Titel:

Projektdokumentation – Corona API

Projektarbeit
Vorgelegt im Kontext des Moduls „Datenerfassung und Datenerhaltung“

am Fachbereich Technische Informatik und Elektrotechnik

Veranstalter: Prof. Dr. Philipp Bruland
Betreuer: Prof. Dr. Philipp Bruland

Vorgelegt von: Luca Lauryn Dube
Bahnhofstraße 24
32825 Blomberg
+49152 29582678
luca.dube@stud.th-owl.de

Datum der Abgabe: 30.09.2024

Inhaltsverzeichnis

Abkürzungen

API – Application Programming Interfaces

DTO – Data Transfer Object

MVT – Model-View-Template

DTL – Django Template Language

GUI – Graphical User Interface

UI – User Interface

HTML – Hypertext Markup Language

CSS – Cascading Style Sheets

JSON – JavaScript Object Notation

URL – Uniform Resource Locator

Abbildungen

1 Einleitung

Die COVID-19 Pandemie hat die Menschheit seit dem Jahr 2020 auf die Probe gestellt. Insgesamt gab es auf der Welt bis heute knapp 700 Millionen registrierte Infektionen und fast 7 Millionen Menschen sind direkt oder indirekt durch die Krankheit gestorben. Auch wenn die Pandemie vom Bundesgesundheitsminister im April 2023 offiziel für beendet erklärt wurde, infizieren sich weiterhin tausende Menschen wöchentlich.

Im Rahmen dieser Projektarbeit wurde eine Anwendung zur Visualisierung verschiedener Daten, die im Zusammenhang mit COVID-19 stehen, entwickelt. Hierbei wurden die theoretisch gelernten Konzepte aus dem Modul „Datenerhaltung und Datenerfassung 1“ genutzt um ein vollständig funktionierendes Resultat zu erschaffen, welches gängige Konzepte der Programmierung nutzt.

Die folgende Dokumentation befasst sich mit den verwendeten Technologien und Methoden, die für die Programmierung der Anwendung genutzt wurden, als auch mit dem Aufbau der Datenbank und API.

2 Material und Methoden

Für Das Projekt wurde eine Webanwendung mithilfe des Frameworks Django gebaut, welche COVID-19 Daten aus einer API abfragt und grafisch in der Oberfläche aufbereitet. Zur Abfrage der Daten aus der API benötigt es zudem eine Datenbank mit verschiedensten deutschen Standorten. Um all diese Dinge in einer Anwendung zu ermöglichen, wurden folgende Materialien verwendet:

- **Django**: Das Webframework Django, welches auf der Programmiersprache Python basiert, verbindet mithilfe des MVT das Frontend und Backend auf eine einfache Art und Weise.

- **Bootstrap**: Bootstrap bietet ein grundlegendes Design für das Frontend.

- **SQLite**: Als Integrierte Datenbank in Django ist SQLite einfach und effizient zu nutzen.

- **API**: Die API (https://api.corona-zahlen.org) wurde genutzt um angefragte COVID-19 Daten in die Webanwendung laden zu können.

- **GitHub**: GitHub ermöglicht eine einfache Quellcodeverwaltung, welche außerdem die Entwicklung auf mehreren Geräten unterstützt.

3 Django

Django ist ein strukturiertes, auf Python basierendes, Open Source Webframework zur einfachen Erstellung von Webanwendungen, welches auf dem etablierten Software Pattern Model-View-Controller aufbaut.

3.1 Model-View-Template

Das Software Pattern „Model-View-Template“ ist ein von Django eingeführtes Pattern, welches auf dem „Model-View-Controller“ aufbaut. Die einzelnen Komponenten funktionieren wie folgt:

- **Model**: Die Models werden in Django genutzt, um mit der Datenbank zu interagieren. Jedes Model spiegelt eine Tabelle innerhalb der Datenbank ab.
- **View**: Die Views kümmern sich um die Verarbeitung der HTTP-Anfragen aus dem Frontend und stellen das Bindeglied zwischen Frontend und Backend dar. Außerdem geben sie die entsprechenden HTTP-Antworten an das Frontend zurück, mit deren Hilfe die angeforderte Seite aufgebaut werden kann.
- **Template**: Das Template kümmert sich um die Darstellung der Seiten im Frontend. Es wird genutzt, um HTML-Seiten mit den Daten der Views zu generieren.

3.2 Routing

Django verwendet ein URL-Routing-System, um die Anfragen der URL-Seiten direkt an die entsprechenden Views weiterzuleiten. In den Dateien „urls.py“ werden URL-Pfade definiert, welche dann mit einem Namen versehen und mit den Views verknüpft werden.

```python
# Datei: corona/urls.py:5-14
urlpatterns = [
    # Frontend
    path("", home_view, name="home"),
    path("search/", search_view, name="search"),
    path("details/<str:name>", details_view, name="details"),

    # Backend
    path("backend/sync-database", sync_database_view, name="sync_database"),
    path("backend/live-search/", live_search, name="live-search"),
]
```

Abbildung 1: URL-Definitionen in Django (eigene Darstellung)

