteninhalte und Templates, für eine RESTful-API könnten diese Inhalte ordnungsgemäß formatierte JSON-Daten sein. Es gibt zwei Arten, wie Views definiert werden können. Views werden entweder durch eine Python-Funktion oder eine Methode einer Python-Klasse dargestellt. In den frühen Tagen von Django gab es nur funktionsbasierte View, aber als Django im Laufe der Jahre gewachsen ist, haben die Entwickler von Django klassenbasierte Ansichten zu Django hinzugefügt. Klassenbasierte Views erweitern die Ansichten von Django sowie integrierte Ansichten, die das Erstellen allgemeiner Ansichten (wie das Anzeigen einer Liste von Artikeln) einfacher zu implementieren machen. Um die Programmierer zu entlasten, verfügen viele gängige Anzeigeaufgaben über integrierte Ansichten in Django (z. B. Seite konnte nicht gefunden werden, 404 View)[21].
Views in Django zu implementieren ist eine der Hauptaufgaben, wenn eine Webanwendung mittels Django realisiert werden soll. Insbesondere wenn Django nur als Backend benutzt wird, ist dies das Hauptaugenmerk. Eine API sendet Java Objekt Notation (kurz JSON) formatierte Daten als Antwort auf Anfragen. Dies kann in Django mithilfe der

Funktion JsonResponse getan werden. Diese formatiert die Antwort als JSON, indem ein Python Dictionary ausgewertet wird, wie in Zeile 8 im Quellcode 5 gezeigt. Sobald diese aufgerufen wird, über die verlinkte URL wird die Antwort angezeigt: „"Hallo Welt": True".

```
1  from django.http import HttpResponseRedirect, JsonResponse
2  from django.shortcuts import render
3
4  def index(request):
5    render(request, "index.html", {'kontext':kontext})
6
7  def MyAPI(request):
8    return JsonResponse({"Hallo Welt": True})
```

Quellcode 5: Ein Beispiel Django View für Front oder Backend

Views können in Django über Uniform Resource Locators (kurz URL) aufgerufen werden, diese sind mit der gewünschten Python-Funktion verlinkt, z.B. Index oder MyAPI wie oben dargestellt. Wie URLs in Django mit Python-Funktionen verbunden werden, wird im Folgenden erläutert.

3.2.5 Hypertext Transfer Protokoll und Uniform Resource Locator

Webanwendungen müssen in der Lage sein, den Nutzer innerhalb der Anwendung navigieren zu können, um zu entscheiden, was wann dem Nutzer gezeigt wird. Die Navigation innerhalb einer Django-Applikation wird auch wie üblich für jede Webanwendung mittels Uniform Resource Locator (kurz URLs) realisiert. Sobald innerhalb einer Django-Applikation auf einen Link geklickt wird, wird die Anfrage direkt an Django gesendet, wie in Abbildung 8 gezeigt. Django sucht nun in „urls.py“ nach dem passenden Link und führt die darin verknüpfte View aus (entweder als Python Funktion oder als Python Funktion einer View Klasse), wie in Abbildung 10 beispielhaft illustriert.

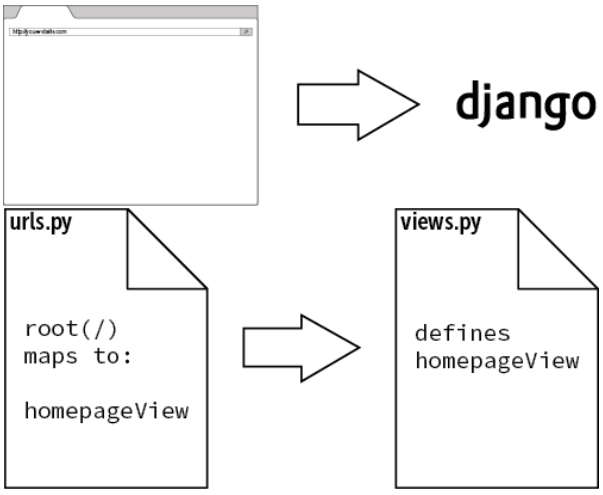

Abbildung 8: Beispielhafter Aufruf einer Django Webseite in einem Browser und Zuordnung einer angeforderten URL zu einer View [21].

Nach dem Ausführen wird beispielsweise ein neues HTML-Dokument dem Client bereitgestellt. Abbildung 9 veranschaulicht diesen Sachverhalt. Wie schon erwähnt, kann ein HTML-Dokument mittels der Django Funktion „render“ ausgewertet werden [21].

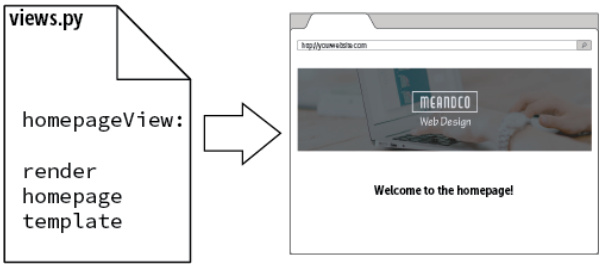

Abbildung 9: Beispielhafte Darstellung einer Webseite als Antwort einer Webserveranfrage in Django [21].

4 Implementation

Wie in Abbildung 1 bereits kurz dargestellt, wird ein Überblick über den bisher realisierten Backend-Teil der Anwendung geschaffen. Zuerst wird die generelle Struktur und Organisation des Backends erläutert. Danach wird beispielhaft eine realisierte API beschrieben, welche zur Umkreissuche von einer künstlichen Bevölkerung genutzt wird.

4.1 Quellcodestruktur und Versionsverwaltung mittels Git

Git ist inzwischen der Standard zum Versionsverwalten eines Softwareprojekts geworden. Aus diesem Grund wurde sich auch dazu entschieden, die Softwareentwicklung mittels Git zur Versionierung und zentral auf Github für alle Beteiligten zur Verfügung zu stellen. Zu diesem Zweck wurde eine Organisation unter dem Namen „morizon-digital-twin“ auf Github erstellt. Das Backend wird in dem Repository „backend_Django“ entwickelt. Für die Entwicklung wurde ein Zweig namens „development“ erzeugt, in diesem befinden sich alle Features, welche sich aktuell in der Entwicklung befinden. Nach der Fertigstellung werden diese im main Zweig gemerged. Um alle notwendigen Python Bibliotheken in der virtuellen Umgebung aufzulisten, wurde „requirements.txt“ mit Pip erzeugt (`$ pip freeze > requirements.txt`). So wird sichergestellt, dass alle essentiellen Bibliotheken bei jedem Nutzer direkt installiert werden und die Anwendung somit startbereit ist (`$ pip install -r requirements.txt`). In Abbildung 10 wird ein grafischer Überblick verschafft. Das Backend wurde in zwei Teile unterteilt. Einmal das Django Projekt, wo Konfigurationen des Server vorgenommen werden und Apps registriert werden. Des Weiteren beinhaltet dieser Bereich des Backends eine Datei namens „manage.py“, welche, wie im Kapitel 3 bereits erläutert, eine Art Kommandozeilenprogramm ist und für das Managen des Backend verantwortlich ist. Der zweite Bereich des Quellcodes sind Apps. Diese enthalten die eigentlichen APIs und Datenbankmodelle. Dabei bietet die App „perimeter_search“ eine API für eine geografische Umkreissuche von einer synthetischen Bevölkerung an. Die App „doctor_service“ enthält ein Datenbankmodell. Es enthält zudem eine Funktion, womit selbst angelegte Ärzte aus einer Excel Tabelle oder eine JSON Datei, welche automatisch heruntergeladene Ärzte hochgeladen werden können und eine Schnittstelle, womit Ärzte ausgegeben werden können. Die letzte App „BasicGeoDataDB“ enthält ein Datenbankmodell für Geodaten.

