zhehe

Release 1.0

Jul 22, 2020

Inhaltsverzeichnis

Hochschule Augsburg
Fakultät für Informatik
Python Webentwicklung
Sommersemester 2020

Table 1: Autor

Vorname	Nachname	Kontakt	Semester
Jonas	Winkler	Jonas.Winkler@Hs-Augsburg.de	06

CHAPTER 1

Einleitung

1.1 Projektbeschreibung

Thema des Projekts ist die Konvertierung verschiedener Markupsprachen in PDF-Dokumente. Nutzern soll es ermöglicht werden beispielsweise ein Markdown-Dokument durch simplen Input umzuwandeln. Des Weiteren soll dem Benutzer ein Dokumentenverwaltungssystem bereitgestellt werden. Hier können erstelle PDF-Dokument editiert, gelöscht oder für weitere Zwecke heruntergeladen werden können. Um diesen Service nutzen zu können, wird eine E-Mail Adresse benötigt, die validiert werden muss.

1.2 Ressourcen

Für die Programmiersprache Python wurden in der Vergangenheit bereits einige Web-Frameworks entwickelt. Die bekanntesten darunter sind: Flusk und Django. Beide stellen gleichartige Ressourcen zur Verfügung, jedoch bietet Django durch vielzählige Publikationen sowie Tutorials und ähnlichen Hilfsmitteln eine sehr viel geringere Einstiegshürde, weshalb ich mich für dieses Projekt für das Python-Web-Framework Django entschieden habe. Für das Frontend gibt es verschiedenste Möglichkeiten, von React über CSS Frameworks wie Bootstrap bis hin zur eigenen HTML/CSS Implementation. Ein Framework wie React oder Flutter für die Erstellung der Webseite zu verwenden ist nach heutigem Standard selbstredend effizienter, jedoch entgegnet man hier einer sehr hohen Einstiegshürde. Aus diesem Grund habe ich mich in diesem Projekt aus zeitlichen Gründen für Bootstrap entschieden. Bootstrap bietet die Möglichkeit, reaktionsfähige Webseiten mithilfe der implementierten CSS-Klassen vereinfacht zu erstellen. Jedoch ist auch dieses Vorgehen mit einigen Hürden verbunden, da es bis heute noch keine gute Literatur gibt, welche das Arbeiten mit Bootstrap 4 erläutert.

CHAPTER 2

Django Spezifisch

2.1 Forms

2.1.1 Setup

Forms dienen der Kommunikation mit Webseiten. Herkömmlich werden diese in einfachem HTML Kontext deklariert und designed. Dies führt jedoch oft zu Komplikationen bezüglich der Darstellung dieser Forms. Bootstrap stellt an dieser Stelle einen sinnvollen Ausweg zur Verfügung, um diese Forms für alle Geräte und Browser wie gedacht darzustellen. Das Django-Framework hingegen bietet in diesem Fall eigene Lösungen, um Forms zu implementieren, um übertragene Informationen im backend einfach zu verarbeiten sind. Hier kommt das **crispy_forms** Package zum Einsatz. Dieses verbindet die Kompatibilität der Django Forms und die Dahrstellungsüberlegenheit von Bootstrap.

Crispy Forms können sowohl unter Anacoonda als auch unter PIP installiert werden.

Anaconda *(vgl. Anaconda Inc. o. J.)*

```
$ conda install -c conda-forge django-crispy-forms
```

PIP *(vgl. Python Software Foundation 2020)*

```
$ pip install django-crispy-forms
```

Um das Package unter Django zu nutzen, muss es unter **settings.py** zu den Apps hinzugefügt werden.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'crispy_forms',
]
```

Um die crispy forms nun auch in den Templates verwenden zu können, müssen diese wie static Files geladen werden.
`{% load crispy_forms_tags %}`

2.1.2 Forms erstellen

forms.py

Django Forms sollten in den jeweiligen Applikation in der Datei **forms.py** erstellt werden.