3.3 Django Template Language

Django hat für die Nutzung der Templates (HTML-Seiten) ein spezielles Template-Rendering-System namens Django-Template-Language (DTL) entwickelt. Dieses System ermöglicht es, HTML-Seiten dynamisch generieren zu lassen. Im gezeigten Beispiel sieht man die Generierung der Suchergebnisse auf der „search“-Seite:

```
<!-- Datei: corona/templates/pages/search.html:16-25 -->
{% if querySucceeded %}
    {% for item in results %}
        {% if item.type|lower == "stadt" %}
            <a class="dropdown-item" href='{% url "details" item.id %}?type={{ item.type }}'>{{ item.name }} (PLZ: {{ item.plz }})</a>
        {% elif item.type|lower == "landkreis" %}
            <a class="dropdown-item" href='{% url "details" item.id %}?type={{ item.type }}'>{{ item.name }} ({{ item.type }})</a>
        {% else %}
            <a class="dropdown-item" href='{% url "details" item.abbreviation %}?type={{ item.type }}'>{{ item.name }} ({{ item.type }})</a>
        {% endif %}
    {% empty %}
```

Abbildung 2: Django Template Language Beispiel (eigene Darstellung)

Hier wird die Liste „results“, welche durch die View an das Template übergeben wurde, von einer For-Schleife iteriert. Entsprechend des Typs der Standorte, wird der angezeigte Text auf der „search“-Seite angepasst.

Mithilfe der DTL kann man Steuerungsanweisungen und Variablen innerhalb des HTML-Templates durch „Template-Tags“ verwenden. Die Steuerungsanweisungen werden in geschweiften Klammern mit Prozentzeichen „{% … %}“ und die Variablen in doppelten geschweiften Klammern „{{ … }}“ angegeben.

Die in diesem Projekt am häufigsten genutzten Funktionen der DTL sind die Vererbungstags. Dazu gehören hauptsächlich die beiden Tags „{% include … %}“ und „{% extends … %}“, welche eine Art Vererbung zwischen HTML-Seiten ermöglichen. Dazu wurde eine „base“ HTML-Seite erstellt, die das grundlegende Design der Website beinhaltet. Alle anderen Seiten nutzen den Tag: {% extends "pages/base.html" %} am Anfang um diese Seite zu “erweitern”. Durch die Angabe von “Content-Blöcken” ist es möglich nun an spezielle Stellen innerhalb der “base” HTML-Seite neue Inhalte einzufügen.

```
<!-- Datei: corona/templates/pages/search.html:10-12 -->
{% block navbar %}
    {% include "includes/navbar.html" %}
{% endblock %}
```

Abbildung 3: Django Vererbungstags (eigene Darstellung)

4 Frontend-Architektur

Das Frontend wurde basierend auf dem open source Framework Bootstrap aufgebaut. Es unterstützt verschiedene Features wie die „Live-Suche“ für die Suchfelder und das Abbilden der Daten in Charts.

4.1 Bootstrap

Bootstrap ist ein Open Source CSS-Framework, welches für das Frontend genutzt wurde. Es bietet ein grundlegendes Styling für verschiedenste UI-Komponenten, die innerhalb des Projektes genutzt und mithilfe eigener CSS-Klassen erweitert wurden. Die Nutzung von Bootstrap wird am folgenden Beispiel erklärt:

```
<!-- Datei: corona/templates/pages/details.html:27 -->
<div class="col-lg-3 d-flex flex-column justify-content-between side-charts">
```

Abbildung 4: Nutzung von Bootstrap (eigene Darstellung)

- Bootstrap-Klassen:
    - **col-lg-3**: Das legt die Größe des Elementes fest. Auf großen Bildschirmen nimmt dieses Element 3 von 12 Spalten auf dem Bildschirm ein.
    - **d-flex** & **flex-column**: Hier wird die Flexbox aktiviert und vertikal ausgerichtet, sodass alle child-Elemente untereinander angeordnet werden.
    - **justify-content-between**: Mit dieser Klasse werden alle child-Elemente mit gleichem Abstand zueinander angeordnet.
- Eigene Klasse:
    - **side-charts**: Diese Klasse wurde selbst erstellt, um die kleinen Diagramme auf der Seite nach den gewünschten Anforderungen, in Farbe, Größe und Anordnung, selbst gestalten zu können.

4.2 Datenvisualisierung

Die Daten werden auf der „details“-Seite der Anwendung innerhalb von vier verschiedenen Linien-diagrammen dargestellt. Zur Darstellung der Linien-diagramme wird die JavaScript Bibliothek Chart.js genutzt. Für die Umsetzung in HTML ist lediglich ein Canvas mit einer bestimmten ID nötig (Appendix A.1: Code 1). Im JavaScript wird zunächst ein neues Chart-Objekt erstellt (Appendix A.1: Code 2), welches das Canvas, sowie Konfigurationen für den Chart im JSON-Format benötigt. Diese Konfigurationen beinhalten gleichzeitig die Achsenbeschriftungen und Daten der Charts.