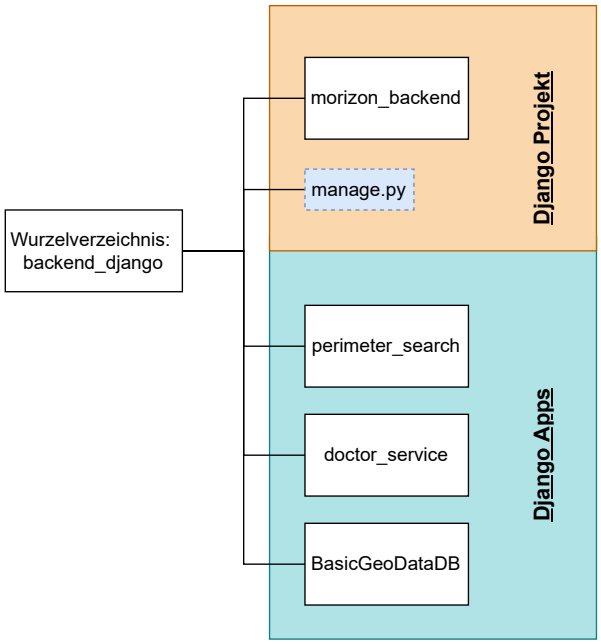

Abbildung 10: Architektur des implementierten Django-Backends.

Abbildung 11 illustriert schematisch die grundlegende Funktionsweise der Django Apps des Backends. Wie man sieht, wird direkt die HTTP Anfrage in die URLS geführt, diese sucht den entsprechenden Link und führt die verlinkte Funktion aus. In Abhängigkeit, welche URL angefragt und mit welcher HTTP-Methode, wird eine JSON Antwort generiert. Diese setzt sich aus den Daten der Datenbank zusammen und werden ggf. in einer zusätzlichen Python-Klasse verarbeitet. Wird eine Anfrage an eine View mit der falschen HTTP-Methode gesendet, so reagiert diese darauf mit einem „wrong HTTP Method“ und ignoriert die übergebenen Argumente. So wird sichergestellt, dass der Datenverkehr sicher und auf zuverlässigem Wege abläuft.

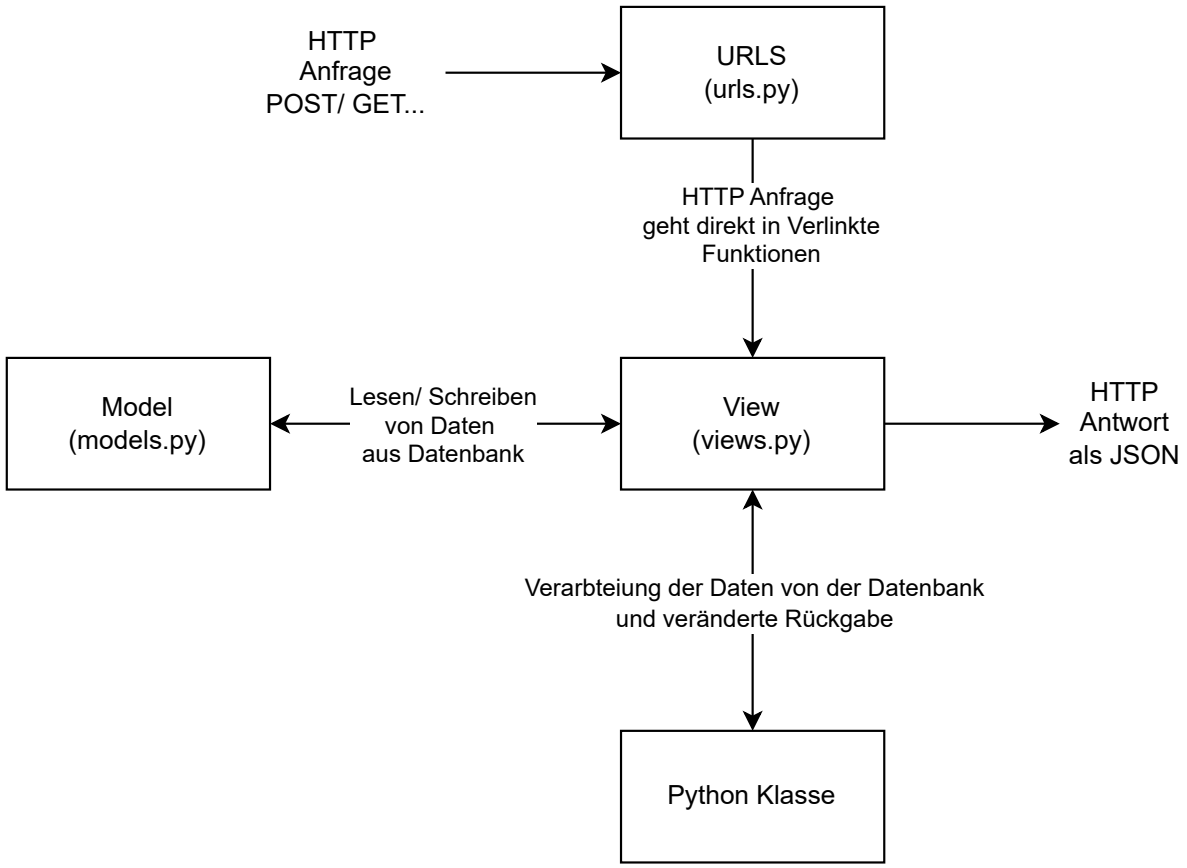

Abbildung 11: Grundlegende Funktionsweise der Apps des Backends.

4.2 Django Konfiguration und Uniform Resource Locator für Navigation zu den APIs

Wie bereits angedeutet, werden sämtliche Einstellungen wie die Verbindung zur Datenbank in der „settings.py“ des Django Projekts (nicht einer Django App) vorgenommen. Eine Django Einstellung besteht im wesentlichen aus Listen und Dictionaries, wo z.B. Django Apps registriert werden können (wie in Quellcode 6 gezeigt).

