Das Django Web-Framework dargestellt anhand des Praktischen Beispieles eines Inventarisierungssystemes

Clemens Dautermann

2. Januar 2019 bis 8. Februar 2019

Inhaltsverzeichnis

1 Einleitung **3**

2 Struktur **3**
2.1 Erstellung . . . 4
2.2 manage.py . . . 4
2.3 db.sqlite3 . . . 5
2.4 server/server . . . 5
2.4.1 __init__.py . . . 5
2.4.2 settings.py . . . 5
2.4.3 urls.py . . . 5
2.4.4 wsgi.py . . . 6
2.5 server/app1 . . . 6
2.5.1 admin.py . . . 6
2.5.2 apps.py . . . 8
2.5.3 forms.py . . . 8
2.5.4 models.py . . . 9
2.5.5 tests.py . . . 10
2.5.6 views.py . . . 10
2.5.7 migrations . . . 11

3 Routing in Django **11**
3.1 Root Url Config . . . 11
3.2 Weitere Url Konfigurationsdateien . . . 12
3.3 View . . . 12

4 Template rendering **12**
4.1 Variablen an ein Template übergeben . . . 13
4.2 ”Django template language” . . . 13
4.2.1 Ein Beispiel . . . 13
4.3 Rendering . . . 14

5 Den Inventarisierungsserver einrichten **14**
5.1 Installation . . . 14
5.2 Einen Datenbankbenutzer erstellen . . . 14

6 Erklärung der Benutzeroberfläche **15**

1 Einleitung

Diese Facharbeit soll einen grundlegenden Überblick über die wichtigsten Funktionen des Django Web-Frameworks geben.
Das Django Web-Framework ist ein größtenteils in Python geschriebenes[1] Framework zum entwickeln von Webservern. Es ist aufgrund seiner ausgeprägten Modularität und der Existenz einer Vielzahl von Datenbanktreibern besonders gut für die Entwicklung von Webservern geeignet, die eine Datenbank erfordern.
Django stellt eine grundlegende Struktur für die Entwicklung zur Verfügung. So zum Beispiel:

- Eine settings.py Die genutzt werden kann um Konfigurationsmöglichkeiten zentral zu bündeln
- Eine library um einfache Zugriffe auf Datenbanken zu tätigen und sogenannte Models um Datenbankobjekte zu verwalten
- Ein Routingsystem um eine einfachere Verwaltung von Urls zu gewährleisten
- Eine Grundstruktur, die Modularität unterstützt und das einfache Installieren oder Entfernen von sogenannten ”Apps” ermöglicht

Es ist also kaum notwendig, jedoch durchaus möglich, als Entwickler noch SQL zu schreiben wenn man mit dem Django Web-Framework entwickelt.

2 Struktur

Ein typischer Django Server ist aus sogenannten ”Apps” aufgebaut. Diese werden entweder vom Entwickler selber geschrieben oder können via pip (dem Python Paket Manager) installiert werden. Ein standard Verzeichnisaufbau ist in Abbildung 1 dargestellt.

[1] Offizielle Django GitHub Seite https://github.com/django/django

```
server
├── manage.py
├── db.sqlite3
├── server
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── app1
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── forms.py
    ├── models.py
    ├── tests.py
    ├── urls.py
    ├── views.py
    └── migrations
        └── 0001_initial.py
```

