Django

Installation

1. Python herunterladen von python.org
2. Installation prüfen mit `python --version` im Terminal
3. pip Paket installieren mit `pip install pipenv` (optional)
4. Download VS Code
5. VS Code einrichten
6. Extension herunterladen (Python, Prettier)

erstes Projekt anlegen

1. Ordner erstellen `mkdir new-ordner`
2. Django installieren `pipenv install django`
   pipenv hat eine virtuelle Umgebung installiert und hat django mit rein installiert
3. run `pipenv shell` (bei Neustart des PC wieder neu eingeben!)
4. `django-Admin` zeigt dir die verschiedenen Befehle an!
5. `django-admin startproject name .` legt das Django Projekt an
6. `python manage.py migrate`
7. Dev Server starten `python manage.py runserver`

Projekt Strukturen

- ***db.sqllite3*** = Database SQL Lite Version 3 (Datenbank)
- *mySQL* und *Mongo* sind auch Datenbanken die mit Django funtkionieren
- ***__init__.py*** ist dafür um zu sagen dass es sich um ein Package handelt(wird nichts geändert)
- ***wsgi.py*** = Web Server Gateway Interface(hier wird ebenfalls nichts geändert)
- ***asgi.py*** = Asynchronois Server Gateway interface (handled die Asynchronität des Servers)
- ***urls.py*** in dieser Datei findet man alle Urls die im Projekt angelegt worden sind

Test Ausgabe in der urls.py

```
from django.http import JsonResponse

def hallo(request):
    return JsonResponse("Hallo Welt!", safe=False)
```

```
urlpatterns = [path("", hallo)]
```

! INFO JsonResponse wird häufig für API anfragen genutzt!

APPS

Apps sind kleine Teile des Projektes, dass eine eigene kleine Website ist.

1. Das Projekt
2. App mit den eigenen Funktion (es können mehrere Apps enthalten sein)
    - URLs
    - Templates
    - Views
    - Models

- Um eine App zu kreieren `python manage.py startapp name`
- mit der admin.py registriert man **models** , **Benutzer** und über den Browser der Webanwendung einloggen(admin interface)
- in der apps.py werden die Haupteinstellungen für die App vorgenommen
- models.py beinhaltet die **Datenmodelle** der App und definiert die Struktur der Datenbank(wird das tatsächlich design hergestellt)
- tests.py kann man units Tests erstellen, um die Anwendung zu testen
- views.py ist eine ***wichtige*** Datei und enthält alle Views in Form von Klassen, stellt dass interface da mit welchem der Nutzer interagiert.

Django weiß noch nicht dass die neue App existiert, deshalb muss diese in dem Kern Projekt registriert werden:

1. settings.py aufrufen
2. ```
   INSTALLED_APPS = ['meinProjekt.apps.MeinprojektConfig']
   ```
   ```

(am besten den genauen Pfad angeben NameDesProjekts.apps.NameDerClassInApps)

INSTALLED APPS

1. `'django.contrib.admin'` (erstellt ein Admin Interface)
2. `'django.contrib.auth'` (hilft uns unsere Nutzer zu Authentifizieren)
3. `'django.contrib.contenttypes'` (unterstützt mit einem Highlevel Interface, um mit meinen Datenmodels zu interagieren)
4. `'django.contrib.session'` (hinterlegt einen temporären speicher, um zb Nutzerdaten zu verwalten)
5. `'django.contrib.messages'` (um den Nutzer Notifications anzeigen zulassen)
6. `'django.contrib.staticfiles'` (um statische Dateien wie Bilder und CSS ausgeben zu lassen)

VIEWS

Views haben mit dem Frontend nichts zu tun, es sind einfach nur ***Request Handler***. Sie verarbeiten urls, die ein Nutzer aufruft.

- Views können in 2 Formen implemitiert werden **funktionsbasiert** oder **klassenbasiert** (django wird meist mit den funktionsbasierten Views genutzt)
- nimmt eine Anfrage(Request) entgegen und antwortet darauf(Response)

URLs

Kümmert sich in den *URLPATTERNS* die URLs mit den entsprechenden *Views* zu mappen.

1. In der APP eine neue Datei erstellen mit dem Namen ***urls.py***
2. Die Imports und die View verknüpfen

```
    from django.urls import path
    from . import views

    urlpatterns = [
    path(
          "hallo/", views.HelloWorld, name="hallo"
       )  # kann 3 Argumente entgegen nehmen, den URL, die View die gemappt
und aufgerufen werden soll, ein Name
    ]
```

3. die App ***urls.py*** in der overall urls.py registrieren, damit Django weiß das diese urls.py existiert.

```
    from django.contrib import admin
    from django.http import JsonResponse
    from django.urls import include, path
```

```
    # includes muss importiert werden

    urlpatterns = [path("admin/", admin.site.urls), path("",
include("meinProjekt.urls"))]
    # und hier ausgeführt werden mit dem Verweis auf welche urls.py er
zugreifen soll
```

Templates

Templates sind für das Frontend in Django verantwortlich

1. Templates Ordner im Overall anlegen / kann man auch in der App anlegen, für größere Projekte vom Vorteil
2. Im Templater Ordner eine start.html anlegen
3. die Templates müssen in der **settings.py** unter TEMPLATES registriert werden

```
    TEMPLATES = [
        # anderer Code
        "DIR": [BASE_DIR / "templates"]
    ]
    # BASE_DIR steht für das Overall Verzeichnis
```

4. dann die Templates in den jeweiligen Views hinzufügen

```
    def start(request):
        return render(request, "start.hml")


    # render function erwartet 2 parameter und einen optional 3ten
```

! INFO Don´t repeat yourself!