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'corsheaders',
    'django.contrib.sessions',
    'django.contrib.messages',
```

```
8       'django.contrib.staticfiles',
9       'doctor_service',
10      'perimeter_search',
11      'BasicGeoDataDB'
12  ]
```

Quellcode 6: Django Einstellung, Registrierte Apps.

Jedes Django Projekt hat einen sogenannten „SECRET_KEY“ (deutsch Geheimschlüssel) in den Einstellungen (vgl. Quellcode 7). Dieser Schlüssel wird verwendet, um kryptografische Signaturen bereitzustellen. Diese Signaturen werden benutzt, um Daten zu verschlüsseln. Es handelt sich hier um eine sogenannte Hashfunktion (oder auch Streufunktion). Es gibt unterschiedliche Hash Funktionen, standardmäßig verwendet wird SHA (Secure Hash Algorithm)-256 [24]. Dieser sollte aus diesem Grund immer geheim bleiben und in keinem Django Repository als Klartext gespeichert werden. Deshalb wurde das Python Modul „environ “ genutzt. Hier können empfindliche Informationen in eine .env Datei abgelegt werden und ins .gitignore hinzugefügt werden. So wird sichergestellt, dass empfindliche Informationen wie „SECRET_KEY“ oder Datenbankverbindungen (also Passwort, IP und Benutzername) nicht auf github veröffentlicht werden. Des Weiteren können Fehlerausgaben zum Testen in Django angezeigt werden (Zeile 20). In diesen werden bei einem Fehler automatisch Debug-Seiten angezeigt und enthalten sämtliche Informationen über den aufgetretenen Fehler. In Zeile 22 kann festgelegt werden, unter welcher IP-Adresse der Webserver erreichbar ist.

```
1  from pathlib import Path
2  import os
3
4  import environ
5  env = environ.Env()
6  environ.Env.read_env("./morizon_backend/env")
7
8
9  # Build paths inside the project like this: BASE_DIR / 'subdir'.
10 BASE_DIR = Path(__file__).resolve().parent.parent
11
12
13 # Quick-start development settings - unsuitable for production
14 # See https://docs.djangoproject.com/en/4.1/howto/deployment/checklist/
15
16 # SECURITY WARNING: keep the secret key used in production secret!
```

```
17 SECRET_KEY = env("SECRET_KEY")
18
19 # SECURITY WARNING: don't run with debug turned on in production!
20 DEBUG = True
21
22 ALLOWED_HOSTS = [
23     'localhost',
24     '127.0.0.1',
25 ]
```

Quellcode 7: Django Einstellung Wichtige Vorkonfigurationen.

Um eine Verbindung zur Datenbank zu erzeugen, muss der „DATABASES“ Eintrag angepasst werden, da standarmäßig Django SQLlite als Datenbank verwendet und für die Anwendung eine Postgres-SQL Datenbank vorgesehen ist. Dies kann in `'ENGINE'` eingestellt werden. Die anderen Einträge legen Benutzername, Passwort und unter welcher IP bzw. Port die Datenbank angesprochen werden kann. Auch diese Informationen befinden sich in der .env Datei.

```
1 DATABASES = {
2     'default': {
3         'ENGINE': 'django.db.backends.postgresql_psycopg2',
4         'NAME': env("DB_NAME"),
5         'USER': env("DB_USER"),
6         'PASSWORD': env("DB_PASSWORD"),
7         'HOST': env("DB_HOST"),
8         'PORT': env("DB_PORT"),
9     }
10 }
```

Quellcode 8: Django Einstellung Verbindung zur Datenbank.

4.3 Umkreissuche der synthetischen Population

Diese App des Backends soll künftig für eine Personenumkreissuche (eine Filterung von Datenpunkten an einem gewünschten Ort mit gewünschter Entfernung in Metern) einer künstlich angelegten Bevölkerung eingesetzt werden. Außerdem wird im Laufe der Entwicklung ein Datenbankmodell entwickelt, welche im späteren Verlauf alle synthetischen Bevölkerungsdaten beinhalten und mit der Simulation interagieren sollen, wenn

Änderungen stattfinden. Als erster Entwicklungsschritt wurde auf der Modell zuerst verzichtet und die Daten in eine GeoJSON Datei lokal ausgelesen, um so zuallererst die Funktionalität zu implementieren sowie zu testen.
Hierzu wurde eine neue Python Klasse angelegt, welche mithilfe von Geopandas, ein Python Modul für geografische Datenverarbeitung, namens „GeoDataProcessing“. Eine Person wird als Punkt im Koordinatensystem der Karte angezeigt und besitzt gewissen Attribute (wie Alter und Geschlecht), welche später in eine Datenbank ausgelagert werden soll, hierzu wird ein Django Modell benötigt. Prinzipiell funktioniert die Umkreissuche wie folgt:
Zuerst wird ein Kreis um einen gewählten Punkt des Koordinatensystems der Karte angelegt, die Koordinaten werden dabei vom Frontend vom Benutzer gewählt. Die Distanz innerhalb welcher die geforderten Daten liegen sollen, werden zudem auch vom Benutzer festgelegt und übertragen. In Abhängigkeit dieser Parameter wird nun der bereits angelegte Kreis annähernd vergrößert. Der Radius von diesem Punkt entspricht in etwa der gewünschten Entfernung in Metern. Im nächsten Schritt wird die eigentliche Filtrierung der Daten berechnet. Mathematisch entspricht dies einer Schnittmenge zwischen dem erzeugten großen Kreis und der kleinen Punkte von Personen. Quellcode 9 zeigt hierzu die entsprechende Python Funktion innerhalb der Klasse.

```
1     def perimeter_search(self, lat:float=51.080182693882925, long:float
    =12.112089520387727, dist:int=2000):
2         #Check If Data is Avail:
3         if(type(long)== float and type(lat) == float and type(dist) ==
    int and self.DataIsSet==True):
4             #EG: 51.080182693882925, 12.112089520387727
5             # lat is x (51.080182693882925)
6             # long is y (12.112089520387727) in epsg=4326!
7 
8             #Create Dataframe of Point:
9             #Reset Old Data: (Dynamik list)
10            self.numberDistricts.clear()
11            self.percentageDistricts.clear()
12 
13            try:
14                df = pd.DataFrame({'x': [lat], 'y': [long]})
15                self.dataframe_buffer_geo=gpd.GeoDataFrame(df, geometry
    =gpd.points_from_xy(df['y'], df['x']),  crs="EPSG:4326")
16                # transform in meters
17                self.dataframe_buffer_geo=self.dataframe_buffer_geo.
    to_crs("EPSG:25832")
```

