Webprogrammierung mit Python

Spieltag 8 – Session- und Datei-Behandlung, Benutzer-Profil

WiSe 2020/21 Prof. Dr. Amy Siu

Lernziel(e)

- Django's Session-Behandlung kennenlernen
- `static`-Funktionalität vertiefen
- Django's Datei-Behandlung kennenlernen
- Ansätze zum Benutzer-Profil kennenlernen

Session-Behandlung

Web-Session

Was?

- Verbindung eines Web-Clients mit einem Web-Server
- Beginnt mit Login
- Ended mit Logout

Wie bei Django realisiert?

- Cookies mit Session-ID
- Implizite bzw. versteckte Behandlung per `SessionMiddleware`
- Informationen in `request.session` zugreifbar

Dokumentation: `https://docs.djangoproject.com/en/3.0/topics/http/sessions/`

Sicherheit

Eingebaute Maßnahmen bei Django:

- Cross site scripting (XSS) protection
  Benutzer setzt Client-Side-Script in den Browser von einem anderen Benutzer
- Cross site request forgery (CSRF) protection
  Benutzer verkleidet sich als ein anderer Benutzer und führt eine Aktion aus
- SQL injection protection
  Benutzer löst die Ausführung eines SQL-Codes aus
- Clickjacking protection
  Eine Website hüllt (wraps) eine andere Website als ein Frame ein

Verschiedene Middlewares sind dafür zuständig

Dokumentation: `https://docs.djangoproject.com/en/3.0/topics/security/`

`https`

`https` ...

- ist nicht ganz so einfach, in `localhost` zu realisieren
- benötigt praktisch kein Code, nur System-Einstellungen
  $\rightarrow$ typischerweise erst in Produktions-System ermöglicht

`static`

static Dateien

- Dateien wie Grafik, Java-Skripte und css sind static, d.h. unverändert
- `django.contrib.staticfiles` ist die dafür zuständige Django-App

Dokumentationen:

`https://docs.djangoproject.com/en/3.0/ref/contrib/staticfiles/`

`https://docs.djangoproject.com/en/3.0/ref/settings/#static-files/`

Während Entwicklungs-Phase

Django-App ist nach Standard-Einstellung schon aktiviert:

```
# settings.py
INSTALLED_APPS = [
    'django.contrib.staticfiles',
]
```

Nach Standard-Einstellungen sind static Dateien in 2 Methoden gesucht:

```
# wuerde in settings.py sein, aber nur zum Anpassen offenbaren
STATICFILES_FINDERS = [
    'django.contrib.staticfiles.finders.FileSystemFinder', # Ganze Website
    'django.contrib.staticfiles.finders.AppDirectoriesFinder', # App-Spezifisch
]
```

Während Entwicklungs-Phase

Ordner erstellen, Dateien dort entsprechend speichern:

```
# Ganze Website
static/css/
static/images/ # usw.

# App-spezifisch
app_A/static/app_A/
app_B/static/app_B/ # usw.
```

Weitere Einstellungen:

```
# settings.py
STATIC_URL = '/static/'

STATICFILES_DIRS = [
    BASE_DIR / 'static',
]
```

Während Entwicklungs-Phase

Dateien im Template zugreifen:

```
{% load static %}

<link rel="stylesheet" type="text/css"
    href="{% static 'css/datei_name_C.css' %}">

<link rel="stylesheet" type="text/css"
    href="{% static 'app_A/datei_name_D.css' %}">
```

Datei-Behandlung, oder `media`

`static` vs. `media`

`static`:

- Gehört zum System, relativ unveränderte Dateien

`media`:

- Gehört zum Modell, z.B. mp3-Dateien bei Spotify
- Dateien von Benutzern, typischerweise hochgeladen

Einstellungen

`media`-Ordner und evtl. z.B. auch `media/images`-Ordner erstellen, danach:

```
# settings.py
MEDIA_ROOT = BASE_DIR / 'media'
MEDIA_URL = '/media/'
```

```
# urls.py
from django.conf import settings
from django.conf.urls.static import static

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root= settings.MEDIA_ROOT)
```

Model, Form, Template

- In Model, `FileField` und `ImageField` benutzen
  Dokumentationen:

  `https://docs.djangoproject.com/en/3.0/topics/files/#using-files-in-models`

  `https://docs.djangoproject.com/en/3.0/ref/files/`
- In Form, die entsprechenden Fields ganz normal auflisten
- In Template, den URL mit `class.field.url` darstellen und

```
<form method="post" enctype="multipart/form-data">
```

Hinweise:

- `ImageField` benötigt das `Pillow`-Package
  Mit `pip install Pillow` installieren
- Zugang durch Django-Admin-Portal nicht vergessen!