Abbildung 1: Die typische Verzeichnisstruktur eines Django Servers

2.1 Erstellung

Ein Django Projekt kann mit dem Befehl $ django-admin startproject server initialisiert werden. Dadurch wird folgende Ordnerstruktur erstellt:

```
server
├── manage.py
└── server
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Abbildung 2: Verzeichnisstruktur, die der $ django-admin startproject server Befehl erzeugt

2.2 manage.py

Die manage.py wird, wie der Name schon sagt, verwendet um den Server zu verwalten. Mit Hilfe der manage.py können beispielsweise Migrierungen an der Datenbank erstellt werden, Datenbanknutzer erstellt werden oder der Testserver zur Entwicklung kann gestartet werden. Die gleiche Funktionalität stellt auch der django-admin Befehl zur Verfügung[2].

[2]Django Dokumentation https://docs.djangoproject.com/en/2.1/ref/django-admin/

2.3 db.sqlite3

In dieser Datei wird die SQL Datenbank gespeichert, die der Server nutzt. Sie wird automatisch erstellt. Es können jedoch auch andere Datenbanken, wie zum Beispiel MongoDB oder eine extern gehostete Datenbank, verwendet werden.

2.4 server/server

2.4.1 __init__.py

Diese Datei befindet sich im Wurzelverzeichnis jeder App. Sie macht für Python erkennbar, dass es sich bei dem Inhalt dieses Ordners um ein Python Modul handelt. Somit kann die App einfach geteilt und von anderen Nutzern verwendet werden.[linenos, frame=lines, framesep=2mm]Python

2.4.2 settings.py

In dieser Datei befinden sich die Einstellungen für den Django Server. Mit ihrer Hilfe werden Zeitzone, Sprache, Datenbankkonfiguration und viele andere Konfigurationen verwaltet. Man kann sie auch verwenden um eigene Einstellungsmöglichkeiten anzubieten. Dafür definiert man eine Konstante (in Python typischerweise durch Großbuchstaben ausgedrückt) und einen Wert. Zum Beispiel
LOGIN_REDIRECT_URL = "/". Im Falle dieses Projektes wurde beispielsweise die Konstante LOGFILE = 'serverlog.log' definiert um zentral auf die Logdatei zugreifen zu können. Auf die in der settings.py definierten Werte kann aus jeder App zugegriffen werden, indem unter Benutzung der Anweisung
`from django.conf import settings` diese importiert wird. Anschließend kann via `file = settings.LOGFILE` beispielsweise auf die Konstante LOGFILE zugegriffen werden.

2.4.3 urls.py

Diese Datei ist der erste und wichtigste Teil des Django Routing Systems. Über sie werden die grundlegenden URL Strukturen des Servers definiert. Sie importiert urls.py Dateien aus anderen Apps und definiert wie auf eine Bestimmte URL reagiert werden soll. Im Falle dieses Projektes sieht sie folgendermaßen aus:

```
1 from django.contrib import admin
2 from django.urls import path, include
3
4 urlpatterns = [
5 path('admin/', admin.site.urls),
6 path('accounts/', include('django.contrib.auth.urls')),
7 path('', include('usermanager.urls')),
8 path('add/', include('objectadder.urls')),
9 path('list/', include('objectlister.urls')),
10 path('settings/', include('settingsapp.urls')),
11 ]
```

Hier werden zuerst Pakete für das admin-Interface und Pakete für das URL-Routing importiert. Anschließend wird eine Liste urlpatterns definiert. Diese enthält alle URL Pfade. include() Importiert dabei die urls.py Dateien aus den anderen apps.

2.4.4 wsgi.py

Diese Datei stellt ein ”application” genanntes WSGI Objekt zur Verfügung. Es wird zum starten des Testservers genutzt. In Verbindung mit ”mod_wsgi” kann auch Apache dieses Objekt nutzen.[3]

2.5 server/app1

Dieses Verzeichnis enthält alle Dateien, die eine App ausmachen. In diesem Fall heißt die App beispielhaft ”app1”. Die Dateien, die in Abb 1 erscheinen, jedoch hier nicht erwähnt werden, haben die selbe Funktion wie die gleichnamigen Dateien im ”server/server” Ordner.

2.5.1 admin.py

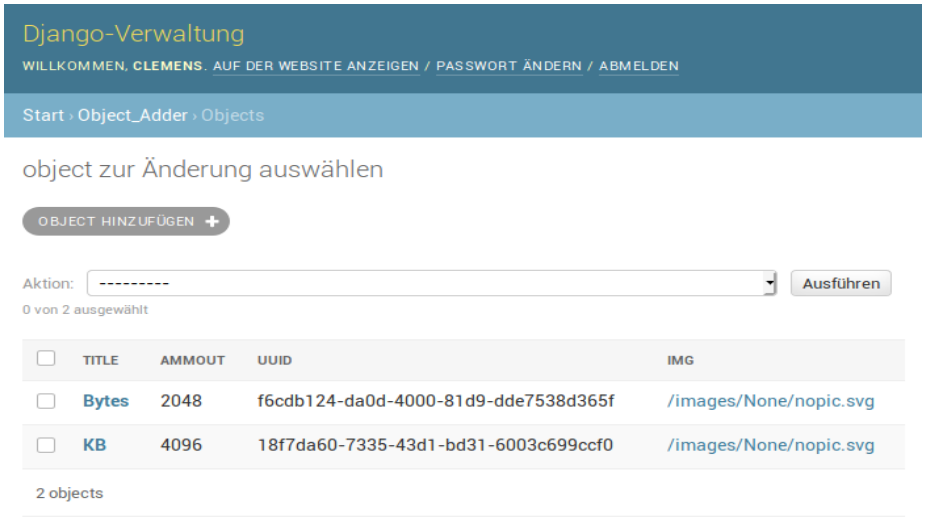
Die admin.py dient dazu das Django Admin Interface zu konfigurieren. In ihr werden die Models registriert, die auf der admin Seite zu sehen sind, und wie diese dargestellt werden sollen. Sie sieht beispielsweise in der ”object_adder” app folgendermaßen aus:

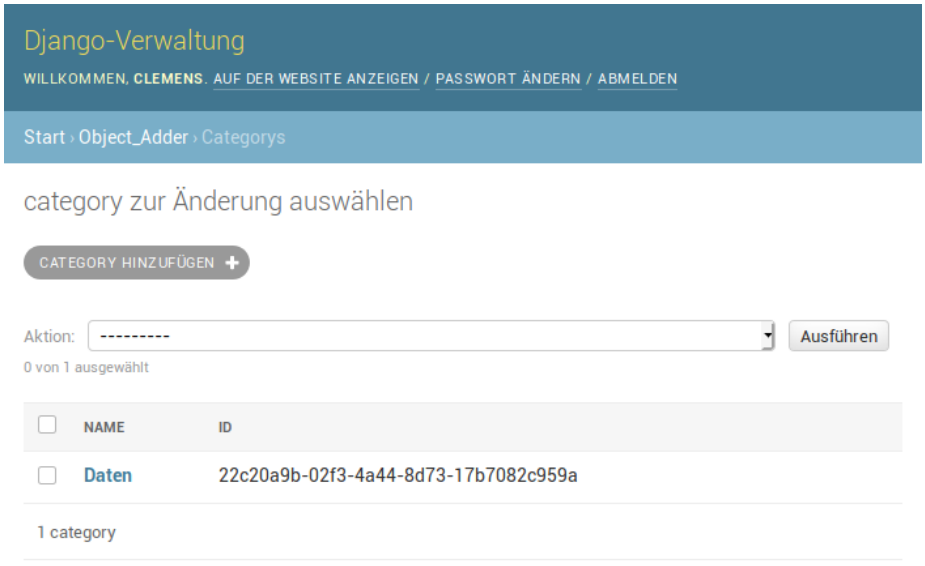
```
1  from django.contrib import admin
2  from .models import Object, Category
3
4
5  class ObjectAdmin(admin.ModelAdmin):
6          listdisplay = ('title', 'ammout', 'uuid', 'img')
7
8
9  class CategoryAdmin(admin.ModelAdmin):
10         listdisplay = ('name', 'id')
11
12
13 # Register your models here.
14 admin.site.register(Object, ObjectAdmin)
15 admin.site.register(Category, CategoryAdmin)
```

In Zeile 2 werden die in der models.py definierten Models importiert um Zugriff auf diese zu erlangen.
Anschließend wird für jedes Model eine Klasse erstellt. In dieser Klasse wird jeweils eine ”list_display” Variable definiert, die ein Tupel enthält, mit dem alle darzustellenden Attribute an die admin library übergeben werden können. Diese Abmin-Klassen

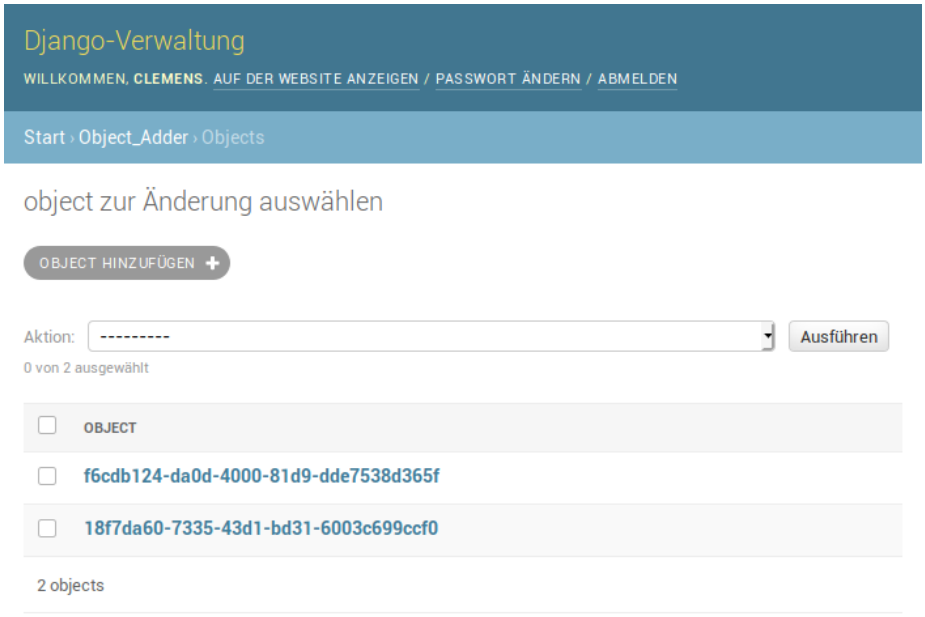
[3] Django Dokumentation https://docs.djangoproject.com/en/2.1/howto/deployment/wsgi/

müssen jetzt noch zusammen mit dem Model an die admin library übergeben werden. Dies erfolgt duch die Befehle in Zeile 14 und 15. Dieser Code erzeugt demnach folgendes Admin-Interface:





Ohne die ”list_display” Angebe würde nur der primary key des Models angezeigt und das Interface sähe folgendermaßen aus:



2.5.2 apps.py

Hier ist der Name der app definiert. Diese Datei ist außerdem notwendig, damit die App von Django als solche erkannt wird.