```
# Django Form import
from django import forms
# Crispy Form Helper import
from crispy_forms.helper import FormHelper
# Crispy CSS Form Helper imports
from crispy_forms.layout import Layout, Div
# Crispy further CSS Form Helper imports
from crispy_forms import bootstrap, layout

# New Class defining the Form to use in Python-Django-Framework
class Newsletter(forms.Form):
    """
    Custom designed newsletter form for bootstrap 4 viewport
    """
    # In this case we use two form-inputs.
    # The placeholder defines what will be displayed to the user prior to the users␣
↪input
    name = forms.CharField(widget=forms.TextInput(attrs={'placeholder': 'Name'}))
    email = forms.EmailField(widget=forms.EmailInput(attrs={'placeholder': 'Email'}))

    # Initialize the form with Bootstrap classes
    def __init__(self, *args, **kwargs):
        """
        Add layout to form attributes
        """
        super().__init__(*args, **kwargs)
        # First create a FormHelper
        self.helper = FormHelper()
        # Define what Class the overall form shall inherit
        self.helper.form_class = 'form-inline justify-content-center'
        # Define Bootstrap design for children of the form
        self.helper.layout = Layout(
            # First we define the field name which in this case ought to be at the␣
↪center of the page
            Div('name', css_class=['form-group', 'justify-content-center']),
            # Second we define the field email which is to the right of name
            Div('email', css_class=['form-group', 'justify-content-center']),
            # We define an BootstrapAction to receive a properly defined bootstrap␣
↪button to submit the user input
            bootstrap.FormActions(
                # First Argument ist the action to be taken when clicked
                # Secondary the label to be displayed on the butten
                # Third the Bootstrap Class to be inherited
                layout.Submit('submit', 'Abonnieren!', css_class='btn btn-primary'))
        )
```

view.py

Nach Erstellung der Form, muss diese der Webseite zur Verfügung gestellt werden. Dies kann durch das **Context-Dictionary** in der **views.py** druchgeführt werden. Hier ein Beispiel, wie dies druchgeführt werden kann.

```
def index(request):
    # Wenn das Formular ausgefüllt wurde, handelt es sich um einen POST request
    if request.method == 'POST':
        newsletter_form: Newsletter = Newsletter(request.POST)
        # Weitere Aktionen mit den gesammelten Daten druchführen
    # Wenn die normale Webseite mit dem Formular geladen wird
    elif request.method == 'GET':
        template_name: str = "zhehe_index/index_main.html"
        # Bereitstellen des Formulars
        form = Newsletter
        # Mithilfe des Context-Dictionaries in der render-funktion, kann die Form an␣
↪das Template geschickt werden
        return render(request=request, template_name=template_name, status=200,␣
↪context={'form': form})
```

template.html

Anschließend muss die Form nur noch in das Template eingebunden werden. Dies ist vergleichsweise einfach, da bereits alle wichtigen Styles übergeben wurden. Um die **crispy_forms** in einem Template nutzen zu können, müssen diese zunächst *geladen* werden.

```
{% block newsletter %}
    {% crispy form %}
{% endblock %}
```

Diese Schreibweise genügt, um eine vollständige Form in HTML darzustellen, es werden keine weiteren Styles oder Ähnliches benötigt. Diese sehr unkomplizierte Implementation ermöglicht es dem Entwickler eine erhebliche Zeitersparnis, da die Form sehr einfach wiederverwendet werden kann und die Nutzerdaten sicher und einfach abgegriffen werden können.

Weiteres über **Crispy Forms** kann selbstverständlich in der Dokumentation nachgelesen werden.

2.2 Django Allauth

2.2.1 Beschreibung

2.2.2 CSSing Forms

Einem Medium Artikel verfasst von Gavin Wiener zu folgen ist es möglich, den allauth-formen CSS-Klassen hinzuzufügen.