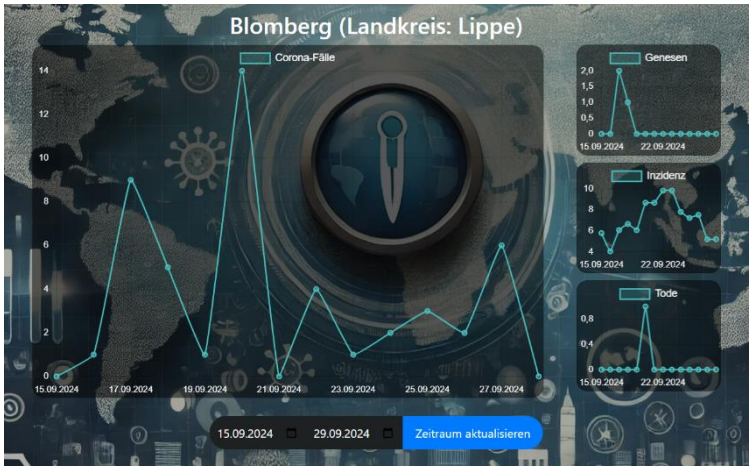

Abbildung 5: Darstellung der Daten in der GUI (eigene Darstellung)

Die „details“-Seite unterstützt außerdem das Wechseln des Hauptdiagramms. Dazu muss der Nutzer lediglich mit der Maus auf das zu vergrößernde Seitendiagramm klicken. Anschließend werden die betroffenen Canvases geleert und neue Charts mit den neuen Konfigurationen erstellt (Appendix A.1: Code 3). Zuletzt werden nur noch die IDs zum Nachvollziehen der korrekten Reihenfolge der Charts getauscht.

4.3 Suchvorschläge

Die Anwendung unterstützt für alle Suchfelder automatische Suchvorschläge, basierend auf der Eingabe des Nutzers. Dazu wird mithilfe von JavaScript die Eingabe des Nutzers ausgelesen und als Parameter an eine der Backend-URLs (folgend in 4.4 erklärt) geschickt. Das Backend liefert der Seite eine JSON-Response (Appendix A.2: Response 1), welche durch JavaScript verarbeitet werden kann (Appendix A.1: Code 4). Dabei werden in einem zuvor erstellten Container „search-results“ neue Linktexts hinzugefügt, welche dann im Frontend unterhalb der Suchleiste als

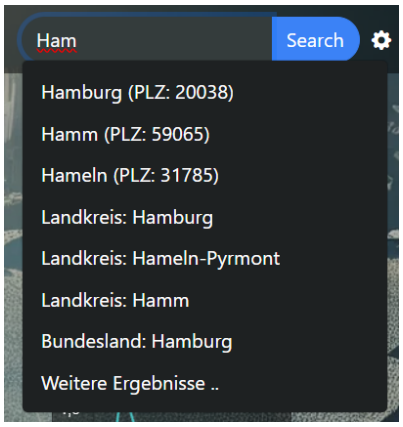

Abbildung 6: Suchvorschläge in der GUI (eigene Darstellung)

Vorschläge sichtbar sind. Je nach Standorttyp unterscheidet sich der angezeigte Text innerhalb der Vorschläge, zur Unterscheidung der einzelnen Standorte.

4.4 URLs

Die URLs des Projektes sind in Frontend und Backend aufgeteilt. Wie in 3.2 schon einmal gezeigt, werden diese innerhalb der „urls.py“ definiert und der Pfad der einzelnen URLs wird dort festgelegt.

Es gibt drei verschiedene Frontend URLs: die „main“-Seite (Startseite), die „search“-Seite (Suchergebnisse) und die „details“-Seite (Anzeige der Daten). Die entsprechenden Views der Seiten liefern mithilfe der „render“-Funktion immer eine HTTP-Response an das Frontend zurück, damit die Entsprechende Seite geladen werden kann.

```python
# Datei: corona/views.py:21-26
def home_view(request):
    context = {
        "sync_function": sync_database,
        "navbarSearch": False,
        }
    return render(request, "pages/home.html", context)
```

Abbildung 7: Nutzung der "render" Funktion von Django (eigene Darstellung)

Außerdem sind insgesamt 2 verschiedene Backend URLs definiert: „sync-database“ (zum Starten der Datenbanksynchronisation) und „live-search“ (Zum Generieren der Suchvorschläge). Diese Views liefern keine HTTP-Response, da sie keine HTML-Seite anzeigen sollen. Stattdessen schicken sie entweder eine JSON-Response oder einen Redirect auf eine andere URL.

5 Backend-Architektur

5.1 Datenbankaufbau

Der Aufbau der Datenbank wurde mithilfe der Website drawsql.app (siehe Referenzen) visualisiert. Es spiegelt den Aufbau der Models in Django wider.

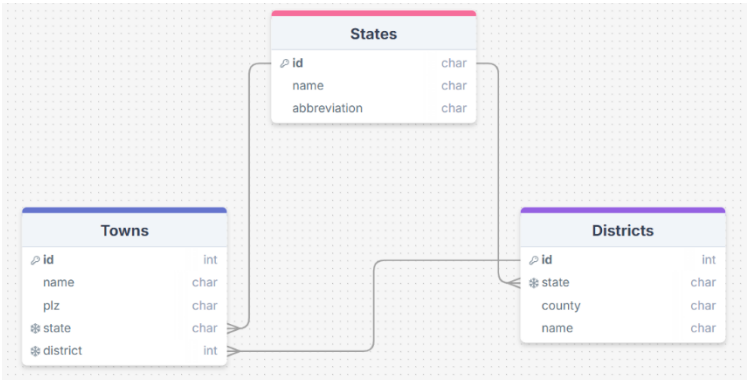

Abbildung 8: Datenbankaufbau (eigene Darstellung)