5. Eine Base HTML struktur festlegen:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Django</title>
  </head>

  <body>
    {% block content %} {% endblock content %}
```

```
  </body>
</html>
```

6. in den anderen Templates hinzufügen:

```
{% extends 'main.html' %} <br />
{% block content %}

<h1>Startseite</h1>
{% endblock content %}
```

eine weitere Möglichkeit zu **extends** zum ableiten der *Templates*, ist es **include** hinzuzufügen.
`{% include 'navbar.html' %}`

desweiteren unterstüzt die Django Engine Template noch weitere features wie zb schleifen, if-else-statements und Variablen!

dritter Parameter für die Render function

- kann eine Variable oder ein Python dictonary sein:
  `render(request, 'template.html', {'name': 'name'})`
- im Template wird es dann in den Mustaches ausgeben:
  `{{ name }}`

programmier Logik einfügen

- ein if-else-statement geht wie folgt:

```
{% extends 'main.html' %} {% block content %} {%if name%}
<h1>Hallo {{ name }}</h1>

{% else %}
<h2>Name nicht vorhanden</h2>
{% endif %} {% endblock content %}
```

Models

Die Models in Django Repräsentieren die Datenbank Tabellen aus der Datenbank, als Classes in deinem Code.

Du kannst auch mit den Models verschiedene Beziehungen mit den Datenbank Tabellen herstellen. (1:1, 1:n, n:n)

Um die Models in Django zu verwenden benutzen wir das ORM, welches Django gleich mitbringt. (ORM = *Objektrelationale Abbildung / object-relational mapping*)

Beispiele für Tabellenfelder:

- Boolenfield = True or False
- CharField = String Feld wo man eine bestimmte länge mit angeben kann
- DateTimeField = um das Datum hinzuzufügen
- mehr hier

1. erstelle eine Python Class in models.py:

```
class beispiel(models.Model):
   titel = models.charfield(max_length=100)
   beschreibung = models.TextField(null = True, blank = True)
   text = models.TextField(null = True, blank = True)
```

2. eine String rückgabe Function erstellen:

```
class beispiel(models.Model):
   titel = models.charfield(max_length=100)
   beschreibung = models.TextField(null = True, blank = True)
   text = models.TextField(null = True, blank = True)

   def __string__(self):
      return self.titel
```

3. Wenn man sein Model angelegt hat ist dass erste was man tun soll, eine migration anzulegen:

   `python manage.py makemigrations`

Dass Terminal sagt dir wenn es eine migrations erstellt wurde.

4. Immer wenn du eine neue Migrations erstellt hast müssen diese aktiviert werden: `python manage.py migrate`

5. Jetzt ist die neue Tabelle in der SQL Lite Datenbank gespeichert!

Admin Interface

1. Der DevServer muss laufen und die Entwicklungsungebung
   `pipenv shell`
   `python manage.py runserver`

2. ruf im Browser localhost:800/admin auf

3. erstelle einen Admin User: `python manage.py createsuperuser`

! INFO Mit `python manage.py` kannst du dir alle Befehle ausgeben lassen!

4. damit man seine eigene Tabelle sieht muss diese aktiviert werden, dazu geht man in die APP auf die admin.py und gibt folgendes ein:

```
from .models import nameModel

admin.site.register(nameModel)
```

5. Jetzt kann man auf der Django-Admin Seite Elemente zur Tabelle hinzufügen und speichern!

Elemente aus der Tabelle ausgeben

1. In der views.py, dass Model importieren welches abgefragt werden soll:

```
from .models import modelsName # import

def start(request):
newName = modelsName.objects.all # variable erstellen
return render(request, "start.html", {"newName": newName})
# den 3ten parameter für die neue Variable verwenden
```

Info:

- **objects** ist ein model Manager der es uns ermöglicht die abfrage an das Model zu stellen
- `all()` werden alle Elemente von der Model Datenbank zurückgeben (alternative: `get(für ein bestimmtes Element), filter(um die Elemente nach einem Filter zu filtern), exclude(um bestimmte Elemente auszusotieren)`)

2. dann auf das Template in welchem das Model ausgeben werden soll:

```
{% extends 'main.html' %} {% block content %}
<h1>Startseite</h1>

<div>
  {% for artikel in artikels %}
  <!--for-schleife um jedes Element auszugebn-->
  <div>
    <h3>{{artikel.title}}: {{artikel.text}}</h3>
    <!-- wie bei vue.js-->
  </div>
  {% endfor%}
  <!--wichtig ist die schleife zu beenden-->
</div>
{% endblock content %}
```

Debbugin Django Toolbar

- eine debbugin Möglichkeit ist die Django Toolbar die direkt im WebBrowser angezeigt wird `pipenv install django-debug-toolbar` `pip install django-debug-toolbar`
    - in der settings.py bei installed apps registrieren `debug_toolbar`
    - dann noch bei der MIDDLEWARE an erster Stelle einfügen: `"debug_toolbar.middleware.DebugToolbarMiddleware"`
    - eine INTERNAL_IPS anlegen:

```
INTERNAL_IPS = ["127.0.0.1"]
```

        - urls.py muss die Toolbar importiert werden:

```
import debug_toolbar #toolbar importieren
from django.contrib import admin
from django.http import JsonResponse
from django.urls import include, path

urlpatterns = [
   path("admin/", admin.site.urls),
   path("", include("meinProjekt.urls")),
   path("__debug__", include(debug_toolbar.urls)),
] #path für die toolbar erstellen
```