```
class MyCustomSignupForm(SignupForm):

def __init__(self, *args, **kwargs):
    super(MyCustomSignupForm, self).__init__(*args, **kwargs)
    self.fields['email'].widget.attrs.update({
        'class': 'red-border'
    })
```

Nach vielen Versuchen und Tests konnte ich dieses Verhalten nicht bestätigen. Es war mir nicht möglich, auf diese Art den Formen CSS-Klassen hinzuzufügen. Weshalb ich mich für eine etwas unsaubere jedoch funktionierende Front-End-Variante entschieden habe.

```
$(function () {
    // Add form-control css-classes to input
    $('#id_email').addClass('form-control');
    $('#id_password1').addClass('form-control');
    $('#id_password2').addClass('form-control');
    $('#id_password').addClass('form-control');
    $('#id_login').addClass('form-control');
    $('#id_oldpassword').addClass('form-control');
});
```

Durch diese Javascriptfunktion ist es möglich, nachträglich CSS-Classen zu den Formen hinzuzufügen. Und so auch weiterhin Bootstrap für das Stylen dieser zu verwenden. Die Funktion sucht lediglich nach der ID des HTML Elements, welche für eindeutig ist und fügt dessen Element die Klasse `form-control` hinzu. Dies ist eine Bootstrap CSS-Klasse, um Formen zu stylen.

2.2.3 Verfication Email

Die Informationen zu Googles SMTP-Einstellungen können hier nachgelesen werden.

In der `settings.py` sollte die Einrichtung der E-Mail-Adresse dann wie folgt aufgebaut werden.

Des Weiteren muss unter Verwendung dieser Einstellung im Google-Konto eine Einstellung vorgenommen werden, damit Google das Versenden der Validierungsmail nicht verhindert und den Log-in durch Django ermöglicht. Weniger sichere Apps gestatten

Manchmal kann es durch Captchas ebenfalls zu Problemen mit dem Google-Konto kommen. Hier muss man gleichermaßen Einstellungen im Konto vornehmen.

CHAPTER 3

Dokumentation

3.1 Read The Docs

ReadTheDocs.org ist nicht nur verantwortlich für das gleichnamige Sphinx-Theme. Hier kann auch jeder Entwickler mit einem Git-Repository seine erstelle Dokumentation hosten. Readthedocs erhebt hierbei keine Kosten. Ein neuer push auf das Repository wird hier automatisch verarbeitet und in die Webseite mit aufgenommen. Das Theme kann hier ebenfalls frei eingestellt werden *(vgl. Read the Docs, Inc & contributors o. J.b)*. Leider ist es jedoch nicht für GitLab Projekte auf den Servern der Hochschule Augsburg geeignet, da keine geeignete Log-in-Funktion für Private Server angeboten wird. Aus diesen Gründen habe ich mich für dieses Projekt für ein GitHub Repository entschieden.

3.1.1 Einstellungen

Um ReadTheDocs mit einer eigenen Konfigurationsdatei verwenden zu können ist zu beachten, dass ReadTheDocs unter Defaulteinstellungen nach einer **contents.rst** Datei sucht. Default von Sphinx ist hier jedoch die **index.rst** Datei. Aus Designgründen habe ich mich in diesem Projekt für die Sphinx-Umgebung entschieden. Dafür muss in der **config.py** ein Eintrag für das Master-Dokument hinzugefügt werden.

```
master_doc = 'index'
```

Durch diesen Einschub sucht ReadTheDocs nach einer index.rst Datei und die Dokumentation kann erzeugt werden.

3.2 Besonderheiten

3.2.1 Literaturverzeichnis

Da zitieren mit Sphinx nicht klassisch implementiert ist, muss man hier eigene Wege gehen. Ich habe mich in diesem Projekt dazu entschieden **ref** Links zu verwenden. **Reflinks** ermöglichen es Hyperlinks zu erzeugen, welche auf Stellen in anderen Dokumenten verweißen.

Beispiel

Zunächst benötigt man das Schlüsselwort **ref**.
Anschließen kommt der Hyperlink Text in (‘).
Jetzt wird der Verweiß in einem anderen Dokument in <Verweiß> angegeben.
Und zuletzt wird die Anweißung wieder mit (‘) geschlossen.