Quelle der Lego-Avatars: `https://www.flaticon.com/authors/smashicons`

Benutzer-Profil

`User`-Model erweitern

Um Djangos `User`-Model zu erweitern, gibt es mehrere Möglichkeiten.

Keine neue Fields erwünscht:

- Proxy-Model: Nur Verhalten erweitern

Neue Fields erwünscht:

- `OneToOneField` benutzen
- `AbstractUser` beerben
- `AbstractBaseUser` beerben – viel Arbeit!

`OneToOneField`

`OneToOneField`

- Model: `User`-Klasse mittels `OneToOneField` mit einer neuen Klasse verknüpfen, dazu extra Fields einfügen
- Views, Forms, Templates entsprechend implementieren

Offizielle Hilfestellung:

`https://docs.djangoproject.com/en/3.0/topics/auth/customizing/`

`#extending-the-existing-user-model`

`AbstractUser` erweitern

`AbstractUser` erweitern

- Superclass importieren:

```
# models.py
from django.contrib.auth.models import AbstractUser
```

- Superclass erben:

```
class MyUser(AbstractUser):
```

  Danach je nach Bedarf die Subclass schreiben

- Neue eigene Klasse einstellen:

```
# settings.py
AUTH_USER_MODEL = 'App_Name.MyUser'
```

- Typischerweise muss die Datenbank “von Anfang an” erneut erstellt werden

`AbstractUser` erweitern

Offizielle Hilfestellung:

`https://docs.djangoproject.com/en/3.0/topics/auth/customizing/`

`#substituting-a-custom-user-model`

Tutorial im Internet:

`https://wsvincent.com/django-custom-user-model-tutorial/`

`AbstractUser` erweitern

Neue Fields:

- `date_of_birth`
- `profile_picture`
- `is_a_cat`

Nicht vergessen:

```
python manage.py makemigrations
python manage.py migrate
```

Weitere Django-Klassen erweitern

`UserCreationForm` erweitern

- Superclass importieren:

```
# forms.py
from django.contrib.auth.forms import UserCreationForm
```

- Superclass erben:

```
class MySignUp(UserCreationForm):
```

- Eigenes User-Model benutzen:

```
class Meta:
    model = MyUser
```

- Nicht vergessen, Template anzupassen, z.B.:

```
<form method="post" enctype="multipart/form-data">
```

`UserCreationForm` erweitern

- URL-Verknüpfung nicht vergessen:

```
# Useradmin/urls.py
urlpatterns = [
    path('signup/', views.MySignUpView.as_view(), name='signup'),
]
```

`LoginView` erweitern

- Superclass importieren:

```
# views.py
from django.contrib.auth.views import LoginView
```

- Superclass erben:

```
class MyLoginView(LoginView):
```

- Attribut überschreiben, damit `registration`-Ordner weg wird:

```
template_name = 'login.html'
```

- `form_valid()`-Methode überschreiben, damit eigenem Code nach Login ausgeführt wird:

```
def form_valid(self, form):
    ...
    form.get_user().execute_after_login()  # Custom code
    ...
```

`LoginView` erweitern

- URL-Verknüpfung nicht vergessen:

```
# Useradmin/urls.py
urlpatterns = [
    path('login/', views.MyLoginView.as_view(), name='login'),
]
```

`TemplateView` erweitern

- Superclass importieren:

```python
# views.py
from django.views.generic.base import TemplateView
```

- Superclass erben:

```python
class HomeBirthdayView(TemplateView):
```

- `get_context_data()`-Methode überschreiben, damit Information aus eigenem User-Model eingebaut wird:

```python
def get_context_data(self, **kwargs):
    user = self.request.user
    # Information aus user zugreifen
    # context entsprechend ergaenzen
```

`TemplateView` erweitern

- Context im Template benutzen:

```
{% if user_has_birthday_today %}
    <p>Happy Birthday!</p>
{% endif %}
```

`TemplateView` erweitern

- URL-Verknüpfung nicht vergessen:

```
# urls.py
urlpatterns = [
  path('home/', HomeBirthdayView.as_view(template_name='home.html'),
       name='home'),
]
```