```
18                  # Compute Buffer around distance:
19                  self.dataframe_buffer_geo=self.dataframe_buffer_geo.
    buffer(dist) # compute buffer
20                  self.dataframe_buffer_geo=self.dataframe_buffer_geo.
    to_crs("EPSG:4326") # transform to long lat
21                  # setup buffer df:
22                  self.dataframe_buffer_geo=gpd.GeoDataFrame(geometry=gpd
    .GeoSeries(self.dataframe_buffer_geo)) # create new geodataframe
23                  #now filter data
24              except:
25                  print("GEODATAPROCESSING::perimeter_search:  Failed to
    Buffer Data...")
26                  self.DataPerimeterIsSet=False
27              try:
28                  self.df_Gender_pop_permiter=gpd.sjoin(self.
    df_Gender_pop,self.dataframe_buffer_geo, how="inner")
29                  print("GEODATAPROCESSING::perimeter_search: Perimeter
    successful...")
30                  self.DataPerimeterIsSet=True
31                  return True
32              except:
33                  print("GEODATAPROCESSING::perimeter_search: Failed to
    Compute perimeter!")
34                  self.DataPerimeterIsSet=False
35                  return False
36          else:
37              print("GEODATAPROCESSING::perimeter_search: DATA is not
    setup!")
38              self.DataPerimeterIsSet=False
39              return False
```

Quellcode 9: Implemtierte Umkreissuche mithilfe von Geopandas.

Wie man sieht, wird als erstes geprüft, ob die Argumente vollständig sind und den richtigen Datentypen entsprechen. Dies ist wichtig, da sonst bei der Berechnung Fehler auftreten könnten. Im nächsten Schritt werden alte analysierte Daten, welche in einer dynamischen Liste abgelegt werden, geleert. Dann wird zuerst versucht ein Pandas Datenrahmen mit den Argumenten zu erzeugen. Danach wird ein Geopandas Datenrahmen aus dem Pandas Rahmen erzeugt und ein Punkt als Geometrie angelegt. Diese müssen noch in ein anderes Koordinatensystem transformiert werden, da sie in „EPSG4326“ im Frontend dargestellt sind. Es verwendet als Einheit den Winkel und nicht die Länge. Aus

diesem Grund wird in „EPSG25832“ transformiert, innerhalb dieses Koordinatensystems wird auch die Einheit Meter verwendet. In Zeile 19 wird nun der angelegte Punkt um die geforderte Distanz annähernd vergrößert und wieder in die ursprüngliche Darstellung transformiert.
Im nächsten Schritt wird in Zeile 28 die Schnittmenge berechnet und eine neue Membervariable „df_Gender_pop_perimeter“ gespeichert. Das Ergebnis enthält alle Punkte, die sich innerhalb des Distanzkreises befinden. Wenn alle Schritte erfolgreich waren, wird ein `return True` ausgegeben, wenn nicht wird ein `return False` zurückgegeben. So wird sichergestellt, das alle notwendigen Schritte bis zur Berechnung erfolgreich durchgeführt wurden. Alle Punkte werden in einer anderen Funktion analysiert und als JSON dem View Django zurückgegeben. Um diese Funktion nun vom Frontend aufzurufen, wurde eine Django View erzeugt, welche unter der URL `http://127.0.0.1:8000/apis/perimeter_search?long=12&lat=51&dist=2000` per HTTP GET aufrufbar ist. Die Variablen long, lat (Koordinaten des gewünschten Standorts) und dist (Umkreis in Metern, in welchem gesucht werden soll) müssen hier mit übergeben werden, da sonst keine Umkreissuche berechnet wird, sondern ein Fehler ausgegeben wird. In Quellcode 10 kann die realisierte Django View gefunden werden. Wie man sieht, wird zunächst die Klasse GeoDataProcessing als Objekt namens DataProcessing gespeichert (Zeile 6). Das Objekt wird zur Laufzeit einmalig angelegt. Als nächstes ist die mit der URL verknüpften Python Funktion zu finden. In dieser (ab Zeile 11) wird als erstes geprüft, ob alle notwendigen Daten vom Client übergeben worden sind. Sollte dies nicht der Fall sein, bekommt dieser eine Fehlermeldung. Danach werden alle Variablen aus der Anfrage gespeichert und die Berechnung erfolgt nun. Ist diese erfolgreich, so wird dies in der booleschen Variable IsComputed aus `True` gesetzt. Dementsprechend wird als nächstes geprüft, ob die Berechnung erfolgreich war oder nicht. Zum Schluss werden alle Ergebnisse der Suche und des erzeugten Kreis als Geometrie in GeoJson formatiert und den Client als Antwort gesendet (Zeile 47).

```
1 from perimeter_search.GeoDataProcessing.GeoDataProcessing import
     GeoDataProcessing # Class for Dataprocessing (perimeter_search)
2 from django.http import JsonResponse
3 import numpy as np
4
5 # Global Var for Data Processing:
6 DataProcessing = GeoDataProcessing()
7
8 # View for Perimeter_search
9 # long, lat : Coorindates
```

```
# dist: Distance in meter
def PerimeterSearch(request):
    IsComputed = False

    # Check if all Vars are here
    if (
        request.method == "GET"
        and "lat" in request.GET
        and "long" in request.GET
        and "dist" in request.GET
    ):
        # Save variable (long, lat, dist)
        lat = float(request.GET["lat"])
        long = float(request.GET["long"])
        dist = int(request.GET["dist"])
        IsComputed = DataProcessing.perimeter_search(lat, long, dist) # Compute Perimeter Search
        if IsComputed: # format output to geojson
            JsonToSend = DataProcessing.Get_PerimeterData()
            coords = np.asarray(
                DataProcessing.geodesic_point_buffer(lat, long, dist) # Geometrie for Buffer Circle as list
            ).tolist()

            geoJSON = {
                "type": "FeatureCollection",
                "features": [
                    {
                        "type": "Feature",
                        "properties": JsonToSend,
                        "geometry": {"type": "Polygon", "coordinates": [coords]},
                    }
                ],
            }

            jsonObject = {"analytics": JsonToSend, "geo": geoJSON} # final format

            print("PERIMETER_SEARCH:    calculation successful") # print output
            print(JsonToSend)
```

```
47              return JsonResponse(jsonObject) # send geojson to client
48          else:
49              print("PERIMETER_SEARCH:    calculation failed") # print
    output
50              JsonResponse({"RESPONSE": "Perimeter Search Failed...
    Error in calculation!"})
51      else:
52          print("PERIMETER_SEARCH:    parameters missing") # print output
53          return JsonResponse({"RESPONSE": "Perimeter Search Failed...
      long lat oder dist are missing!"})
```

Quellcode 10: Implemtierte Django View für die Erreichbarkeit der Umkreissuche.

Die bisher genutzte synthetische Population enthält Informationen wie Alter und Geschlecht. Die relativen und absoluten Anteile werden in einer Methode der Klasse berechnet und Dictionarie formatiert ausgegeben. In Quellcode 11 kann die Methode gefunden werden, welche die notwendigen einzelnen Analysemethoden (Zeile 2 bis 5) aufruft.