```
1 from django.apps import AppConfig
2
3 class ObjectAdderConfig(AppConfig):
4         name = 'objectadder'
```

2.5.3 forms.py

In dieser Datei werden Formulare definiert. So beispielsweise das Formular zur Erstellung von Objekten und Kategorien, das folgendermaßen aussieht:

```
1 from django.forms import ModelForm, TextInput
2 from .models import Object, Category
3
```

```
4
5  class ObjectForm(ModelForm):
6          class Meta:
7                  model = Object
8                  fields = ('ammout', 'title', 'img',
9                  'description', 'category')
10                 widgets = -
11                         'title': TextInput(),
12
13
14
15 class CategoryForm(ModelForm):
16         class Meta:
17                 model = Category
18                 fields = ['name']
19                 widgets = -
20                         'name': TextInput()
21
```

Hier werden zwei Formulare definiert. Es wird angegeben mit welchem model das Formular asoziiert werden soll und welche Felder angezeigt werden sollen. Außerdem wird definiert, dass für das name und das title Feld ein ”TextInput()” Feld genutzt werden soll.

2.5.4 models.py

In der models.py werden Models definiert, die in Verbindung mit der Datenbank genutzt werden können. Diese werden innerhalb von Django als Objekte repräsentiert, können jedoch trotzdem in einer SQL Datenbank gespeichert werden. Eine mopels.py kann beispielsweise folgendermaßen aussehen:

```
1  from django.db import models
2  import uuid
3
4          class Category(models.Model):
5                  name = models.TextField(blank=False, maxlength=150)
6                  id = models.UUIDField(
7                  primarykey=True, default=uuid.uuid4, editable=False
8                  )
9                  categories = models.Manager()
10
11                 def str(self):
12                         return self.name
```

In dieser Datei wird ein Model definiert. Es heißt ”Category” und hat zwei Eigenschaften. Eine ID und einen Namen. Es wird dafür erst eine Klasse namens ”Category” er-

stellt, die Subklasse der django.db.models.model Klasse ist. In dieser Klasse werden jetzt die Datenfelder des Objektes definiert, diese sind später in der Datenbank die Spalten. den Datenfeldern können sogenannte Felder zugewiesen werden, die angeben welcher Datentyp in dieser Spalte steht[4]. Ein UUIDField fasst zum Beispiel eine UUID, ein TextField Text ein DateField ein Datum und so weiter. Der Parameter ”blank” gibt an ob das Feld bei der Erstellung des Objektes blank gelassen werden darf. ”max_length” ist ein für das TextField spezifischer Parameter, die maximale Länge des Strings angibt, der gespeichert werden kann. ”primary_key” gibt an ob dieses Feld der primary Key in der Datenbank ist. Dieser Wert kann nur einem Feld zugewiesen werden. Wenn kein Feld als primary Key gesetzt wurde, erstellt Django automatisch ein ID Feld, das dann primary Key ist. Der ”default” Wert gibt an, Welcher Standardwert in das Feld eingespeichert werden soll, wenn vom Nutzer kein Wert übergeben wurde. ”editable” gibt an, ob das Feld manuell bearbeitet werden kann, zum Beispiel im admin Interface.
der Aufruf ”categories = models.Manager()” ist notwendig um via ”categories = Category.categories.all()” alle Kategorien auf einmal als Liste abfragen zu können.
”def __str__(self):” ist eine sogenannte ”magic method” in Python. Sie ermöglicht es anzugeben, wie das Objekt als String repräsentiert werden soll. Wenn also ”print(category)” aufgerufen wird, wird aufgrund dieser Methode an stelle des primary Keys, der UUID, der Name ausgegeben.

2.5.5 tests.py

Django weist ein umfangreiches System zur Unterstützung von testgetriebener Entwicklung auf. Dieses Testsystem basiert auf dem ”unittest” Pythonmodul. Die tests.py ist die Datei, in der die Tests für diese App geschrieben werden sollen [5]. Mit Hilfe der Tests können beispielsweise Methoden von Models getestet werden oder es kann überprüft werden, ob das richtige Template mit dem richtigen Kontext gerendert wird.

2.5.6 views.py

Diese Datei enthält alle ”views”. Ein View ist eine Methode, die auf die Anfrage des Nutzers reagiert, alle nötigen Berechnungen tätigt und letztendlich das gerenderte Template zurück gibt. Ein View muss immer ein Objekt des Typen ”HttpResponse” oder einer seiner Subklassen zurück geben. Ein einfacher View kann zum Beispiel folgendermaßen aussehen:

```
1 from django.shortcuts import render
2
3 def mainview(request):
4         context = -'a': 'b', 'beispiel': True
5     return render(request, 'exampleapp/index.html', context)
```

[4]Django Model field reference https://docs.djangoproject.com/en/2.1/ref/models/fields/
[5]Writing and running tests Django Dokumentation
https://docs.djangoproject.com/en/2.1/topics/testing/overview/

Die ”context” Varieble, wird von der render() Methode genutzt um Variablen im Template zu rendern. Die Werte im context können beispielsweise aus der Datenbank abgefragt werden.

2.5.7 migrations

Dieser Ordner enthält Dateien, die durch den makemigrations Befehl erstellt wurden. Sie geben an, ob und wie die Models verändert wurden. Man muss diese Dateien in der Regel als Entwickler nicht selber bearbeiten.

3 Routing in Django

Das grundlegende und wichtigste Prinzip in Django ist das sogenannte ”Routing”. Dieses Prinzip beschreibt den Weg, den eine Anfrage zurücklegt um zu einer Antwort zu führen. Routing in Django funktioniert, indem die Url mit Hilfe von Regular Expressions in Teile aufgespalten wird, die dann einzeln bis letztendlich ein HTML Template zurück gegeben wird verfolgt werden.

3.1 Root Url Config

Diese Datei ist die grundlegende Url Konfigurationsdatei. In ihr schaut der Server zuerst nach. Sie befindet sich im server/server/urls.py Ordner und gehört zur Hauptapp des Servers. Sie kann beispielsweise folgendermaßen aussehen:

```
"""invsystem URL Configuration

The urlpatterns list routes URLs to views. For more
information please see:
https://docs.djangoproject.com/en/2.1/topics/http/urls/
Examples:
Function views
1. Add an import:  from myapp import views
2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
1. Add an import:  from otherapp.views import Home
2. Add a URL to urlpatterns:  path('', Home.asview(), name='home')
Including another URLconf
1. Import the include() function:
from django.urls import include, path
2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
```

```
22 path('admin/', admin.site.urls),
23 path('accounts/', include('django.contrib.auth.urls')),
24 path('', include('usermanager.urls')),
25 path('add/', include('objectadder.urls')),
26 path('list/', include('objectlister.urls')),
27 path('settings/', include('settingsapp.urls')),
28 ]
```

Die Kommentare werden beim Erstellen des Projektes automatisch generiert. Diese Datei erzeugt also sechs URL Pfade. Alle Pfade bis auf den ”/admin” Pfad zeigen hier auf die urls.py Datei einer anderen App. Dies wird mit dem ”include()” Statement erreicht. Die ”/admin” Url wird mit hilfe des Admin Paketes gehandhabt, dass automatisch ein Admin Interface zur Datenbankverwaltung erzeugt.

3.2 Weitere Url Konfigurationsdateien

Mithilfe der ”include()” Statements kann die Anfrage jetzt theoretisch durch eine unbegrenzte Zahl von urls.py Dateien geleitet werden. In diesem Beispiel wird in der settings_app.urls jedoch direkt der sogenannte ”View” zurück gegeben.

```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5 path('', views.index, name='settingsindex'),
6 ]
```

Dies liegt daran, dass in der ”path()” Funktion kein ”include()” Statement mehr steht sondern ”views.index”. Außerdem wird in der zweiten Zeile die ”views.py” der App importiert.

3.3 View

Im sogenannten View wird findet die eigentliche Programmlogik statt. Hier werden Daten an die ”render()” Funktion übergeben, die das gerenderte HTML Template zurück gibt. Diese Daten können beispielsweise aus einem Formular stammen oder von der Datenbank abgefragt sein. Hier werden auch POST Anfragen bearbeitet.

4 Template rendering

Ein weiterer zentraler Aspekt des Django Frameworks ist das sogenannte ”Template rendering”. Es ermöglicht Variabln und dynamische Inhalte in das HTML Dokument einzubinden.

4.1 Variablen an ein Template übergeben

Um Variablen an das Template weiterzugeben wird im entsprechenden view als dritte Variable ein Dictionary übergeben. Auf die Werte in diesem Dictionary cann jetzt innerhalb des Templates beim rendern zugegriffen werden.