Innerhalb des Projektes gibt es drei verschiedene Tabellen (Models) für die verschiedenen Arten der Standorte: Towns (für Städte), Districts (für Landkreise) und States (für Bundesländer).

```python
# Datei: corona/models.py:4-19
class States(models.Model):
        id = models.CharField(primary_key=True, max_length=2)
        name = models.CharField(max_length=200)
        abbreviation = models.CharField(max_length=2)

class Districts(models.Model):
        id = models.IntegerField(primary_key=True)
        state = models.ForeignKey(States, on_delete=models.PROTECT)
        county = models.CharField(max_length=200)
        name = models.CharField(max_length=200)

class Towns(models.Model):
        name = models.CharField(max_length=200)
        plz = models.CharField(max_length=10, primary_key=True)
        district = models.ForeignKey(Districts, on_delete=models.PROTECT)
        state = models.ForeignKey(States, on_delete=models.PROTECT)
```

Abbildung 9: Definition der Modelle in Django (eigene Darstellung)

Alle Models werden innerhalb der „models.py“ definiert. Vor der Nutzung der Datenbank müssen diese Models einmal migriert werden, was innerhalb der README.md ausführlicher erklärt wird.

Die Datenbank wird hauptsächlich genutzt, um die Suchvorschläge und API-URLs generieren zu können. Dazu fragen die Views den „db_service“ (dazu in 5.4.1 mehr) an, welcher die einzelnen Inhalte der Tabellen als DTOs (dazu in 5.2 mehr) wiedergibt. Damit die Datenbank alle nötigen Städte, Landkreise und Bundesländer beinhaltet, müssen diese vor der Nutzung des Programms einmalig synchronisiert werden. Dazu gibt es innerhalb der Navigationsleiste ein Einstellungssymbol mit der Unterkategorie „Datenbank synchronisieren“.

5.2 Data Transfer Objects

Data Transfer Objects (DTOs) werden innerhalb des Projektes genutzt um Daten zwischen verschiedenen Methoden, Dateien oder zwischen Front- und Backend zu übertragen. Die DTOs ermöglichen damit eine einheitliche Datenstruktur innerhalb des Projektes, welche die Verständlichkeit steigert und das Programmieren vereinfacht.

Definiert sind fünf verschiedene DTOs, welche zum einen für die Übertragung der Standorte und der „eigentlichen“ Datenpunkte genutzt werden.

Die Klasse „DatapointsDTO“ enthält verschiedene Parameter um für einen beliebigen Standort (hier als „endpoint“ bezeichnet) alle benötigten Informationen in das Frontend zu übertragen. Dazu wird ein erstelltes Objekt an das Frontend überliefert und mithilfe der DTL und JavaScript dort verarbeitet.

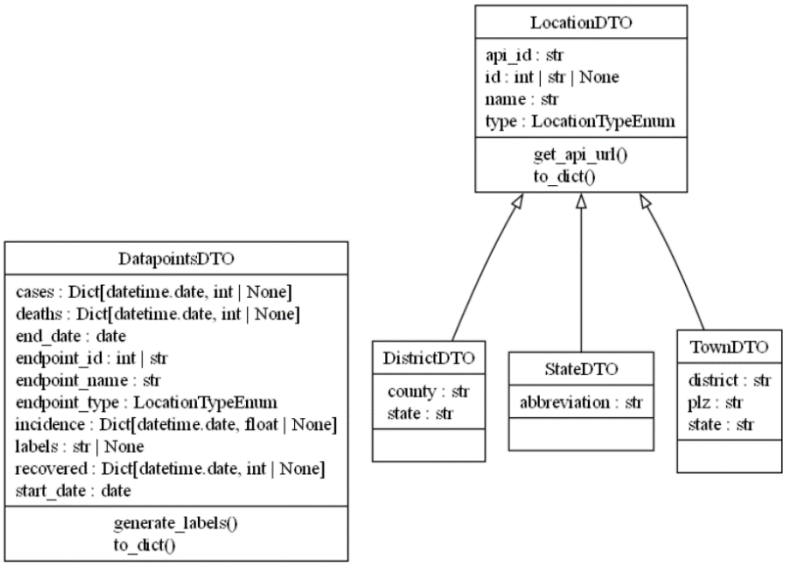

Abbildung 10: Aufbau der DTO Klassen (eigene Darstellung)

Die restlichen DTO Klassen werden verwendet um die verschiedenen Standorte in korrekter Form an die jeweiligen Methoden zu übergeben. Es gibt eine Oberklasse „LocationDTO“, welche die grundlegenden Parameter und Methoden beinhaltet, sowie die von dieser Klasse erbenden Klassen: „DistrictDTO“ (Bezirke), „StateDTO“ (Bundesländer) und „TownDTO“ (Städte), welche für die entsprechenden Standorttypen verwendet werden. Durch die Vererbung der DTO Klassen ist es möglich das Konzept der Polymorphie zu nutzen. Besonders hilfreich ist dies in dem „data_service“ (dazu in 5.4.2 mehr), wo es genutzt werden kann um die verschiedenen Standorttypen, welche auch mit verschiedenen Endpunkten verknüpft sind, mithilfe einer einzigen Funktion von der API abfragen zu können. Dazu wird innerhalb der Views eine DTO Unterklasse an die Funktion „get_location_datapoints“ übergeben, welche diese aber als „LocationDTO“ (Oberklasse) nutzt und damit weiter arbeitet.