```
1   def Get_PerimeterData(self):
2       self.ToPandasDataFrame()
3       self.CountGender()
4       self.CountAgeClass()
5       self.CountDistrict()
6
7       if(self.Count==0):
8           data={"Respone of":str("Perimeter Search API"),
9           "No Population found!":None}
10          return data
11
12      data={"Respone of":str("Perimeter Search API"),
13          "Total Number": int(self.Count),
14          "Number Male": int(self.numberMale),
15          "Number Female": int(self.numberMale),
16          "Relative Male":float(self.percentageMale),
17          "Relative Female:": float(self.percentageFemale),
18          "Number AgeClass 0-25":int(self.numberAgeClass_0_25),
19          "Number AgeClass 25-45":int(self.numberAgeClass_25_45),
20          "Number AgeClass 65-100":int(self.numberAgeClass_65_100),
21          "Relative AgeClass 0-25":float(self.percentageAgeClass_0_25
    ),
```

```
22              "Relative AgeClass 25-45": float(self.
    percentageAgeClass_25_45),
23              "Relative AgeClass 65-100":float(self.
    percentageAgeClass_65_100)}
24          data.update(self.numberDistricts)
25          data.update(self.percentageDistricts)
26          return data
```

Quellcode 11: Hilffunktion zur Formatierung der Analyse in JSON.

Insgesamt wurden vier Untermethoden entwickelt, um eine kleine Statistik der künstlichen Bevölkerung durchzuführen. Zuerst werden die bisher in Geopandas vorhandenen Daten zum Pandas Datenrahmen umgewandelt, da sich in Pandas Analysen besser berechnen lassen. Die Methode CountGender wertet die Geschlechter der Bevölkerung aus, CountAgeClass wertet die Altersklassen aus und CountDistrict analysiert, in welchen Orten die Bevölkerung verteilt ist. Die Ergebnisse werden absolut und relativ in % ausgegeben. Zu Testzwecken wurde ein Ort übergeben, wo später mal ein Ärztehaus ($long$ = 12.112089520387727 und lat = 51.080182693882925) gebaut werden soll. Es wurde in einem Umkreis von 2000 m ($dist$ = 2000) nach Personen gesucht. Dabei entsteht eine Ausgabe wie in Quellcode 12 dargestellt. Relative Anteile werden durch `"Relative"` gekennzeichnet. Beispielsweise wurden in der Suche insgesamt 3296 Personen gefunden, davon sind 47.91 % weiblich und 52.09 % männlich. Außerdem Leben 928 Personen in Zeitz, was 28.16 % entspricht.

```
{ "analytics":
  {
    "Respone of": "Perimeter Search API",
    "Total Number": 3296,
    "Number Male": 1579,
    "Number Female": 1579,
    "Relative Male": 47.91,
    "Relative Female:": 52.09,
    "Number AgeClass 0-25": 604,
    "Number AgeClass 25-45": 561,
    "Number AgeClass 65-100": 2131,
    "Relative AgeClass 0-25": 18.33,
    "Relative AgeClass 25-45": 17.02,
    "Relative AgeClass 65-100": 64.65,
    "Luckenau": 172,
    "Nonnewitz": 512,
```

```
17      "Theissen": 1684,
18      "Zeitz": 928,
19      "Relativ Luckenau": 5.22,
20      "Relativ Nonnewitz": 15.53,
21      "Relativ Theissen": 51.09,
22      "Relativ Zeitz": 28.16
23    }
24  }
```

Quellcode 12: Beispiel Auszug aus der Ausgabe von der Umkreissuch API.

4.4 Datenbankmodellierung

Dieses Unterkapitel beschäftigt sich mit der Implementation der Schnittstellen und Modellierung einer relationalen Datenbank. Relationen in der Datenbank werden in Django mittels ORM implementiert (vgl. Kapitel 3.2.3). Die Verbindung zur Datenbank übernimmt das Framework. Es müssen nur die Verbindungsparameter in Einstellungen hinterlegt werden (Vergleich Quellcode 8). Im Vorfeld wurde sich dazu entschieden eine PostgresSQL Datenbank zu nutzen, da diese eine Erweiterung speziell für geografische Daten anbietet. In jeder erzeugten Django App gibt es die Datei namens „models.py“. Innerhalb dieser können Pythonklassen erzeugt werden, welche als Tabelle in der Datenbank sind. Es wurden mehrere Datenbankmodelle realisiert. Das erste Modell beinhaltet Informationen über selbst angelegte und echte Ärzte („doctor_service“). In der Django App „BasicGeoDB“ befinden sich Modelle, wo geografische Informationen gespeichert werden können.

4.4.1 Ärzte Datenbank

Quellcode 13 illustriert die hier realisierte Pythonklasse für eine Relation, welche künstlich angelegte und real existierende Ärzte enthält. Dabei ist zu beachten, dass die Klasse von `models.Model` erben muss, da dies die Basisklasse ist und alle notwendigen Voreinstellungen und Funktionen liefert, welche für Datenbankoperationen notwendig sind wie das Abfragen (`Doctors.objects.get(...)`). Ein Attribut entspricht dabei einer Membervariable, die Typen der Attribute können von der Klasse „models.py“ festgelegt werden.

Die hier gezeigte Klasse speichert Informationen eines Arztes wie Name als Zeichenkette (maximal 100 Zeichen lang), Adresse und Postleitzahl (zip). Das Attribut externalID ist eine ID von einer externen Datenbank von der AOK. Die AOK bietet eine bundesweite Arztsuche an (Erreichbar unter AOK Ärztesuche). Anhand der URL sieht man, dass es sich hierbei um eine API handelt, welche öffentlich zugänglich ist. Deshalb wurde hierzu ein R-Skript entwickelt, welches alle Ärzte Deutschlands von der AOK Arztsuche in eine JSON Datei speichert. Diese Datei wird dann genutzt, um alle Ärzte in der eigenen Datenbank zu speichern. Wie man sieht, darf die externe ID auch nur einmal in der Tabelle vorkommen (`unique=True`), aber auch leer sein (`blank=True` und `null=True`), da selbst angelegte Ärzte diese ID nicht besitzen. In dem Fall steht `null` in dem Attribut. Die Variable „Services“ listet alle Dienstleistungen auf (z. B. Zahnmedizin, Allgemeinmedizin...), welche der Arzt anbietet, diese werden durch ein „|“ getrennt.