```
# Referezlink
:ref:`HyperlinkText<Referenz-in-anderem-Dokument>`
# Beispiel für Referenz-in anderem Dokument
.. _Referenz-in-anderem-Dokument
```

Dadurch ist es möglich ein Literaturverzeichnis in einer eigenen Datei anzulegen. Dieser Zustand macht das Verzeichnis einfacher zu verwalten und Trennt die verschiedenen Einheiten der Dokumentation.

CHAPTER 4

Produktionsstatus

4.1 Docker - Django

In diesem Teil wird versucht, Docker in Verbindung mit dem Python Django Framework für das Deployment auf einem Server einzurichten. Hierfür werden verschiedene Docker-Container (Nginx, Python, GUnicorn, PostgreSqlDB..) erstellt und mit Docker-Compose ein komplettes System zusammen gesetzt, welches Systemunabhängig auf dem Server eines Kunden deployed werden kann.

Inhalt

4.1.1 Installation

Im nachfolgenden Projekt wird der APT-Packetmanager verwendet, falls Sie einen anderen Packetmanager verwenden, lesen Sie bitte die Installationsdokumentation von Docker an.

Um die neueste Version von Docker zu installieren, sollten zunächst alle vorherigen Docker-Installationen vom Ihrem System entfernt werden.

```
$ sudo apt-get remove docker docker-engine docker.io containerd runc
```

4.1.2 Linux Mint

Wie Docker auf der Homepage erwähnt, untstützt und tested die Docker-Community Installationen lediglich auf Mainstream Plattformen wie Ubunut, Debian, Windows usw. Linux Mint ist ein Abkömmling von Ubuntu mit einer eignen Oberfläche und eigenen Settings und Ressourcen. Aus diesem Grund funktioniert die Installationsanleitung von Docker nicht für Linux Mint, wesshalb ich hier eine kurze Installationsanleitung verfasst habe.

1. Das System auf den neusten Stand bringen

```
$ sudo apt-get update
```

2. Site-Packages für den Zugriff über Https

```
$ sudo apt-get install apt-transport-https ca-certificates curl gnupg-
↪agent software-properties-common
```

3. Hinzufügen des Docker GPG-Keys

```
$ sudo apt-get install apt-transport-https ca-certificates
```

4. Den Key verifizieren

```
$ sudo apt-key fingerprint 0EBFCD88
```

5. Repository hinzufügen

```
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/
↪linux/ubuntu bionic stable"
```

6. Den Packetmanager neu laden

```
$ sudo apt-get update
```

7. Docker und Docker-Compose installieren

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-
↪compose
```

4.1.3 Andere Betriebssysteme

Für die meisten anderen Betriebssystem kann der Intstallationsanleitung der Docker Dokumentation gefolgt werden, um Docker und Docker-Compose zu installieren.