5.3 Enumeratoren

Ein Enumerator ist eine spezielle Klasse, welche konstante Werte definiert und diese als eine Art Zustand benutzen kann. Genutzt werden in diesem Projekt zwei verschiedene Enumeratoren: „LocationTypeEnum“ (Beschreibung des Typs des Standorts) und „APIDataTypeEnum“ (Beschreibung des Datentyps der API). Die Klasse

```python
# Datei: corona/objects/enums.py:31-35
class APIDataTypeEnum(Enum):
    CASES = "cases"
    DEATHS = "deaths"
    INCIDENCE = "incidence"
    RECOVERED = "recovered"
```

Abbildung 11: Definition vom "APIDataTypeEnum" (eigene Darstellung)

„LocationTypeEnum“ beinhaltet zusätzlich noch zwei spezielle Funktionen (Appendix A.1: Code 7). Zum einen die Funktion „from_string“, welche es ermöglicht die vom Frontend übergebenen Strings in den richtigen Enumerator Wert zu übersetzten. Zum anderen gibt es die Funktion „api_str“, welche es ermöglicht auf Basis des Types des Standorts die richtige API URL zu generieren.

```python
# Datei: corona/objects/enums.py:3-6
class LocationTypeEnum(Enum):
    TOWN = "Stadt"
    DISTRICT = "Landkreis"
    STATE = "Bundesland"
```

Abbildung 12: Definition von "LocationTypeEnum" (eigene Darstellung)

5.4 Services

Zur vereinfachten Verarbeitung / Verwaltung der Daten wurden drei verschiedene Services programmiert (im Projekt unter „corona/services/…“ zu finden). Diese kümmern sich um die Kommunikation mit der Datenbank (db_service.py) und der API (data_service.py), sowie um die Verwaltung der Standorte (location_service.py). Dabei verarbeiten alle Service ihre Daten so, dass sie als DTOs weitergeleitet bzw. weiterverarbeitet werden können.

5.4.1 Datenbank

Die Kommunikation mit der Datenbank verläuft ausschließlich über den „db_service.py“, welcher alle nötigen Funktionen zum Hinzufügen, als auch zum Lesen eines Standortes in der Datenbank beinhaltet (Beispiel in: Appendix A.1: Code 5). Zudem beinhaltet diese Datei auch die Funktion „filter_all_tables“, welche ein sehr wichtigen Part in den Suchen spielt (Appendix A.1: Code 6).

5.4.2 API

Um die Kommunikation mit der API kümmert sich der „data_service.py“. Dazu stellt dieser alle nötigen Funktionen zur Abfrage und Verarbeitung der Daten bereit. Außerdem ermöglicht die Funktion „read_file_lines“ das Lesen der CSV-Datei (dazu in 6.1 mehr)

5.4.3 Standorte

Wenn Daten mit Bezug zu Standorten verarbeitet werden müssen, wird der Service „location_service.py“ benötigt. Dieser Service ermöglicht mithilfe der Funktionen „get_all_states“ o.ä. (Appendix A.1: Code 8) das Herausfiltern aller genannten Standorte innerhalb der API oder CSV. Außerdem kann mithilfe des Services innerhalb der Datenbank nach einem Standort gesucht („search_for_location“) oder ein bestimmter Standort als DTO aus der Datenbank erhalten werden („get_location“).

6 CSV-Daten

6.1 Nutzung

Aufgrund der Fehlenden API Endpunkte für Städte (dazu in 7.2 mehr) war es notwendig deutsche Städte über einen anderen Weg zu importieren. Am einfachsten geschieht das mithilfe von CSV-Dateien. Dazu wurde die Datei „corona/static/files/Liste-Staedte-in-Deutschland.csv“ zum Projekt hinzugefügt. Diese ermöglicht es extern Städte zu importieren und mit den in der Datenbank hinterlegten Bezirken und Bundesländern zu Verknüpfen.

6.2 Verarbeitung

Um die CSV-Datei richtig verarbeiten zu können, mussten einige Schreibweisen und Benennungen an die in der Datenbank hinterlegten Bezirke manuell angepasst werden. Dadurch ist es nun möglich mithilfe der „sync-function“ innerhalb der Navigationsleiste im Frontend die CSV-Datei einzulesen und alle benötigten Städte in der Datenbank zu hinterlegen. Dazu wird innerhalb der helpers.py die Funktion „get_all_towns“ aus dem, „location_service.py“ aufgerufen. Diese liest die CSV-Datei ein und wandelt diese

```python
# Datei: corona/services/location_service.py:32-42
def get_all_towns() -> List[TownDTO]:
    datapoints = []

    lines = read_file_lines()

    for line in lines:
        line = line.strip().split(";")
        town = TownDTO(name=line[1].strip(), plz=line[2].strip(), district=line[4].strip(), state=line[3].strip())
        datapoints.append(town)

    return datapoints
```

Abbildung 13: Funktion "get_all_towns" (eigene Darstellung)

anschließend in eine Liste an „TownDTOs“ um. Diese Liste wird mithilfe der Methode „insert_all_towns“ aus dem Datenbank-Service mit den Bezirken und Bundesländern aus der Datenbank verknüpft und anschließend auch dort abgespeichert.

```python
# Datei: corona/services/db_service.py:48-57
def insert_all_towns(towns: List[TownDTO]):
    for townDto in towns:
        state = States.objects.get(name = townDto.state)
        if Districts.objects.filter(name = townDto.name).exists():
            district = Districts.objects.get(name = townDto.name)
        else:
            district = Districts.objects.get(name = townDto.district)

        town = Towns(name = townDto.name, plz = townDto.plz, district = district, state = state)
        town.save()
```

Abbildung 14: Funktion "insert_all_towns" (eigene Darstellung)

7 API-Integration

Durch die Einstellung des Dienstes der gegebenen Covid19-API (https://covid19api.com/), musste im Projekt auf eine andere API (https://api.corona-zahlen.org/) ausgewichen werden. Im Folgenden wird sich daher nur auf die in diesem Projekt genutzte API bezogen.