```
from django.db import models
from djgeojson.fields import PointField
from datetime import datetime

class Doctors(models.Model):
    Name=models.CharField(max_length=100, default="")
    Address=models.CharField(max_length=100)
    zip = models.IntegerField()
    location=models.CharField(max_length=100)
    externalID=models.CharField(unique=True, null=True, blank=True,
   max_length=100)
    Services=models.CharField(max_length=1000, default="")
    manually_created=models.BooleanField(default=True)

    created=models.DateTimeField(default=datetime.now())
    geom = PointField() #https://django-geojson.readthedocs.io/en/
   latest/models.html
```

Quellcode 13: Implementierte Relation für die Ärzte Datenbank.

Das Attribut „manually_created“ hat die Aufgabe zu unterscheiden, ob es sich um einen künstlich angelegten oder einen real existierenden Arzt handelt. Dabei gibt es neben den SQL Datentypen auch spezielle Felder, wie das hier verwendete „PoinField“. In diesem Attribut („geom“) wird der geografische Standort des Arztes gespeichert in Form eines Punktes.
Im Quellcode 14 kann die implementierte API gefunden werden, welche für das Hoch-

laden der JSON Datei der Ärzte zuständig ist. Das JSON wird im HTTP POST Body übertragen und zuerst in UTF-8 decodiert. Als nächstes wird aus der langen Zeichenkette ein JSON object formatiert (`json.loads(Body)`), dies entspricht in Python dem Datentypen Dictionary. Auf die einzelnen Attribute kann nun mittels der Schlüssel zugegriffen werden. Da es sich hierbei um alle Ärzte deutschlandweit handelt, muss zeilenweise durch das Objekt iteriert werden. In den darauf folgenden Zeilen werden alle notwendigen Attribute in Variablen aus dem JSON Objekt gespeichert. In Zeile 13 werden nun alle zuvor gesammelten Attribute dem Konstruktor der ORM Klasse übergeben und ein neues Objekt wird erzeugt. In der nächsten Zeile wird das Objekt in der Datenbank gespeichert. Nachdem alle Zeilen des JSON hochgeladen sind, wird ein `"Scracped Doctors Uploaded" :"True"` ausgegeben. Die Funktion `db_obj.save()` stammt dabei aus Djangos Basisklasse `from django.db import models`.

```
1  def AddJsonDoctors(request):
2      Body=request.body.decode('utf-8')
3      json_array = json.loads(Body)
4      services={}
5      for item in json_array:
6          exID=item['id']
7          name=item['locationName']
8          street_number=str(item['address']['street'])
9          plz=item['address']['zip']
10         stadt=item['address']['city']
11         leistungen=item['subjectArea']
12         geomerty = {'type': 'Point', 'coordinates': [item['coordinates'
       ]['lat'], item['coordinates']['lon']]}
13         db_obj=Doctors.objects.create(Name=name, Address=street_number,
        zip=plz, location=stadt, Services=leistungen,
14         geom=geomerty,externalID=exID ,  manually_created=False)
15         db_obj.save() # upload to db
16     return JsonResponse({"Scrapped Doctors Uploaded":"True"})
```

Quellcode 14: Implementierte API für das Hochladen von Ärzten als JSON.

4.4.2 Basisdatenbank für geografische Daten

Um die vorhanden geografische Informationen systematisch zu speichern und Redundanzen zu vermeiden, wurde „Object_Key“ als Primärschlüssel eingeführt. Dieser identifiziert eindeutig die Shapefiles und setzt sich aus behördlichen Kennzahlen (vgl. Abbildung 12) und von Behörden vergebene IDs zusammen. Ein „_“ dient dabei als Separator.

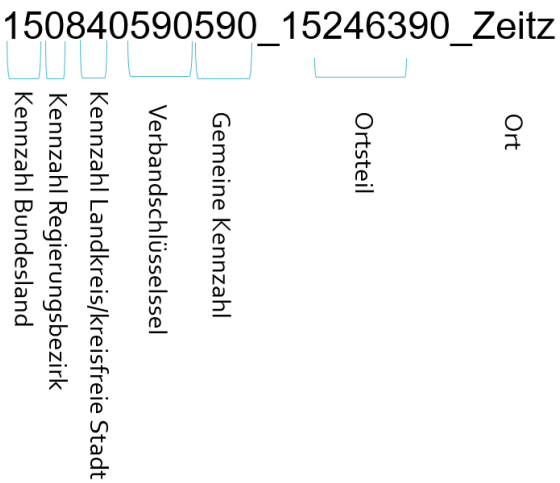

Abbildung 12: Architektur des implementierten Django-Backends.

Quellcode 15 zeigt hierbei die realisierten Relationalen, um die geografische Informationen in der Datenbank zu speichern. Es wurden drei neue Tabellen in der Datenbank erzeugt. Die Klasse „meta_data_admin_shapefiles“ beschreibt die mit den geografischen Daten verbundenen Metainformationen wie Ortsname (ObjectName), den zuvor erläuterten Primärschlüssel und zusätzliche Informationen, wann der jeweilige Eintrag veröffentlicht wurde (date_published) und Datum der letzten Änderung (date_modified). Die letzten beide Attribute wurden in allen Relationen eingebaut, um zu sehen, wann welche Änderungen stattfanden. Die Methode `def __str__` ist Dabei eine Python Standardmethode, diese wird genutzt um Datenbankinformationen auf dem Django Admin Ansicht darzustellen. Diese wird in jeder Klasse überschrieben, damit der Nutzer weiß, um welches Objekt es sich hierbei handelt. Im anderen Fall würde Django nur den Na-

men der Klasse mit der ID des Objekts versehen, womit in der Regel die Endnutzer weniger anfangen können als eine übersichtlichere Darstellung. Das Attribut „adminLevel“ beschreibt, um welche Ebene der Ansicht es sich hierbei handelt (von Komunaler Ansicht bis hinzu einzelnen Orten, 3 bis 6).

```
from django.db import models
from datetime import date


class meta_data_admin_shapefiles(models.Model):
    objectName=models.CharField(max_length=100)
    object_key=models.TextField(unique=True)
    date_modified = models.DateTimeField(auto_now=True)
    date_published = models.DateTimeField(auto_now_add=True)
    adminLevel=models.IntegerField()
    def __str__(self):
        return self.objectName+" Last updated:"+str(self.date_modified)

class Admin_Shapefiles_Level_3_6(models.Model):
    metainformation=models.ForeignKey(meta_data_admin_shapefiles,
   on_delete=models.CASCADE, default="")
    date_modified = models.DateTimeField(auto_now=True)
    date_published = models.DateTimeField(auto_now_add=True)
    geometrie=models.JSONField()

    def __str__(self):
        return self.metainformation.objectName+" Internal ID:"+ str(
   self.id)+" Last updated:"+str(self.date_modified)

class Population_Infos_Admin_Level_6(models.Model):
    metainformationen=models.ForeignKey(meta_data_admin_shapefiles,
   on_delete=models.CASCADE, default="")
    date_modified = models.DateTimeField(auto_now=True)
    date_published = models.DateTimeField(auto_now_add=True)
    Object_Name=models.CharField(max_length=100)
    year=models.IntegerField()
    age_groups=models.JSONField()
    TOTAL_SUM=models.IntegerField()
    W_SUM=models.IntegerField()
    M_SUM=models.IntegerField()