4.1.4 Projektstruktur

docker-compose.yml

Die *docker-compose.yml* Datei dient des Aufbaus für Docker-Compose. Hier werden einige Konfigurationsdaten eingestellt. Es wird eine Datenbank benötigt, auf welche die Applikation zugreifen kann, um beispielsweiße Userdaten oder ähnliches abzuspeichern. Zusätzlich wird die Dango-Applikation in einem Container gestartet. Dazu benötigen wir eine eigenz kozipierte DockerFile, um lokal Daten zu verarbeiten. Des Weiteren wird ein GUnicorn-Wsgi-Webserver benötigt, um die Tasks von Django zu verarbeitent. Dieser startet dann die Webapplikation. Um Statische Dateien konventionell ablegen zu können, benötigen wir noch einen Web-Server, hier Nginx.

```
version: '3.1'
volumes:
  pgdata:
  static_files:                                          # Volume für static files
  media_content:                                         # Volume für media files
services:
  postgre_db:                                            # DNS für Datenbank-Service
    image: library/postgres:latest                       # Image für Postgre-DB
    environment:
      POSTGRES_DB: postgreDB                             # Datenbank Name
      POSTGRES_USER: user                                # Datenbank User Name
      POSTGRES_PASSWORD: password                        # Datenbank User Passwort
    volumes:
      - pgdata:/var/utils/posgresql/data                 # Default Volume für Postgres␣
↪Datenbank
    ports:
      - 5432:5432                                        # Port für Zugriff auf die␣
↪Datenbank
  django_app:                                            # DNS für Web-Applikation
      build:
        context: .                                       # Root Verzeichnis des␣
↪Projekts
        dockerfile: docker/webapp/Dockerfile             # Dockerfile, wird benötigt,␣
↪um Container zu bauen
      restart: always                                    # Starte Applikation neu,␣
↪wenn Fehler auftritt
      volumes:
      - ./webapp:/webapp                                 # Das Volumen/Die App, welche␣
↪eingebunden werden soll
      - static_files:/static_files                       # Volumen für static file
      - media_content:/media_content                     # Volumen für media files
      ports:
      - 8000:8000                                            # Port, auf welchem die␣
↪Webseite ereicht werden kann
      command: gunicorn -w 4 webapp.wsgi -b 0.0.0.0:8000     # GUnicorn Middleware-
↪Webserver für Produktion
  nginx:                                                     # DNS für Nginx Server
      build:
        context: .                                       # Root Verzeichnis für Nginx␣
↪Server
        dockerfile: docker/nginx/Dockerfile              # Dockerfile, wird benötigt␣
↪um Server zu konfigurieren
      volumes:
        - static_files:/static_files                     # shared volume mit django-
↪web-applikation für static files
        - media_content:/media_content                   # shared volume mit django-
↪web-applikation für media files
      ports:
        - 8080:80                                        # Leitet port 8080 an Port 80␣
↪weiter
```

nginx Verzeichnis

Das *Nginx Verzeichnis* beinhaltet alle wichtigen Dateien, um den Webserver zu starten. Sowie das Image, welches für Docker verwendet wird.

DockerFile

Um einen eigenn Docker-Container zu erzeugen, wird eine *DockerFile* benötigt. Es sind verschiedene Kommandos möglich und die Datei folgt einer strikten Syntax. Mehr über die DockerFile kann hier in der DockerFile-Dokumentation in Erfahrung gebracht werden.

```
FROM nginx:latest                                          # Docker Image für Nginx-
↪Web-Server

RUN rm /etc/nginx/conf.d/default.conf                      # Löschen der Standart␣
↪Konfigurationsdatei

COPY ./docker/nginx/webapp.conf /etc/nginx/conf.d/         # Kopieren unserer␣
↪Konfigurationsdatei
```

Nginx-Konfiguration

Um den Server mit den nötigen Einstellungen auszustatten, wie beispielsweiße dem nötigen Static und Media Kontent abzuspeichern, muss der Server konfiguriert werden. Dies lässt sich mit einer *.conf* Datei bewältigen. Mehr über die Konfiguration von Nginx kann in der Nginx-Dokumentation nachgelesen werden.

```
server {
listen 80;                                  # Port auf dem http requests eingehen
server_name localhost;                      # Name des Servers
access_log  /var/log/nginx/example.log;     # Verzeichnis für logs
server_tokens off;                          # Verhindert, dass Serverinformationen␣
↪nach außen sichtbar sind.

location /media/ {                          # Verzeichnis für media content
    autoindex off;                          # Abschalten der automatisch generierten␣
↪Index Seite (Dort werden die Dateien im Verzeichnis angezeigt: html, css, ...)
    alias /media_content/;                  # Weiterleitung an den tatsächlichen␣
↪Speicherort
}

location /static/ {                         # Wie bei media Dateien
    autoindex off;
    alias /static_files/;
}

location / {                                # Root Verzeichnis
    try_files $uri $uri/ @python_django;    # Wenn requested URI eine Datei oder ein␣
↪Ordner ist, wird dieser versandt. Andernfalls weiterleitung an @python_django
}

location @python_django {
    proxy_pass http://django_app:8000;                                  # Weiterleitung␣
↪an GUnicorn Server auf Port 8000
    proxy_pass_request_headers on;                                      # Der Host wird␣
↪geforwarded (Beispiel -> Django_app.com)
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;    # Der Request␣
↪wird an Django weitergeleitet (Django hat keine Ahnung von Proxy, deshalb werden IP
↪'s weitergeleitet werden)
    proxy_set_header Host $http_host;                                   # Proxy kennt den␣
↪Host-Header nicht, deshalb muss auch dieser weitergeleitet werden
```