7.1 Aufbau und Abfragen

Die drei großen Endpunkte der API sind: /germany, /districts (Daten für Bezirke) und /states (Daten für Bundesländer). Bei den beiden letzteren Endpunkten muss nach dem Typ immer eine ID kommen. Diese ist bei den Districts eine vier- bis fünfstellige Zahl und bei den Bundesländern die Internationale Abkürzung (zwei Buchstaben). Anschließend kann man mit dem Muster: „/history/typ/zahl“ eine definierte Anzahl (hier „zahl“) an Datenpunkten von der API zu einem Datentypen (hier „typ“) anfragen. Die verschiedenen Datentypen sind alle ausführlich in der Dokumentation der API beschrieben, in diesem Projekt jedoch werden nur die vier Typen: /cases, /deaths, /incidence und /recovered verwendet. Eine Beispiel URL sieht folgendermaßen aus:

https://api.corona-zahlen.org/districts/1001/history/cases/50

Im Code wird die URL innerhalb der Funktion „get_location_data“ im Data-Service generiert. Dazu wird die Funktion „get_api_url“ (Appendix A.1: Code 10) des DTO-Objektes genutzt und mit dem „APIDataTypeEnum“ kombiniert. Dadurch erhält man die gesamte Historie an Datenpunkten, die die API zur Verfügung stellen kann.

```python
# Datei: corona/services/data_service.py:21-22
def get_location_data(location: LocationDTO, startDay: date, endDay: date, dataType: APIDataTypeEnum) -> Dict[date, int | float | None]:
    url = location.get_api_url() + str(dataType)
    response = request_api(url)
```

Abbildung 15: Generierung der URL (eigene Darstellung)

7.2 Probleme bei der Nutzung

Bei der Nutzung der API sind zwei Hauptprobleme aufgetreten. Zum einen stellt die API keine Städte zur Verfügung. Da der User jedoch selten nach einem Bezirk sucht, wurden die Städte manuell implementiert und mit den Bezirken verknüpft. Bei der Suche nach einer Stadt werden dann die Daten des Landkreises genutzt, was dem User zusätzlich bei der Überschrift im Frontend gezeigt wird. Außerdem ist die API beim Abfragen von Bezirksdaten langsam. Eine Anfrage für jegliche Art von Daten dauert zwischen zwei und drei Sekunden. Da für das Frontend insgesamt vier Anfragen getätigt werden, dauert das Laden einer Bezirks bzw. Stadt Seite bis zu 10 Sekunden.

8 Fehlerbehandlung

Zum erleichtertem Debuggen wurden eigene Exceptiones geschrieben (Appendix A.1: Code 10), welche innerhalb des Projektes zum genauen Definieren der Probleme genutzt werden. Dadurch ist es möglich verschiedene Fehler direkt abzufangen und die dazugehörigen Antworten an den Nutzer weiterzuleiten.

Als Anwendungsbeispiel für die Fehlerbehandlung (Exception Handling) wird folgend der „TooManyResultsError“ genutzt. Dieser wird innerhalb der Suchen genutzt um dem Nutzer zu informieren, dass seine Suche zu viele Ergebnisse geliefert hat. Das wird benötigt, damit die Live-Suche, als auch die „search“-Seite nicht zu viele Ergebnisse anzeigen, wodurch das Design gestört werden könnte. Die Ergebnisse werden innerhalb des Errors dennoch abgespeichert, was es ermöglicht diese manuell auf eine bestimmte Anzahl zu begrenzen und dem User im Frontend zu benachrichtigen.

```python
# Datei: corona/views.py:113-134
def live_search(request):
    query = request.GET.get("q", "").lower().strip()
    if query:
        try:
            states, districts, towns = search_for_location(query, max_results=10)

            results = {
                "towns": [town.to_dict() for town in towns],
                "districts": [district.to_dict() for district in districts],
                "states": [state.to_dict() for state in states],
                "querySucceeded": True,
            }

        except TooManyResultsError as e:
            results = {
                "towns": [x.to_dict() for x in e.results["towns"][:3]],
                "districts":  [x.to_dict() for x in e.results["districts"][:3]],
                "states":  [x.to_dict() for x in e.results["states"][:3]],
                "querySucceeded": False,
            }
    else:
        results = {"towns": [], "districts": [], "states": [], "querySucceeded": True}

    return JsonResponse(results)
```

Abbildung 16: Nutzung des "TooManyResultsError" (eigene Darstellung)

9 Zusammenfassung und Ausblick

Die in diesem Projekt entwickelte Webanwendung informiert dem User umfassend über verschiedene Daten innerhalb der gesuchten Region, die im Umgang mit dem Corona-Virus eine rolle spielen. Dabei wurden grundlegende Konzepte aus dem Modul „Datenerhaltung und Datenerfassung 1“ eingebunden und mithilfe des Projektes weiter vertieft. Das Hauptaugenmerk dieser Grundlagen stellt die Einbindung einer normalisierten, relationalen Datenbank, als auch die Nutzung einer externen API dar. Zudem wurden verschiedene Konzepte wie das Erstellen einer Webbasierten Anwendung und die Nutzung von DTO-Objekten innerhalb des Projektes erlernt bzw. weiter vertieft.