    def __str__(self):
```

```
35          return self.Object_Name+" Internal ID:"+str(self.id)+ " Last
    updated:"+str(self.date_modified)
```

Quellcode 15: Realisierte Datenbankmodelle für geografische Daten.

Die Klasse „Admin_Shapefiles_level_3_6“ nutzt die SQL Beziehung „Many to One“, um jeden Eintrag mit den verbundenen Metainformationen zu verknüpfen. Dies wird in Django realisiert, indem `models.ForeignKey` als Feld angegeben wird. Diese Information (Object_key usw.) wird mit der ID der Relation von Metainformation verknüpft, weshalb auch die Klasse als Argument übergeben wird. In dem Feld „geometrie“ werden die geometrische inklusive Standort Informationen (engl. kurz Shape) als JSON gespeichert. Diese liegen entweder als ein Punkt (Orte) oder Polygon (Gebiete) vor. Die Klasse „Population_Infos_Level_6“ beschreibt eine Relation in der Datenbank, welche Bevölkerungsdaten enthalten. Geschlecht sowie Altersgruppe werden im JSON Format (Feld namens „age_groups“) gespeichert. Zum Beispiel (vgl. Quellcode 16) werden alle männliche Personen in einem Alter von 0 bis 3 Jahren in `"M3"` gespeichert. In dem Beispiel wären es 17 männliche Personen. Neben männlich und weiblich (16 Personen) wird auch die Summe der Altersgruppe gespeichert (`"G3"`..., 33 Personen). Andere wichtige Informationen wie aus welchem Jahr stammen die Daten, Gesamtsumme der männlichen oder weiblichen Personen und dessen Einzelsumme werden in separaten Feldern als Integerformat gespeichert.

```
[
  {
    ...,
    "M3": 17,
    "W3": 16,
    "G3": 33,
    "M6": 23,
    "W6": 19,
    "G6": 42,
    "M9": 23,
    "W9": 23,
    "G9": 46,
    "M12": 22,
    "W12": 21,
    "G12": 43,
    ...,
  },
  ...
```

```
]
```

Quellcode 16: Beispiel Auszug aus den gespeicherten Bevölkerungsdaten in JSON.

Alle bisher gesammelten Daten wurden in der Datenbank abgelegt. Um auf diese im Frontend zuzugreifen, wurde eine API entwickelt, welche mittels der „object_key“ (Primärschlüssel) funktioniert. Die Funktion der API (get_shapfileById) kann in Quellcode 17 gefunden werden. Neben der Filterung per „object_key“ (Zeichenkette), kann auch über die eigens vergebene interne Datenbank ID gefiltert werden. Hierzu kann über die URL die Variable `"object_key"` oder `"internal_id"` übergeben werden. In Abhängigkeit dessen werden unterschiedliche Hilfsfunktionen aufgerufen, die genau mit dem übergebenen Schlüssel die geforderten Funktionen filtern. Die Funktion „QuerryByObjectKey“ (Zeile 27) versucht dabei eine Filterung mit dem Primärschlüssel durchzuführen. Gelingt dies, wird die Hilfsfunktion (FormatOneJSON) zum Formatierten in JSON Format genutzt, um in ein zuvor festgelegtes Format zur formatieren. Dabei steht immer als erstes der Name der API, danach folgen interne ID und Primärschlüssel (Zeile 45).

```
def get_shapefileById(request):
    if(request.method=="GET"):
        if("object_key" in request.GET):
            key=request.GET['object_key']
            return JsonResponse(QueryByObjectKey(key), safe=False)
        elif("internal_id" in request.GET):
            ID=request.GET['internal_id']
            return JsonResponse(QuerryByInternalID(ID))

    else:
        return JsonResponse({"UpdatePopulation_Infos RESPONSE": "WRONG
   HTTP METHOD OR MISSING ID?"})

def QuerryByInternalID(internalID):
    try:
        shapeOBJ=Admin_Shapefiles_Level_3_6.objects.get(id=internalID)
        meta=shapeOBJ.metainformation
        return FormatOneJSON(shapeOBJ=shapeOBJ,meta=meta)
    except Exception as e:
        error={
            "api_name":"get_shapefileById",
            "querry_error":"not_found",
```

```
            "sended_key": internalID
            }
        return error


def QueryByObjectKey(key):
    try:
        meta=meta_data_admin_shapefiles.objects.get(object_key=key)
        shapeOBJ=Admin_Shapefiles_Level_3_6.objects.get(metainformation
   =meta.id)
        return FormatOneJSON(shapeOBJ=shapeOBJ,meta=meta)
    except:
        error={
            "api_name":"get_shapefileById",
            "querry_error":"not_found",
            "sended_key": key
            }
        return error


def FormatOneJSON(shapeOBJ,meta):
    json={
        "api_name":"get_shapefileById",
        "identifier":{
        "internal_id":shapeOBJ.id,
        "object_key":meta.object_key
        },
        "object_name":meta.objectName,
        "analytics":{},
        "geo":shapeOBJ.geometrie,
        "style":0
        }
    return json
```

Quellcode 17: Realisierte Schnittstelle zur geografischen Datenbank und deren Hilfsfunktionen.

In dem Schlüssel `"analytics"` werden im späteren Verlauf Ergebnisse von Analysen der Bevölkerungsdaten stehen. In `"geo"` steht die geografische Information in Form von einer Geometrie (entweder Polygon bei ganzen Gebieten oder Punkte bei Orten), welche in der Datenbank schon als JSON hinterlegt sind. Der Datentyp Dictionary in Python entspricht dabei eines JSON Formats. Diese lassen sich zudem in Python beliebig verschachteln. Eine Beispielhafte Ausgabe für eine den Ort Könderitz kann in Quellcode 18 gefunden werden.