(continues on next page)

(continued from previous page)

```
    proxy_set_header X-Forwarded-Proto $scheme;                    # Beispiel wenn␣
↪nginx hat ssl proxy, dann muss das an django weitergeleitet werden
    proxy_redirect off;                                            # Sollte von␣
↪Django übernommen werden
}
}
```

django_app Verzeichnis

In diesem Verzeichnis werden alle DockerFile für das Deployment sowie für die Entwicklung hinterlegt. Besonders für die Entwicklung wird eine spezielle Konfiguration benötigt.

DockerFile

```
FROM python:3.8.3-buster
MAINTAINER user@localhost

COPY ./django_app /django_app

WORKDIR /django_app

RUN pip install -r requirements/deploy.txt                    # hier requirements/
↪deploy.txt oder requirements/dev.txt
                                                              # Um die benötigten␣
↪Python-Packages zu installieren.
                                                              # deploy.txt benötigt␣
↪beispielsweiße gunicorn, dev jedoch nicht.
COPY ./docker/webapp/entrypoint.sh /entrypoint.sh

RUN chmod +x /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]
```

Um beim Start des Dockers eine Ausgabe auf der Konsole zu sehen, macht es Sinn ein solches Entryskript zu verwenden.

entrypoint.sh

```
#!/bin/bash

echo "Running command '$*'"
exec /bin/bash -c "$*"
```

4.1.5 Django

Auch in Django müssen ein paar Änderungen vorgenommen werden. Zunächst muss die Datenbank richtig initialisiert werden. In diesem Beispiel verwenden wir eine Postgresql Datenbank. Eine relationale Datenbank, die von den Django Entwicklern empfohlen wird. Um da zu bewerkstelligen, muss die Datenbank in der *settings.py* eingerichtet werden.

```
DATABASES = {
'default': {
    'ENGINE': 'django.db.backends.postgresql_psycopg2',
    'NAME': 'database_name',
    'USER': 'user_name',
    'PASSWORD': 'user_password',
    'HOST': 'postgre_db',           # Der Hostname, welcher in der docker-compose.yml
↪als Service für die Datenbank eingetragen wurde
    'PORT': '5432',                 # Der Port, welcher für die Datenbank gedacht
↪wurde --> docker-compose.yml
    }
}
```

Des weiteren müssen die *Static und Media Files* noch umgeleitet werden. Dazu benötigen wir die Volumes, welche in der docker-compose.yml festgelegt wurden. In diesem Projekt, sähe das wie folgt in der *settings.py* Datei aus.

```
STATIC_ROOT = '/static_files/'
MEDIA_ROOT = '/media_content/'
```

Zu guterletzt sollte der Debugmodus von Django noch deaktiviert werden. Dies kann ebenfall in der *settings.py* vorgenommen werden.

```
DEBUG = False
```

4.1.6 Docker-Compose

Um die die Docker-Container jetzt zusammenzusetzen benötigen wir *docker compose*. Um die Umgebung für die Produktion zu erstellen, muss dieses zunächst *gebuilded* werden.

```
$ docker-compose build
```

Um den Service nun zustarten, genügt es in der Konsole folgenden Befehl einzugeben.

```
$ docker-compose up
```

Der Nginx-Webserver läuft jetzt mit 4 Gunicorn Workern stabil und kann auf einem beliebiegen Server installiert werden.