Durch den modularen Aufbau der Anwendung hinsichtlich des Front- und Backends ist es möglich in der Zukunft weitere Datentypen hinzuzufügen. Genauer soll die Möglichkeit der API hervorgehoben werden, Daten basierend auf dem Geschlecht oder des Alters zur Verfügung zu stellen. Das ist einer der großen Punkte, die in diesem Projekt noch hinzugefügt werden könnten.

Referenzen

- Django Software Foundation. (2024). *Django Documentation (Version 5.1)*. https://docs.djangoproject.com/en/5.1/
- Bootstrap. (2024). *Bootstrap Documentation (Version 5.3)*. https://getbootstrap.com/docs/5.3/
- Lückert, M. (2024). *API for COVID-19 Data*. https://api.corona-zahlen.org/
- Lauryn Dube. (2024). *Corona API* [GitHub Repository]. https://github.com/LauLauGaamer/Corona-API
- Lauryn Dube. (2024). Corona API [drawsql Diagramm]. https://drawsql.app/teams/lauryn-1/diagrams/corona-api
- andrena objects ag. (2024). Liste-Staedte-in-Deutschland [CSV]. https://github.com/andrena/java8-workshop/blob/master/demos/Liste-Staedte-in-Deutschland.csv

Appendix

A.1 - Codebeispiele

Code 1

```
<!-- Datei: corona/templates/pages/details.html:33 -->
<canvas id="chart-main"></canvas>
```

Code 2

```
// Datei: corona/static/js/details.js:43-46
charts['chart-main'] = new Chart(
    document.getElementById('chart-main').getContext('2d'),
    chartConfigs['chart-main']
);
```

Code 3

```
// Datei: corona/static/js/details.js:113-142
window.moveToCenter = function(chartId) {
    // delete old charts
    if (charts['chart-main']) {
        charts['chart-main'].destroy();
        clearCanvas(0);
    }

    if (charts['chart' + String(chartId)]) {
        charts['chart' + String(chartId)].destroy();
        clearCanvas(chartId);
    }

    chartConfigs[currentChartIDs[0]] = createLineChartConfig(chartConfigs[currentChartIDs[0]].data.datasets[0].label, chartConfig
    chartConfigs[currentChartIDs[chartId]] = createLineChartConfig(chartConfigs[currentChartIDs[chartId]].data.datasets[0].label,

    // Create new Charts
    charts['chart-main'] = new Chart(
        getChartContext(0).getContext('2d'),
        chartConfigs[currentChartIDs[chartId]]
    );
    charts['chart' + String(chartId)] = new Chart(
        getChartContext(chartId).getContext('2d'),
        chartConfigs[currentChartIDs[0]]
    );

    // update chart IDs
    var mainChart = currentChartIDs[0];
    currentChartIDs[0] = currentChartIDs[chartId]
    currentChartIDs[chartId] = mainChart;
};
```

Code 4

```
// Datei: corona/static/js/live_search.js:4-38
if (query.length > 1) {
    fetch(`${window.location.origin}/corona/backend/live-search/?q=${query}`)
        .then(response => {
            if (!response.ok) {
                throw new Error(`HTTP error! status: ${response.status}`);
            }
            return response.json();
        })
        .then(data => {
            resultsDiv.innerHTML = '';

            if (data.towns.length > 0 || data.districts.length > 0 || data.states.length > 0) {
                data.towns.forEach(function(town) {
                    const div = document.createElement('a');
                    div.classList.add('dropdown-item');
                    div.textContent = `${town.name} (PLZ: ${town.plz})`;
                    div.setAttribute("href", 'http://127.0.0.1:8000/corona/details/' + String(town.id) + "?type=" + town.type)
                    resultsDiv.appendChild(div);
                });

                data.districts.forEach(function(district) {
                    const div = document.createElement('a');
                    div.classList.add('dropdown-item');
                    div.textContent = "Landkreis: " + district.name;
                    div.setAttribute("href", 'http://127.0.0.1:8000/corona/details/' + String(district.id) + "?type=" + district.type
                    resultsDiv.appendChild(div);
                });

                data.states.forEach(function(state) {
                    const div = document.createElement('a');
                    div.classList.add('dropdown-item');
                    div.textContent = "Bundesland: " + state.name;
                    div.setAttribute("href", 'http://127.0.0.1:8000/corona/details/' + String(state.abbreviation) + "?type=" + state.
                    resultsDiv.appendChild(div);
                });
```

Code 5

```
# Datei: corona/services/dp_service.py:13-29
def get_state(abbreviation:str) -> StateDTO:
    if abbreviation is not None:
        try:
            state = States.objects.get(abbreviation__iexact=abbreviation)
        except:
            raise MissingDatabaseEntryError(key=abbreviation, table=LocationTypeEnum.STATE)

    return StateDTO(name = state.name, abbreviation = state.abbreviation)


def insert_all_disctricts(districts: List[DistrictDTO]):
    for districtDto in districts:
        if not Districts.objects.filter(name = districtDto.name).exists():
            state = States.objects.get(name = districtDto.state)

            district = Districts(id = districtDto.id, state = state, county = districtDto.county, name = districtDto.name)
            district.save()
```