```
1 from django.shortcuts import render
2
3 def mainview(request):
4         context = -'a': 'b', 'beispiel': True, 'liste': range(10)
5         return render(request, 'exampleapp/index.html', context)
```

In diesem Beispiel wird context = 'a': 'b', 'beispiel': True, 'liste': range(10) als Kontextdictionary übergeben. Das heißt innerhalb des HTML Templates kann auf die Variablen "a", "beispiel" und "liste" zugegriffen werden.

4.2 "Django template language"

In django existiert eine kleine eigene Skriptsprache, die "django template language". Mit ihrer Hilfe kann in Templates auf den übergebenen Kontext eingegangen werden. Sie enthält if-Abfragen, for-Schleifen und vieles mehr, kann aber auch einfach Variablen einbinden. " var " gibt dabei immer eine einzubindende Variable an, "{%%}" zeigt Befehlssequenzen an.

4.2.1 Ein Beispiel

Im folgenden Beispiel wird eine Liste namens "Items" übergeben. Es soll erst überprüft werden ob diese wirklich übergeben worden ist, und dann jedes Element ausgegeben werden. Der <head> tag wurde im Beispiel weggelassen, da er für diese Demonstration irrelevant ist.

```
<html>
  <body>
  {% if Items is not None %}
  <ul>
      {% for item in items %}
        <li>{{item.name}}</li>
      {% endfor %}
  </ul>
  {% else %}
        <p>Es wurde keine Liste übergeben</p>
  {% endif %}
  </body>
</html>
```

4.3 Rendering

Das ”rendering” findet in dem Moment statt, wo in dem View die ”render()” Methode aufgerufen wird. Diese erstellt aus dem Kontext und dem Template eine fertige HTML Datei, die dem Benutzer angezeigt wird. Sie beachtet dazu die Kontrollsequenzen im Template und setzt die entsprechenden Variablen aus dem Kontext ein.

5 Den Inventarisierungsserver einrichten

Der Inventarisierungsserver basiert auf einem Virutualisierungssystem namens Docker. Um den Server einzurichten wird daher Docker sowie das zugehörige Programm Docker compose benötigt.

5.1 Installation

1. Installieren sie Docker und Docker compose. Sie sind auf `https://www.docker.com/get-started` und `https://docs.docker.com/compose/install/` verfügbar. Dort ist auch der Installationsvorgang genauer beschrieben.

2. Clonen sie das GitHub Repository von `git@github.com:Clemens-Dautermann/Inventarium.git`

3. Entpacken sie das Repository und öffnen sie ein Terminal im entpackten Ordner, wo sich die Datei ”docker-compose.yml” befindet.

4. Führen sie den Befehl ”docker-compose up” aus. Dieser wird automatisch Container für eine PostgresSQL Datenbank und den Inventarisierungsserver herunterladen und konfigurieren. Die Ausführung dieses Befehls kann einige Zeit dauern und benötigt eine aktive Internetverbindung sowie einiges an Festplattenspeicherplatz

5. Sie müssen jetzt einen Datenbankbenutzer erstellen

5.2 Einen Datenbankbenutzer erstellen

Aus Sicherheitsgründen ist das gesamte Webinterface des Servers passwortgeschützt. Das heißt, es wird ein Nutzerkonto benötigt um auf dieses zugreifen zu können. Da aber noch kein Benutzer existiert um überhaupt auf die Datenbank zuzugreifen und weitere Nutzer zu erstellen, muss direkt an der Datenbank ein Nutzer erstellt werden. Dies tun sie folgendermaßen:

1. Führen sie den Befehl ”docker ps” aus während der Server läuft. Die Ausgabe sollte etwa folgendermaßen aussehen:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
f1b377c3700f	invsystem_web	”python3 manage.py r…”	1 minute ago	Up 1 minute
1289f9bcdd04	postgres	”docker-entrypoint.s…”	1 minute ago	Up 1 minute

2. Kopieren sie die CONTAINER ID von invsystem_web und führen sie den Befehl ”docker exec -i -t <CONTAINER ID> /bin/bash” aus. Dieser öffnet eine Shell im Container des Servers.

6 Erklärung der Benutzeroberfläche