CHAPTER 5

Installation

5.1 Server

Da es sich bei dem Projekt um ein Produktionsfähiges System handelt, kann die Applikation nicht durch den Django-Debug-Server gestartet werden. Um die verschiedenen Docker-Einheiten und damit den Server zu initialisieren, müssen die Images aufgebaut und gestartet werden. `Dieser Process kann einmalig bis zu 15 Minuten dauern`

```
$ docker-compose up --build
```

Anschließend müssen innherhalb des Zhehe Docker-Containers verschiedene Operationen ausgeführt werden. Dies sollte aus Sicherheitsgründen stehts von dem Verwaltenten Administrator durchgeführt werden, da dies andernfall zu einer Sicherheitslücke führen kann.

Wenn Docker den Aufbau der Images fertiggestellt hat, welchseln Sie in ein neues Fenster und geben Sie folgenden Befehl ein, um zu sehen welche Images auf Ihrem System derzeit laufen.

```
$ docker ps
```

Der Output des Befehls sollte in etwa so aussehen.

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS              PORTS                    NAMES
e430143a6129        python-web-wpf_zhehe   "/entrypoint.sh guni…"   16 hours ago        Up 16 hours         0.0.0.0:8000->8000/tcp   python-web-wpf_zhehe_1
acc5434944c7        postgres:latest        "docker-entrypoint.s…"   17 hours ago        Up 10 seconds       0.0.0.0:5432->5432/tcp   python-web-wpf_postgre_db_1
90637bb00d1f        python-web-wpf_nginx   "/docker-entrypoint.…"   17 hours ago        Up 11 seconds       0.0.0.0:8080->80/tcp     python-web-wpf_nginx_1
```

Ganz links sehen Sie die `ID` des Containers und in der zweiten Spalte den `Namen` des Images. Wechseln Sie jetzt in den Container mit dem Namen **python-web-wpf-zhehe** mit folgendem Befehl. Ändern Sie die ID des Containers entsprechend Ihrer `docker ps` Ausgabe.

```
$ docker exec -it e430143a6129 bash
```

Sie befinden sich nun innerhalb des Docker-Containers. Führen Sie jetzt nur die `init.sh` aus und füllen Sie die Daten für den Superuser aus.

Um die Initialisierung innerhalb des Docker-Containers zu vereinfachen habe ich ein Skript `init.sh` erstellt, welches die verschiedenen Operationen durchführt. Das Skript arbeitet folgende Schritte ab.

---

python manage.py migrate

Dieses Kommando erstellt die Tabellen in der Datenbank.
Des Weiteren wird der Name der Domain auf `zhehe.com` verändert,
sowie auch der Name bezogen auf die Webseite.

---

python manage.py createsuperuser

Dieses Kommando ermöglicht es dem Administrator einen Super-User anzulegen, mit
welchem Benutzerdaten verwaltet werden können.

---

python manage.py collectstatic

Dieses Kommando verschiebt alle statischen Dateien in den in der `settings.py` angegebenen
`STATIC_ROOT` Ordner, damit der Webserver diese zur Verfügung stellen kann.

Nach der Ausführung der Initialisierung kann die Webseite nun unter `0.0.0.0:8080` aufgefunden und ausgeführt werden.

CHAPTER 6

Quellen

6.1 Literaturverzeichnis

Anaconda Inc. (o. J.): Django Crispy Forms :: Anaconda Cloud, in: Anaconda, [online] https://anaconda.org/conda-forge/django-crispy-forms [10.07.2020].

Python Software Foundation (2020): Django-crispy-forms, in: PyPI, [online] https://pypi.org/project/django-crispy-forms/ [10.07.2020].

Read the Docs, Inc & contributors (o. J.): Home | Read the Docs, in: readthedocs, [online] https://readthedocs.org/ [07.07.2020].