Code 6

```
# Datei: corona/services/db_service.py:76-86
def filter_all_tables(name: str) -> Tuple[List[TownDTO], List[DistrictDTO], List[StateDTO]]:
    towns = Towns.objects.filter(Q(name__icontains=name) | Q(plz__icontains=name)).all()
    districts = Districts.objects.filter(name__icontains=name).all()
    states = States.objects.filter(name__icontains=name).all()

    # Convert into Lists of DTOs
    towns = [TownDTO(name=x.name, plz=x.plz, district=x.district.id, state=x.state.id) for x in towns]
    districts = [DistrictDTO(id=x.id, state=x.state.id, county=x.county, name=x.name) for x in districts]
    states = [StateDTO(name=x.name, abbreviation=x.abbreviation) for x in states]

    return towns, districts, states
```

Code 7

```python
# Datei: corona/objects/enums.py:3-29
class LocationTypeEnum(Enum):
    TOWN = "Stadt"
    DISTRICT = "Landkreis"
    STATE = "Bundesland"

    def __str__(self):
        return self.value

    @classmethod
    def from_string(cls, string_value):
        if string_value.strip() == "" or string_value is None:
            raise ValueError('Bitte gebe einen Typen als LocationTypeEnum an!')

        for name in cls:
            if name.value.lower() == string_value.lower().strip():
                return name

        raise ValueError(f'{string_value} ist kein Gültiger Wert für LocationTypeEnum!')

    def api_str(self):
        match self:
            case LocationTypeEnum.TOWN:
                return "districts"
            case LocationTypeEnum.DISTRICT:
                return "districts"
            case LocationTypeEnum.STATE:
                return "states"
```

Code 8

```python
# Datei: corona/services/location_service.py:21-30
def get_all_states() -> List[StateDTO]:
    datapoints = []

    request_data = request_api("https://api.corona-zahlen.org/states")

    for abbreviation, data in request_data["data"].items():
        state = StateDTO(name=data["name"], abbreviation=abbreviation)
        datapoints.append(state)

    return datapoints
```

Code 9

```python
# Datei: corona/objects/location_dtos.py:7-21
@dataclass
class LocationDTO:
    name: str
    id: int | str | None
    type: LocationTypeEnum
    api_id: str

    def to_dict(self):
        data = asdict(self)
        if self.type is not None:
            data["type"] = str(self.type)
        return data

    def get_api_url(self):
        return f'https://api.corona-zahlen.org/{self.type.api_str()}/{self.api_id}/history/'
```

Code 10

```python
# Datei: corona/objects/exceptions.py:4-41
class TooManyResultsError(Exception):
    def __init__(self, results_length: int, message: str = "Die Suche hat zu viele Ergebnisse geliefert.", max_results: int = 15, results = {}):
        self.message = message
        self.max_results = max_results
        self.results_length = results_length
        self.results = results

        super().__init__(self.message)

    def __str__(self):
        return f"Die Suche hat zu viele Ergebnisse geliefert. Maximal Erlaubt sind: {self.max_results} (Deine Ergebnisse: {self.results_length})"

class MissingDataPointError(Exception):
    def __init__(self, start_day: date, end_day: date, missing_day: date, message:str = f"Es wurde ein fehlender Datenpunkt innerhalb der Daten festgestellt!"):
        self.message = message
        self.start_day = start_day
        self.end_day = end_day
        self.missing_day = missing_day

        super().__init__(self.message)

    def __str__(self) -> str:
        return f"Es wurde ein fehlender Datenpunkt (Tag: {self.missing_day}) innerhalb der angefragten Punkte gefunden! (Zeitraum: {self.start_day} - {self.end_day})"

class MissingDatabaseEntryError(Exception):
    def __init__(self, key: str | int, table: LocationTypeEnum, message="Ein Eintrag wurde nicht in einer Tabelle gefunden!"):
        self.message = message
        self.key = key
        self.table = table

        super().__init__(self.message)

    def __str__(self) -> str:
        return f"Der Eintrag {self.key} existiert in {self.table} nicht.'"


class MissingParameterError(Exception):
    pass
```

A.2 – Responses

Response 1 (URL: http://127.0.0.1:8000/corona/backend/live-search/?q=Ham)

{"towns": [{"name": "Hamburg", "id": "20038", "type": "Stadt", "api_id": 2000, "plz": "20038", "district": 2000, "state": "HH"}, {"name": "Hamm", "id": "59065", "type": "Stadt", "api_id": 5915, "plz": "59065", "district": 5915, "state": "NW"}, {"name": "Hameln", "id": "31785", "type": "Stadt", "api_id": 3252, "plz": "31785", "district": 3252, "state": "NI"}], "districts": [{"name": "Hamburg", "id": 2000, "type": "Landkreis", "api_id": 2000, "state": "HH", "county": "SK Hamburg"}, {"name": "Hameln-Pyrmont", "id": 3252, "type": "Landkreis", "api_id": 3252, "state": "NI", "county": "LK Hameln-Pyrmont"}, {"name": "Hamm", "id": 5915, "type": "Landkreis", "api_id": 5915, "state": "NW", "county": "SK Hamm"}], "states": [{"name": "Hamburg", "id": "HH", "type": "Bundesland", "api_id": "HH", "abbreviation": "HH"}], "querySucceeded": false}

Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Blomberg, 09 Oktober 2024