```
{
  "api_name": "get_shapefileById",
  "identifier": {
    "internal_id": 227,
    "object_key": "150840130130_15246170_Koenderitz"
  },
  "object_name": "Koenderitz",
  "analytics": {},
  "geo": {
    "type": "Point",
    "coordinates": [
    12.2121674,
    51.0867611
    ]
  },
  "style": 0
}
```

Quellcode 18: Beispiel Ausgabe vom Ort Koendernitz der realisierten Schnittstelle zur Geodatenbank.

5 Zusammenfassung und Ausblick

Alles in allem wurde erfolgreich eine erste Codebasis geschaffen, um das Backend der Anwendung zu realisieren. Es wurde analysiert, welches Framework diese Aufgabe am Besten erfüllen kann. Zudem wurde eine geeignete Datenbank ausgesucht, welche Geodaten verarbeiten und speichern kann. Dabei wurde das Backend modular in drei Teile aufgeteilt. Der erste Teil enthält ein Aplication Programming Interface (kurz API), welche eine Umkreissuche von einer künstlichen Bevölkerung erlaubt. Das Ergebnis wird analysiert und in JSON Format dem Frontend übergeben. Im zweiten Teil wurde eine Ärzte Datenbank implementiert, welche später für Erreichbarkeitsanalysen genutzt werden kann. Im dritten Teil wurde eine Basis Datenbank erzeugt für geografische Informationen wie Orte. Diese können als Geometrie im Frontend dargestellt werden. Hierzu wurde eine API programmiert, welche in Abhängigkeit einer ID die gewünschte Information strukturiert und im JSON Format ausgibt. Im späteren Verlauf wird dieser Teil der Anwendung genutzt, um die bisher in lokalen JSON Daten gespeicherten und simulierte künstliche Bevölkerung in eine Datenbank zu speichern. Dabei muss der bisherige Quellcode auf Effizienz untersucht werden und gegebenenfalls optimiert werden. Als nächstes muss die Simulation in Java im Backend eingebunden werden und angepasst werden, um dynamische Änderungen in der Infrastruktur in Echtzeit anzeigen zu können. Es kann zudem überlegt werden, ob der Einsatz von Maschine Learning oder klassische algorithmische Lösungen wie der Dijkstra Algorithmus zur optimalen Wegfindung sinnvoll wäre, weil es sich im späteren Verlauf um größere Datenmengen handeln wird, können Korrelationen genutzt werden, welche bisher nicht erkannt wurden und Python als Programmiersprache gewählt wurde.

Literaturverzeichnis

[1] M. Meinköhn, "Nahverkehrsstudie zeigt: 55 Millionen Deutsche ohne ausreichenden ÖPVN-Zugang," *STERN.de*, 27.10.2021. [Online]. Available: https://www.stern.de/wirtschaft/news/nahverkehrsstudie-zeigt--55-millionen-deutsche-ohne-ausreichenden-oepvn-zugang-30870014.html

[2] I. Editor, "Baustelle Gesundheitssystem: Fachkräftemangel mit Abstand größtes Problem," *www.ipsos.com*, 15.12.2021. [Online]. Available: https://www.ipsos.com/de-de/baustelle-gesundheitssystem-fachkraftemangel-mit-abstand-grosstes-problem

[3] D. Tischlinger, "Frontend und Backend: Was ist der Unterschied?" 25.04.2022. [Online]. Available: https://blog.hubspot.de/website/frontend-backend

[4] "Microservices," 2022. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Microservices&oldid=227753200

[5] Statista, "Beliebteste Programmiersprachen weltweit 2023 | Statista," 24.02.2023. [Online]. Available: https://de.statista.com/statistik/daten/studie/678732/umfrage/beliebteste-programmiersprachen-weltweit-laut-pypl-index/

[6] S. Ivanov, "Die zehn besten Backend-Sprachen," *Low-code backend to build modern apps*, 22.07.2021. [Online]. Available: https://blog.back4app.com/de/backend-sprachen/

[7] IONOS Digital Guide, "Flask vs. Django: Die beiden Python-Frameworks im Vergleich," 22.02.2023. [Online]. Available: https://www.ionos.de/digitalguide/websites/web-entwicklung/flask-vs-django/

[8] "Was sind Geodaten (raumbezogene Daten)? IBM," 03.03.2023. [Online]. Available: https://www.ibm.com/de-de/topics/geospatial-data

[9] Bundesverband Geothermie, "Koordinaten, geographische," 2023. [Online]. Available: https://www.geothermie.de/bibliothek/lexikon-der-geothermie/k/koordinaten-geographische.html

[10] "Geographische Koordinaten," 2023. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Geographische_Koordinaten&oldid=231385123

[11] "8. Koordinatenbezugssysteme — QGIS Documentation Dokumentation," 01.03.2023. [Online]. Available: https://docs.qgis.org/3.22/de/docs/gentle_gis_introduction/coordinate_reference_systems.html

[12] "European Petroleum Survey Group Geodesy," 2022. [Online]. Available: https://de.wikipedia.org/w/index.php?title=European_Petroleum_Survey_Group_Geodesy&oldid=226603921

[13] GIS Analyse, "Was ist die Shapefile? - GIS Analyse," 13.03.2022. [Online]. Available: https://gisanalyse.de/was-ist-die-shapefile

[14] "JavaScript Object Notation," 2023. [Online]. Available: https://de.wikipedia.org/w/index.php?title=JavaScript_Object_Notation&oldid=229374937

[15] codecentric, "GeoJSON Tutorial," 14.03.2023. [Online]. Available: https://blog.codecentric.de/geojson-tutorial

[16] "Django (Framework)," 2022. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Django_(Framework)&oldid=227497137

[17] "Was ist Django?" 02.03.2023. [Online]. Available: https://www.djangoblog.ch/de/django/was-ist-django/

[18] Django Project, "Django," 02.03.2023. [Online]. Available: https://docs.djangoproject.com/en/4.1/intro/tutorial01/

[19] "Django-Superkraft: Eine CRUD-Web-App in 60 Minuten," 02.03.2023. [Online]. Available: https://www.djangoblog.ch/de/django/django-superkraft-eine-crud-web-app-in-60-minuten-teil-1/

[20] R. Naushad, "Difference between WSGI and ASGI ? - Analytics Vidhya - Medium," *Analytics Vidhya*, 24.07.2020. [Online]. Available: https://medium.com/analytics-vidhya/difference-between-wsgi-and-asgi-807158ed1d4c

[21] B. Nige, "Beginner Lesson 2: How Django Works," *Djangobook*, 05.01.2022. [Online]. Available: https://masteringdjango.com/django-tutorials/beginner-lesson-2-how-django-works/

[22] "Introduction to Django," 02.03.2023. [Online]. Available: https://www.w3schools.com/django/django_intro.php

[23] Django Project, "Django," 02.03.2023. [Online]. Available: https://docs.djangoproject.com/en/4.1/topics/templates/

[24] ——, "Django," 14.03.2023. [Online]. Available: https://docs.djangoproject.com/en/4.1/topics/signing/

Listings

Abbildungsverzeichnis

Danksagung

Dank an alle. An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung der Masterprojektarbeit unterstützt und motiviert haben. Zuerst gebührt mein Dank meiner Frau. Sie hat mich nicht nur seelischen Beistand geleistet, motiviert und unterstützt, sondern auch gerne in meiner Arbeit vorkommenden Rechtschreibfehler korrigiert.
Ich bedanke mich zudem bei Dr. Jan Schlüter, welcher mir vor allem die Möglichkeit gab, diese Arbeit zu verfassen. Außerdem danke ich Herrn Prof. Dr. Rer. Nat. Roman Grothausmann für die Vermittlung zur Forschungsgruppe und der Sichtung meiner Arbeit. Ein besonderer Dank geht an Alex Röhrs und Andy Bossert für die tatkräftige Unterstützung bei der Realisierung der Quellcodes sowie Ideenfindung. Ich bedanke mich zudem bei Nico für die Unterstützung der Realisierung.

Christian Grünewald

Göttingen, den 26. April 2023